DEBS Challenge '24 – International Conference on Distributed and Event-based Systems

Abner Abstete, Jonas Gottal

OT6: Cloud Computing and Big Data Applications at INSA
(Institut National des Sciences Appliquées de Lyon)

In the context of the "Cloud Computing and Big Data Applications" course (OT6) at INSA (Institut National des Sciences Appliquées) Lyon, we tackled the DEBS Grand Challenge 2024, aiming to efficiently address provided queries on a data stream. Our approach leveraged PySpark, the Python version of Apache Spark. This presentation outlines our problem structuring and solution-finding process.

Keywords: Stream Processing, Apache Spark, Python, K-Means Clustering

1. INTRODUCTION

Data streaming offers real-time insights, low-latency processing, and scalability, making it crucial in applications requiring immediate decision-making, adaptability to dynamic environments, and efficient handling of massive volumes of data. In the era of big data and event-driven architectures, streaming is essential for continuous learning models, reducing latency, and supporting the requirements of modern, data-intensive applications.

1.1. DEBS Challenge

The ACM International Conference on Distributed and Event-Based Systems (DEBS) serves as a leading platform for discussing cutting-edge research in event-based computing, focusing on Big Data, AI/ML, IoT, and Distributed Systems. The 2024 DEBS Grand Challenge centers around real-time complex event processing using real-world telemetry data sourced from Backblaze. This challenge involves analyzing SMART data from over 200,000 hard drives. The Self-Monitoring, Analysis, and Reporting Technology (SMART), is an integrated monitoring system in SDDs and HDDs, designed to detect and report reliability indicators. This allows the prediction of potential imminent hardware failures with machine learning techniques like clustering to prevent data loss. Thus, this year's DEBS Grand Challenge requires us to implement two queries:

- **Query 1:** Count of the recent number of failures detected for each vault (group by storage servers) (Continuous Querying)
- **Query 2:** Use this number to continuously compute a cluster of the drives (K-Means)

The provided input data in form of a CSV file `drivedata-small.csv` entails:

- SMART readings for a list of drives (`s_1, ..., s_242`)
- `vault_ids`: a list of vault identifiers of interest for this batch (used in Q1, see below)
- `date`: a Timestamp for our sliding window
- `failure`: a binary variable indicating a failure

We have two more CSV files for query 2: `clusters.csv` with the initial centeroids for K-Means and `norm.csv` providing a range for each variable to normalize the data.

1.2. Our Approach

We considered many frameworks to build upon, with Flink already discussed in the course and tested with an example. However, we ultimately chose Apache Spark. This decision was influenced not only by some team members having given a short presentation on Spark but also by its polyglot nature and extensive implementation of various libraries. Moreover, the ease of initiating a data stream from a folder of CSV files was a key factor. Additionally, PySpark, with its structured streaming capabilities, serves as an extension of the core Spark API, making it particularly advantageous for team members familiar with Python.

2. IMPLEMENTATION

We initiated a structured data stream from CSV files as part of our data processing approach. Initially, we thoroughly examined all provided data, investigating their formats, ranges, and identifying any missing data. Subsequently, we focused on schema design, ensuring

a structured and organized foundation for our data processing pipeline, as described in Listing 1.

Listing 1. Schema for SMART Data

```
# Schema for PySpark data stream
schema = StructType([
    StructField("date", TimestampType(),
        ↪ True),
    StructField("serial_number",
        ↪ StringType(), True),
    StructField("model", StringType(),
        ↪ True),
    StructField("failure", IntegerType(),
        ↪ True),
    StructField("vault_id", IntegerType(),
        ↪  True),
    StructField("s1_read_error_rate",
        ↪ IntegerType(), True),
    ...
    StructField("s242_total_lbas_read",
        ↪ IntegerType(), True)
])
```

2.1. API Call

Our API integration posed challenges despite the guidance offered in the provided tutorial. Nevertheless, we successfully adjusted our query to efficiently transmit data to the designated page. Leveraging the initially provided modules, we established a connection with the server.

The code orchestrates a multi-stage process for transmitting data to the gRPC server. Initially, a batch of data undergoes processing using Spark, and the outcomes are persistently stored. Subsequently, each row of this result is then sent to the gRPC server, recording which result corresponds to which batch. Lastly, the system verifies whether it is processing the last batch before concluding the benchmark.

2.2. First Query

The first query is as described above: continuously count the recent number of failures detected for each vault. This means we design a sliding window based on the provided information:

- Size = 30 days
- Slide = 1 day
- For every vault v, count the number of failures NF_v in a sliding window W
- For a given day i, NF_v^i is the count of failures in the window that starts at $i-31$ (included) and ends at $i-1$ (included)
- The first window closes at day 0: you can assume 0 failures for every day $i <= 0$
- Each batch of input data will contain the identifiers of 5 vaults: you are to return the current value of NF_v for those vaults

Listing 2. Heart of `def processTheBatchQ1(batch)`

```
# Apply Spark operations
windowedData = df \
    .withWatermark("date", "31␣days") \
    .groupBy(
        df.vault_id,
        window(df.date, "30␣days", "1␣day"
            ↪ ),
        df.model
    ) \
    .agg(_sum("failure").alias("
        ↪ total_failures"))

selectedData = windowedData.select("window
    ↪ .start", "vault_id", "
    ↪ total_failures")
filteredData = selectedData.filter(
    ↪ selectedData.total_failures > 0)
```

So first, with Apache Spark's Structured Streaming API we read the CSV files as a data stream and apply the specified schema to map the data to Spark's structure.

The provided code snippet in Listing 2 is written in PySpark to answered our query 1 to "Count the recent number of failures detected for each vault":

1. `withWatermark("date", "31 days")`:
   The `withWatermark` function is used to set a watermark on the "date" column. The watermark is a threshold for late data. In this case, it is set to 31 days, meaning the system will consider data arriving up to 31 days late for processing.
2. `groupBy(df.vault_id, window(df.date, "30 days", "1 day"), df.model)`:
   The `groupBy` operation groups the data by "vault_id", a 30-day window based on the "date" column with a sliding interval of 1 day, and "model". It essentially forms groups of data based on these criteria for further aggregation.
3. `agg(_sum("failure").alias("total_failures"))`:
   The `agg` function performs an aggregation operation on each group created by the `groupBy` clause. In this case, it calculates the sum of the "failure" column for each group and names the resulting column as "total_failures".
4. `selectedData = windowedData.select("window.start", "vault_id", "total_failures")`:
   The `select` operation is used to choose specific columns from the `windowedData`. It selects the start time of the window, "vault_id", and the aggregated "total_failures" column.
5. `filteredData = selectedData.filter(selectedData.total_failures > 0)`:
   Finally, the `filter` operation is applied to keep only the rows where the "total_failures" column is greater than 0. This step is essentially removing records where there were no failures.

Start Time	Vault ID	Total Failures
2023-03-31 02:00:00	1149	1
2023-06-05 02:00:00	1042	1
2023-06-01 02:00:00	1097	1
2023-05-24 02:00:00	1042	1
2023-06-17 02:00:00	1406	1
2023-03-13 01:00:00	1124	1
2023-06-06 02:00:00	1406	1
2023-06-02 02:00:00	1406	1
2023-05-31 02:00:00	1042	1
2023-04-09 02:00:00	1149	1
2023-06-11 02:00:00	1042	1
2023-05-27 02:00:00	1406	1
2023-06-01 02:00:00	1406	1
...	...	...
2023-03-20 01:00:00	1124	1

TABLE 1. Results for Query 1

Applied on our CSV file, we obtain the following result as live output in Table 1.

And included as a function in our benchmark on the platform, we score the following result in Figure 1:

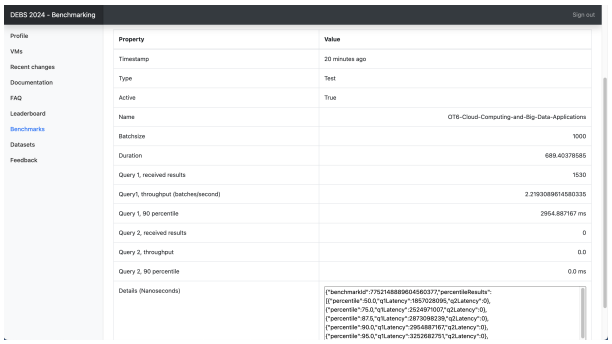

FIGURE 1. Benchmark platform.

2.3. Second Query

The second query is elaborately structured to achieve several key objectives. For each daily reading at day i belonging to a drive d within vault v, the query accumulates a value denoted as NF_v^i for that specific reading. Following this, a crucial step involves re-scaling and normalizing the SMART values to fit within their specified ranges from the provided CSV file. The dynamic K-means clustering process unfolds in sequential stages, beginning with the assignment of incoming readings to the nearest centroids. As each day concludes, marked by a "day_end" flag in the batch, the centroids' positions are updated. This update involves computing the average coordinates of all readings currently associated with a particular centroid. The initial coordinates of the centroids are provided in a CSV file. Each batch of input data is characterized by a list of cluster identifiers ranging from 0 to 49. The ultimate goal is to return the count of drives associated with each of these clusters.

Our approach to executing this query encompasses initial cleaning steps, such as dropping the "c35" column from "clusters.csv," utilizing a data stream sourced from a folder containing CSV files, batching by day due to the absence of day_end flags.

To tackle this query, we initiate the process with essential cleaning steps, including the removal of the "c35" column from the "clusters.csv" dataset. We leverage a data stream sourced from a folder containing CSV files and batch by day due to the absence of the "day_end" flags.

Listing 3. Your Python Code

```
# Your Python code here
def min_max_normalization(df, column,
    ↪ norm_data, min_val, max_val):
```

To normalize the data within specified ranges – as implemented in Listing 3, we use min-max scaling:

$$x_{\text{normalized}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

This renders the data normalized in a range of 0 to 1 – meaning we need to scale it to the desired range with data from the `norm.csv`. We do so by multiplying with the difference from the provided upper and lower bounds and adding the min:

$$x_{\text{scaled}} = (x_{\text{upper}} - x_{\text{lower}}) \times x_{\text{normalized}} + x_{\text{lower}}$$

This renders a stream of normalized data in the same format as our incoming data. The next step is to implement a custom K-Means model initialized by the provided centeroids – meaning we already know our $k = 50$ and we only need to allocate the lowest euclidean distance:

1. **Step 1:** Perform the cross product of the data stream and center points, generating combinations of all incoming data points (ID: timestamp) and the current centroids (ID: label).
2. **Step 2:** Calculate the distance between all vectors using PySpark's `Vectors.squared_distance()`.
3. **Step 3:** Aggregate by selecting the minimum distance for all available non-identical pairings for each data stream ID (timestamp).
4. **Step 4:** Assign a dedicated label for the returned minimum distance.
5. **Step 5:** Count the number of drives for each label.
6. **Step 6:** Calculate the average vector for each label and overwrite the centroids.

Although this approach makes sense from a theoretical point of view, its practical efficiency is limited. Additionally, we encountered significant challenges in managing and consolidating various aggregations, ultimately resulting in a non-functional implementation. Attempts with PySpark's out-of-the-box solutions proved unsuitable for meeting query 2 of the DEBS grand challenge.

3. CONCLUSION

While PySpark offered a valuable learning experience, it didn't emerge as the ideal solution for streaming data. Although the initial query was straightforward, the processing speed didn't quite meet our expectations. The integration into the benchmarking platform presented challenges and the process proved to be quite time-consuming. And due the very recent launch of access and the tight deadline, we had only a few days left to complete it. The second query, based on the theoretical approach discussed earlier, posed implementation challenges, partly due to limited time for a deeper exploration of PySpark functionalities. Despite these challenges, both of us intend to keep using PySpark in future projects, recognizing its learning curve and versatile applications.