## Введение

**Цель работы:** В лабораторной работе необходимо изучить работу в VS Code с Data Wrangler, включая очистку данных, визуализацию на карте с помощью folium, а также выполнение задания со звездочкой с использованием Docker.

## Ход работы

Скачала данные о местах падения метеоритов в формате csv и установила пакет **Data Wrangler** (рисунок 1 и 2).

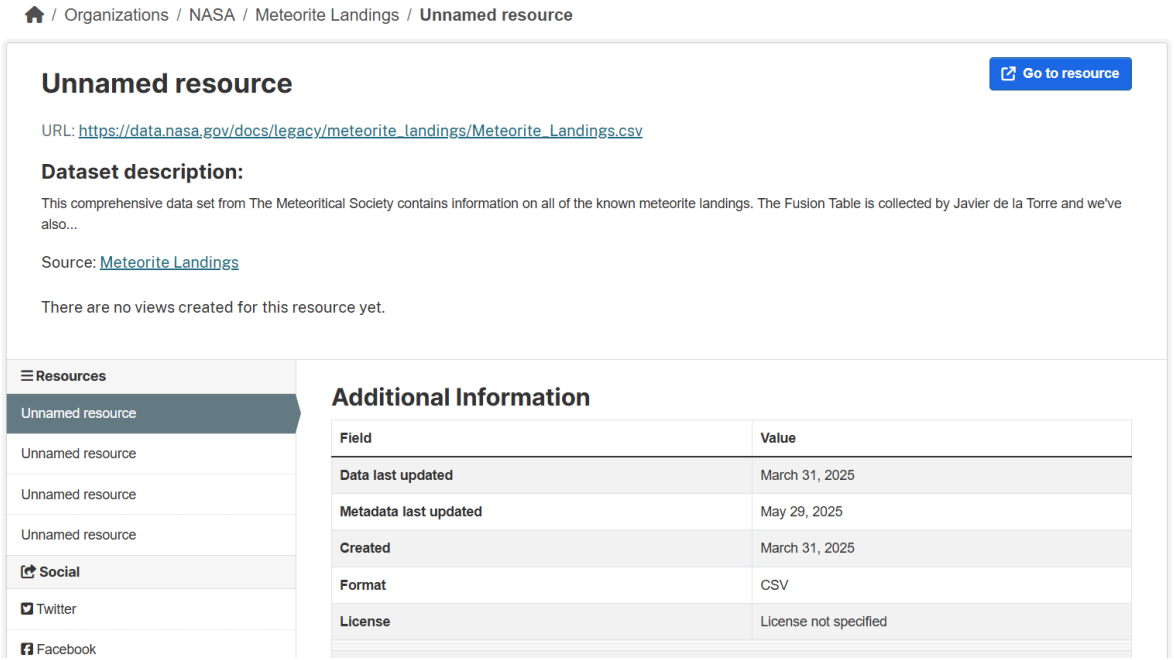



Рисунок 1 – Сайт для выгрузки данных

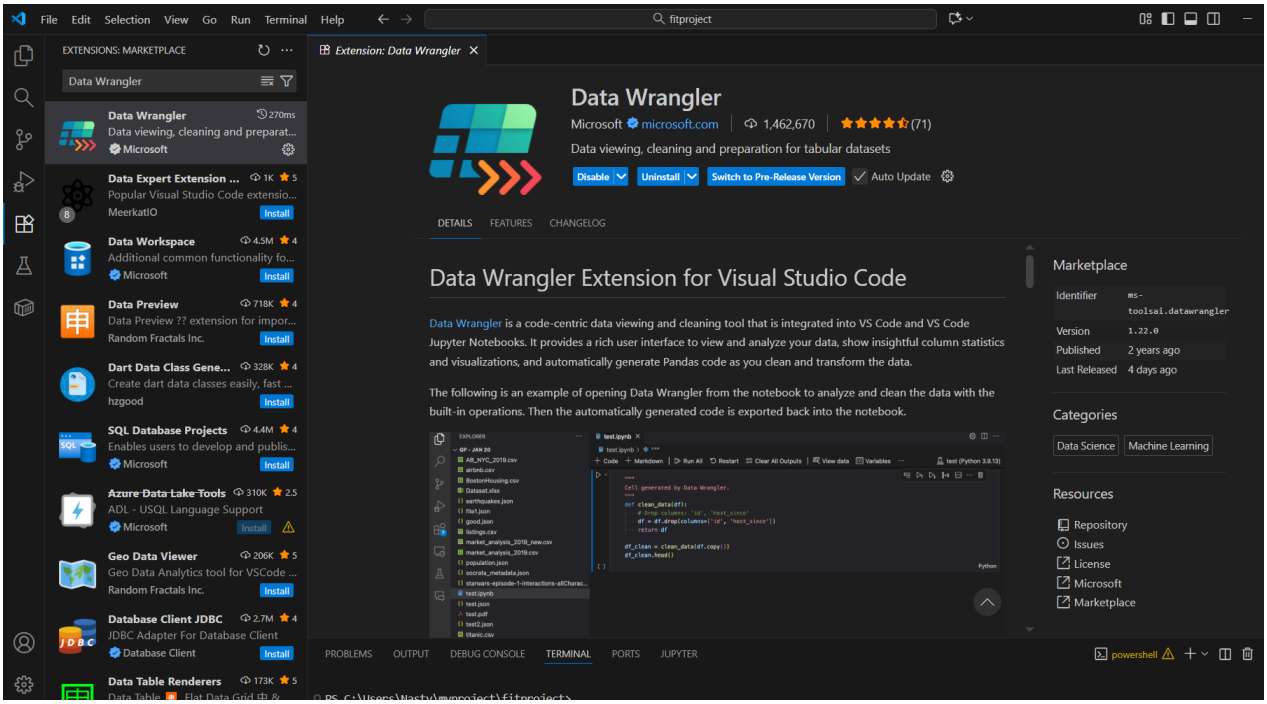



Рисунок 2 – Установка расширения

Далее я открыла набор данных Meteorite_Landings.csv через расширение Data Wrangler (рисунок 3).

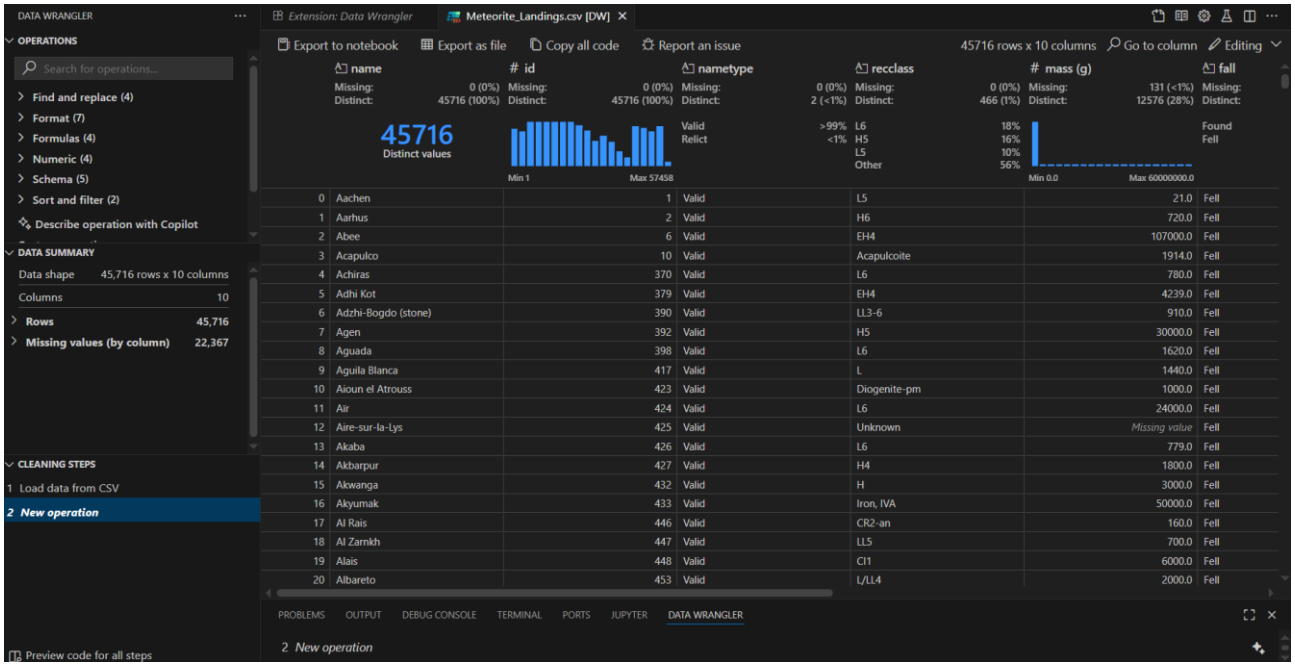



Рисунок 3 – Набор данных

Далее нужно почистить данные и убрать пропуски, это делается через фильтры. Отфильтровала колонки широта и долгота по параметру «без пропусков» (рисунок 4).

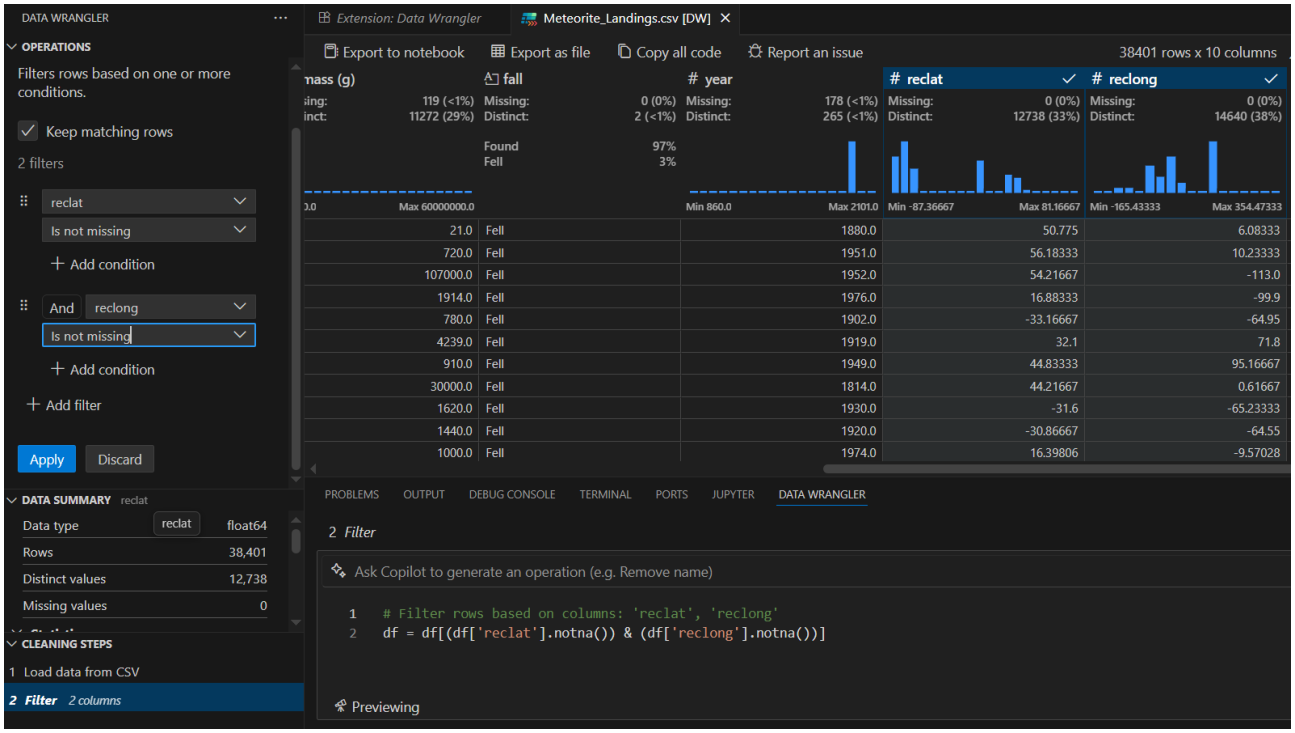



Рисунок 4 – Фильтрация для очистки данных

Далее создала проект, написала первую часть кода для проверки данных. Также для наглядности отфильтрую данные, оставив только метеориты, упавшие после 1970 года, чтобы карта не была перегружена (рисунок 5-6).

```python
# 1. Загружаем данные
import pandas as pd
df = pd.read_csv('Meteorite_Landings.csv')

# 2. Преобразуем столбец 'year' в числовой формат.
#    errors='coerce' превратит все некорректные значения (не числа) в NaT (Not a Time),
#    аналог NaN для дат.
df_cleaned['year'] = pd.to_numeric(df_cleaned['year'], errors='coerce')

# 3. Удаляем строки, где год не был распознан
df_cleaned.dropna(subset=['year'], inplace=True)

# 4. Преобразуем год в целый тип, чтобы убрать ".0"
df_cleaned['year'] = df_cleaned['year'].astype(int)

# 5. Фильтруем данные: оставляем только метеориты, упавшие после 1970 года
df_filtered = df_cleaned[df_cleaned['year'] > 1970].copy()
df_filtered.dropna(subset=['mass (g)'], inplace=True)

# Выведем информацию, чтобы убедиться, что все сработало
print("Данные после полной очистки и фильтрации:")
print(df_filtered.info())
```

[2] ✓ 0.3s Python

Рисунок 5 – Код

```
Данные после полной очистки и фильтрации:
<class 'pandas.core.frame.DataFrame'>
Index: 35856 entries, 3 to 45715
Data columns (total 10 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   name         35856 non-null  object
 1   id           35856 non-null  int64
 2   nametype     35856 non-null  object
 3   recclass     35856 non-null  object
 4   mass (g)     35856 non-null  float64
 5   fall         35856 non-null  object
 6   year         35856 non-null  int64
 7   reclat       35856 non-null  float64
 8   reclong      35856 non-null  float64
 9   GeoLocation  35856 non-null  object
dtypes: float64(3), int64(2), object(5)
memory usage: 3.0+ MB
None
```

Рисунок 6 – Результат

Теперь нужно визуализировать метеориты на карте с помощью Folium. Маркеры настроены так, чтобы их цвет зависел от массы, а также при нажатии на интерактивной карте выводилась полезная информация (рисунок 7-9).

```python
import folium

# Создаем базовую карту мира. location - начальная точка, zoom_start - масштаб
m = folium.Map(location=[20, 0], zoom_start=2)

# Чтобы карта не "тормозила", возьмем для примера первые 1000 метеоритов из отфильтрованного списка
data_for_map = df_filtered.head(1000)

# Проходим по каждой строке в наших данных и добавляем маркер на карту
for index, row in data_for_map.iterrows():

    # 1. Определяем цвет пина в зависимости от массы
    mass = row['mass (g)']
    if mass > 100000:  # Больше 100 кг
        pin_color = 'red'
    elif mass > 10000: # Больше 10 кг
        pin_color = 'orange'
    else:
        pin_color = 'blue'

    # 2. Создаем текст для всплывающего окна
    popup_text = f"""
    <b>Название:</b> {row['name']}<br>
    <b>Год падения:</b> {row['year']}<br>
    <b>Масса:</b> {mass} г.<br>
    <b>Класс:</b> {row['recclass']}
    """

    # 3. Добавляем сам маркер на карту
    folium.Marker(
        location=[row['reclat'], row['reclong']],
        popup=folium.Popup(popup_text, max_width=200),
        icon=folium.Icon(color=pin_color, icon='fa-star', prefix='fa') # иконка звездочки
    ).add_to(m)

# 4. Сохраняем карту в HTML-файл. Его можно будет открыть в браузере.
map_filename = 'meteorite_landings_map.html'
m.save(map_filename)

print(f"Карта сохранена в файл: {map_filename}")

# Чтобы увидеть карту прямо в ноутбуке
m
```

[3] ✓ 7.4s Python

```
Карта сохранена в файл: meteorite_landings_map.html
```

Рисунок 7 – Код для визуализации

Рисунок 8 – Карта

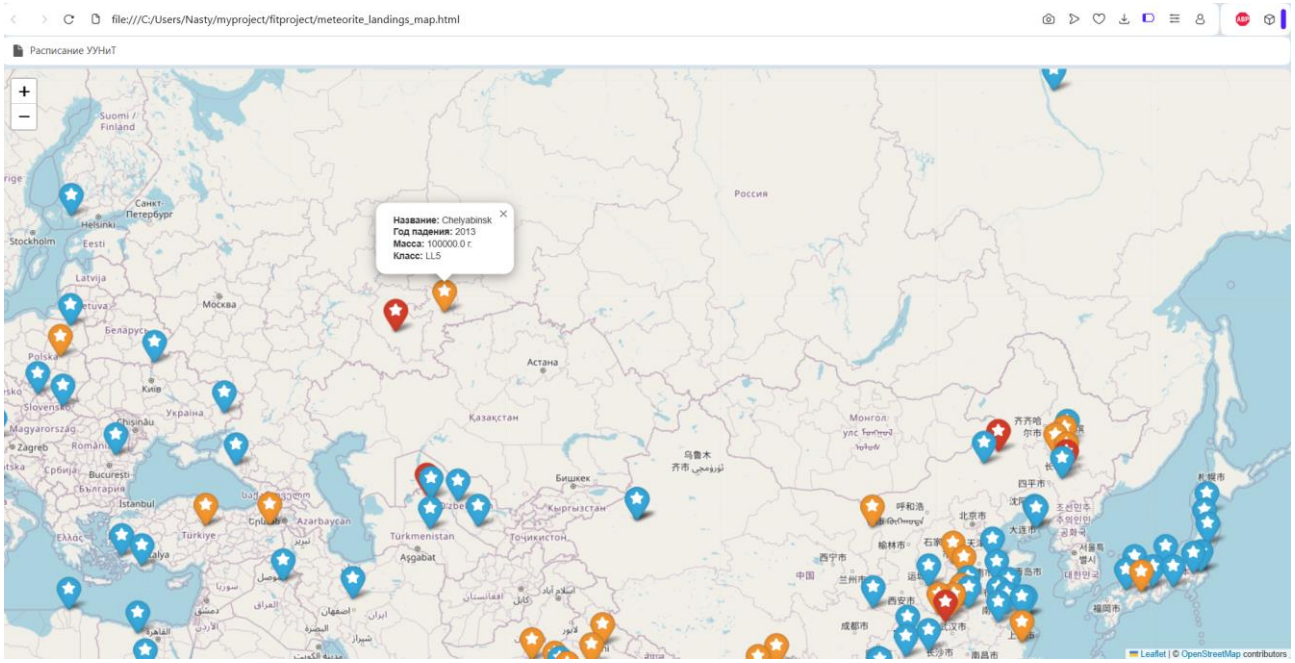



Рисунок 9 – Информация о метеорите

Глядя на карту, можно заметить скопления в определенных местах: Антарктида, пустыня Сахара в Африке, Северная Америка, Европа.

Пространственная дислокация найденных метеоритов крайне неоднородна. Это связано не с тем, что метеориты падают в эти места чаще, а с условиями их обнаружения:

В Антарктиде и пустынях - темные метеориты очень легко заметить на фоне льда или песка. Специальные экспедиции целенаправленно ищут их именно там.

В густонаселенных районах (Европа, США) - больше людей = больше шансов, что кто-то увидит падение метеорита или случайно найдет необычный камень.

В океанах, джунглях или сибирской тайге метеориты падают не реже, но найти их там практически невозможно.

Далее нужно изучить работу в Docker. В нем не будет расширения Data Wrangler. Поэтому всю очистку данных нужно будет делать кодом.

Для начала нужно запустить Docker, открыть PowerShell. Скачать готовую среду Jupyter (рисунок 10).

```
PS C:\Users\Nasty> docker pull jupyter/datascience-notebook
>>
Using default tag: latest
latest: Pulling from jupyter/datascience-notebook
aece8493d397: Pull complete
ef8373d600b0: Pull complete
4f4fb700ef54: Pull complete
9a165b6e9dc7: Pull complete
fd92c719666c: Pull complete
5689442fd4e1: Pull complete
77e45ee945dc: Pull complete
2c4c69587ee4: Pull complete
dc42adc7eb73: Pull complete
75d33599f5f2: Pull complete
088f11eb1e74: Pull complete
abaa8376a650: Pull complete
734ea0c3d94e: Pull complete
9a6a202f62a6: Pull complete
e488194bf535: Pull complete
f5302bfd25be: Pull complete
02c2173301db: Pull complete
de2cdd875fa8: Pull complete
31973ea82470: Pull complete
d25166dcdc7b: Pull complete
964fc3e4ff9f: Pull complete
d19266e0cb17: Pull complete
a30f89a0af6c: Pull complete
96ee7e4439c7: Pull complete
a21a167f7127: Pull complete
1f9ad23c07ac: Pull complete
5201d3116fb6: Pull complete
aa099bb9e49a: Pull complete
822c4cbcf6a6: Pull complete
Digest: sha256:476c6e673e7d5d8b5059f8680b1c6a988942a79263da651bf302dc696ab311f2
Status: Downloaded newer image for jupyter/datascience-notebook:latest
docker.io/jupyter/datascience-notebook:latest
```

Рисунок 10 – Скачивание среды

Далее перейти в папку с проектом cd C:\Users\Nasty\myproject\fitproject и выполнить команду для запуска контейнера (рисунок 11).

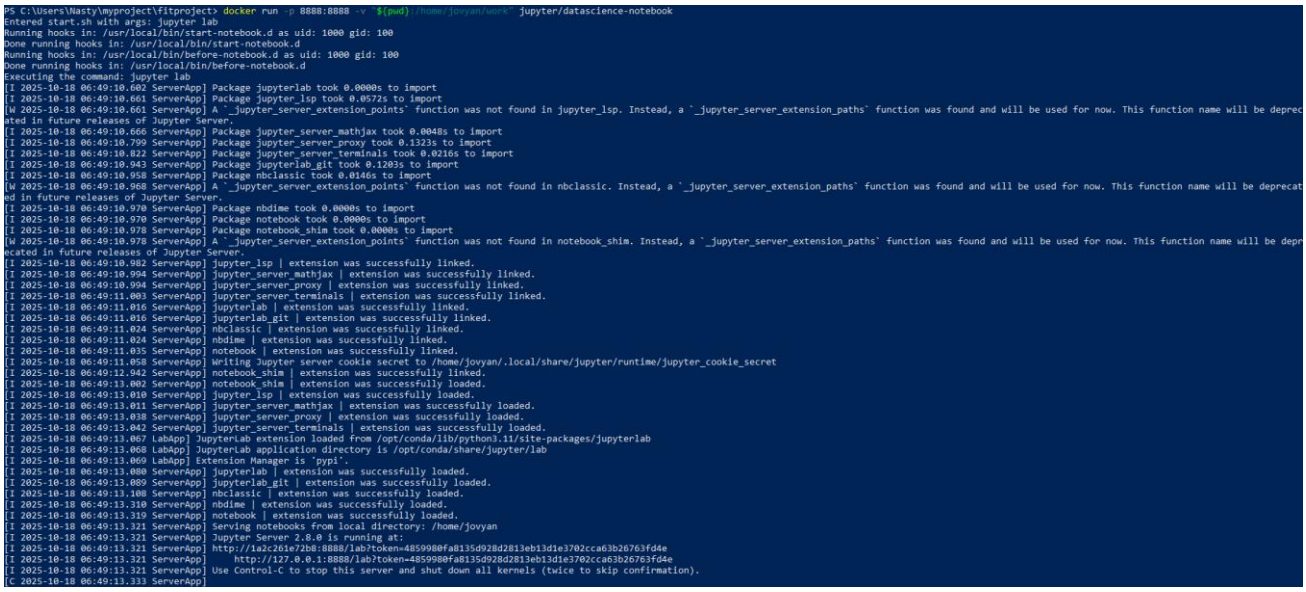



Рисунок 11 – Результат выполнения команды

Скопировала сгенерированную ссылку и открыла в браузере Opera. Открылось пространство для работы. Создала новый файл, установила библиотеку, так как пространство новое (рисунок 12).

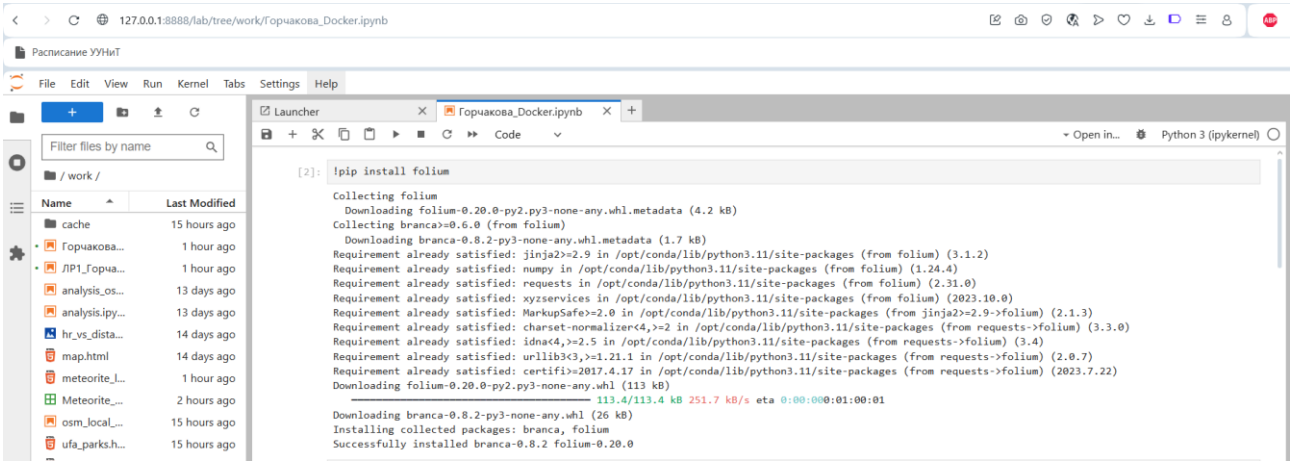



Рисунок 12 – Установка библиотеки

Далее вставила код, который разрабатывала в VS Code и дополнила строчкой очистки широты и долготы от отсутствия данных (df_cleaned = df.dropna(subset=['reclat', 'reclong']).copy()). Код работает исправно. Результаты приведены на рисунке 13.

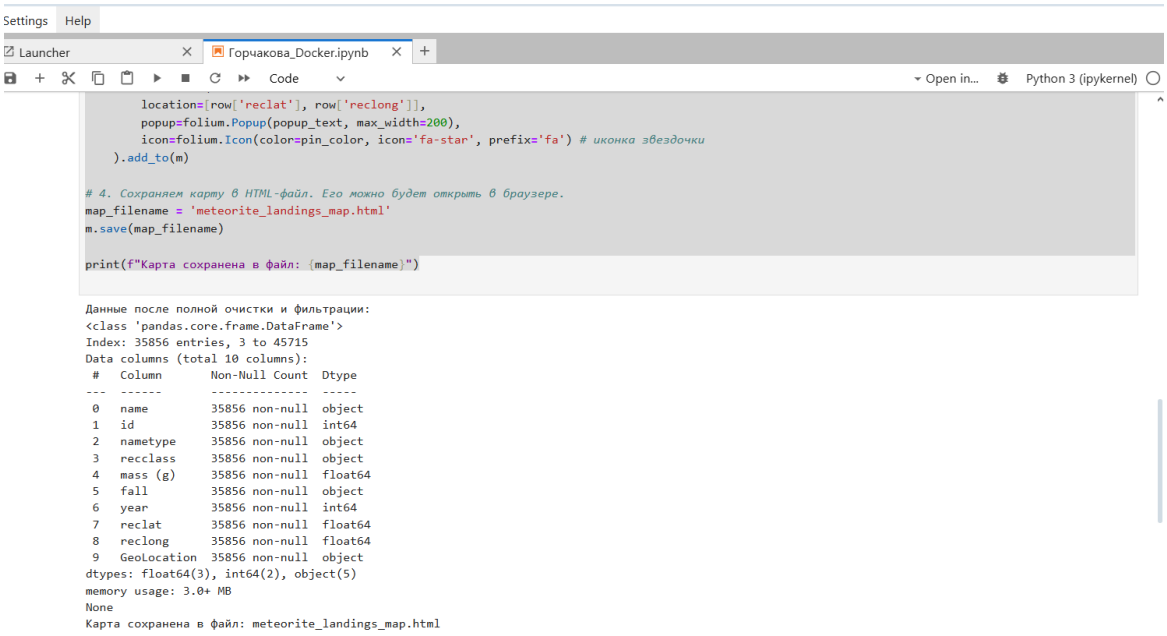



Рисунок 13 – Результат работы кода в Docker

## Заключение

В ходе выполнения лабораторной работы были получены два файла формата .ipynb. Один для работы с Data Wrangler, другой без участия данного расширения. Также в результате работы была получена интерактивная карта падения метеоритов в формате HTML, где при нажатии появляется информация о метеорите. В данной лабораторной работе мы научились первично обрабатывать данные в VS Jupyter Notebook и запускать образ jupyter/datascience-notebook с использованием Docker Desktop.

**Введение**

**Цель работы:** изучить возможности пространственной базы данных PostGIS, научиться выполнять операции импорта геоданных (в формате GeoJSON и CSV) в базу данных PostgreSQL, а также освоить выполнение пространственных запросов для анализа географических данных.

**Ход работы**

Открываю терминал в VS Code для запуска контейнера с PostGIS, задаю имя, логин/пароль/имя БД, открываю порт для pgAdmin 4, подключаю каталог и запускаю в фоне. Далее проверяю запуск (рисунок 1). Так как возникает проблема с созданием сервера в pgAdmin 4 на порте 5432, использую проброс с 5433 и разрешаю этот порт в PowerShell (рисунок 2). После этого создаю сервер в pgAdmin 4 (рисунок 3).

```
PS C:\Users\Nasty\myproject\fitproject> & C:/Users/Nasty/myproject/fitproject/.venv/Scripts/Activate.ps1
(.venv) PS C:\Users\Nasty\myproject\fitproject> docker run --name postgis_lab `
>>   -e POSTGRES_PASSWORD=postgres `
>>   -e POSTGRES_USER=postgres `
>>   -e POSTGRES_DB=meteorites_db `
>>   -p 5433:5432 `
>>   -v C:\Users\Nasty\myproject\fitproject:/data `
>>   -d postgis/postgis
>>
a;cbdada0d-6512-449d-b487-a75273def1ceed500637273f3f0a42e680e2e194a419496c04ebb7fca4b41cf51d49e9617bcf
(.venv) PS C:\Users\Nasty\myproject\fitproject> docker ps
>>
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS                                       NAMES
ed500637273f   postgis/postgis   "docker-entrypoint.s…"   12 seconds ago   Up 11 seconds   0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp   postgis_lab
(.venv) PS C:\Users\Nasty\myproject\fitproject>
```

Рисунок 1 – Запуск контейнера

```
Администратор: Windows PowerShell
Windows PowerShell
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Попробуйте новую кроссплатформенную оболочку PowerShell (https://aka.ms/pscore6)

PS C:\WINDOWS\system32> New-NetFirewallRule -DisplayName "Allow Docker Postgres" -Direction Inbound -LocalPort 5433 -Protocol TCP -Action Allow
>>


Name                          : {c68e5c99-7c82-4fb4-acb3-d4c13a3bf707}
DisplayName                   : Allow Docker Postgres
Description                   :
DisplayGroup                  :
Group                         :
Enabled                       : True
Profile                       : Any
Platform                      : {}
Direction                     : Inbound
Action                        : Allow
EdgeTraversalPolicy           : Block
LooseSourceMapping            : False
LocalOnlyMapping              : False
Owner                         :
PrimaryStatus                 : OK
Status                        : Правило было успешно проанализировано из хранилища. (65536)
EnforcementStatus             : NotApplicable
PolicyStoreSource             : PersistentStore
PolicyStoreSourceType         : Local
RemoteDynamicKeywordAddresses :
PolicyAppId                   :



PS C:\WINDOWS\system32> netstat -ano | findstr 5433
>>
  TCP    0.0.0.0:5433           0.0.0.0:0              LISTENING       11244
  TCP    127.0.0.1:54336        127.0.0.1:54042        TIME_WAIT       0
  TCP    192.168.0.100:54331    40.79.173.40:443       TIME_WAIT       0
  TCP    192.168.0.100:54335    13.107.213.53:443      TIME_WAIT       0
  TCP    192.168.0.100:54337    94.131.53.144:9066     TIME_WAIT       0
  TCP    192.168.0.100:54339    40.79.173.40:443       TIME_WAIT       0
  TCP    [::]:5433              [::]:0                 LISTENING       11244
  TCP    [::1]:5433             [::]:0                 LISTENING       8420
PS C:\WINDOWS\system32>
```

Рисунок 2 – Использование PowerShell

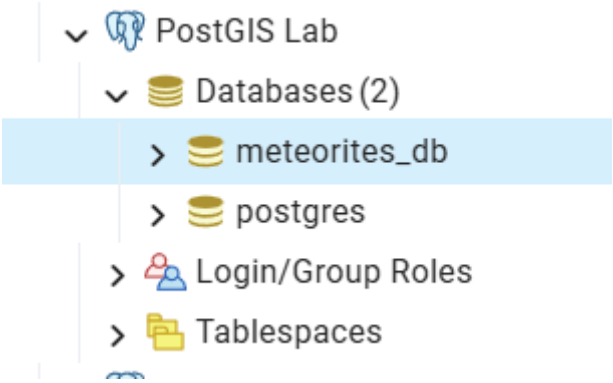



Рисунок 3 – Созданный сервер в pgAdmin 4

Далее включаю расширение PostGIS (рисунок 4).

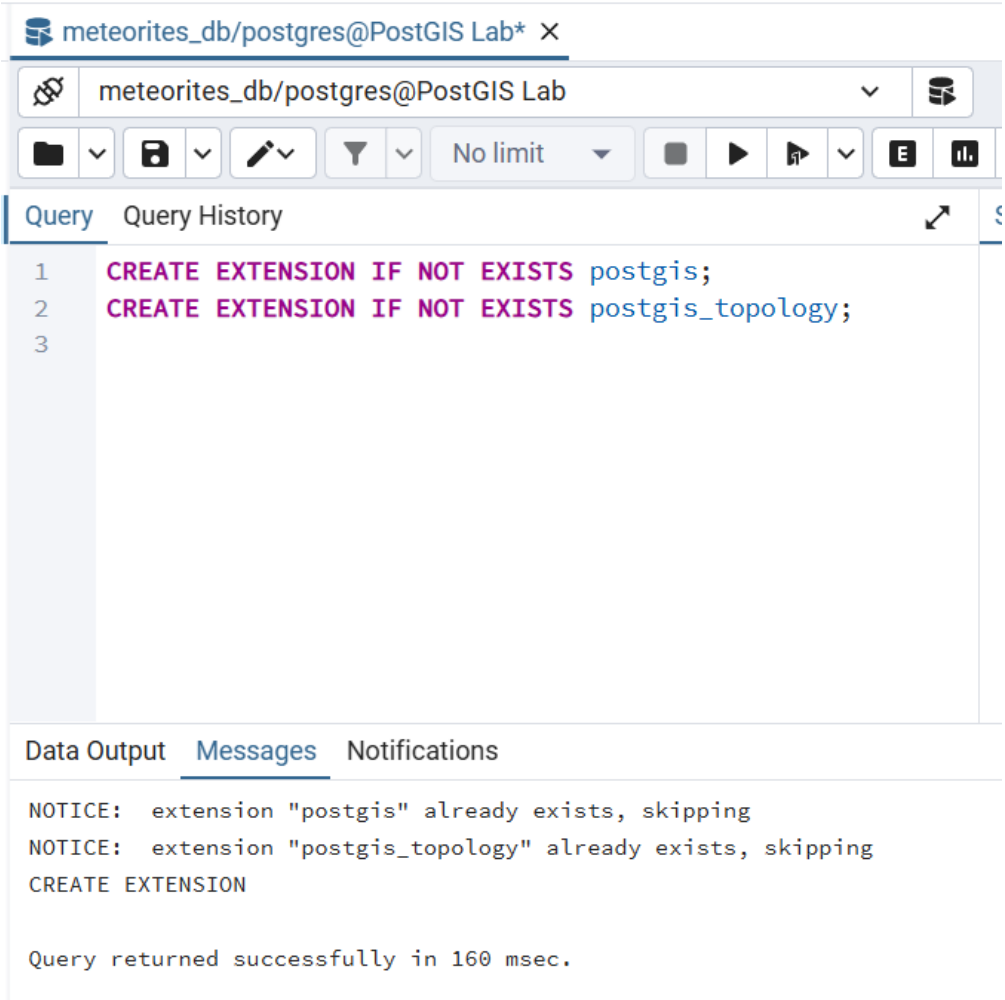
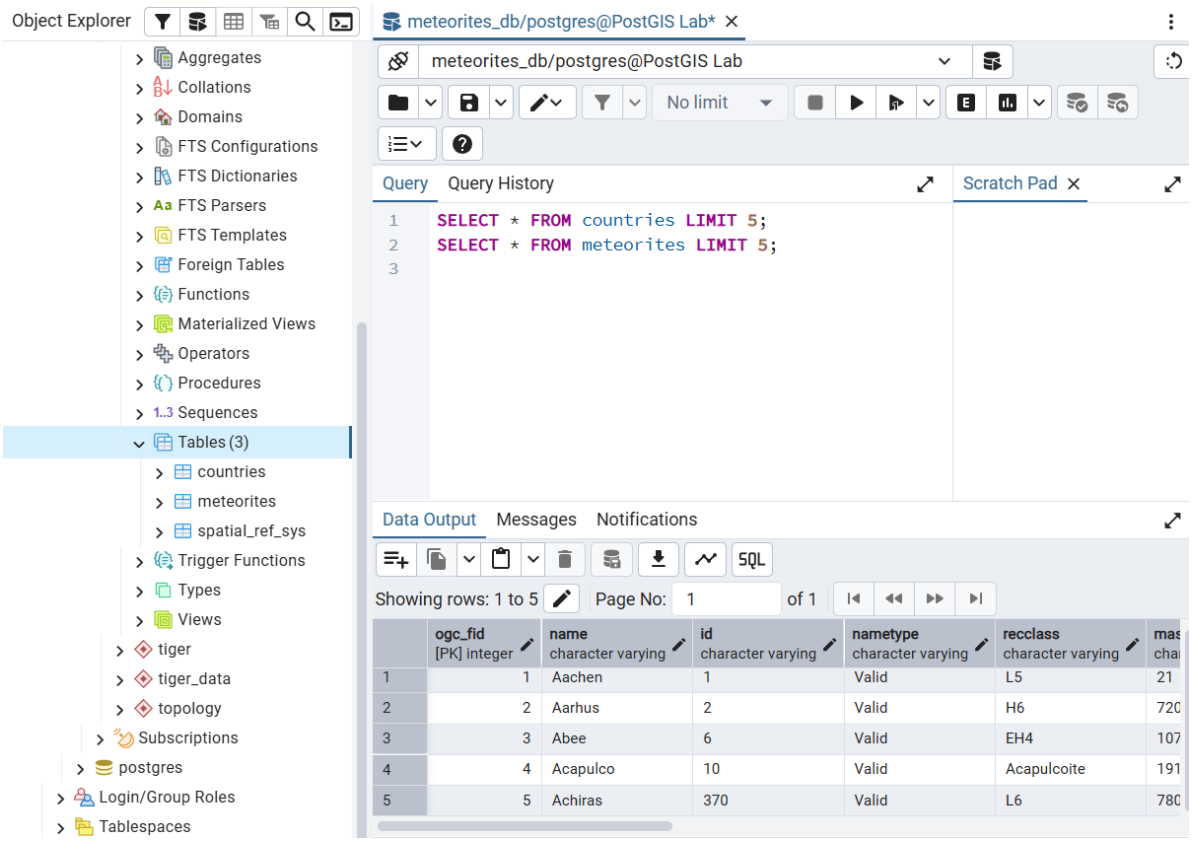



Рисунок 4 – Включение расширения

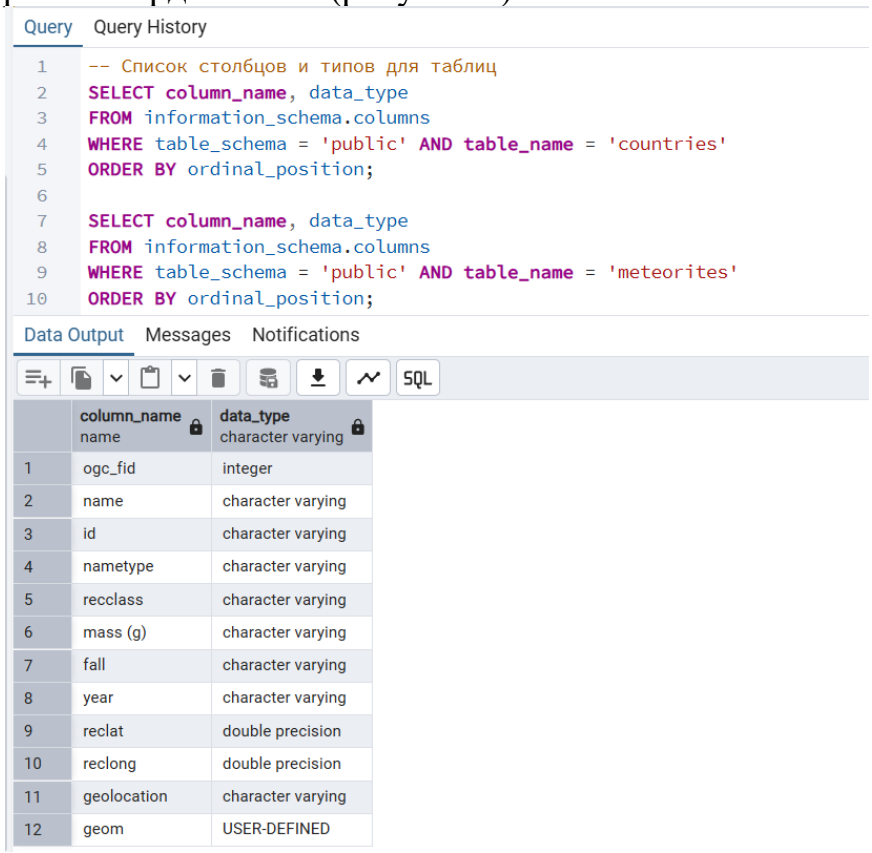
Далее импортирую файлы с падениями метеоритов и границами стран через OSGeo4W. В pgAdmin 4 появились таблицы стран и метеоритов, которые мы проверили (рисунок 5).




| | ogc_fid [PK] integer | name character varying | id character varying | nametype character varying | recclass character varying |
|---|---|---|---|---|---|
| 1 | 1 | Aachen | 1 | Valid | L5 |
| 2 | 2 | Aarhus | 2 | Valid | H6 |
| 3 | 3 | Abee | 6 | Valid | EH4 |
| 4 | 4 | Acapulco | 10 | Valid | Acapulcoite |
| 5 | 5 | Achiras | 370 | Valid | L6 |

Рисунок 5 – Проверка загруженных файлов

Далее нужно убедиться, какие поля в таблицах и как именуются поля с названием страны/координатами (рисунок 6).




| | column_name name | data_type character varying |
|---|---|---|
| 1 | ogc_fid | integer |
| 2 | name | character varying |
| 3 | id | character varying |
| 4 | nametype | character varying |
| 5 | recclass | character varying |
| 6 | mass (g) | character varying |
| 7 | fall | character varying |
| 8 | year | character varying |
| 9 | reclat | double precision |
| 10 | reclong | double precision |
| 11 | geolocation | character varying |
| 12 | geom | USER-DEFINED |

Рисунок 6 – Проверка структуры таблиц

Далее пишу основной запрос, который покажет топ 20 стран по количеству падений метеоритов, попавших на их территорию (рисунок 7).

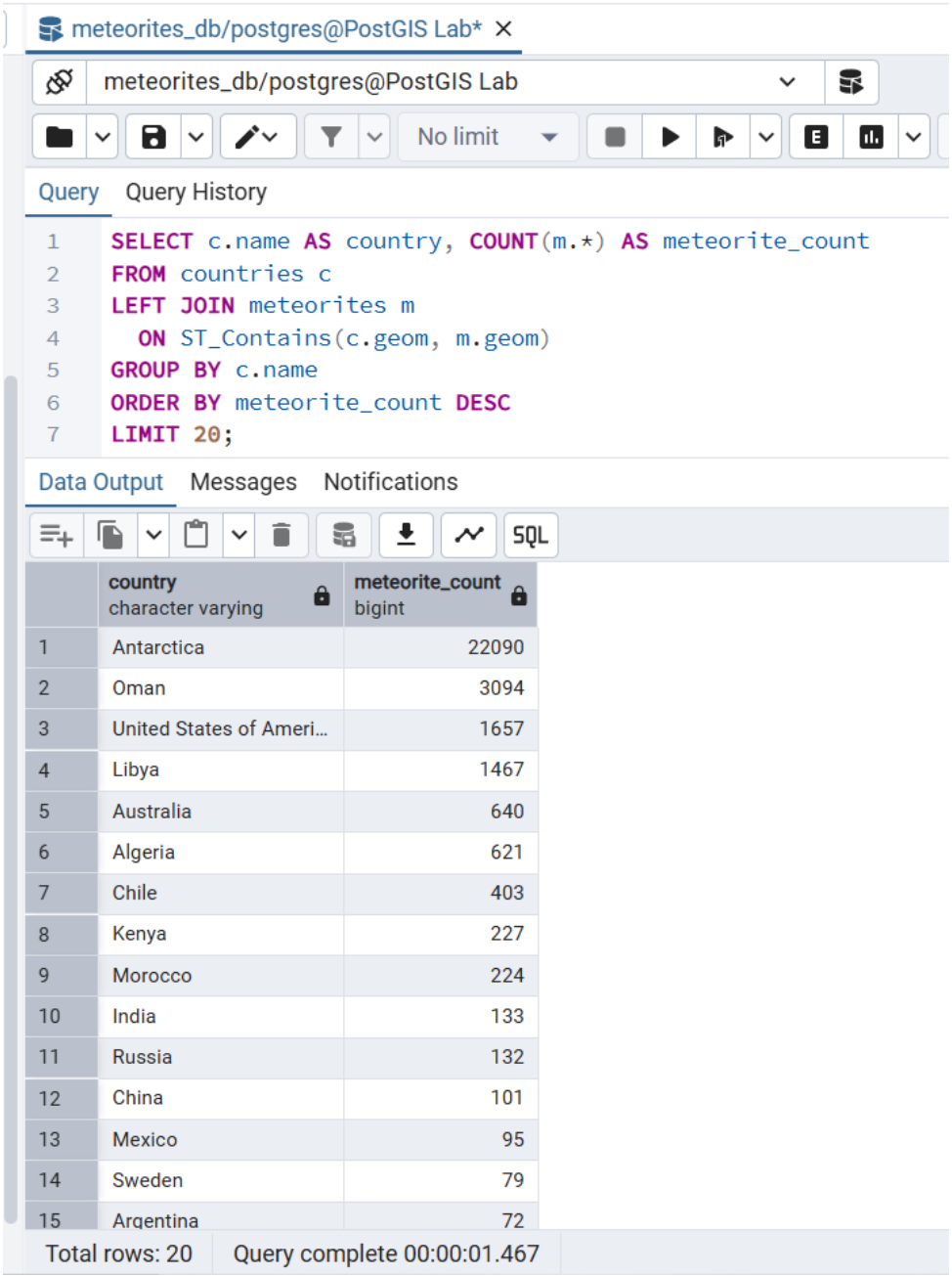



Рисунок 7 – Метеориты по странам

Далее ищу метеориты, которые находятся вне всех стран (рисунок 8).

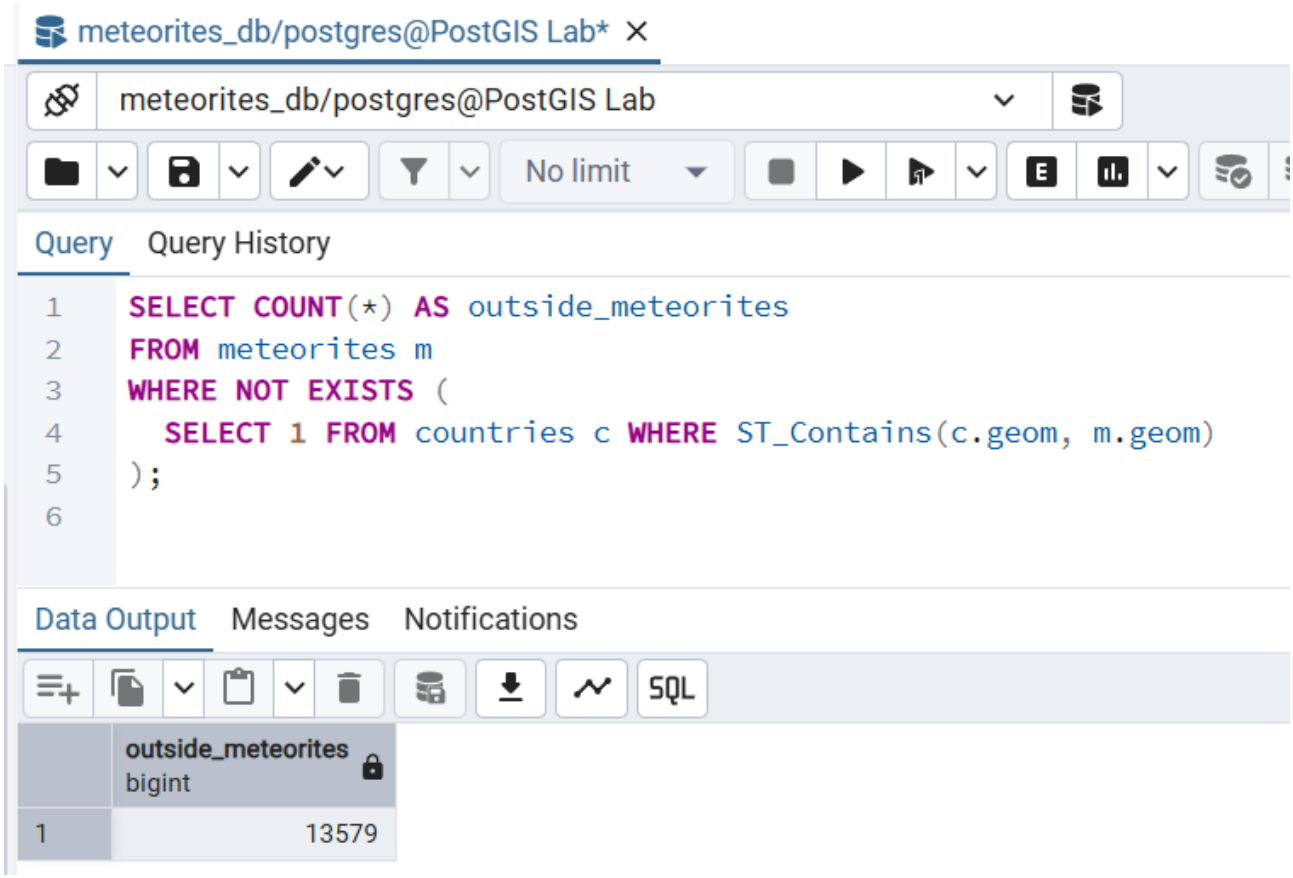



Рисунок 8 – Метеориты вне всех стран

Далее провожу временной анализ по годам (так как year хранится в строке нужно перевести в числа) (рисунок 9).

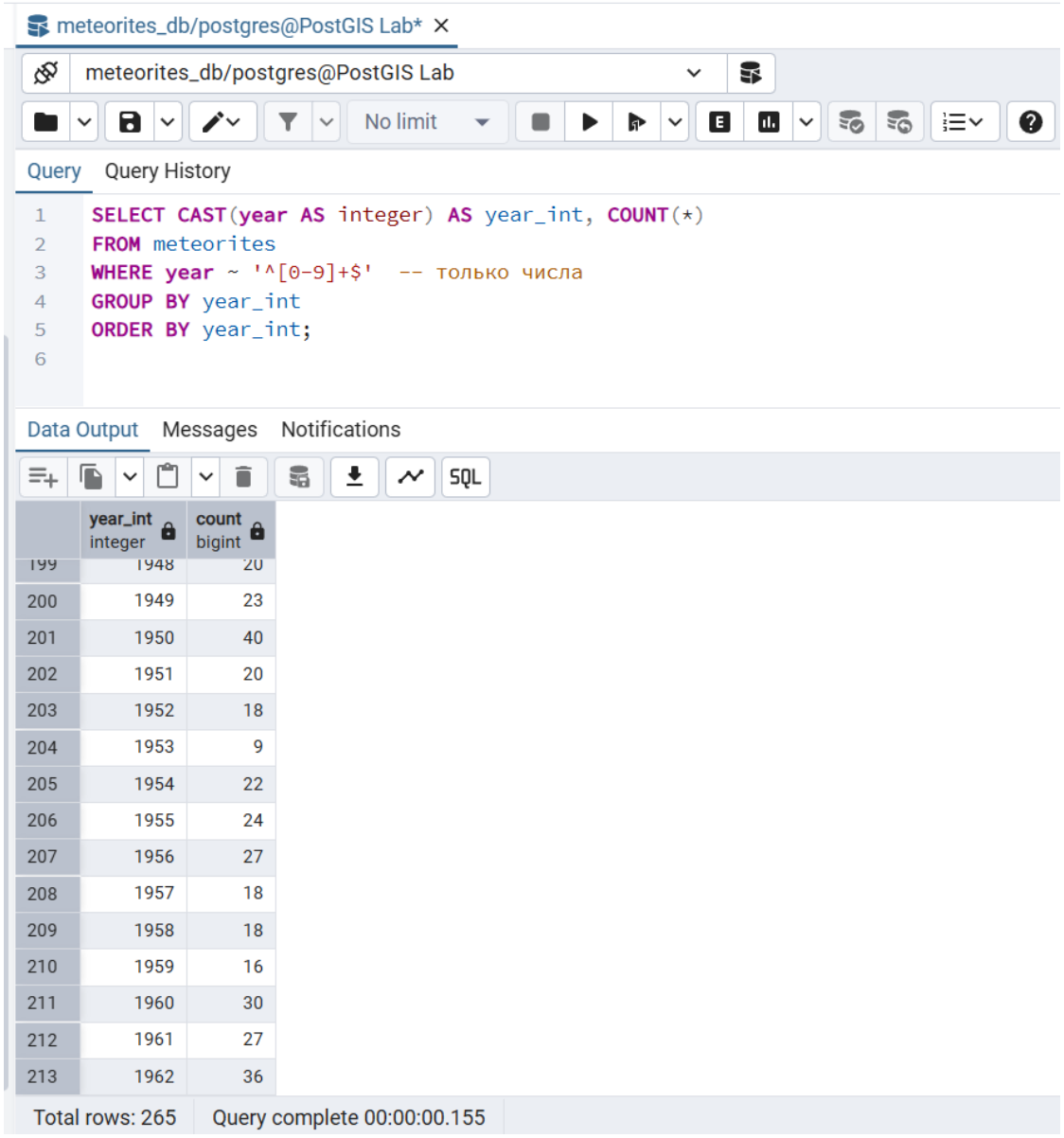



Рисунок 9 – Анализ по годам

Далее создаю и экспортирую результаты (топ 100) в формате csv (рисунок 10-11).

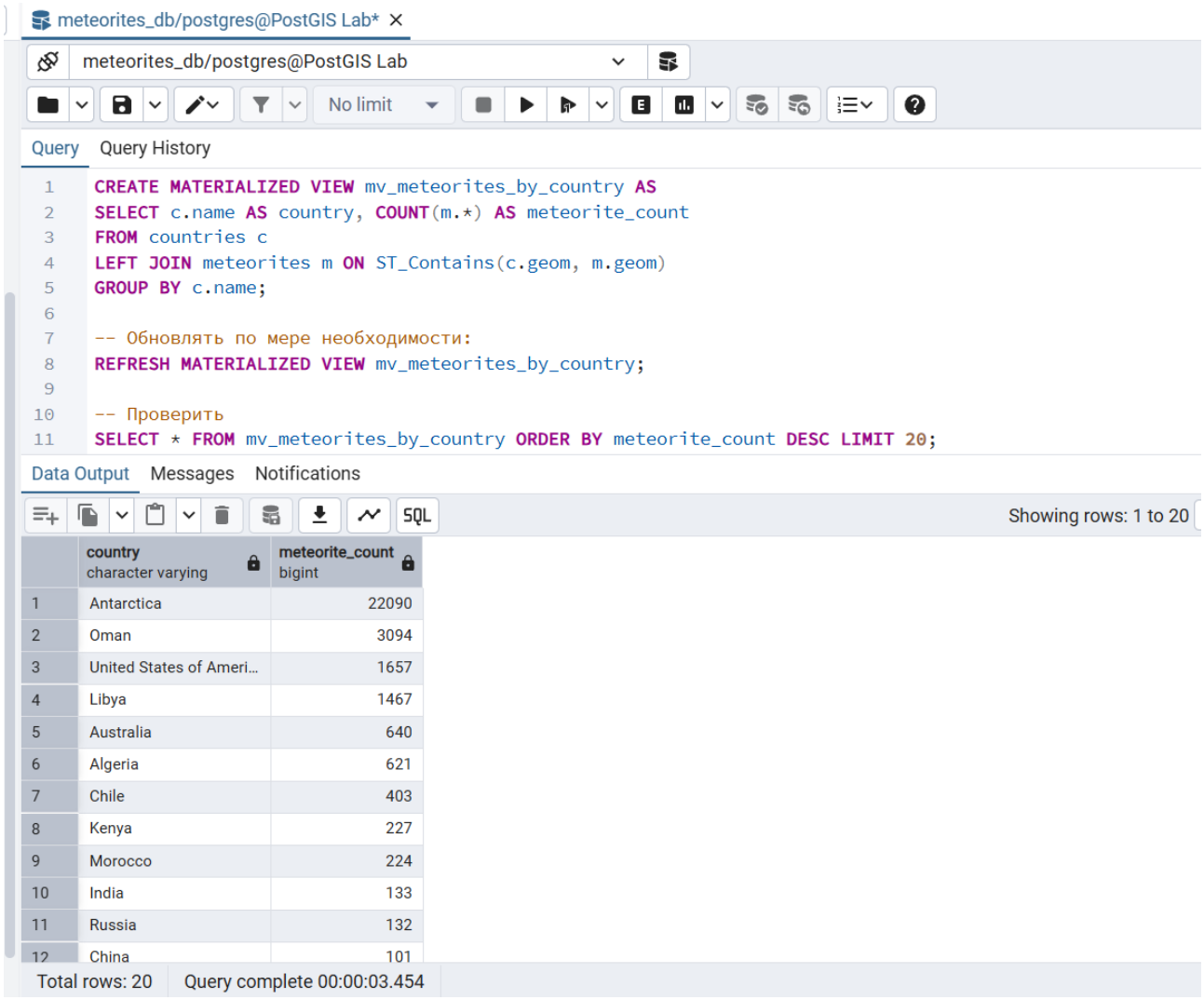
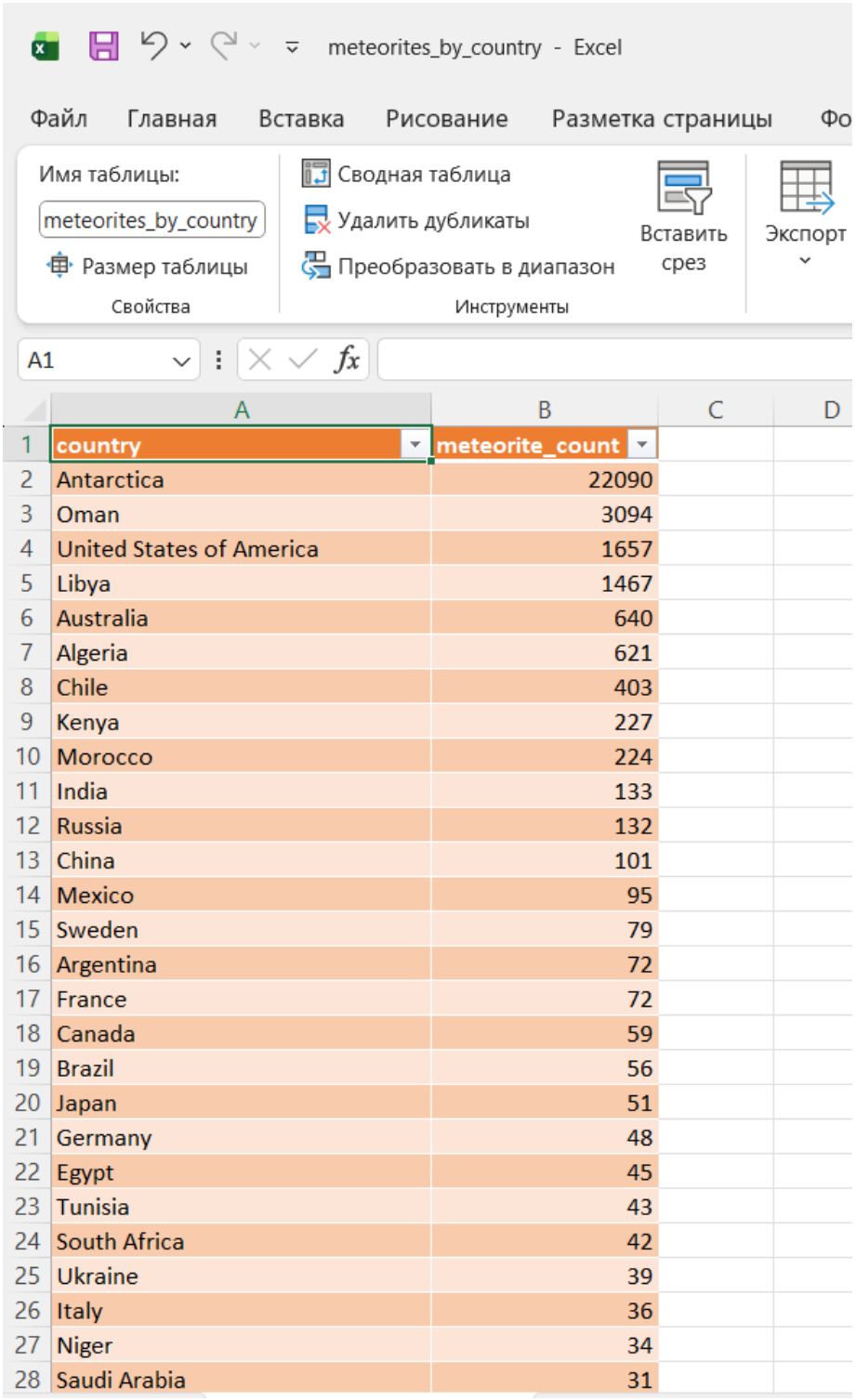



Рисунок 10 – Создание готовой таблицы для отчетов




| | A | B | C | D |
|---|---|---|---|---|
| 1 | country | meteorite_count | | |
| 2 | Antarctica | 22090 | | |
| 3 | Oman | 3094 | | |
| 4 | United States of America | 1657 | | |
| 5 | Libya | 1467 | | |
| 6 | Australia | 640 | | |
| 7 | Algeria | 621 | | |
| 8 | Chile | 403 | | |
| 9 | Kenya | 227 | | |
| 10 | Morocco | 224 | | |
| 11 | India | 133 | | |
| 12 | Russia | 132 | | |
| 13 | China | 101 | | |
| 14 | Mexico | 95 | | |
| 15 | Sweden | 79 | | |
| 16 | Argentina | 72 | | |
| 17 | France | 72 | | |
| 18 | Canada | 59 | | |
| 19 | Brazil | 56 | | |
| 20 | Japan | 51 | | |
| 21 | Germany | 48 | | |
| 22 | Egypt | 45 | | |
| 23 | Tunisia | 43 | | |
| 24 | South Africa | 42 | | |
| 25 | Ukraine | 39 | | |
| 26 | Italy | 36 | | |
| 27 | Niger | 34 | | |
| 28 | Saudi Arabia | 31 | | |

Рисунок 11 – Полученный файл

Далее перехожу в VS Code, где для начала установила нужные библиотеки через терминал pip install jupyter geopandas sqlalchemy psycopg2-binary. Создала новый файл, импортировала библиотеки, подключилась к БД (рисунок 12).

```
# --- Ваши параметры подключения ---
db_host = 'localhost'
db_port = '5433'          # <-- Ваш порт из docker -p
db_user = 'postgres'      # <-- Ваш POSTGRES_USER
db_pass = 'postgres'      # <-- Ваш POSTGRES_PASSWORD
db_name = 'meteorites_db'  # <-- Ваш POSTGRES_DB

# Создаем строку подключения
db_connection_str = f"postgresql://{db_user}:{db_pass}@{db_host}:{db_port}/{db_name}"

# Создаем "движок" подключения
try:
    db_connection = create_engine(db_connection_str)
    print(f"Успешное подключение к базе: {db_name}")
except Exception as e:
    print(f"ОШИБКА ПОДКЛЮЧЕНИЯ: {e}")
```

[2] ✓ 0.5s

```
Успешное подключение к базе: meteorites_db
```

Рисунок 12 – Подключение к БД

Теперь я выполню два запроса, чтобы забрать таблицы countries и meteorites из PostGIS в Python. Я использую GeoPandas, чтобы прочитать таблицы PostGIS сразу как гео-датафреймы (рисунок 13).

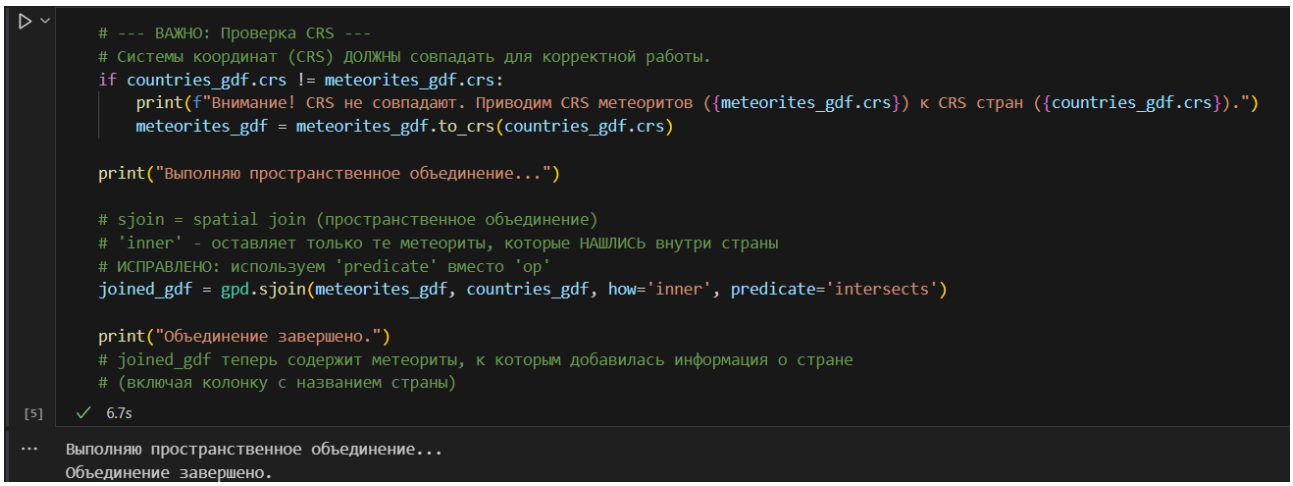
```
# Загружаем полигоны стран
try:
    countries_gdf = gpd.read_postgis(
        'SELECT * FROM countries',
        db_connection,
        geom_col='geom'
    )
    print(f"Загружено {len(countries_gdf)} стран.")

    # Загружаем точки метеоритов
    meteorites_gdf = gpd.read_postgis(
        'SELECT * FROM meteorites',
        db_connection,
        geom_col='geom'
    )
    print(f"Загружено {len(meteorites_gdf)} метеоритов.")

except Exception as e:
    print(f"ОШИБКА ЗАГРУЗКИ ДАННЫХ: {e}")
    print("Проверьте, что таблицы 'countries' и 'meteorites' существуют в базе 'meteorites_db'")
```

[3] ✓ 4.0s

```
Загружено 258 стран.
Загружено 45716 метеоритов.
```

Рисунок 13 – Загрузка данных из БД в GeoPandas

Теперь использую функцию gpd.sjoin (spatial join) для поиска метеоритов, которые «пересекаются» с полигонами стран (рисунок 14).



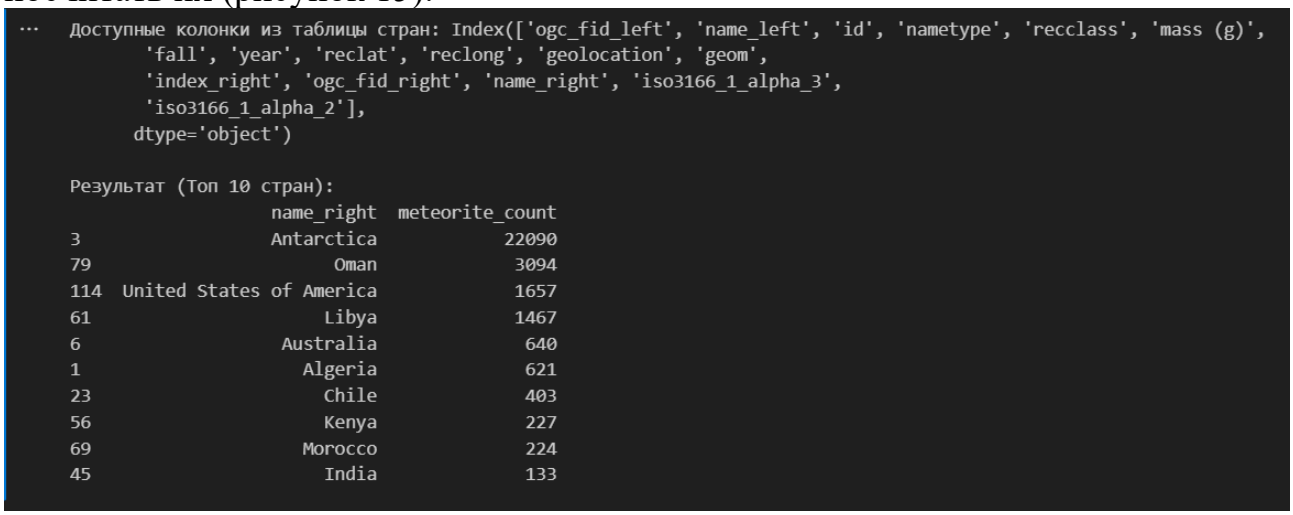
```
# --- ВАЖНО: Проверка CRS ---
# Системы координат (CRS) ДОЛЖНЫ совпадать для корректной работы.
if countries_gdf.crs != meteorites_gdf.crs:
    print(f"Внимание! CRS не совпадают. Приводим CRS метеоритов ({meteorites_gdf.crs}) к CRS стран ({countries_gdf.crs}).")
    meteorites_gdf = meteorites_gdf.to_crs(countries_gdf.crs)

print("Выполняю пространственное объединение...")

# sjoin = spatial join (пространственное объединение)
# 'inner' - оставляет только те метеориты, которые НАШЛИСЬ внутри страны
# ИСПРАВЛЕНО: используем 'predicate' вместо 'op'
joined_gdf = gpd.sjoin(meteorites_gdf, countries_gdf, how='inner', predicate='intersects')

print("Объединение завершено.")
# joined_gdf теперь содержит метеориты, к которым добавилась информация о стране
# (включая колонку с названием страны)
```

[5] ✓ 6.7s

```
Выполняю пространственное объединение...
Объединение завершено.
```

Рисунок 14 – Анализ (пространственное соединение)

Финальным шагом является группировка и получение результатов. Теперь есть таблица, где у каждого метеорита есть «метка» страны. Осталось просто посчитать их (рисунок 15).



```
Доступные колонки из таблицы стран: Index(['ogc_fid_left', 'name_left', 'id', 'nametype', 'recclass', 'mass (g)',
       'fall', 'year', 'reclat', 'reclong', 'geolocation', 'geom',
       'index_right', 'ogc_fid_right', 'name_right', 'iso3166_1_alpha_3',
       'iso3166_1_alpha_2'],
      dtype='object')

Результат (Топ 10 стран):
                  name_right  meteorite_count
3                 Antarctica            22090
79                      Oman             3094
114  United States of America             1657
61                     Libya             1467
6                  Australia              640
1                    Algeria              621
23                     Chile              403
56                     Kenya              227
69                   Morocco              224
45                     India              133
```

Рисунок 15 – Группировка и получение результата

Можно увидеть, что результаты и с использованием SQL-запросов, и с использованием Python-кода аналогичны, что подтверждает корректность работы.

**Вопросы:**

1. Насколько стабильными во времени являются анализируемые данные?

Анализируемые пространственные данные обладают разной степенью стабильности. Слой метеоритов является очень стабильным, так как координаты падения или обнаружения метеорита, упавшего даже столетие назад, не меняются. База данных может пополняться новыми находками, но исторические данные остаются статичными. В то же время, слой границ стран является крайне нестабильным. Политические границы постоянно подвержены изменениям вследствие военных конфликтов, распадов государств (например, СССР или Югославия), объединений или образования новых стран (например, Южный

Судан). Таким образом, используемый файл countries.geojson представляет собой лишь моментальный снимок современных политических границ.

2. Насколько корректным будет заключение о количестве падений метеоритов на территорию той или иной страны, если учитывать дату обнаружения (падения) метеоритов?

Корректность ограничена. Дата обнаружения и сама дата падения не всегда совпадают. Метеорит мог упасть в прошлом, а быть обнаружен позже и в другом политическом статусе территории. Также границы государств могли измениться.

Для корректного утверждения нужно учитывать: время падения, временной контекст границы (границы страны на момент падения) и статус обнаружения. Без таких уточнений выводы будут приближенными.

3. Какие варианты вы предложили бы для визуализации на карте таких явлений как природные катаклизмы (извержения вулканов, землетрясения, ураганы и т.п.), которые имеют трансграничный характер? Как можно отразить на карте динамику явления?

Чтобы наглядно показать на карте природные катаклизмы, которые не привязаны к границам стран, нужно использовать два ключевых метода.

Во-первых, для отображения пространства и силы явления лучше всего подходят тепловые карты. Они показывают «горячие точки», где активность (например, землетрясений) самая высокая, полностью игнорируя политические границы. Для ураганов можно нарисовать линию, показывающую их путь, при этом толщина или цвет этой линии сразу покажет, насколько сильным был ураган. Важно, что на таких картах границы стран нужно делать очень бледными, чтобы они не отвлекали внимание от самого природного события.

Во-вторых, для отражения динамики, то есть движения во времени, самым простым интерактивным решением является временной слайдер (Time Slider). Это ползунок, который пользователь может двигать, и карта будет показывать состояние явления (например, где находится ураган или как далеко распространилось облако пепла) только на выбранный момент времени. Также для отчетов часто создают анимированные карты в виде видео или GIF-файлов, которые показывают движение катаклизма кадр за кадром, что является наиболее наглядным способом демонстрации развития процесса.

**Вывод:** лабораторная работа позволила получить практический опыт работы со всем стеком современных ГИС-инструментов: от контейнеризации и управления пространственной БД до выполнения сложных аналитических операций как внутри самого сервера (SQL), так и программно (Python). Полученные навыки являются фундаментом для дальнейшей работы в области геоинформатики.

**Введение**

**Цель работы:** в лабораторной работе необходимо изучить возможности пространственной базы данных PostGIS, научиться выполнять операции импорта геоданных в базу данных PostgreSQL, а также освоить выполнение пространственных запросов для анализа географических данных через Jupyter Notebook.

**Ход работы**

Запускаю раннее созданную базу данных (рисунок 1).

```
PS C:\Users\Nasty\myproject\fitproject> & C:/Users/Nasty/myproject/fitproject/.venv/Scripts/Activate.ps1
(.venv) PS C:\Users\Nasty\myproject\fitproject> docker start postgis_lab
postgis_lab
(.venv) PS C:\Users\Nasty\myproject\fitproject> docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED      STATUS         PORTS                                       NAMES
ed500637273f   postgis/postgis    "docker-entrypoint.s…"   8 days ago   Up 9 seconds   0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp   postgis_lab
(.venv) PS C:\Users\Nasty\myproject\fitproject>
```

Рисунок 1 – Запуск докера

Далее устанавливаю библиотеки (рисунок 2).

```
Generate  Code  Markdown

# Устанавливаем библиотеки (если не установлены)
# !pip install SQLAlchemy psycopg2-binary
from sqlalchemy import create_engine, text
from sqlalchemy.exc import SQLAlchemyError

print("✅ Все библиотеки импортированы")

[1]  ✓ 0.3s                                                    Python

…  ✅ Все библиотеки импортированы
```

Рисунок 2 – Установка библиотек

Я использовала подключение к БД, работающей в Docker-контейнере, подключение было выполнено через create_engine() библиотеки SQLAlchemy (рисунок 3).

```
# Подключение к PostGIS (через Docker)
DB_URL = "postgresql+psycopg2://postgres:postgres@localhost:5433/meteorites_db"
engine = create_engine(DB_URL, future=True, echo=False)

print("✅ Подключение к базе данных успешно")

[2]  ✓ 0.2s                                                    Python

…  ✅ Подключение к базе данных успешно
```

Рисунок 3 – Подключение к БД

Запрос 1 – метеориты, упавшие в России. Запрос выбирает первые 10 метеоритов, геометрия которых попадает в геометрию страны «Russia». Этот запрос демонстрирует работу с пространственными данными в PostGIS (рисунок 4).

```
query1 = """
SELECT m.name, m.recclass, m."mass (g)", m.year
FROM meteorites m
JOIN countries c ON ST_Contains(c.geom, m.geom)
WHERE c.name = 'Russia'
LIMIT 10;
"""

with engine.connect() as conn:
    result = conn.execute(text(query1))
    rows = result.mappings().all()

print("🔹 Первые 10 метеоритов, упавших в России:")
for r in rows:
    print(dict(r))
```

[4] ✓ 0.2s Python

```
🔹 Первые 10 метеоритов, упавших в России:
{'name': 'Barnaul', 'recclass': 'H5', 'mass (g)': '23.2', 'year': '1904'}
{'name': 'Boguslavka', 'recclass': 'Iron, IIAB', 'mass (g)': '256000', 'year': '1916'}
{'name': 'Boriskino', 'recclass': 'CM2', 'mass (g)': '1342', 'year': '1930'}
{'name': 'Borodino', 'recclass': 'H5', 'mass (g)': '500', 'year': '1812'}
{'name': 'Brient', 'recclass': 'Eucrite-pmict', 'mass (g)': '219', 'year': '1933'}
{'name': 'Chelyabinsk', 'recclass': 'LL5', 'mass (g)': '100000', 'year': '2013'}
{'name': 'Demina', 'recclass': 'L6', 'mass (g)': '16400', 'year': '1911'}
{'name': 'Doroninsk', 'recclass': 'H5-7', 'mass (g)': '3891', 'year': '1805'}
{'name': 'Glasatovo', 'recclass': 'H4', 'mass (g)': '152000', 'year': '1918'}
{'name': 'Grosnaja', 'recclass': 'CV3', 'mass (g)': '3500', 'year': '1861'}
```

Рисунок 4 – Первый запрос

Запрос 2 – метеориты, найденные в период 1990-2000 гг. Были отфильтрованы строки, у которых год – корректное четырёхзначное число, и преобразован в integer. Запрос позволяет получить список метеоритов за указанный временной диапазон (рисунок 5).

```
query2 = """
SELECT name, year, recclass, "mass (g)" AS mass_g
FROM meteorites
WHERE year ~ '^[0-9]{4}$'
  AND CAST(year AS integer) BETWEEN 1990 AND 2000
ORDER BY CAST(year AS integer)
LIMIT 10;
"""

with engine.connect() as conn:
    rows = conn.execute(text(query2)).mappings().all()

print("🔹 Метеориты 1990–2000 гг:")
for r in rows:
    print(dict(r))
```

[6] ✓ 0.3s Python

```
🔹 Метеориты 1990–2000 гг:
{'name': 'Yanzhuang', 'year': '1990', 'recclass': 'H6', 'mass_g': '3500'}
{'name': 'Acfer 066', 'year': '1990', 'recclass': 'LL3.8-6', 'mass_g': '517'}
{'name': 'Itqiy', 'year': '1990', 'recclass': 'EH7-an', 'mass_g': '4720'}
{'name': 'Sterlitamak', 'year': '1990', 'recclass': 'Iron, IIIAB', 'mass_g': '325000'}
{'name': 'Magombedze', 'year': '1990', 'recclass': 'H3-5', 'mass_g': '666.6'}
{'name': 'Jalanash', 'year': '1990', 'recclass': 'Ureilite', 'mass_g': '700'}
{'name': 'Burnwell', 'year': '1990', 'recclass': 'H4-an', 'mass_g': '1504'}
{'name': 'Glanerbrug', 'year': '1990', 'recclass': 'L/LL5', 'mass_g': '670'}
{'name': 'Quija', 'year': '1990', 'recclass': 'H', 'mass_g': '17450'}
{'name': 'Acfer 067', 'year': '1990', 'recclass': 'H5', 'mass_g': '383'}
```

Рисунок 5 – Второй запрос

Запрос 3 – самые тяжелые метеориты (больше 10 кг). Так как в таблице столбец массы имеет имя «mass (g)», пришлось учитывать кавычки, исключить строки с пустыми значениями, фильтровать только числовые строки, преобразовывать значение в numeric. Запрос выбирает метеориты с массой больше 10 000 г (рисунок 6). Этот запрос показывает работу с преобразованием данных.

```
query3 = """
SELECT name, recclass, "mass (g)" AS mass_g, year
FROM meteorites
WHERE "mass (g)" IS NOT NULL
  AND "mass (g)" <> ''        -- исключаем пустые строки
  AND "mass (g)" ~ '^[0-9]+$' -- оставляем только строки, состоящие из цифр
  AND ("mass (g)")::numeric > 10000
ORDER BY ("mass (g)")::numeric DESC
LIMIT 10;
"""

with engine.connect() as conn:
    rows = conn.execute(text(query3)).mappings().all()

print("🔹 Самые тяжёлые метеориты (>10 кг):")
for r in rows:
    print(dict(r))
```

[8] ✓ 0.0s Python

```
🔹 Самые тяжёлые метеориты (>10 кг):
{'name': 'Hoba', 'recclass': 'Iron, IVB', 'mass_g': '60000000', 'year': '1920'}
{'name': 'Cape York', 'recclass': 'Iron, IIIAB', 'mass_g': '58200000', 'year': '1818'}
{'name': 'Campo del Cielo', 'recclass': 'Iron, IAB-MG', 'mass_g': '50000000', 'year': '1575'}
{'name': 'Canyon Diablo', 'recclass': 'Iron, IAB-MG', 'mass_g': '30000000', 'year': '1891'}
{'name': 'Armanty', 'recclass': 'Iron, IIIE', 'mass_g': '28000000', 'year': '1898'}
{'name': 'Gibeon', 'recclass': 'Iron, IVA', 'mass_g': '26000000', 'year': '1836'}
{'name': 'Chupaderos', 'recclass': 'Iron, IIIAB', 'mass_g': '24300000', 'year': '1852'}
{'name': 'Mundrabilla', 'recclass': 'Iron, IAB-ung', 'mass_g': '24000000', 'year': '1911'}
{'name': 'Sikhote-Alin', 'recclass': 'Iron, IIAB', 'mass_g': '23000000', 'year': '1947'}
{'name': 'Bacubirito', 'recclass': 'Iron, ungrouped', 'mass_g': '22000000', 'year': '1863'}
```

Рисунок 6 – Третий запрос

Создание запроса на добавление данных (INSERT). В таблицу meteorites была добавлена тестовая запись – имя: Notebook Test Meteorite, масса: 2500, координаты: (37.62, 55.75). После выполнения был выведен результат вставки (рисунок 7).

```
insert_query = """
INSERT INTO meteorites (name, id, nametype, recclass, "mass (g)", fall, year, reclat, reclong, geom)
VALUES (:name, :id, :nametype, :recclass, :mass_g, :fall, :year, :reclat, :reclong,
        ST_SetSRID(ST_MakePoint(:reclong, :reclat), 4326))
RETURNING name, year, recclass, "mass (g)" AS mass_g;
"""

new_meteorite = {
    "name": "Notebook Test Meteorite",
    "id": "ipynb-001",
    "nametype": "Valid",
    "recclass": "L6",
    "mass_g": "2500",
    "fall": "Found",
    "year": "2025",
    "reclat": 55.75,
    "reclong": 37.62
}

with engine.begin() as conn:
    res = conn.execute(text(insert_query), new_meteorite)
    print("✅ Добавлен новый метеорит:")
    for r in res.mappings():
        print(dict(r))
```

[10] ✓ 0.1s Python

```
✅ Добавлен новый метеорит:
{'name': 'Notebook Test Meteorite', 'year': '2025', 'recclass': 'L6', 'mass_g': '2500'}
```

Рисунок 7 – Добавление записи

Отдельным запросом я выбрала в таблице запись по имени Notebook Test Meteorite, чтобы убедиться, что INSERT выполнен успешно (рисунок 8). Также, чтобы база данных осталась в исходном виде, я удалила тестовую запись (рисунок 9). SQLAlchemy вернуло, что запись удалена успешно.


Рисунок 8 – Проверка записи


Рисунок 9 – Возврат к исходным данным

**Вывод:** в ходе лабораторной работы было выполнено программное подключение к базе данных PostGIS, запущенной в Docker, с использованием Python, SQLAlchemy и psycopg2 в среде VS Code/Jupyter Notebook. Были созданы три SQL-запроса для извлечения данных, включая пространственный запрос с функцией ST_Contains, временную выборку и запрос с фильтрацией по числовым данным. Также был выполнен запрос на добавление новой пространственной записи в таблицу метеоритов и проверка корректности её вставки. В завершение тестовая запись была удалена, а соединение с базой данных закрыто. Работа позволила получить практические навыки выполнения geoSQL-запросов и взаимодействия Python с PostGIS.

**Введение**

**Цель работы:** освоить создание простого веб-сервиса на FastAPI, научиться подключать FastAPI к пространственной базе данных PostGIS, считывать данные из таблицы с геометрией (метеориты) с помощью SQLAlchemy и GeoAlchemy2, а также реализовать базовые REST-методы: получение данных (GET) и добавление данных (POST).

**Ход работы**

В VS Code создаю новый каталог для проекта. Устанавливаю FastAPI и uvicorn, запускаю сервер с минимальным кодом для проверки работоспособности (рисунок 1).

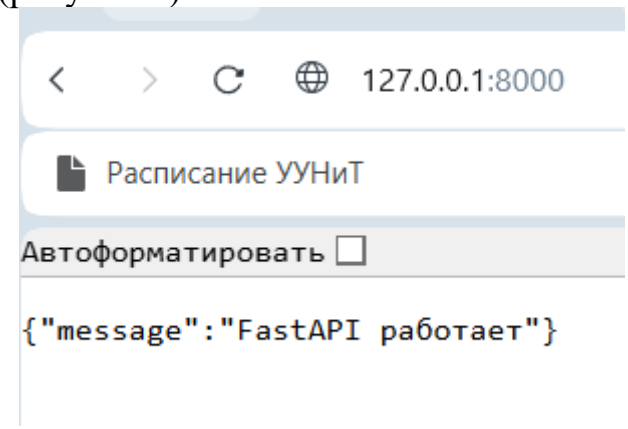



Рисунок 1 – Рабочий сервер

Для взаимодействия FastAPI с базой данных PostGIS и работы с пространственными типами данных были установлены необходимые библиотеки, создаю файл database.py, где была настроена логика подключения к базе данных и определена модель данных, соответствующая таблице метеоритов.

```
from sqlalchemy import create_engine, Column, Integer, String, Float, func
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from geoalchemy2 import Geometry

SQLALCHEMY_DATABASE_URL =
"postgresql://postgres:postgres@localhost:5433/meteorites_db"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL,
    echo=True
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
```

```
    finally:
        db.close()

class Meteorite(Base):
    __tablename__ = "meteorites"

    Привязываем Primary Key к реальному столбцу ogc_fid
    id = Column("ogc_fid", Integer, primary_key=True)

    name = Column(String)
    recclass = Column(String)
    year = Column(String)
    mass = Column("mass (g)", String)
    reclat = Column(Float)
    reclong = Column(Float)
    # ВОЗВРАЩАЕМ ГЕОМЕТРИЮ
    geom = Column(Geometry("POINT"))
```

Далее создаю роутинг для GET-запросов (файл main.py). В этом файле были определены конечные точки (эндпоинты) API для получения данных.

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from sqlalchemy import func # Импортируем func для использования функций PostGIS
from database import get_db, Meteorite
# Импортируем ST_AsText для конвертации геометрии в WKT-строку
from geoalchemy2.functions import ST_AsText
app = FastAPI()

@app.get("/")
def root():
    return {"message": "Сервис метеоритов работает"}

# Используем func.ST_AsText() для возврата геометрии в виде текста
def get_select_list():
    """Возвращает список колонок для SELECT-запроса,
    конвертируя geom в WKT."""
    return [
        Meteorite.id,
        Meteorite.name,
        Meteorite.recclass,
        Meteorite.year,
        Meteorite.mass,
        Meteorite.reclat,
        Meteorite.reclong,
        # Используем ST_AsText для конвертации геометрии в строку WKT
        ST_AsText(Meteorite.geom).label("wkt_geometry")
    ]

@app.get("/meteorites/all")
def get_all(db: Session = Depends(get_db)):
```

```
    data = db.query(*get_select_list()).all()
    # Результат — это список объектов SQLAlchemy. Преобразуем его в список словарей
для чистого JSON.
    # Это важно, так как мы используем query(*select_list)
    return [
        {
            "id": row.id,
            "name": row.name,
            "recclass": row.recclass,
            "year": row.year,
            "mass (g)": row.mass,
            "reclat": row.reclat,
            "reclong": row.reclong,
            "geom": row.wkt_geometry # Поле теперь называется 'geom' и содержит WKT
        }
        for row in data
    ]

@app.get("/meteorites/{limit}")
def get_with_limit(limit: int, db: Session = Depends(get_db)):
    # Используем ту же логику для лимитированных запросов
    data = db.query(*get_select_list()).limit(limit).all()
    return [
        {
            "id": row.id,
            "name": row.name,
            "recclass": row.recclass,
            "year": row.year,
            "mass (g)": row.mass,
            "reclat": row.reclat,
            "reclong": row.reclong,
            "geom": row.wkt_geometry
        }
        for row in data
    ]
```

Для корректной обработки пространственных данных было реализовано преобразование геометрии в текстовый формат. Сервер был запущен командой: uvicorn main:app –reload. Вывод JSON-объектов в браузере с текстовыми полями (name, recclass, year и т.д.) свидетельствует о том, что:

1. FastAPI и Uvicorn работают.
2. Подключение к БД PostGIS установлено и работает.
3. Запрос SQLAlchemy выполнен корректно.
4. Сериализация данных в JSON прошла успешно (рисунок 2).

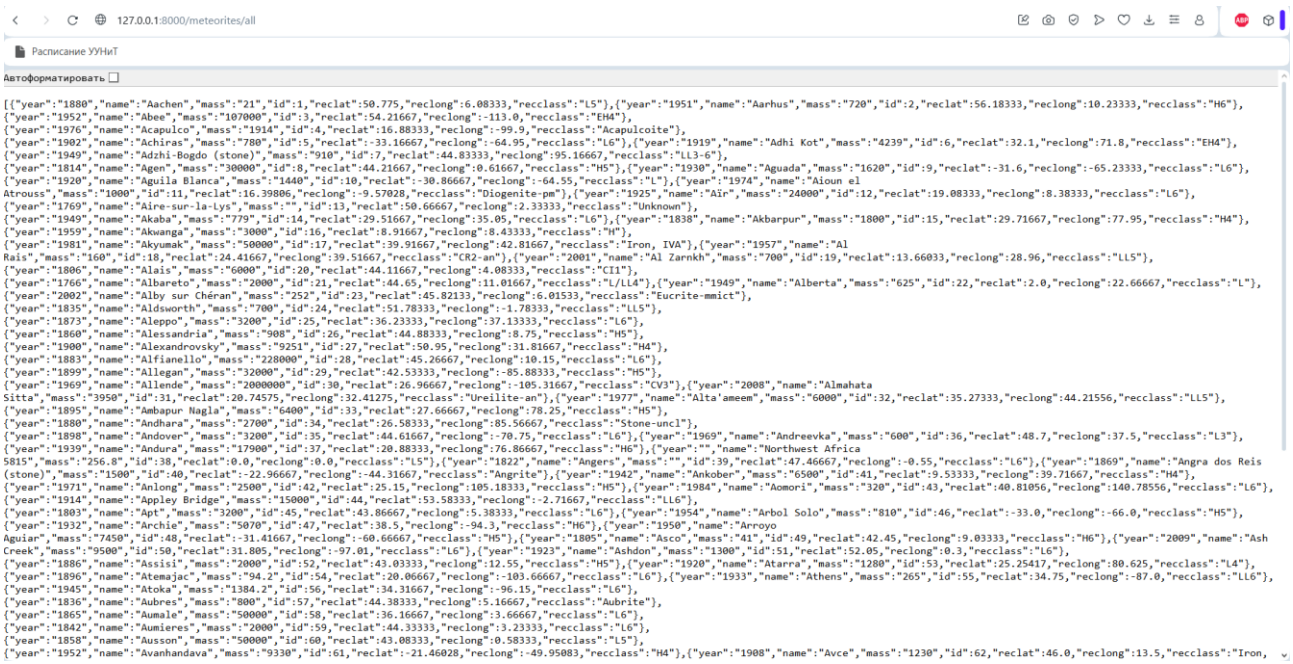



Рисунок 2 – Возврат JSON-объектов

Следующим шагом идет работа с геометрией. Для того чтобы геометрия была сериализуемой, нужно явно преобразовать ее в формат, понятный JSON, а именно, в текстовое представление WKT (Well-Known Text).

Для этого использую функцию ST_AsText из GeoAlchemy2/PostGIS. Эту функцию нужно импортировать. После внесения этих изменений и перезапуска сервера, при обращении к /meteorites/all я увидела в ответе новое поле (рисунок 3).

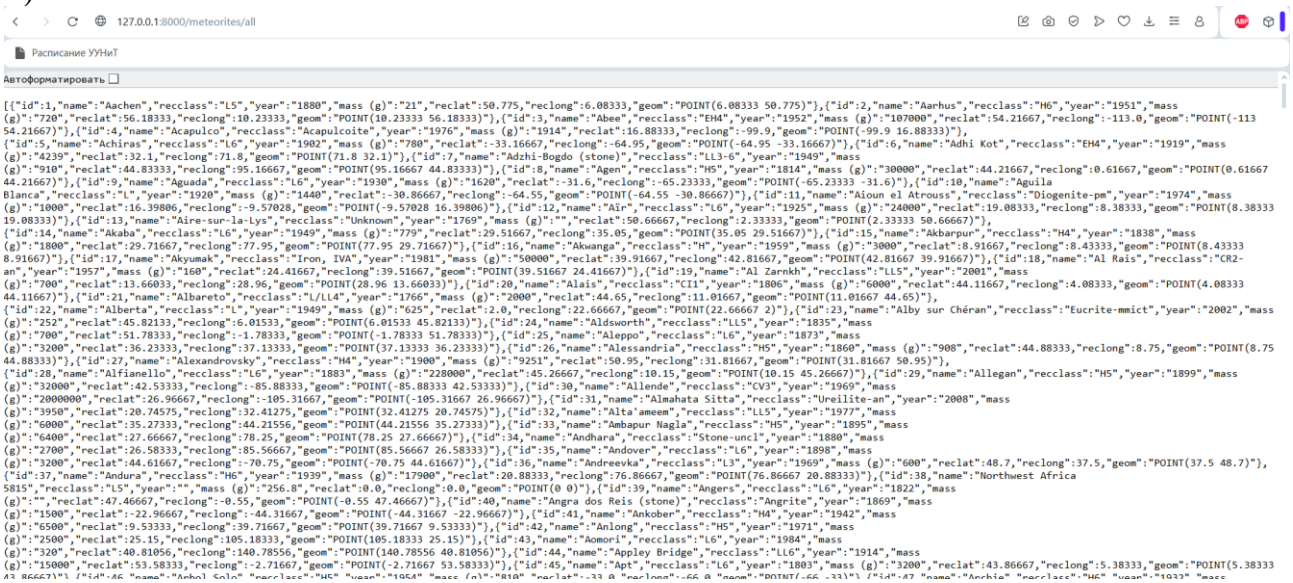



Рисунок 3 – Вывод геометрии

Теперь нужно реализовать POST-запрос в FastAPI, который будет принимать данные о новом метеорите от клиента и выполнять операцию INSERT

в базу данных PostGIS. Для начала добавляю **Pydantic-модель** для валидации и структурирования входящих данных.

```
# Импортируем BaseModel для создания схемы данных
from pydantic import BaseModel
# СХЕМА ДАННЫХ ДЛЯ ВХОДЯЩЕГО ЗАПРОСА (Body)
# Пользователь должен прислать эти поля.
class MeteoriteCreate(BaseModel):
    name: str
    recclass: str
    year: str | None = None  # Может быть пустым (None)
    mass: str | None = None
    reclat: float
    reclong: float

    # Pydantic Configuration
    class Config:
        # Разрешает использовать имена полей с пробелами, но в БД у нас уже 'mass
(g)'
        # В данном случае это не строго необходимо, но полезно для сложных моделей
        orm_mode = True
```

Теперь добавляю POST-маршрут, который будет принимать данные из MeteoriteCreate, создавать объект Meteorite (из database.py) и добавлять его в базу.

```
## 6. Создание POST-запроса
@app.post("/meteorites")
def create_meteorite(
    # Принимаем данные по схеме MeteoriteCreate
    new_meteorite: MeteoriteCreate,
    db: Session = Depends(get_db)
):
    """
    Добавляет новую запись о метеорите в таблицу 'meteorites'.
    """
    try:
        # Создаем новый объект Meteorite, используя данные из запроса.
        # GeoAlchemy2 автоматически создаст POINT из reclat/reclong.
        # Мы используем ST_MakePoint из PostGIS для создания геометрии.
        db_meteorite = Meteorite(
            name=new_meteorite.name,
            recclass=new_meteorite.recclass,
            year=new_meteorite.year,
            mass=new_meteorite.mass,
            reclat=new_meteorite.reclat,
            reclong=new_meteorite.reclong,
            # Создаем геометрию POINT из долготы и широты
            geom=func.ST_SetSRID(func.ST_MakePoint(new_meteorite.reclong,
new_meteorite.reclat), 4326)
        )
```

```
        # Добавляем объект в сессию, выполняем INSERT и фиксируем изменения
(commit)
        db.add(db_meteorite)
        db.commit()

        # Обновляем объект, чтобы получить его ID, присвоенный базой данных
        db.refresh(db_meteorite)

        return {"status": "success", "message": f"Метеорит {db_meteorite.name}
успешно добавлен (ID: {db_meteorite.id})."}

    except Exception as e:
        db.rollback()
        return {"status": "error", "message": f"Ошибка добавления данных: {e}"}
```

Теперь перезапускаю сервер, перехожу в документацию FastAPI http://127.0.0.1:8000/docs. В новом маршруте POST /meteorites вставляю текстовые данные в тело запроса. Запрос был успешно обработан FastAPI-приложением. В секции «Response body» появился ответ, который выводит функция (рисунок 4-5)

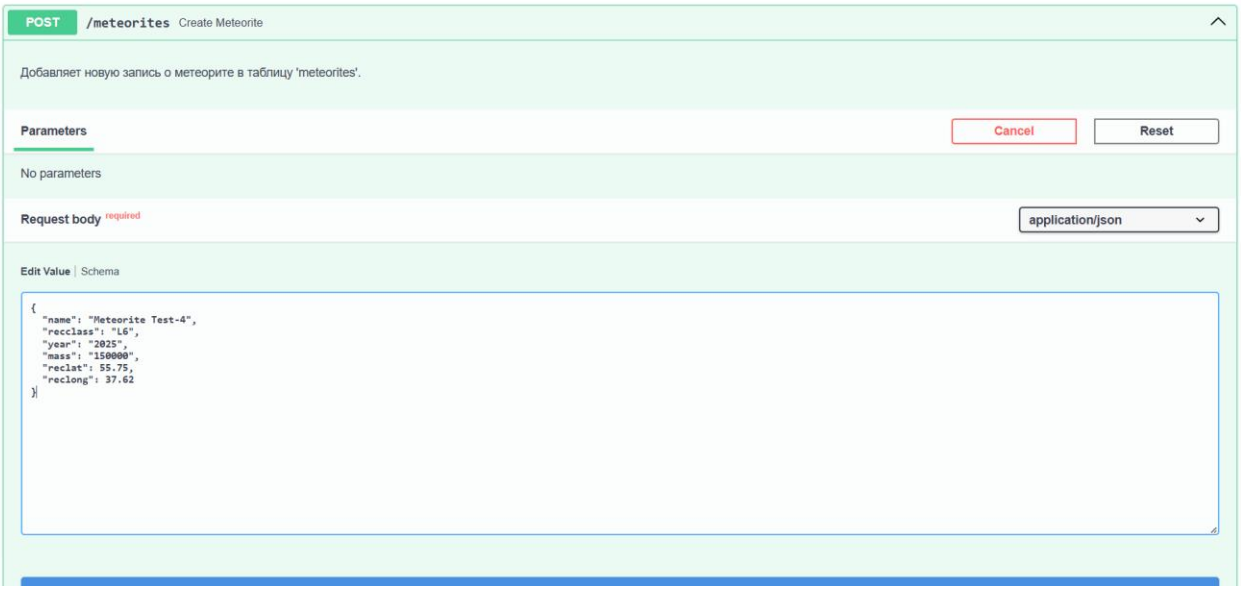



Рисунок 4 – Тело запроса


Рисунок 5 – Результат

**Вывод:** в ходе выполнения лабораторной работы была успешно достигнута цель по созданию **простого геосервиса (API) на базе фреймворка FastAPI**.

Была реализована полная интеграция современного стека технологий: **Python/FastAPI** был соединен с **пространственной базой данных PostGIS**, запущенной в Docker, через ORM-библиотеки **SQLAlchemy** и **GeoAlchemy2**.