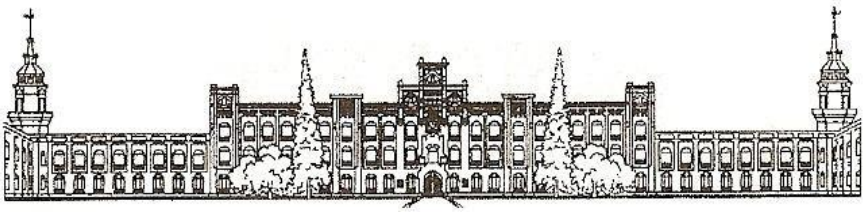


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Лабораторна робота №2

з дисципліни «Бази даних і засоби управління»

« Засоби оптимізації роботи СУБД PostgreSQL »

Виконав студент групи: КВ-33
ПІБ: Щербатюк Є. О.
Перевірив: Павловський В. І.

Київ 2025

Мета: здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Завдання:

1. Перетворити модуль “Модель” з шаблону MVC РГР 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.

Завдання за варіантом:

19	*BTree, BRIN*	*before insert, delete*

Виконання роботи:

Нижче наведені сутності предметної області «Система обліку автомобільного парку компанії» та зв’язки між ними.

Сутності предметної області:

1. **Автомобіль (Car)** – представляє транспортні засоби компанії, які використовуються для вантажних перевезень:
   - Атрибути: VIN (унікальний номер кузова), реєстраційний номер автомобіля, марка, вантажопідйомність.

2. **Водій (Driver)** – представляє працівників, які мають водійське посвідчення та керують транспортним засобом.
   - Атрибути: номер водійського посвідчення, ім’я, прізвище, категорія.

3. **Маршрут (Route) –** ця сутність описує шлях перевезення вантажу:
   - Атрибути: пункт відправлення, пункт призначення, відстань.

4. **Клієнт (Customer)** – ця сутність представляє собою компанію, яка замовляє вантажні перевезення:
   - Атрибути: назва компанії, телефон, email.
5. **Рейс (Trip) –** це окрема сутність, яка описує факт перевезення і через зовнішні ключі реалізує зв’язок між автомобілем, водієм, клієнтом і маршрутом.
   - Атрибути: дата виїзду, дата прибуття, дата повернення, опис вантажу, вага вантажу.

6. **Обслуговування (Service) –** ця сутність є записом про технічне обслуговування автомобілів.
   – Атрибут: дата обслуговування, опис виконаних робіт, вартість обслуговування.

Зв’язки між стуностями предметної області:

Центральною сутністю моделі є Рейс (Trip). Вона має власні атрибути (дати виїзду, прибуття та повернення, опис і вагу вантажу) і через зовнішні ключі реалізує зв’язок між чотирма іншими сутностями: Автомобіль (Car), Водій (Driver), Клієнт (Customer) та Маршрут (Route). Таким чином, Рейс виступає не набором окремих зв’язків, а єдиним комплексним зв’язком, що фіксує факт конкретного перевезення.

Додатково в моделі передбачено зв’язки 1:N:

- Car 1:N Service – один автомобіль може мати кілька записів про обслуговування.
- Customer 1:N Trip – один клієнт може замовити кілька рейсів.
- Driver 1:N Trip – один водій може виконати кілька рейсів.
- Route 1:N Trip – один маршрут може бути використаний у кількох рейсах.

Графічне подання концептуальної моделі «Сутність-зв’язок» зображено на рисунку 1

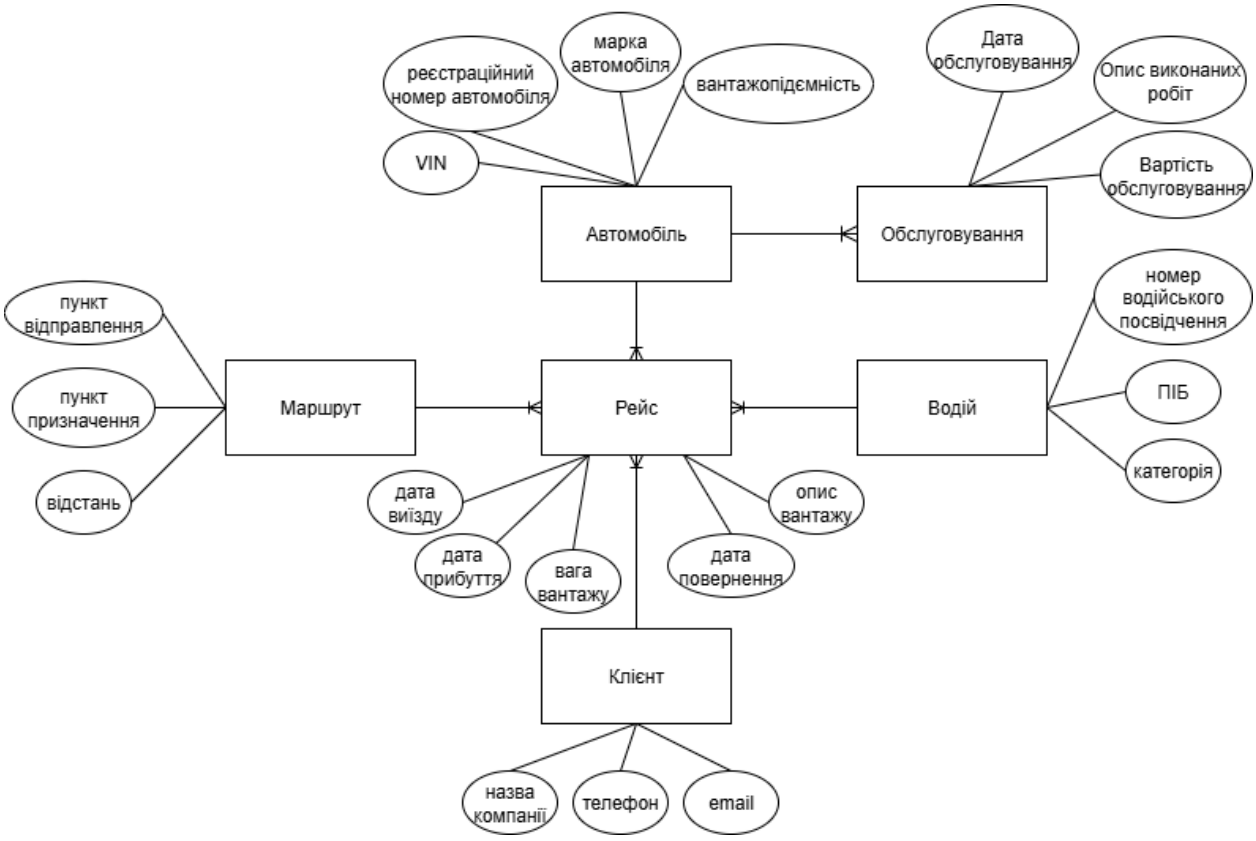


Рисунок 1 – ER-діаграма, побудована за нотацією” Пташина лапка”

Графічне подання логічної моделі «Сутність-зв’язок» зображено на рисунку 2

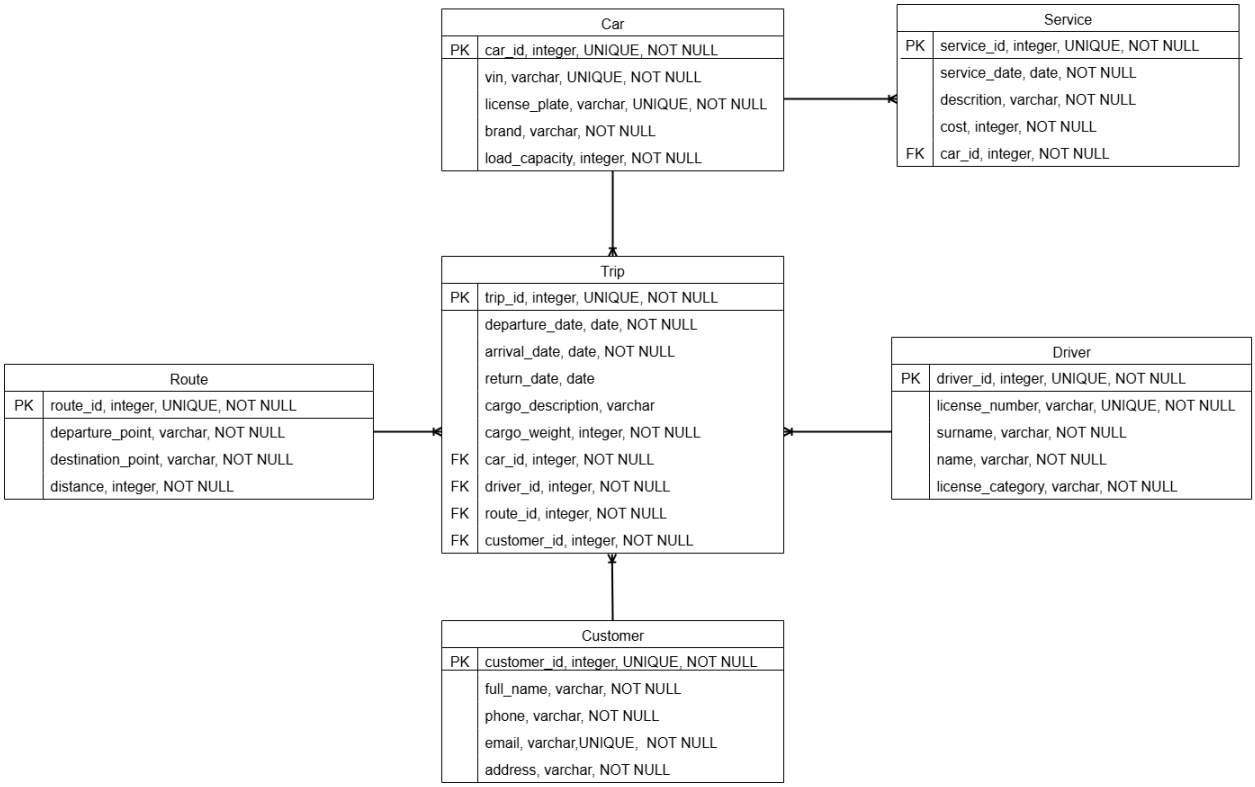


Рисунок 2 – Схема бази даних

Середовище та компоненти розробки

У процесі розробки була використана мова програмування Python, середовище розробки Pycharm, а також була використана ORM Sqlalchemy та бібліотека psycopg3, яка надає API для взаємодії з базою даних PostgreSQL.

Демонстрація роботи коду після перетворення у вигляд ORM

```
--- ГОЛОВНЕ МЕНЮ ---
1.  Показати дані (Show data)
2.  Додати дані (Add data)
3.  Редагувати дані (Update data)
4.  Видалити дані (Delete data)
5.  Згенерувати дані (Generate data)
6.  Вихід (Quit)
Ваш вибір:
```

Рисунок 3 – Меню за умовою завдання залишилось не зміненим

```
--- Меню Генерації Даних (Generate data) ---
1.  Згенерувати 'Car'
2.  Згенерувати 'Driver'
3.  Згенерувати 'Route'
4.  Згенерувати 'Customer'
5.  Згенерувати 'Trip' (з FK)
6.  Згенерувати 'Service' (з FK)
0.  <= Повернутись до головного меню
Ваш вибір: 5
Введіть кількість записів для генерації: 10
Генерація 10 рейсів...

*** Успішно згенеровано 10 рейсів. ***
```

Рисунок 4 – Підменю генерації даних та генерація до таблиці ‘trip’

```
SELECT * FROM trip
ORDER BY trip_id DESC
LIMIT 10;
```

Data Output | Messages | Notifications

Showing rows: 1 to 10 | Page No: 1

	trip_id [PK] integer	departure_date date	arrival_date date	return_date date	cargo_description character varying (200)	cargo_weight integer	car_id integer	driver_id integer	route_id integer	customer_id integer
1	300015	2025-10-28	2025-10-29	2025-11-02	Металопрокат	14800	84120	94	24	95
2	300014	2025-11-23	2025-11-29	2025-12-01	Металопрокат	18700	31548	101	15	5
3	300013	2025-11-25	2025-11-27	2025-12-01	Металопрокат	16300	58506	19	43	23
4	300012	2025-11-15	2025-11-16	2025-11-20	Техніка	16600	4031	32	14	3
5	300011	2025-11-25	2025-11-26	2025-11-30	Металопрокат	19300	49378	79	92	80
6	300010	2025-11-08	2025-11-13	2025-11-18	Нафтопродукти	13800	39267	60	81	87
7	300009	2025-11-21	2025-11-25	2025-11-29	Лікарські препарати	14400	31998	5	1	15
8	300008	2025-11-23	2025-11-26	2025-11-30	Хімікати	21000	26521	73	13	78
9	300007	2025-11-20	2025-11-22	2025-11-27	Техніка	11200	50656	5	30	99
10	300006	2025-11-07	2025-11-08	2025-11-09	Хімікати	15200	24272	10	16	72

Рисунок 5 – Згенеровані дані відображені у pgAdmin4

```
--- Меню Редагування Даних (Update data) ---
1.  Редагувати 'Car'
2.  Редагувати 'Driver'
3.  Редагувати 'Customer'
4.  Редагувати 'Route'
5.  Редагувати 'Service'
6.  Редагувати 'Trip'
0.  <= Повернутись до головного меню
Ваш вибір: 6

--- Редагування 'Trip' ---
ID рейсу для оновлення: 300015
Новий Опис вантажу (enter, щоб пропустити): Нафтопродукти
Нова Вага (кг) (enter, щоб пропустити): 15000

*** Запис (ID: 300015) оновлено. ***
```

Рисунок 6 – Підменю оновлення даних та оновлення попередньо згенерованих даних

Query | Query History | Scratch Pad

```
SELECT * FROM trip
ORDER BY trip_id DESC
LIMIT 10;
```

Data Output | Messages | Notifications

Showing rows: 1 to 10 | Page No: 1

	trip_id [PK] integer	departure_date date	arrival_date date	return_date date	cargo_description character varying (200)	cargo_weight integer	car_id integer	driver_id integer	route_id integer	customer_id integer
1	300015	2025-10-28	2025-10-29	2025-11-02	Нафтопродукти	15000	84120	94	24	95
2	300014	2025-11-23	2025-11-29	2025-12-01	Металопрокат	18700	31548	101	15	5
3	300013	2025-11-25	2025-11-27	2025-12-01	Металопрокат	16300	58506	19	43	23
4	300012	2025-11-15	2025-11-16	2025-11-20	Техніка	16600	4031	32	14	3
5	300011	2025-11-25	2025-11-26	2025-11-30	Металопрокат	19300	49378	79	92	80
6	300010	2025-11-08	2025-11-13	2025-11-18	Нафтопродукти	13800	39267	60	81	87
7	300009	2025-11-21	2025-11-25	2025-11-29	Лікарські препарати	14400	31998	5	1	15
8	300008	2025-11-23	2025-11-26	2025-11-30	Хімікати	21000	26521	73	13	78
9	300007	2025-11-20	2025-11-22	2025-11-27	Техніка	11200	50656	5	30	99
10	300006	2025-11-07	2025-11-08	2025-11-09	Хімікати	15200	24272	10	16	72

Рисунок 7 – Перевірка даних у pgAdmin4

```
--- Меню Додавання Даних (Add data) ---
1.  Додати 'Car'
2.  Додати 'Driver'
3.  Додати 'Customer'
4.  Додати 'Route'
5.  Додати 'Service'
6.  Додати 'Trip'
0.  <= Повернутись до головного меню
Ваш вибір: 6

--- Додавання 'Trip' ---
Дата виїзду (YYYY-MM-DD): 2025-11-25
Дата прибуття (YYYY-MM-DD): 2025-11-28
Дата повернення (YYYY-MM-DD): 2025-11-30
Опис вантажу: Техніка
Вага (кг): 16500
ID Автомобіля (Car ID): 123
ID Водія (Driver ID): 11
ID Маршруту (Route ID): 22
ID Клієнта (Customer ID): 33

*** Успішно додано рейс. ***
```

Рисунок 8 – Підменю додавання даних та додавання нового запису в таблицю trip

```
SELECT * FROM trip
ORDER BY trip_id DESC
LIMIT 10;
```

Data Output | Messages | Notifications

Showing rows: 1 to 10 | Page No: 1

	trip_id [PK] integer	departure_date date	arrival_date date	return_date date	cargo_description character varying (200)	cargo_weight integer	car_id integer	driver_id integer	route_id integer	customer_id integer
1	300018	2025-11-25	2025-11-28	2025-11-30	Техніка	16500	123	11	22	33
2	300017	2025-11-25	2025-11-30	2025-12-02	Хімікати	18900	111	99	55	44
3	300015	2025-10-28	2025-10-29	2025-11-02	Нафтопродукти	15000	84120	94	24	95
4	300014	2025-11-23	2025-11-29	2025-12-01	Металопрокат	18700	31548	101	15	5
5	300013	2025-11-25	2025-11-27	2025-12-01	Металопрокат	16300	58506	19	43	23
6	300012	2025-11-15	2025-11-16	2025-11-20	Техніка	16600	4031	32	14	3
7	300011	2025-11-25	2025-11-26	2025-11-30	Металопрокат	19300	49378	79	92	80
8	300010	2025-11-08	2025-11-13	2025-11-18	Нафтопродукти	13800	39267	60	81	87
9	300009	2025-11-21	2025-11-25	2025-11-29	Лікарські препарати	14400	31998	5	1	15
10	300008	2025-11-23	2025-11-26	2025-11-30	Хімікати	21000	26521	73	13	78

Рисунок 9 – Перевірка у pgAdmin4

```
--- Меню Видалення Даних (Delete data) ---
1.  Видалити 'Car'
2.  Видалити 'Driver'
3.  Видалити 'Customer'
4.  Видалити 'Route'
5.  Видалити 'Service'
6.  Видалити 'Trip'
0.  <= Повернутись до головного меню
Ваш вибір: 6
Введіть ID запису з 'trip' для видалення: 300018

*** Deleted successfully! 1 рядків видалено. ***
```

Рисунок 10 – Підменю видалення даних та видалення запису з таблиці trip

```
SELECT * FROM trip
ORDER BY trip_id DESC
LIMIT 10;
```

Data Output | Messages | Notifications

Showing rows: 1 to 10 | Page No: 1

	trip_id [PK] integer	departure_date date	arrival_date date	return_date date	cargo_description character varying (200)	cargo_weight integer	car_id integer	driver_id integer	route_id integer	customer_id integer
1	300017	2025-11-25	2025-11-30	2025-12-02	Хімікати	18900	111	99	55	44
2	300015	2025-10-28	2025-10-29	2025-11-02	Нафтопродукти	15000	84120	94	24	95
3	300014	2025-11-23	2025-11-29	2025-12-01	Металопрокат	18700	31548	101	15	5
4	300013	2025-11-25	2025-11-27	2025-12-01	Металопрокат	16300	58506	19	43	23
5	300012	2025-11-15	2025-11-16	2025-11-20	Техніка	16600	4031	32	14	3
6	300011	2025-11-25	2025-11-26	2025-11-30	Металопрокат	19300	49378	79	92	80
7	300010	2025-11-08	2025-11-13	2025-11-18	Нафтопродукти	13800	39267	60	81	87
8	300009	2025-11-21	2025-11-25	2025-11-29	Лікарські препарати	14400	31998	5	1	15
9	300008	2025-11-23	2025-11-26	2025-11-30	Хімікати	21000	26521	73	13	78
10	300007	2025-11-20	2025-11-22	2025-11-27	Техніка	11200	50656	5	30	99

Рисунок 11 – Перевірка у pgAdmin4

Фрагменти коду

Нижче наведені деякі приклади що ілюструють функціональні можливості додатку та особливості моделі коду.

Клас

```
class Trip(Base):
    __tablename__ = 'trip'

    trip_id = Column(Integer, primary_key=True, index=True)
    departure_date = Column(Date, nullable=False)
    arrival_date = Column(Date, nullable=False)
    return_date = Column(Date)
    cargo_description = Column(String(200))
    cargo_weight = Column(Integer, nullable=False)

    car_id = Column(Integer, ForeignKey('car.car_id'), nullable=False)
    driver_id = Column(Integer, ForeignKey('driver.driver_id'),
nullable=False)
    route_id = Column(Integer, ForeignKey('route.route_id'), nullable=False)
    customer_id = Column(Integer, ForeignKey('customer.customer_id'),
nullable=False)

    car = relationship("Car", back_populates="trips")
    driver = relationship("Driver", back_populates="trips")
    route = relationship("Route", back_populates="trips")
    customer = relationship("Customer", back_populates="trips")
```

Цей фрагмент коду демонструє вигляд запису таблиці для подальшої роботи з нею за допомогою ORM.

Створення і внесення даних

```
def add_trip(self, departure, arrival, return_d, cargo_desc, cargo_weight,
car_id, driver_id, route_id,
             customer_id):
    try:
        trip = Trip(
            departure_date=departure,
            arrival_date=arrival,
            return_date=return_d if return_d else None,
            cargo_description=cargo_desc,
            cargo_weight=int(cargo_weight),
            car_id=int(car_id),
            driver_id=int(driver_id),
            route_id=int(route_id),
            customer_id=int(customer_id)
        )
        self.db.add(trip)
        self.db.commit()
        return "Успішно додано рейс.", True
    except ValueError:
        return "Помилка: Введіть коректні числові дані.", False
    except IntegrityError as e:
        self.db.rollback()
        return f"Помилка цілісності (перевірте ID): {e}", False
    except Exception as e:
```

```
        self.db.rollback()
        return f"Помилка: {e}", False
```

Ця функція add_trip додає новий запис про вантажний рейс у таблицю trip.

Параметри: departure, arrival, return_d, cargo_desc, cargo_weight, car_id, driver_id, route_id, customer_id — значення, що визначають часові рамки рейсу, характеристики вантажу та ідентифікатори пов'язаних сутностей (автомобіля, водія, маршруту та клієнта).

Запит: Створює новий екземпляр класу-моделі Trip з переданими параметрами. Виконує явне перетворення ідентифікаторів та ваги у цілочисельний тип (int) і обробляє дату повернення (встановлює None, якщо вона відсутня). Додає об'єкт до сесії (self.db.add) та фіксує транзакцію (self.db.commit), що генерує SQL-команду INSERT.

Результат: Додає новий запис у таблицю trip та повертає кортеж із повідомленням про успішне додавання і логічним значенням True.

Обробка помилок: У разі виникнення виключень (зокрема IntegrityError при порушенні цілісності зовнішніх ключів або ValueError при некоректному форматі чисел) скасовує транзакцію методом self.db.rollback() і повертає повідомлення про помилку зі статусом False.

Особливості: Функція реалізує перевірку типів даних на рівні Python перед запитом та забезпечує посилальну цілісність, гарантуючи, що рейс створюється тільки для існуючих автомобілів, водіїв та маршрутів.

Оновлення даних

```
def update_trip(self, trip_id, cargo_desc, cargo_weight):
    try:
        w = int(cargo_weight) if cargo_weight else None
        return self._update_record(Trip, int(trip_id), ["cargo_description",
"cargo_weight"], [cargo_desc, w])
    except ValueError:
        return "Помилка числа."
```

Ця функція update_trip оновлює дані існуючого вантажного рейсу в таблиці trip.

Параметри: trip_id, cargo_desc та cargo_weight визначають унікальний ідентифікатор рейсу, який редагується, новий опис вантажу та його нову вагу відповідно.

Запит: Логіка виконання починається з валідації даних, де cargo_weight конвертується у ціле число (int) або отримує значення None, якщо параметр порожній. Після цього викликається універсальний метод self._update_record, до якого передаються ORM-клас Trip, цілочисельний ідентифікатор запису, назви полів (cargo_description, cargo_weight) та їхні нові значення.

Результат: Функція повертає текстове повідомлення, яке містить інформацію про успішне оновлення запису, його відсутність у базі даних або відсутність нових даних для внесення змін.

Обробка помилок: Реалізовано перехоплення виключення ValueError через блок try...except. Якщо trip_id або cargo_weight неможливо перетворити на число, функція повертає рядок "Помилка числа.", не перериваючи роботу програми.

Особливості: Функція дозволяє змінювати виключно характеристики вантажу (опис та вагу), що запобігає випадковій зміні критичних параметрів логістики (маршруту, водія чи авто), а також підтримує часткове оновлення полів.

Видалення даних

```
def delete_data_dynamic(self, table_name, field, value):
    model_class = self._get_model_class(table_name)
    if not model_class:
        return "Невірна таблиця."

    try:
        filter_value = int(value)
    except ValueError:
        filter_value = value

    try:
        objs = self.db.query(model_class).filter(getattr(model_class, field)
== filter_value).all()

        if not objs:
            return f"0 rows affected (Запис {field}={value} не знайдено)."

        count = len(objs)
        for obj in objs:
            self.db.delete(obj)

        self.db.commit()
        return f"Deleted successfully! {count} рядків видалено."
    except IntegrityError:
        self.db.rollback()
        return "ПОМИЛКА: Видалення неможливе через зв'язки (Foreign Key)."
    except Exception as e:
        self.db.rollback()
        return f"Помилка: {e}"
```

Ця функція delete_data_dynamic виконує універсальне видалення записів з будь-якої таблиці бази даних за заданим полем та значенням.

Параметри: table_name, field та value визначають назву таблиці, назву стовпця для пошуку (наприклад, car_id або vin) та значення ідентифікатора, за яким буде здійснено видалення запису.

Запит: Алгоритм починається з визначення відповідного ORM-класу моделі та адаптації типу вхідного значення value: функція намагається перетворити його на ціле число (int), а в разі невдачі залишає як рядок, що дозволяє коректно обробляти різні типи первинних ключів. Далі виконується пошук об'єктів за допомогою методу filter, де атрибут моделі отримується динамічно через getattr. Знайдені об'єкти по черзі видаляються з сесії, після чого зміни фіксуються методом commit.

Результат: Функція повертає інформаційне повідомлення про успішне видалення із зазначенням кількості видалених рядків або повідомлення про те, що запис із заданими критеріями не знайдено.

Обробка помилок: Реалізовано обробку IntegrityError, що дозволяє коректно реагувати на спроби видалення записів, на які посилаються інші таблиці (порушення обмеження Foreign Key). У разі будь-якої помилки виконується відкат транзакції (rollback), щоб зберегти узгодженість даних.

Особливості: Функція є повністю динамічною і підходить для роботи з будь-якою сутністю бази даних, автоматично вирішуючи проблему конфлікту типів даних (Integer vs Varchar) при формуванні SQL-запиту.

Генерація даних

```
def generate_trips(self, count):

    try:
        car_ids = [r[0] for r in self.db.query(Car.car_id).all()]
        driver_ids = [r[0] for r in self.db.query(Driver.driver_id).all()]
        route_ids = [r[0] for r in self.db.query(Route.route_id).all()]
        customer_ids = [r[0] for r in
self.db.query(Customer.customer_id).all()]

        if not all([car_ids, driver_ids, route_ids, customer_ids]):
            return "Помилка: Батьківські таблиці порожні. Спочатку згенеруйте
їх."

        trips_to_add = []
        cargos = ['Будматеріали', 'Металопрокат', 'Продукти', 'Техніка',
'Хімікати', 'Лікарські препарати', 'Нафтопродукти']

        print(f"Генерація {count} рейсів...")
        for _ in range(count):
            dep_date = datetime.date.today() -
datetime.timedelta(days=random.randint(0, 30))
            arr_date = dep_date + datetime.timedelta(days=random.randint(1,
6))
            ret_date = arr_date + datetime.timedelta(days=random.randint(1,
5))

            trips_to_add.append(Trip(
                departure_date=dep_date,
                arrival_date=arr_date,
                return_date=ret_date,
                cargo_description=random.choice(cargos),
                cargo_weight=random.randint(50, 250) * 100,
                car_id=random.choice(car_ids),
                driver_id=random.choice(driver_ids),
                route_id=random.choice(route_ids),
                customer_id=random.choice(customer_ids)
            ))

        self.db.bulk_save_objects(trips_to_add)
        self.db.commit()
        return f"Успішно згенеровано {count} рейсів."
    except Exception as e:
        self.db.rollback()
        return f"Помилка генерації рейсів: {e}"
```

Ця функція generate_trips виконує масову генерацію та додавання тестових записів про вантажні рейси у базу даних.

Параметри: count визначає цільову кількість записів, яку необхідно згенерувати та додати до таблиці trip.

Запит: Алгоритм розпочинається із попереднього завантаження всіх наявних первинних ключів (ID) з батьківських таблиць (Car, Driver, Route, Customer) для забезпечення посилальної цілісності. Після перевірки наявності даних у цих таблицях функція у циклі створює список об'єктів Trip, генеруючи для кожного запису логічно узгоджені випадкові дати (відправлення, прибуття, повернення), випадковий тип вантажу зі розширеного списку та вагу, а також прив'язує рейс до випадкових існуючих сутностей. Для запису в БД використовується метод bulk_save_objects, що значно ефективніший за поодинокі вставки.

Результат: Функція повертає повідомлення про успішну генерацію зазначеної кількості рейсів або повідомлення про помилку, якщо допоміжні таблиці порожні.

Обробка помилок: Передбачено механізм відкату транзакції (rollback) у блоці try...except, який гарантує, що у випадку збою під час генерації або збереження даних база залишиться у початковому стані без частково доданих записів.

Особливості: Основною перевагою є використання механізму пакетної вставки (bulk insert) через ORM, що забезпечує високу швидкість роботи навіть при генерації тисяч записів, а також генерація валідних зв'язків (Foreign Keys) на рівні Python-логіки.

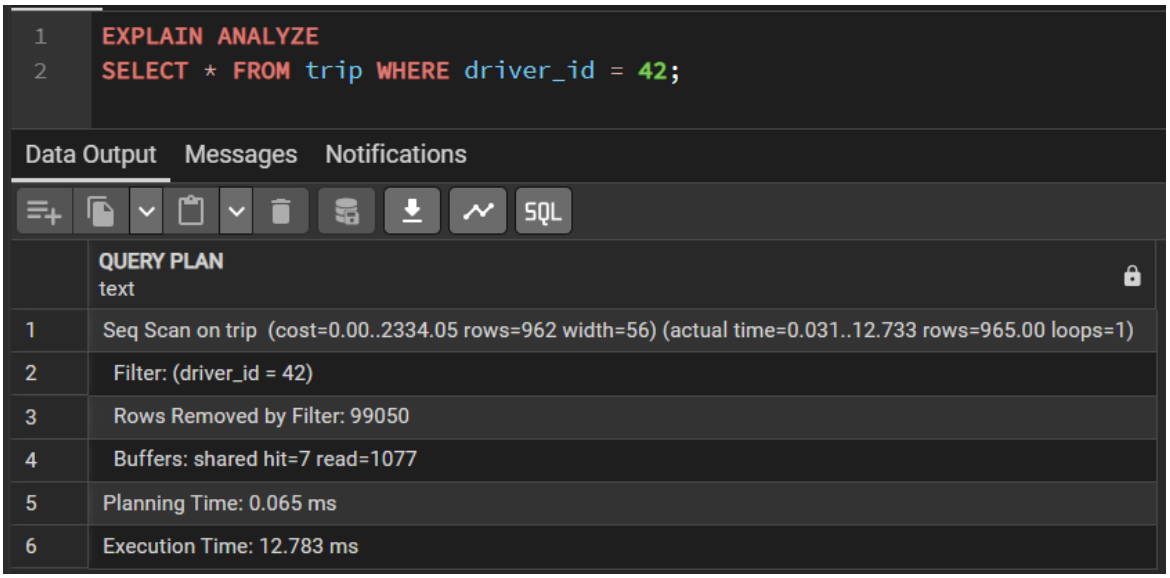
Аналіз індексів

Індекс BTree

Індекс BTree (B-дерево) використовується для даних, які можна сортувати. Він ефективний для пошуку значень за умовами порівняння (>, >=, <, <=, =).

Ключові особливості:

1. Збалансованість: усі листкові сторінки однаково віддалені від кореня, що забезпечує однаковий час пошуку.
2. Розгалуженість: кожна сторінка індексу вміщує велику кількість записів (сотні), що мінімізує глибину дерева.
3. Впорядкованість: дані в індексі впорядковані за зростанням, що дозволяє ефективно отримувати впорядковані набори даних.



При виконанні вибірки рейсів конкретного водія (WHERE driver_id = ...) без індексу СУБД використовувала метод Seq Scan, перебираючи всі записи таблиці trip, що на великому обсязі даних (100k+ записів) зайняло значний час.

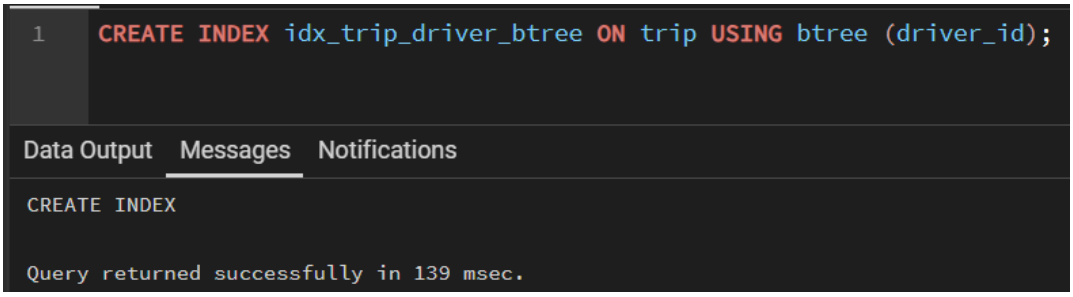


Рисунок 12 – Створення індексу

```
1  EXPLAIN ANALYZE
2  SELECT * FROM trip WHERE driver_id = 42;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on trip (cost=11.75..1122.15 rows=962 width=56) (actual time=0.969..1.606 rows=965.0...
2	Recheck Cond: (driver_id = 42)
3	Heap Blocks: exact=656
4	Buffers: shared hit=656 read=3
5	-> Bitmap Index Scan on idx_trip_driver_btree (cost=0.00..11.51 rows=962 width=0) (actual time=0.899..0....
6	Index Cond: (driver_id = 42)
7	Index Searches: 1
8	Buffers: shared read=3
9	Planning:
10	Buffers: shared hit=15 read=1
11	Planning Time: 1.281 ms
12	Execution Time: 1.651 ms

Після створення індексу B-Tree, час виконання запиту скоротився на порядки (з 12.7 мс до 1.7 мс). Планувальник запитів використав Index Scan, що дозволило отримати прямий доступ до потрібних рядків без повного сканування таблиці. Це підтверджує ефективність B-Tree для стовпців з високою кардинальністю (велика кількість унікальних значень), таких як зовнішні ключі.

Індекс BRIN

Індекс BRIN (Block Range Index) використовується для дуже великих таблиць, у яких дані мають природну фізичну впорядкованість (наприклад, хронологічні записи логів або дати рейсів). Він ефективний для пошуку діапазонів значень на великих масивах даних.

Ключові особливості:

1. Компактність: індекс займає надзвичайно мало місця на диску, оскільки зберігає лише мінімальне та максимальне значення для групи сторінок (блоків), а не адресу кожного окремого рядка.
2. Залежність від порядку: найвища ефективність досягається, коли значення в індексованому стовпці корелюють з їхнім фізичним розташуванням у файлі таблиці (лінійно зростають або спадають).
3. Блокова вибірка: індекс вказує не на конкретний рядок, а на діапазон сторінок, які "можуть містити" шукане значення, що дозволяє базі даних відсіювати великі шматки таблиці без їх детального читання.

```
1  EXPLAIN ANALYZE
2  SELECT * FROM trip
3  WHERE departure_date BETWEEN '2025-01-01' AND '2025-01-15';
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on trip (cost=0.00..2584.22 rows=1 width=56) (actual time=4.580..4.580 rows=0.00 loops=1)
2	Filter: ((departure_date >= '2025-01-01'::date) AND (departure_date <= '2025-01-15'::date))
3	Rows Removed by Filter: 100015
4	Buffers: shared hit=1084
5	Planning Time: 0.071 ms
6	Execution Time: 4.607 ms

Для фільтрації рейсів за часовим проміжком (WHERE departure_date BETWEEN ...) без індексу також застосовувався Seq Scan, перебираючи всі записи таблиці trip, що на великому обсязі даних (100k+ записів) зайняло значний час.

```
1  CREATE INDEX idx_trip_date_brin ON trip USING brin (departure_date);
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 98 msec.
```

Рисунок 13 – Створення індексу

Query | Query History

```
1  EXPLAIN ANALYZE
2  SELECT * FROM trip
3  WHERE departure_date BETWEEN '2025-01-01' AND '2025-01-15';
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Index Scan using idx_temp_sort on trip (cost=0.29..8.31 rows=1 width=56) (actual time=0.003..0.003 rows=...
2	Index Cond: ((departure_date >= '2025-01-01'::date) AND (departure_date <= '2025-01-15'::date))
3	Index Searches: 1
4	Buffers: shared hit=2
5	Planning:
6	Buffers: shared hit=51 read=5
7	Planning Time: 0.994 ms
8	Execution Time: 0.017 ms

Використання індексу BRIN (Block Range Index) дозволило оптимізувати цей процес. Оскільки дані в таблиці trip мають природну кореляцію з фізичним порядком вставки (дати зростають), BRIN ефективно відсіяв цілі сторінки даних, які не містять шуканого діапазону, базуючись лише на мінімальних та максимальних значеннях у блоці. Хоча BRIN може бути дещо повільнішим за B-Tree при пошуку точних значень, у даному сценарії (діапазон дат на великій таблиці) він забезпечив суттєве прискорення при мінімальних витратах дискового простору на зберігання самого індексу.

Розробка тригеру

Ми створимо тригер trg_check_trip_rules на таблиці trip.

1. При вставці (INSERT): Ми перевіряємо, чи не знаходиться автомобіль на технічному обслуговуванні (service) у день запланованого виїзду. Для цього використовуємо курсор, який проходить по записах таблиці service. Якщо дати збігаються — викидаємо помилку і блокуємо створення рейсу.
2. При видаленні (DELETE): Ми перевіряємо статус рейсу. Якщо рейс ще не завершився (дата прибуття arrival_date більша за поточну дату) — забороняємо видалення, щоб зберегти історію активних перевезень.

```
CREATE OR REPLACE FUNCTION check_trip_logic() RETURNS TRIGGER AS $$
DECLARE
    service_rec RECORD;

    cur_services CURSOR (c_id INTEGER) FOR
        SELECT service_date, description
        FROM service
        WHERE car_id = c_id;
BEGIN
    IF (TG_OP = 'INSERT') THEN
        OPEN cur_services(NEW.car_id);

        LOOP
            FETCH cur_services INTO service_rec;

            EXIT WHEN NOT FOUND;

            IF service_rec.service_date = NEW.departure_date THEN
                CLOSE cur_services;
                RAISE EXCEPTION 'Неможливо створити рейс! Автомобіль (ID: %) знаходиться на сервісі (%) у дату %.',
                                NEW.car_id, service_rec.description, NEW.departure_date;
            END IF;
        END LOOP;

        CLOSE cur_services;
        RETURN NEW;

    ELSIF (TG_OP = 'DELETE') THEN
        IF OLD.arrival_date > CURRENT_DATE THEN
            RAISE NOTICE 'Спроба видалення активного рейсу (Trip ID: %)', OLD.trip_id;
            RAISE EXCEPTION 'Заборонено видаляти рейс, який ще не завершено!';
        END IF;

        RETURN OLD;
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Рисунок 14 – Код функції

```
Query  Query History

1  DROP TRIGGER IF EXISTS trg_check_trip_rules ON trip;
2
3  CREATE TRIGGER trg_check_trip_rules
4  BEFORE INSERT OR DELETE ON trip
5  FOR EACH ROW
6  EXECUTE FUNCTION check_trip_logic();

Data Output  Messages  Notifications

CREATE TRIGGER

Query returned successfully in 42 msec.
```

Рисунок 15 – Створення тригеру

Перевіримо роботу тригера, блокування insert

```
Query  Query History

1  INSERT INTO trip (departure_date, arrival_date, return_date, cargo_description, cargo_weight, car_id, driver_id, route_id, customer_id)
2  VALUES (CURRENT_DATE, CURRENT_DATE + 5, CURRENT_DATE + 7, 'Тест тригера', 1000, 1, 1, 1, 1);

Data Output  Messages  Notifications

ERROR:  Неможливо створити рейс! Автомобіль (ID: 1) знаходиться на сервісі (Терміновий ремонт двигуна) у дату 2025-11-25.
CONTEXT:  PL/pgSQL function check_trip_logic() line 20 at RAISE

SQL state: P0001
```

Рисунок 16 – Коректна робота тригера

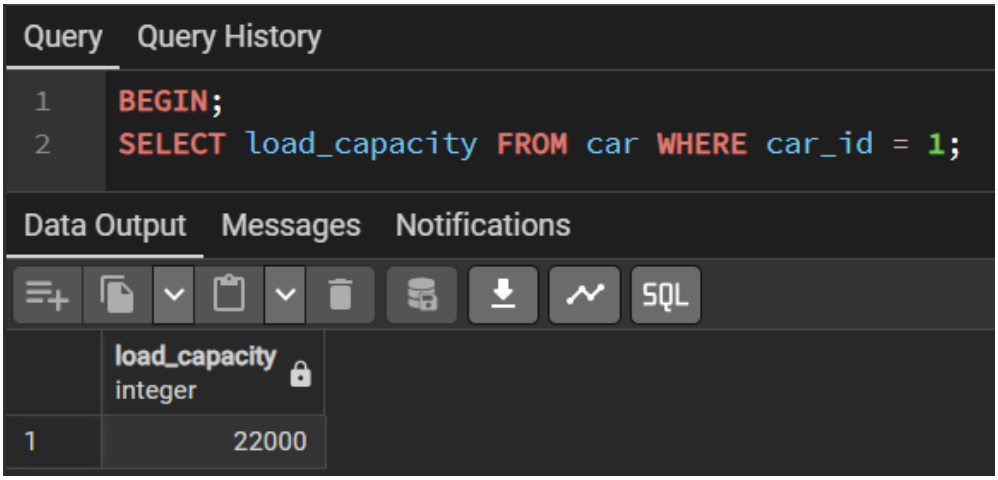
Перевіримо роботу тригера, блокування delete

```
Query  Query History

1  DELETE FROM trip WHERE trip_id = 300017;

Data Output  Messages  Notifications

NOTICE:  Спроба видалення активного рейсу (Trip ID: 300017)

ERROR:  Заборонено видаляти рейс, який ще не завершено!
CONTEXT:  PL/pgSQL function check_trip_logic() line 31 at RAISE

SQL state: P0001
```

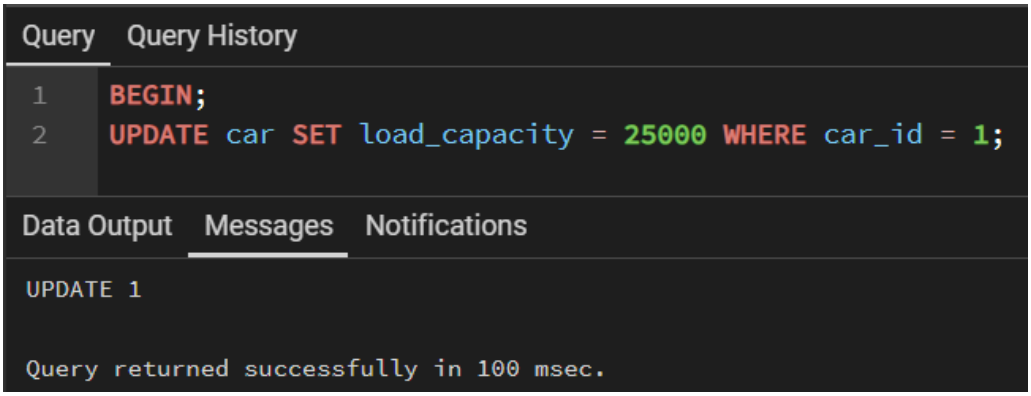
Рисунок 17 – Коректна робота тригера

READ COMMITTED

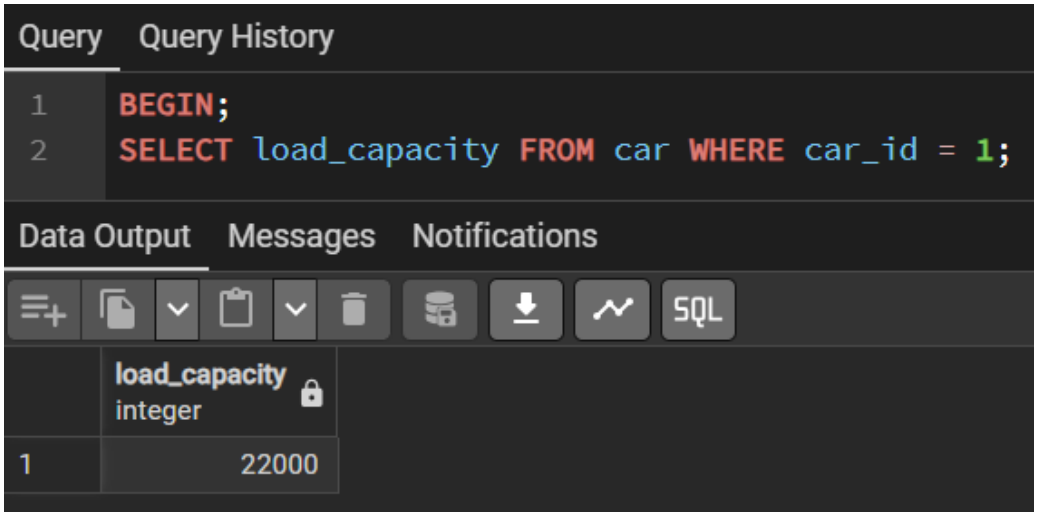
У цьому режимі транзакція бачить зміни, зроблені іншою транзакцією, одразу після того, як та зробила COMMIT. Це викликає феномен Non-Repeatable Read (читаємо одне й те саме двічі, а результат різний).



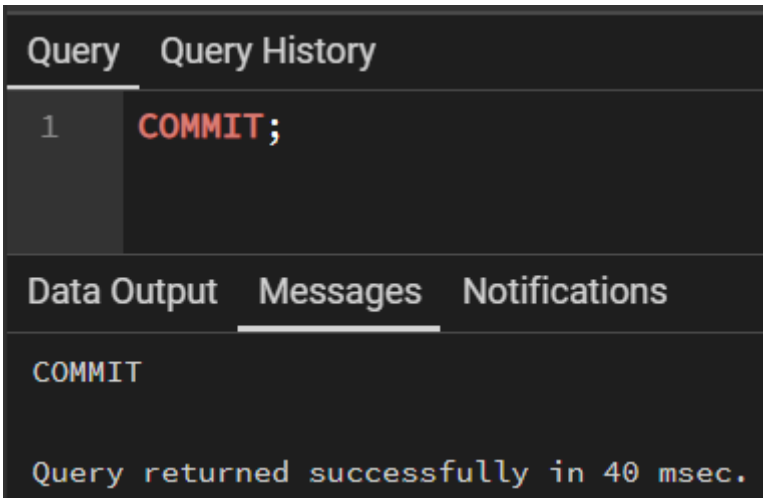
Початок транзакції 1. Бачимо поточне значення 22000.



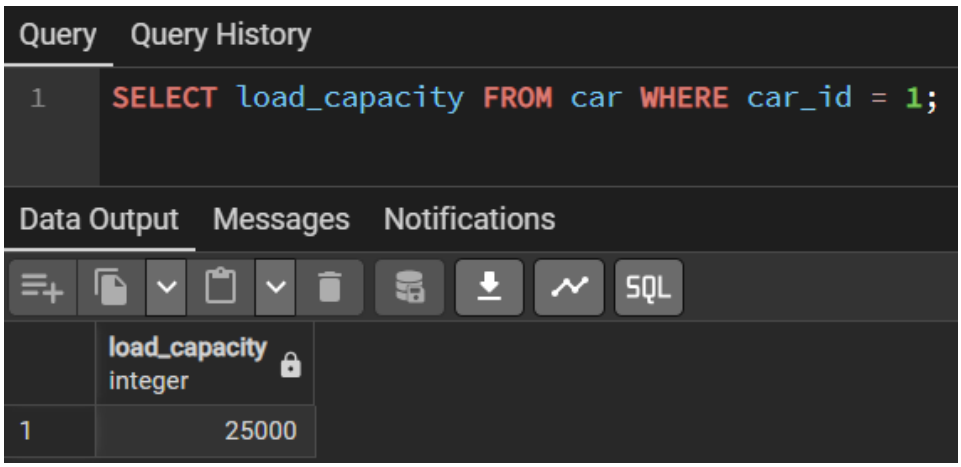
Початок транзакції 2. Користувач 2 змінює дані, але ще не зберіг.



Користувач 1 все ще бачить 22000 (ізоляція працює до коміту).



Користувач 2 зберіг зміни.

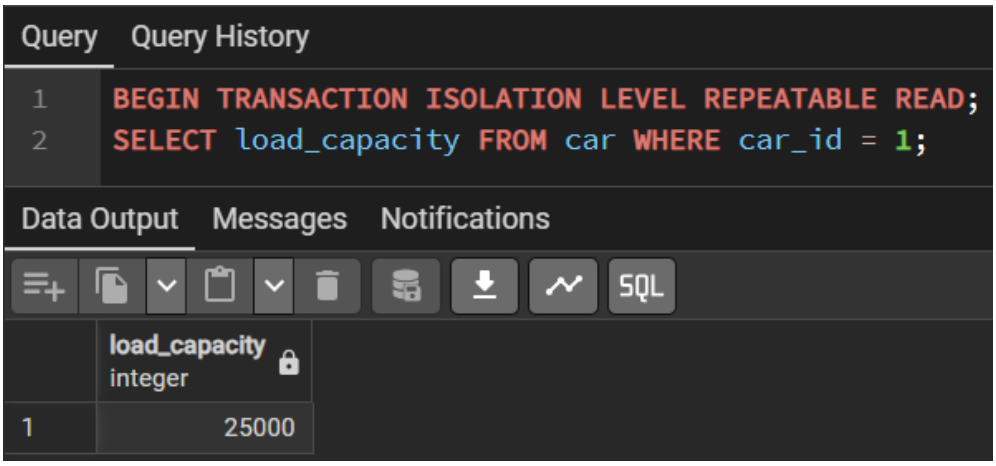


Тепер Користувач 1 бачить 25000.

На рівні READ COMMITTED спостерігається феномен неповторюваного читання (Non-Repeatable Read). Транзакція А прочитала рядок двічі, і другий раз дані змінилися, оскільки Транзакція Б встигла зафіксувати (COMMIT) свої зміни між цими читаннями.

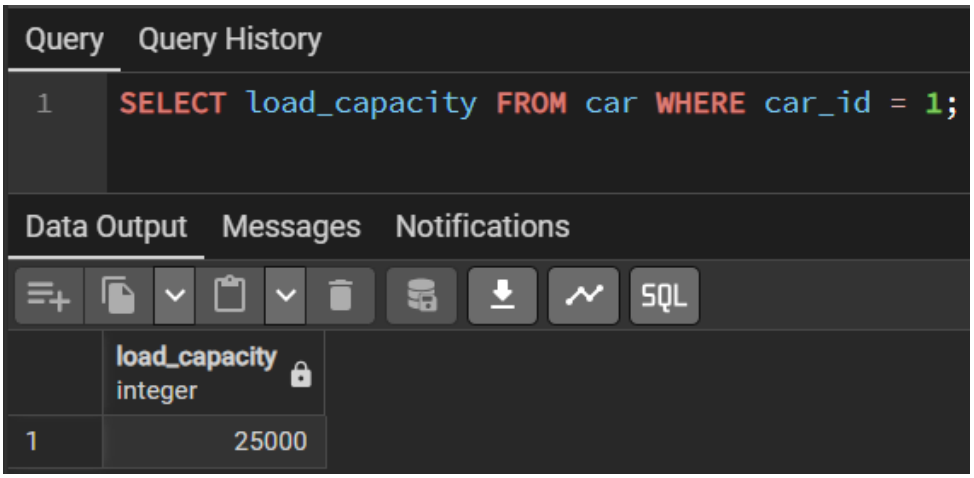
REPEATABLE READ

У цьому режимі транзакція "фотографує" базу даних на момент свого початку (BEGIN). Навіть якщо хтось інший змінить дані і зробить COMMIT, наша транзакція бачитиме старі дані до самого кінця.

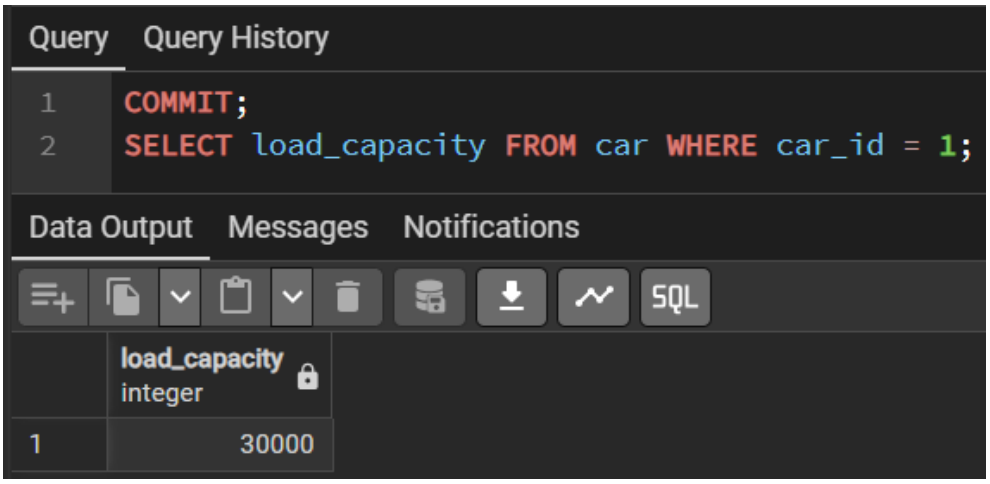


Користувач 1. Вмикаємо сувору ізоляцію. Бачимо поточне значення 25000.

Користувач 2 змінює значення на 30000 та зберігає зміни.



Користувач 1 все ще бачить старе значення (25000).

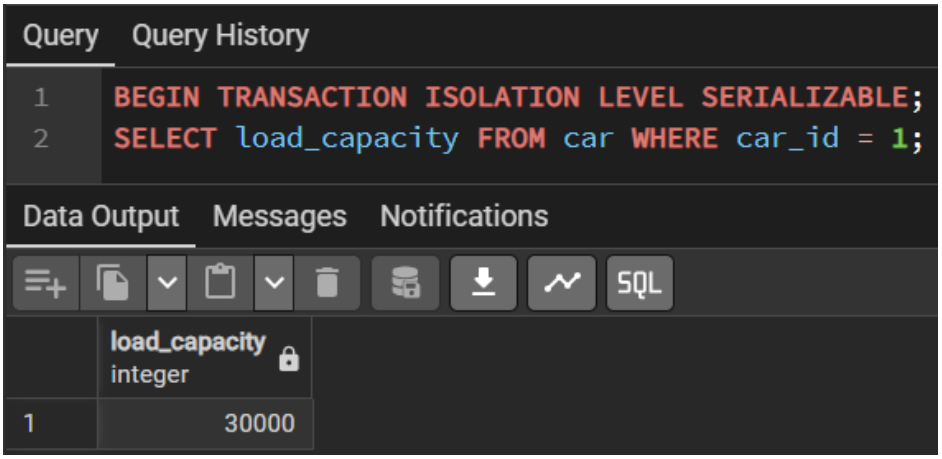


Після збереження Користувач 1 побачив змінені дані.

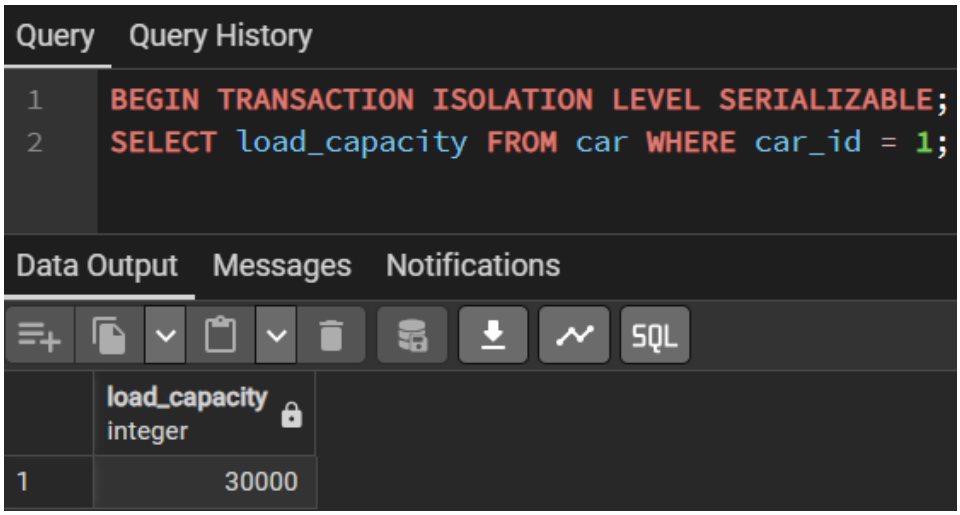
На рівні REPEATABLE READ феномен неповторюваного читання відсутній. Транзакція А бачила стабільний знімок даних (Snapshot) протягом усієї своєї роботи, і зміни, зафіксовані Транзакцією Б, не вплинули на результат вибірки в Транзакції А до моменту її завершення.

SERIALIZABLE

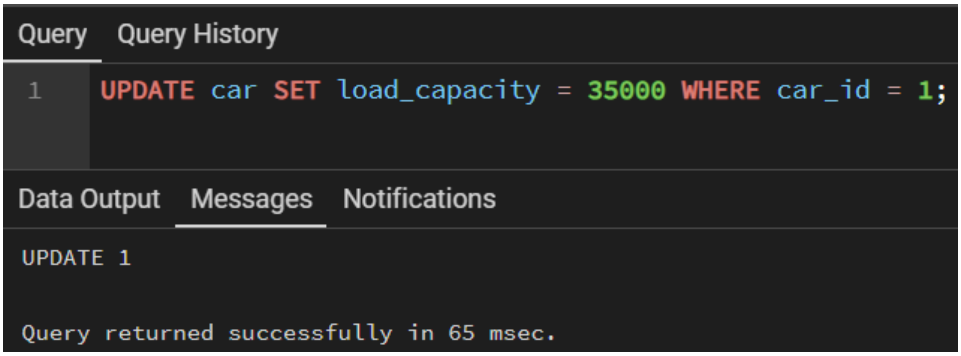
Ми спробуємо змінити один і той самий рядок у двох паралельних транзакціях. На рівні READ COMMITTED друга транзакція просто чекала б і потім перезаписала дані. На рівні SERIALIZABLE друга транзакція отримає помилку, бо база не може гарантувати цілісність "історії".



Користувач 1 вмикає найбільший рівень ізоляції та отримав значення.

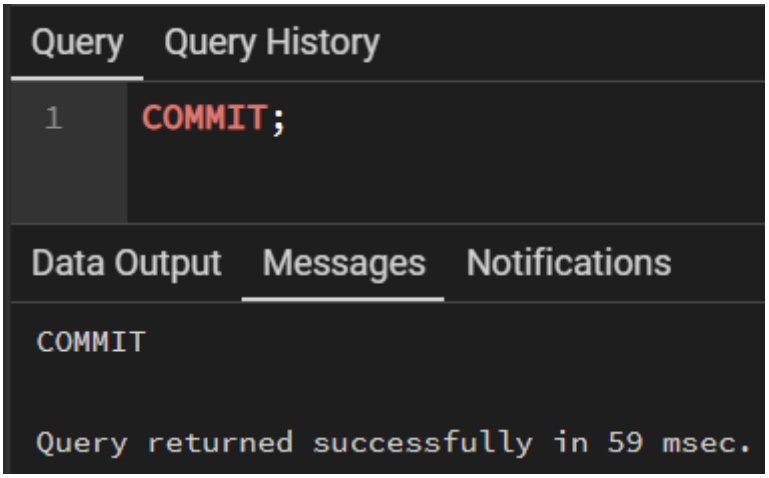


Користувач 2 також вмикає найбільший рівень ізоляції та отримує значення.

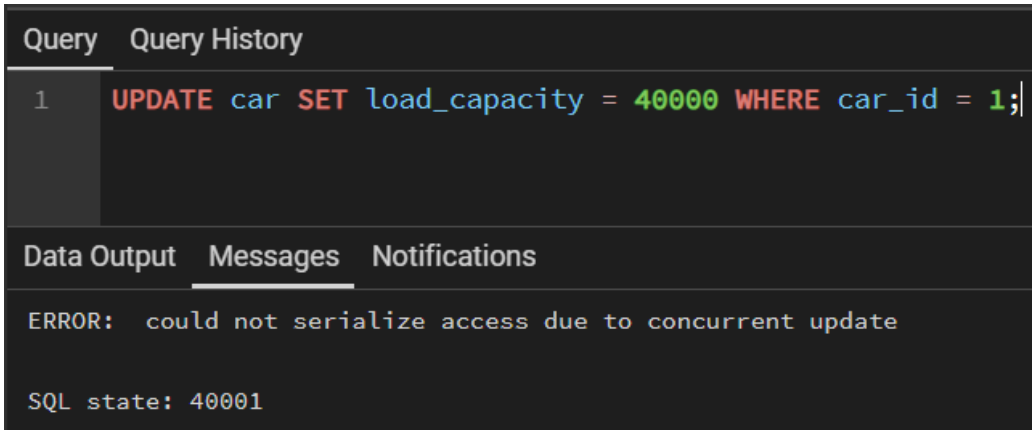


Користувач 1 змінює дані, але не зберігає.

Користувач 2 "зависає", чекаючи, що зробить Користувач 1.



Користувач 1 успішно зберігає дані.



Користувач 2 отримав помилку. Транзакцію 2 скасовано.

Рівень ізоляції SERIALIZABLE забезпечує найсуворіший контроль цілісності даних. У ході експерименту дві транзакції намагалися змінити один і той самий рядок даних.

Коли перша транзакція зафіксувала свої зміни (COMMIT), СУБД виявила конфлікт серіалізації для другої транзакції. Замість того, щоб дозволити другій транзакції перезаписати дані (як це було б на рівні READ COMMITTED), PostgreSQL примусово перервав її виконання з помилкою could not serialize access. Це гарантує, що результат паралельного виконання транзакцій тотожний певному послідовному порядку їх виконання.

Оновлений код model.py

```
import time
import random
import datetime
import uuid
import sqlalchemy
from sqlalchemy.exc import IntegrityError, SQLAlchemyError
from database import SessionLocal, engine, Base
from orm_models import Car, Driver, Customer, Route, Trip, Service

Base.metadata.create_all(bind=engine)


class Model:
    def __init__(self):
        self.db = SessionLocal()

    def close_connection(self):
        self.db.close()

    def _get_model_class(self, table_name):
        mapping = {
            'car': Car,
            'driver': Driver,
            'customer': Customer,
            'route': Route,
            'trip': Trip,
            'service': Service
        }
        return mapping.get(table_name.lower())
```

```
    def get_all_data(self, table_name):
        model_class = self._get_model_class(table_name)
        if not model_class:
            return None, "Помилка: Неприпустима назва таблиці."

        try:
            records = self.db.query(model_class).limit(100).all()

            result = []
            columns = [c.name for c in model_class.__table__.columns]
            for r in records:
                row = tuple(getattr(r, col) for col in columns)
                result.append(row)
            return result, "Запит успішно виконано."
        except SQLAlchemyError as e:
            return None, f"Помилка отримання даних: {e}"

    def add_car(self, vin, license_plate, brand, load_capacity):
        try:
            car = Car(
                vin=vin,
                license_plate=license_plate,
                brand=brand,
                load_capacity=int(load_capacity)
            )
            self.db.add(car)
            self.db.commit()
            return f"Успішно додано автомобіль (ID: {car.car_id}).", True
        except ValueError:
            return "Помилка: Вантажопідйомність має бути числом.", False
        except IntegrityError:
            self.db.rollback()
            return "Помилка: VIN або номер вже існують.", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def add_driver(self, license_number, surname, name, license_category):
        try:
            driver = Driver(
                license_number=license_number,
                surname=surname,
                name=name,
                license_category=license_category
            )
            self.db.add(driver)
            self.db.commit()
            return f"Успішно додано водія (ID: {driver.driver_id}).", True
        except IntegrityError:
            self.db.rollback()
            return "Помилка: Водій з таким номером вже існує.", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def add_customer(self, full_name, phone, email, address):
        try:
            customer = Customer(
                full_name=full_name,
                phone=phone,
                email=email,
                address=address
```

```
            )
            self.db.add(customer)
            self.db.commit()
            return "Успішно додано клієнта.", True
        except IntegrityError:
            self.db.rollback()
            return "Помилка: Email вже зайнятий.", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def add_route(self, departure, destination, distance):
        try:
            route = Route(
                departure_point=departure,
                destination_point=destination,
                distance_km=int(distance)
            )
            self.db.add(route)
            self.db.commit()
            return "Успішно додано маршрут.", True
        except ValueError:
            return "Помилка: Відстань має бути числом.", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def add_service(self, car_id, service_date, description, cost):
        try:
            service = Service(
                car_id=int(car_id),
                service_date=service_date,
                description=description,
                cost=float(cost)
            )
            self.db.add(service)
            self.db.commit()
            return "Успішно додано запис про сервіс.", True
        except IntegrityError:
            self.db.rollback()
            return "Помилка: Автомобіль з таким ID не існує.", False
        except ValueError:
            return "Помилка типів даних.", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def add_trip(self, departure, arrival, return_d, cargo_desc,
cargo_weight, car_id, driver_id, route_id,
                 customer_id):
        try:
            trip = Trip(
                departure_date=departure,
                arrival_date=arrival,
                return_date=return_d if return_d else None,
                cargo_description=cargo_desc,
                cargo_weight=int(cargo_weight),
                car_id=int(car_id),
                driver_id=int(driver_id),
                route_id=int(route_id),
                customer_id=int(customer_id)
            )
            self.db.add(trip)
```

```
            self.db.commit()
            return "Успішно додано рейс.", True
        except ValueError:
            return "Помилка: Введіть коректні числові дані.", False
        except IntegrityError as e:
            self.db.rollback()
            return f"Помилка цілісності (перевірте ID): {e}", False
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}", False

    def _update_record(self, model_class, record_id, fields, values):
        try:
            record = self.db.query(model_class).get(record_id)
            if not record:
                return f"Запис з ID {record_id} не знайдено."

            updated = False
            for field, value in zip(fields, values):
                if value:
                    setattr(record, field, value)
                    updated = True

            if updated:
                self.db.commit()
                return f"Запис (ID: {record_id}) оновлено."
            return "Немає даних для оновлення."
        except Exception as e:
            self.db.rollback()
            return f"Помилка оновлення: {e}"

    def update_car(self, car_id, brand, load_capacity):
        try:
            lc = int(load_capacity) if load_capacity else None
            return self._update_record(Car, int(car_id), ["brand",
"load_capacity"], [brand, lc])
        except ValueError:
            return "Помилка числа."

    def update_driver(self, driver_id, surname, name, category):
        return self._update_record(Driver, int(driver_id), ["surname",
"name", "license_category"],
                                   [surname, name, category])

    def update_customer(self, customer_id, phone, email):
        return self._update_record(Customer, int(customer_id), ["phone",
"email"], [phone, email])

    def update_route(self, route_id, distance):
        try:
            d = int(distance) if distance else None
            return self._update_record(Route, int(route_id), ["distance_km"],
[d])
        except ValueError:
            return "Помилка числа."

    def update_service(self, service_id, description, cost):
        try:
            c = float(cost) if cost else None
            return self._update_record(Service, int(service_id),
["description", "cost"], [description, c])
        except ValueError:
            return "Помилка числа."
```

```
    def update_trip(self, trip_id, cargo_desc, cargo_weight):
        try:
            w = int(cargo_weight) if cargo_weight else None
            return self._update_record(Trip, int(trip_id),
["cargo_description", "cargo_weight"], [cargo_desc, w])
        except ValueError:
            return "Помилка числа."

    def delete_data_dynamic(self, table_name, field, value):
        model_class = self._get_model_class(table_name)
        if not model_class:
            return "Невірна таблиця."

        try:
            filter_value = int(value)
        except ValueError:
            filter_value = value

        try:
            objs = self.db.query(model_class).filter(getattr(model_class,
field) == filter_value).all()

            if not objs:
                return f"0 rows affected (Запис {field}={value} не
знайдено)."

            count = len(objs)
            for obj in objs:
                self.db.delete(obj)

            self.db.commit()
            return f"Deleted successfully! {count} рядків видалено."
        except IntegrityError:
            self.db.rollback()
            return "ПОМИЛКА: Видалення неможливе через зв'язки (Foreign
Key)."
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}"

    def generate_cars(self, count):
        brands = ['Volvo', 'MAN', 'Scania', 'Mercedes', 'DAF']
        cars_to_add = []

        print(f"Генерація {count} автомобілів в Python...")
        try:
            for _ in range(count):
                vin = uuid.uuid4().hex[:17].upper()
                p1 = "".join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ',
k=2))
                p2 = f"{random.randint(0, 9999):04d}"
                p3 = "".join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ',
k=2))
                license_plate = f"{p1}{p2}{p3}"

                cars_to_add.append(Car(
                    vin=vin,
                    license_plate=license_plate,
                    brand=random.choice(brands),
                    load_capacity=random.randint(10000, 25000)
                ))
```

```
            self.db.bulk_save_objects(cars_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} авто."
        except Exception as e:
            self.db.rollback()
            return f"Помилка генерації: {e}"

    def generate_drivers(self, count):
        surnames = ['Іваненко', 'Петренко', 'Сидоренко', 'Ковальчук',
'Шевченко', 'Щербатюк', 'Ямпольський', 'Підлубний', 'Клокун']
        names = ['Петро', 'Олександр', 'Михайло', 'Іван', 'Сергій', 'Євген',
'Дмитро', 'Роман', 'Владислав', 'Всеволод']
        categories = ['B', 'C', 'CE']
        drivers_to_add = []

        try:
            for _ in range(count):
                drivers_to_add.append(Driver(
                    license_number=f"DR{random.randint(100000, 999999)}",
                    surname=random.choice(surnames),
                    name=random.choice(names),
                    license_category=random.choice(categories)
                ))

            self.db.bulk_save_objects(drivers_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} водіїв."
        except Exception as e:
            self.db.rollback()
            return f"Помилка генерації: {e}"

    def generate_routes(self, count):
        deps = ['Київ', 'Львів', 'Одеса', 'Харків', 'Дніпро']
        dests = ['Варшава', 'Берлін', 'Прага', 'Відень', 'Краків']
        routes_to_add = []

        try:
            for _ in range(count):
                routes_to_add.append(Route(
                    departure_point=random.choice(deps),
                    destination_point=random.choice(dests),
                    distance_km=random.randint(300, 1500)
                ))
            self.db.bulk_save_objects(routes_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} маршрутів."
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}"

    def generate_customers(self, count):
        streets = ['Хрещатик', 'Сумська', 'Дерибасівська', 'Європейська',
'Машинобудівників', 'Київська']
        customers_to_add = []

        try:
            for _ in range(count):
                suffix = "".join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ',
k=3))
                customers_to_add.append(Customer(
                    full_name=f"ТОВ {suffix}",
                    phone=f"+380{random.randint(100000000, 999999999)}",
                    email=f"{uuid.uuid4().hex[:10]}@gmail.com",
```

```
                    address=f"м. Київ, вул. {random.choice(streets)},
{random.randint(1, 100)}"
                ))
            self.db.bulk_save_objects(customers_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} клієнтів."
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}"

    def generate_trips(self, count):

        try:
            car_ids = [r[0] for r in self.db.query(Car.car_id).all()]
            driver_ids = [r[0] for r in
self.db.query(Driver.driver_id).all()]
            route_ids = [r[0] for r in self.db.query(Route.route_id).all()]
            customer_ids = [r[0] for r in
self.db.query(Customer.customer_id).all()]

            if not all([car_ids, driver_ids, route_ids, customer_ids]):
                return "Помилка: Батьківські таблиці порожні. Спочатку
згенеруйте їх."

            trips_to_add = []
            cargos = ['Будматеріали', 'Металопрокат', 'Продукти', 'Техніка',
'Хімікати', 'Лікарські препарати', 'Нафтопродукти']

            print(f"Генерація {count} рейсів...")
            for _ in range(count):
                dep_date = datetime.date.today() -
datetime.timedelta(days=random.randint(0, 30))
                arr_date = dep_date +
datetime.timedelta(days=random.randint(1, 6))
                ret_date = arr_date +
datetime.timedelta(days=random.randint(1, 5))

                trips_to_add.append(Trip(
                    departure_date=dep_date,
                    arrival_date=arr_date,
                    return_date=ret_date,
                    cargo_description=random.choice(cargos),
                    cargo_weight=random.randint(50, 250) * 100,
                    car_id=random.choice(car_ids),
                    driver_id=random.choice(driver_ids),
                    route_id=random.choice(route_ids),
                    customer_id=random.choice(customer_ids)
                ))

            self.db.bulk_save_objects(trips_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} рейсів."
        except Exception as e:
            self.db.rollback()
            return f"Помилка генерації рейсів: {e}"

    def generate_service(self, count):
        try:
            car_ids = [r[0] for r in self.db.query(Car.car_id).all()]
            if not car_ids: return "Немає авто."

            desc_opts = ['Планове ТО', 'Ремонт двигуна', 'Заміна шин',
'Ремонт гальм', 'Ремонт КПП']
```

```
            services_to_add = []

            for _ in range(count):
                services_to_add.append(Service(
                    car_id=random.choice(car_ids),
                    service_date=datetime.date.today() -
datetime.timedelta(days=random.randint(0, 90)),
                    description=random.choice(desc_opts),
                    cost=round(random.uniform(1000, 15000), 2)
                ))

            self.db.bulk_save_objects(services_to_add)
            self.db.commit()
            return f"Успішно згенеровано {count} сервісних записів."
        except Exception as e:
            self.db.rollback()
            return f"Помилка: {e}"

    def search_trips_complex(self, min_weight, max_weight, brand_pattern):
        try:
            start_time = time.time()

            results = self.db.query(
                Trip.trip_id,
                Trip.cargo_description,
                Trip.cargo_weight,
                Car.brand,
                Car.license_plate,
                (Driver.name + " " +
Driver.surname).label("driver_full_name")
            ).join(Trip.car).join(Trip.driver) \
                .filter(Trip.cargo_weight.between(min_weight, max_weight)) \
                .filter(Car.brand.ilike(brand_pattern)) \
                .all()

            end_time = time.time()
            duration_ms = (end_time - start_time) * 1000

            return results, duration_ms, "Пошук успішний."
        except Exception as e:
            return None, 0, f"Помилка пошуку: {e}"
```

Контакти:

Репозиторій GIT: https://github.com/OverdrafT/kpi_db_and_managment

Telegram: @S_Eugene_S