Завдання 1.1: Аналіз SQL-запитів

Я порівняв два SQL-запити:

Запит 1 (LEFT JOIN):

```
SELECT *
FROM table1 LEFT JOIN table2 USING (name)
WHERE table2.name = 'John';
```

Запит 2 (INNER JOIN):

```
SELECT *
FROM table1 JOIN table2 USING (name)
WHERE table2.name = 'John';
```

Хоч у першому випадку використовується `LEFT JOIN`, фільтр `WHERE table2.name = 'John'` виключає всі рядки, де `table2.name` є NULL. Таким чином, LEFT JOIN фактично працює як INNER JOIN.

Висновок:
Обидва запити повернуть однаковий результат — лише ті рядки, в яких є відповідність за `name` у обох таблицях **і** значення `table2.name = 'John'`, оскільки фільтр у `WHERE` нівелює різницю між типами JOIN.

Наприклад, якщо `table1` містить:

name	age
John	25
Alice	30

А `table2` містить:

name	city
John	New York
Mark	Chicago

То результат обох запитів буде:

name	age	city
John	25	New York

Завдання 1.2: Перетворення CSV у SQL-базу

Я отримав CSV-файл з даними про замовлення. У процесі аналізу виявив кілька моментів, які заважають прямому імпорту в SQL:

Проблеми у CSV:

- У стовпці `amount` замість крапки використовувалась **кома** (наприклад, `4,5` замість `4.5`) — це порушує формат чисел.
- У значеннях дати був зайвий суфікс `" UTC"`, який заважав конвертації в datetime-формат.

Рішення:

Я написав скрипт на Python з використанням `pandas` і `sqlite3`, який:

1. Зчитує CSV-файл.
2. Замінює кому на крапку в `amount` і конвертує значення у float.
3. Видаляє `" UTC"` з `date_created` і перетворює його в datetime.
4. Створює SQL-базу `orders.db` та імпортує дані у таблицю `orders`.

Код:

```
import pandas as pd
import sqlite3


csv_file = 'Task 1 - Лист1.CSV'
df = pd.read_csv(csv_file)


df['amount'] = df['amount'].astype(str).str.replace(',', '.',
regex=False).astype(float)


df['date_created'] = df['date_created'].str.replace(' UTC', '', regex=False)
df['date_created'] = pd.to_datetime(df['date_created'], format='%Y-%m-%d
%H:%M:%S')


conn = sqlite3.connect('orders.db')
df.to_sql('orders', conn, if_exists='replace', index=False)


print(pd.read_sql('SELECT * FROM orders LIMIT 5', conn))
conn.close()
```

Результат:

CSV-файл успішно перетворено у таблицю SQL без помилок форматування. Всі поля мають правильні типи (float, datetime тощо), і база готова до подальшого аналізу.

Продовження Завдання 1.2: Аналіз покупок по регіонах

Після створення SQL-бази я проаналізував дані та виявив, що **деякі користувачі робили покупки в різних регіонах**.

SQL-запит для перевірки:

```
SELECT
   id_user,
   COUNT(DISTINCT id_region) AS region_count
FROM orders
WHERE status = 'success'
GROUP BY id_user
HAVING region_count > 1
ORDER BY region_count;
```

Результат:

Цей запит показав, що окремі користувачі дійсно мають успішні замовлення у більше ніж одному регіоні.

Висновок для подальшого аналізу:

Щоб коректно обчислити метрику **"усереднений ТС (total spend) користувача по регіонах"** і знайти тих, у кого ТС вищий за середній ТС в кожному регіоні — потрібно використовувати групування за **`id_user` та `id_region`**:
`GROUP BY id_user, id_region`

Це дозволить уникнути змішування даних користувача з різних регіонів і забезпечить точність подальших розрахунків.

Завдання 1.2: Пошук користувачів з вищим за середній ТС у регіоні

Мета:

Знайти користувачів, **чий загальний обсяг покупок (ТС)** перевищує **середній ТС усіх користувачів** у відповідному регіоні.
Що зроблено: Я написав SQL-запит, який складається з двох CTE (тимчасових таблиць):

1. **`user_total_spend`** – рахує **загальні покупки (`total_spend`)** кожного користувача по регіонах (`id_user`, `id_region`) з урахуванням лише успішних транзакцій.
2. **`regional_avg`** – рахує **середній ТС по кожному регіону** і порівнює його з індивідуальним ТС користувача.
3.

SQL-запит:

```
WITH user_total_spend AS (
   SELECT
       id_user,
       id_region,
       SUM(amount) AS total_spend
   FROM orders
   WHERE status = 'success'
   GROUP BY id_user, id_region
),
regional_avg AS (
   SELECT
       id_user,
        id_region,
        total_spend,
       AVG(total_spend) OVER (PARTITION BY id_region) AS
avg_spend_region
   FROM user_total_spend
)
SELECT
   id_user,
   id_region,
   total_spend,
   avg_spend_region
FROM regional_avg
WHERE total_spend > avg_spend_region;
```

Результат:

Цей запит витягує користувачів, **які витрачають більше, ніж у середньому по їхньому регіону**, що дозволяє визначити найактивніших клієнтів у кожному регіоні.
Примітка:
Невдалі покупки (з `status = 'fail'`) не враховувались — підрахунок ТС проводився **виключно на основі успішних покупок (`status = 'success'`)**.

Додатковий крок: отримання унікальних користувачів

У завданні не було чітко вказано, які саме дані необхідно повернути — повну інформацію про ТС чи просто список таких користувачів.
Тому, **на всяк випадок**, я додатково вивів **унікальних користувачів**, чий ТС вищий за середній по їхньому регіону.

SQL-запит:

```
WITH user_total_spend AS (
   SELECT
       id_user,
       id_region,
       SUM(amount) AS total_spend
   FROM orders
   WHERE status = 'success'
   GROUP BY id_user, id_region
),
regional_avg AS (
   SELECT
       id_user,
       id_region,
       total_spend,
       AVG(total_spend) OVER (PARTITION BY id_region) AS avg_spend_region
   FROM user_total_spend
)
SELECT DISTINCT id_user
FROM regional_avg
WHERE total_spend > avg_spend_region;
```

Результат:

Цей запит повертає **унікальний список користувачів**, які **витрачають більше, ніж у середньому по регіону**, без деталізації по сумі чи регіону.
Це дозволяє гнучко використовувати результат — або для подальшого аналізу, або для генерації списку потенційно цінних клієнтів.