UNIVERSITÀ
DELLA CALABRIA

Report

Multi-language Engine: Apache Spark

Big Data Analytics and Reasoning

Written by Kreuzer Florian, Matousch David

Cosenza January 2025

Contents

1 Introduction

Efficient processing of large-scale datasets has become a critical requirement across industries. Apache Spark addresses this need by offering a distributed computing framework designed for speed and scalability. Unlike Hadoop, which relies heavily on disk storage and HDFS for computations, Spark operates primarily in memory, significantly improving performance for iterative tasks.

Initially, Spark introduced Resilient Distributed Datasets (RDDs) as its core abstraction for distributed data processing. However, since Spark 2.0, RDDs have been largely replaced by the Dataset API, which combines the benefits of RDDs with optimizations from Spark's Catalyst query optimizer. This shift reflects a move toward a more structured and efficient approach to data processing.

The Spark stack includes Spark Core, which provides essential functionalities such as task scheduling, memory management, and fault recovery. Built on top of Spark Core are higher-level APIs like DataFrames and Structured Streaming, enabling developers to process structured and unstructured data efficiently. (see Figure 1)

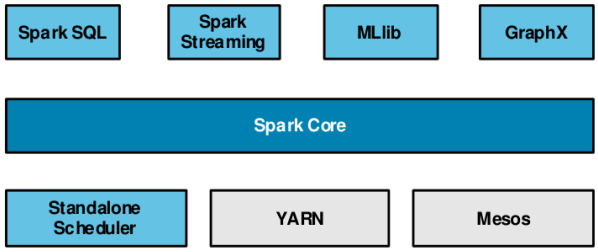

Figure 1: Spark stack and widely used higher-level APIs

2 Dataset Description

The dataset used in this project is sourced from the *OpenWeatherMap API*, a comprehensive platform for accessing weather data. This API provides current and forecasted weather information, along with historical data spanning over 45 years. The key features of the dataset include: [1]

- **Current Weather and Forecasts:**
  - Minute-by-minute forecasts for the next hour.
  - Hourly forecasts for up to 48 hours.
  - Daily forecasts for up to 8 days.
  - Government-issued weather alerts.
- **Historical Weather Data:**
  - Daily aggregated data for the past 45+ years.
- **API Access:** Users can make up to 1,000 API calls per day for free.

2.1 Example Request

To retrieve weather data, an example API call structure is as follows:

```
api.openweathermap.org/data/3.0/onecall?lat={lat}&lon={lon}&exclude={part}&appid={API key}
```

- `lat` and `lon` specify the latitude and longitude of the location.
- `exclude` filters specific data segments, such as current weather or minute forecasts.
- `API key` is the unique key required to authenticate the user.

3 Implementation

3.1 Batch processing

We used *DataFrames* for our batch processing demo. As seen in Listing 1 the fetching of the weather data involves no spark components.

Listing 1: Fetch Weather Data

```python
def fetch_weather_data(cities):
    """Fetch weather data from OpenWeatherMap API for a list of cities."""
    weather_data = []
    for city in cities:
        try:
            response = requests.get(BASE_URL, params={"q": city, "appid": API_KEY, "units":
                "metric"})
            if response.status_code == 200:
                data = response.json()
                weather_data.append({
                    "city": data["name"],
                    "temperature": data["main"]["temp"],
                    "weather": data["weather"][0]["main"],
                    "humidity": data["main"]["humidity"]
                })
            else:
                print(f"Failed to fetch data for {city}: {response.status_code}")
        except Exception as e:
            print(f"Error fetching data for {city}: {e}")
    return weather_data
```

In Listing 2 the weatherdata list is converted into a spark DataFrame.

Listing 2: Convert to DataFrame

```python
spark = SparkSession.builder \
        .appName("Batch Weather Analysis") \
        .getOrCreate()

weather_data = fetch_weather_data(CITIES)

weather_df = spark.createDataFrame(weather_data)

print("Raw Weather Data:")
weather_df.show()
```

Now we can leverage spark operations to analyze the data. In Listing 3 some examples are shown.

Listing 3: Spark Operations on DataFrame

```python
# Average Temperatures per City
avg_temp_df = weather_df.groupBy("city").agg(
    avg("temperature").alias("avg_temperature")
)
print("Average Temperature per City:")
avg_temp_df.show()

# Identify cities with extreme weather conditions
extreme_weather_df = weather_df.withColumn(
    "extreme_weather",
    when((col("temperature") > 35) | (col("temperature") < 0), lit("Yes")).otherwise(lit("No"))
)
print("Cities with Extreme Weather:")
extreme_weather_df.filter(col("extreme_weather") == "Yes").show()

# Group cities by weather condition
```

```
weather_group_df = weather_df.groupBy("weather").count()
print("Cities Grouped by Weather Condition:")
weather_group_df.show()
```

Listing 4: Spark Operations on DataFrame Output

```
Average Temperature per City:
+--------------+---------------+
|          city|avg_temperature|
+--------------+---------------+
|      New York|          -1.4|
|        London|          6.74|
|        Berlin|          0.97|
|         Tokyo|          8.48|
|        Sydney|         24.22|
|        Moscow|         -4.17|
|        Mumbai|         28.99|
|     Cape Town|         21.67|
|Rio de Janeiro|         23.12|
|       Beijing|         -3.06|
+--------------+---------------+

Cities with Extreme Weather:
+--------+--------+-----------+-------+---------------+
|    city|humidity|temperature|weather|extreme_weather|
+--------+--------+-----------+-------+---------------+
|New York|     52|       -1.4| Clear|            Yes|
|  Moscow|     74|     -4.17|  Snow|            Yes|
| Beijing|     19|     -3.06| Clear|            Yes|
+--------+--------+-----------+-------+---------------+

+-------+-----+
|weather|count|
+-------+-----+
|  Clear|    4|
| Clouds|    3|
|   Rain|    1|
|   Snow|    1|
|  Smoke|    1|
+-------+-----+
```

3.2 Stream processing

The stream processing application consists of two parts. A data generator and a data processor.

3.2.1 Data generator

Weather data gets fetched from a random city every 2 seconds. This data is then stored in a new json file.

Listing 5: Data generator

```
def simulate_streaming_data():
    # remove all old files
    shutil.rmtree(OUTPUT_DIR)
    os.makedirs(OUTPUT_DIR)

    count = 0
    while True:
        city = random.choice(CITIES)
        weather_data = fetch_weather(city)
        if weather_data:
            # Generate a unique filename using a counter
```

```
            filename = os.path.join(OUTPUT_DIR, f"weather_{count}.json")
            with open(filename, "w") as f:
                json.dump(weather_data, f)
            print(f"Generated: {filename}")
            count += 1
        time.sleep(2)
```

This function is running in a separate thread.

Listing 6: Start thread

```
threading.Thread(target=simulate_streaming_data, daemon=True).start()
```

After that you just need to attach a spark listener on the generation folder.

Listing 7: streaming DataFrame

```
streaming_df = spark.readStream \
        .schema(schema) \
        .json(OUTPUT_DIR)
```

3.2.2 Data processor

Because the *streaming_df* is already a normal DataFrame we can run any processing we want using the normal transformations. For our test application we chose the following.

Listing 8: processing DataFrame

```
processed_df = streaming_df \
        .withColumn("extreme_weather",
                    when((col("temperature") > 35) | (col("temperature") < 0),
                        lit("Yes")).otherwise(lit("No"))) \
        .withColumn("is_rainy", when(col("weather") == "Rain", lit("Yes")).otherwise(lit("No")))
```

To get the results on the screen we added a simple output to the console.

Listing 9: streaming output

```
query = processed_df.writeStream \
        .outputMode("append") \
        .format("console") \
        .option("truncate", "false") \
        .start()

    query.awaitTermination()
```

4 Results and Insights

5 Challenges and Solutions

5.1 Setup

Setup of spark with docker was the main challenge we faced. Our first approach was using *Scala*. Here we had issues with version mismatches. Some objects were serialized and then deserialized using a different java version, thus failing during the process. After that we switched to the current version that uses python to transform and process dataframes. Setup here was easier but Java version mismatches plagued us here as well. Writing a dockerfile that would work on Arm64 and x86 also proved to be challenging.

5.2 Batch processing

Batch processing was straightforward and using the weather API was easy. We ran into no problems here.

5.3 Stream processing

It is clear that streaming is more difficult than just batching. The main problem here was to understand how the data generator has to generate the data for spark to process it correctly. Creating new json files for every new batch was the solution.

6 Deployment Details

6.1 How to run

The source code is hosted on GitHub. (`https://github.com/matouschdavid/SparkPlayground.git`) All the necessary commands are listed in the `README.md file`. Depending on the internet connection, the first run can take multiple minutes.

6.2 Deployment Implementation

For our solution, we used *docker* as a deployment platform. When starting the `docker-compose` file, the environment variable `PROCESSING` has to be set. It is used to change between the batch/stream processing. The whole deployment is described in a *Dockerfile* and *docker-compose*

Listing 10: Dockerfile

```
FROM python:3.10-slim

# Set environment variables
ENV SPARK_VERSION=3.4.1
ENV HADOOP_VERSION=3
ENV PYSPARK_PYTHON=python3
ENV PYSPARK_DRIVER_PYTHON=python3

# Install java
# ... (not relevant)

# Download and install Spark
RUN wget
    https://archive.apache.org/dist/spark/spark-${SPARK_VERSION}/spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSI
    && \
    tar -xvf spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz && \
    mv spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION} /opt/spark && \
    rm spark-${SPARK_VERSION}-bin-hadoop${HADOOP_VERSION}.tgz

# Set Spark environment variables
ENV SPARK_HOME=/opt/spark
ENV PATH=$SPARK_HOME/bin:$PATH

# Install Python dependencies
## ... (not relevant)

# Set Spark environment variables for the runtime
ENV _JAVA_OPTIONS="-Djava.io.tmpdir=/tmp"
ENV PYSPARK_SUBMIT_ARGS="--driver-java-options='-Djava.io.tmpdir=/tmp' pyspark-shell"

# Default command
CMD ["python", "main.py"]
```

The base is a docker image with python installed. In order to use spark, a java installation is required. The *pyspark* package some environment configuration, which is also described in the Dockerfile. (See Listing 10)

Listing 11: docker-compose-yaml

```
version: "3.9"
```

```yaml
services:
  spark:
    build: .
    container_name: spark-container
    volumes:
      - ./app:/app
    working_dir: /app
    command: ["python", "main.py", "${PROCESSING}"]
```

The `docker-compose` file (Listing 11) uses the `Dockerfile` for the *spark* service. It also defines the processing mode.

7 Conclusion

We think that Spark is only useful for actual big data applications. The scale of our project is way too small to make use of the powerful Spark engine. Knowing this, the difficult setup makes it one of our last choices when developing something similar in the future. The API is nice, but nothing new since C# Linq and Java streams have the same API for years natively in their language with many sources you can attach.
That being said, now that we have a working setup, it is great to just spin up a docker container that executes a job and stores everything to our server if we ever need it.

References

[1] OpenWeather. Openweather api documentation. `https://openweathermap.org/api/one-call-3`, 2025. Accessed: 2025-01-08.