# Real-time Analysis of Disk Monitoring Events with Apache Flink

EMANUELE VALZANO, 0341634, University of Rome Tor Vergata, Italy

This paper aims to present the system architecture, strategies, and evaluation of key metrics of a real-time analysis performed on hard disks telemetry data.

## 1 Introduction

The goal of this project is to answer specific queries related to real-time telemetry data events from approximately 200,000 hard disks in data centers managed by Backblaze. This is accomplished using the Apache Flink framework for stream processing. To enable stream processing, real-time event generation from the original dataset is emulated. The dataset comprises S.M.A.R.T. (Self-Monitoring, Analysis, and Reporting Technology) monitoring data, enhanced with additional attributes collected by Backblaze. Each event in the dataset represents the S.M.A.R.T. status of an individual hard disk on a given day. For this analysis, a reduced dataset of approximately 3 million events is used, down from the original 5 million events.

## 2 System Architecture

The system architecture for this project has been emulated on a single node using Docker Compose.

- **Apache Kafka**: Apache Kafka is utilized to store the monitoring events generated by a Producer and eventually preprocessed data by the Faust-Preprocessor (see later). These data streams are produced in designated Kafka topics and are subsequently consumed by Flink jobs for stream processing. Furthermore, Kafka serves as the message queue for query results. Flink sinks the results into separate topics, categorized by query type and time interval, thereby facilitating an effective data retrieval.
- **Stream Emulator Producer**: This component is responsible for generating events in real-time, emulating the original dataset event generation, thus replying the dataset. The stream emulator simply reads the CSV file from a local source, iterates through the CSV rows, extracting the date field and properly scaling down the interval between events (to allow a faster reply), sleeping for such interval. Therefore events are sequentially produced in a Kafka topic during the emulator iteration. To trigger the execution of all windows in Flink, the emulator produces an artificial tuple at the end of the production, with an event time set to one week later than the last event. The values assigned to the fields of such tuple are designed to ensure that the tuple passes all filters, guaranteeing the reach of windows. Note that this tuple will not influence the query results, as the window to which it is assigned will never be triggered. The stream emulator producer can operate in one of the following modes:
  - **Fast Mode**: In this mode, there is a scaling factor of 36,000, meaning that one hour is scaled down into 0.1 seconds. Events occurring on the same day are produced in bursts, with flushing occurring every 0.5 seconds to prevent the producer queue from becoming overloaded.
  - **Non-Fast Mode**: This mode provides a more accurate representation of real-time data. Here, the scaling factor is 3,600, so one hour is represented as one second. Events occurring on the same day are spaced out with a random uniformly distributed interval between 0 and 0.1 seconds, and flushing happens every five seconds.

Author's Contact Information: Emanuele Valzano, 0341634, emanuele.valzano@students.uniroma2.eu, University of Rome Tor Vergata, Rome, Italy.

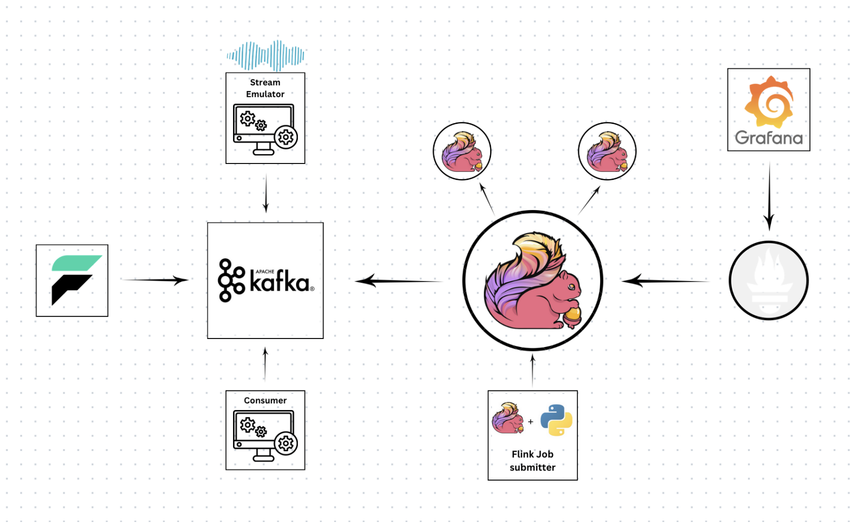

Fig. 1. System Architecture

- **Faust-Preprocessor**: The Faust-Preprocessor is a Kafka consumer and producer that facilitates Faust, a python-based stream processing library that exploits async/await Python constructs. It preprocesses the tuples received from the producer. It then writes the preprocessed data to another Kafka topic, making it ready for Flink to process without additional preprocessing. This component may be inactive during deployment, in such case Flink handles preprocessing directly. The Faust-Preprocessor adds a level of indirection, therefore it decouples the system components, however the trade-off is that it introduces additional latency.
- **Flink**: Flink is the core component of the architecture, consisting of a job manager and task manager. It is responsible for processing queries in real-time by reading data from a Kafka topic. Flink is provided with a Kafka connector, responsible for running both a Kafka consumer and producer (used to sink query results). When the Faust-Preprocessor is inactive, Flink also performs the necessary preprocessing tasks on the consumed tuples.
- **Flink Job Submitter**: This component is responsible of submitting jobs to the Flink Job Manager.
- **Results Consumer**: The consumer component is responsible for consuming query results in real-time, produced by Flink, and writing these results into a CSV file.
- **Prometheus**: Prometheus is used for monitoring the metrics exposed by the Flink metrics system. It particularly focuses on two key metrics: throughput and latency during stream processing. Flink is configured with a metrics reporter, which Prometheus queries periodically to collect the relevant performance metrics.
- **Grafana**: Grafana is utilized to visualize the metrics collected by Prometheus. It provides graphical representations, making the analysis of the metrics easier for the user.

## 3 Preprocessing

Preprocessing operations are either performed by the Faust-Preprocessor or Flink, based on the deployment configuration. Although the framework differs between the two configurations, the operations performed are identical. Upon receiving tuples from the Kafka *disk-monitoring* topic, the following operations are applied:

- Selection of fields relevant to the implemented queries. *s9_power_on_hours* has not been selected since only the first two queries have been implemented.The fields selected are the following:
  - **date**
  - **serial_number**
  - **model**
  - **failure**
  - **vault_id**
  - **s194_temperature_celsius**
- Filtering out tuples where the `failure`, `vault_id`, or `s194_temperature_celsius` fields are set to `NULL`. The first two are essential due to the nature of the data, while tuples with `s194_temperature_celsius` set to `NULL` have been discarded because it was observed that nearly all such tuples also had all other S.M.A.R.T. attributes set to `NULL`. Filtering out these events helps to improve the overall accuracy of the queries.
- Filtering of tuples with models and serial numbers that do not match the following regular expressions:
  - Serial Number Regex:
    `^[A-Z0-9_-]+$`
  - Model Regex:
    `^[A-Za-z0-9 _.-]+$`
    This regex is similar to the previous one but less strict, allowing lowercase characters and additional special characters, especially whitespace.

  All tuples satisfied the model's regex. An interesting result was found for serial numbers: tuples that did not meet the serial number regex (because all characters were lowercase) had all S.M.A.R.T. based attributes set to NULL, with only the date, serial_number, model, failure, and vault_id fields different from NULL. These tuples, following the same lowercase pattern for serial numbers and lacking relevant monitoring data, are treated as invalid.

Since event fields are all of the string type when consumed from Kafka (the producer writes JSON based streams), to ensure type safety, event fields were type casted to the proper data type. Values of the *date* field have been converted to UNIX timestamps and then multiplied by 1000, for Flink timestamps and watermarks. On the Faust-Preprocessor, these operations where implemented using the *faust* library, designed for stream-based applications, leveraging the python async/await constructs. On the other hand, in Flink the transformations on the data stream have been applied using the Flink DataStream API through the python based SDK.

## 4 Processing

Query processing has been carried out using Flink, leveraging the DataStream API, which enables the application of transformations on the incoming data stream. Following the stream preprocessing (executed by either Flink or the Faust-Preprocessor) and prior to the query execution, timestamps and watermarks were assigned to the incoming stream, to allow Flink to properly handle the stream, especially out-of-order events. The watermark strategy used is *'WatermarkStrategy.for_monotonous_timestamps()'*, which assumes strictly increasing timestamps, with a custom timestamp assigner that simply extracts the timestamp field from the preprocessed datastream, based on the event date field. Once the datastream has been prepared, the actual query processing starts. Finally, after processing, the resulting datastream is sinked to a Kafka topic through a Producer.

Given the specifications, both queries have been computed on the following timing windows:

- 1 Day
- 3 Days
- From the beginning of the dataset

To enable aggregation based computation and trigger windows, stream elements have been assigned using a **tumbling window** assigner, that assigns each element to a window of a specified window size. Tumbling windows have a fixed size and do not overlap. This allows the splitting of the stream into “buckets” of finite size, over which we can apply computations.

Query execution involves iterating over the three different window assigners and executing the queries accordingly. To optimize the process and avoid redundant computations, a partial execution is performed beforehand. This involves computing a partial data stream that can be reused in each iteration over the window assigners.

### 4.1 Query 1

The pre-window operation stream computation involves the following steps:

1. Filter the data stream to retain tuples where the *vault_id* field is between 1000 and 1020.
2. Select the relevant fields (*vault_id* and *temperature*) from each filtered tuple using a map transformation.
3. Key the data stream by the *vault_id*, creating a keyed stream that will be used for the subsequent keyed window assignment.

For each window assigner, the following steps are performed:

1. Assign the window of interest to the data stream. using the *window()* operation.
2. Apply an *Aggregate Function* to compute statistical measures (count, mean, M2) using the Welford algorithm[1]. Here, count represents the number of events, and mean and M2 represent the mean value and M2 value of the temperature, respectively.
3. Use the *parallel_variance()* function for merging, as described in the references.
4. After accumulating the above statistics, compute the standard deviation by first calculating the sample variance and then the standard deviation using a *ProcessWindowFunction*, that also produces the final window-based result, which includes tuples containing the start timestamp of the window, the *vault_id*, the count of events, the mean temperature, and the calculated standard deviation.

### 4.2 Query 2

Before the query iterations over the query assigners, a partial datastream has been computed through the following phases:

1. The data stream is filtered, keeping only records where the failure field is True.
2. Filtered records are then mapped to the tuple (*vault_id*, 1, [(*model*, *serial_number*)]), where 1 represents a failure of that vault during a day and then the last element is a list containing one tuple, that holds the model and serial number of the hard disk that failed within the vault.
3. The data stream is keyed by *vault_id* and assigned a tumbling event-time window of one day. This step allows the accumulation of failure counts and hard disks associated to the faulty vault in a single day.

[1] Welford’s online algorithm

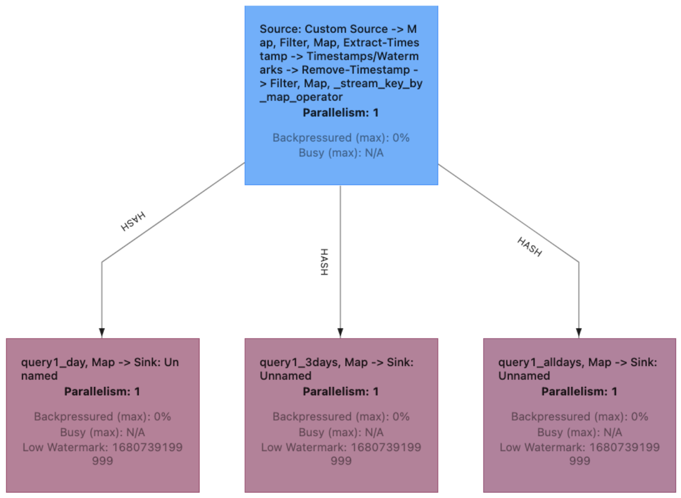
Source: Custom Source -> Map, Filter, Map, Extract-Timestamp -> Timestamps/Watermarks -> Remove-Timestamp -> Filter, Map, _stream_key_by_map_operator
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
HASH
HASH
HASH
query1_day, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
Low Watermark: 1680739199999
query1_3days, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
Low Watermark: 1680739199999
query1_alldays, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
Low Watermark: 1680739199999

Fig. 2. Query 1

4. A reduce function is applied to the window of one day, by means of accumulating the number of failures per vault and the list of (model, serial_number) tuples, within each day. Duplicate pairs of the hard disks list are avoided using the function set(). The window is triggered once a watermark associated to the next window arrives, thus triggering the reduce computation that outputs a datastream with a single element per vault, within the associated window.

At this point, for each window assigner associated to the time interval of interest the following is performed:

1. In order to group the datastream from the previous window, the query involves assigning an additional window (using the *window_all()* operation, which is applied on a non-keyed datastream), for aggregating vault daily failures across all vaults and computing the top 10 ranking.
2. The *AggregateFunction* uses an accumulator list to hold the top 10 vaults that experienced the most number of failures in a day, reporting the vault, failures count and list of models and serial numbers. To handle multiple tuples associated to the same vault within different days (for the three days and all days windows), we leveraged the binary search algorithm to find the index (if exits) at which a vault failure has been stored. In this way for incoming tuples associated to the same vault, we only keep the one with the maximum failure count. Once elements are appended or substituted, we perform a sorting operation on the accumulator list. For the merging operation, we compute a new list with the updated ranking by first adding (without order at first), one by one, elements of the concatenated list and using a dictionary to store the index at which an element has been stored, so that we can handle two elements associated to the same vault within the two different lists and take the maximum of the two. Lastly, we perform the sorting on the resulting list.
3. Finally, once the ranking has been accumulated, a *ProcessWindowFunction* is used to just output the final windowed result, consisting of a tuple containing the start timestamp of the window and the ranking.

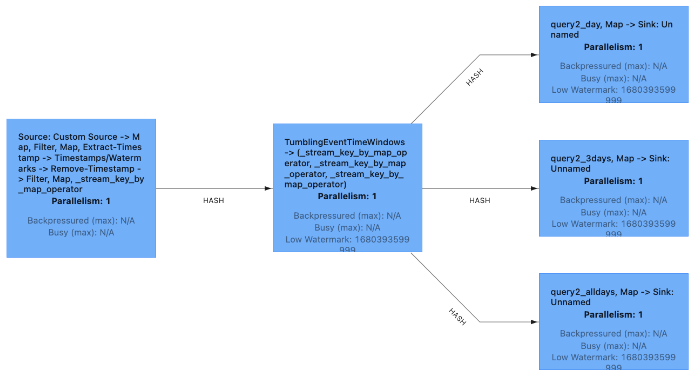
Source: Custom Source -> Map, Filter, Map, Extract-Timestamp -> Timestamps/Watermarks -> Remove-Timestamp -> Filter, Map, _stream_key_by_map_operator
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
HASH
TumblingEventTimeWindows -> (_stream_key_by_map_operator, _stream_key_by_map_operator, _stream_key_by_map_operator)
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
Low Watermark: 1680393599999
HASH
query2_day, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
Low Watermark: 1680393599999
HASH
query2_3days, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
Low Watermark: 1680393599999
HASH
query2_alldays, Map -> Sink: Unnamed
Parallelism: 1
Backpressured (max): N/A
Busy (max): N/A
Low Watermark: 1680393599999

Fig. 3. Query 2

The implementation is simple and robust, however it has a limitation; the second window is not triggered as soon as it receives the data associated with the last day of that window, this is due to the nature of window based processing in Flink that utilizes watermarks. Indeed, the second window must wait a watermark of the following window, but that watermark arrives with a delay of one day, due to the presence of the previous one day window, that does no produce streams continuously, but only once per day.

## 5 Metrics Evaluation

The simulations were executed on a single node using Docker Compose. The Flink configuration used for the simulations is as follows:

- Parallelism: 1
- Number of Task Managers: 1
- Number of Task Slots: 10

The evaluations were conducted using two key metrics, leveraging the Flink metrics system for both.

The key metrics evaluated were:

- **Throughput**:
  To measure throughput, it has been leveraged a Flink's metric suggested by the documentation[2]. The metric is **numRecordsOutPerSecond** within the scope of an operator. This metric represents the number of records an operator emits per second. It has been monitored the mean records out per second computed on all operators within each job, where each job handles the execution associated with a different time slot. However, for this metric, there are no expectations to have significant differences between the different timing windows of the same query. This can be attributed to two main reasons:
  - The operators applied are essentially the same.
  - The number of values measured for window based operators are negligible with respect to the operators that operate on the stream.

[2] Monitoring Apache Flink Applications

- **Latency**:
  To measure latency, it has been used a Flink's metric, by enabling the Latency Tracking feature. However, using the Flink's metric for latency raised the following issues:
  - It is a histogram-type metric, and as such, it only provides quantiles. To estimate an upper bound, we decided to consider the 0.999 quantile, viewing it as a good approximation of the maximum latency.
  - The latency values are not provided solely end-to-end but for each step through all the operators in the query. These operators are identified by their ID, which is a hash computed by Flink at startup, making it difficult for a user to identify when requesting the value. This led to the inability to specifically identify the Sink among the various operators. So, if an operator is between the source and the Sink, the latency between the source and the operator is necessarily less than the latency between the source and the Sink. Therefore, it was deemed appropriate to select the maximum value among the returned latencies. Latency tracking, leverages a latency marker, an artificial tuple, that does not interfere with application specific records, that navigates through the Flink's topology, identifying bottlenecks because it stops at operators that are overloaded.

  In conclusion, the statistic used for latency is the maximum of the 0.999 quantiles of the latencies of the various operators.

The metrics evaluation has taken place with the following execution environment:

- Each query was executed individually.
- For each query, executions corresponding to different time slots were handled by separate jobs.
- Especially to evaluate the throughput, the stream emulator was executed using both the fast configuration and the more realistic one to observe the impact of the arrival rate.

## 5.1 Throughput Analysis

As expected, the throughput measurements led to basically the same results independently of the time slot, therefore an analysis is performed simply based on the two queries, without making a distinction between the different windows.

- **Query1**

  - **High arrival rate:** the effect of having a really high arrival rate (especially within one day) is evident; in fact the metric increases linearly as records arrive and finds an upper bound at approximately 9000 *numRecordsOut / s*, and remains as such until the end of the query execution, this is because a task executor is never in an idle state and always has tuples to emit (for most of the initial operators that work over the continuous stream) since arrivals happen in bursts and are queued to operators. The utilization of the task manager is consequentially quite high since the metric finds a clear upper bound.
  - **Uniformly distributed arrivals:** When tuples within each day are interleaved based on a random uniform distribution, the results change substantially. In fact there is no upper bound and the values oscillates between 10.8 and 11.8 *numRecordsOut/s*. So, given the arrival rate, the system is able to process out records nearly at the same rate at which these arrive (not completely the same because some records get filtered out by operators), indicating a low utilization. This is also proved from the previous results, in fact the system processed out records almost at one order of magnitude faster.

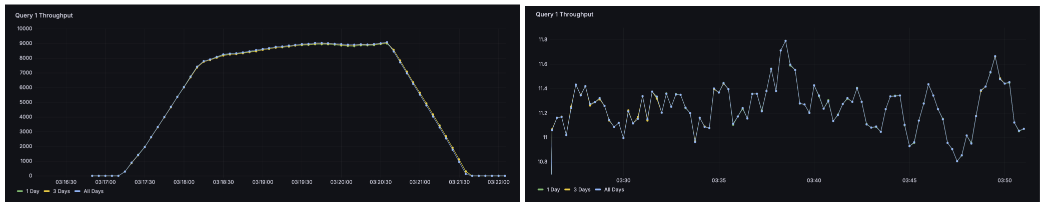

Fig. 4. Query 1 Throughput

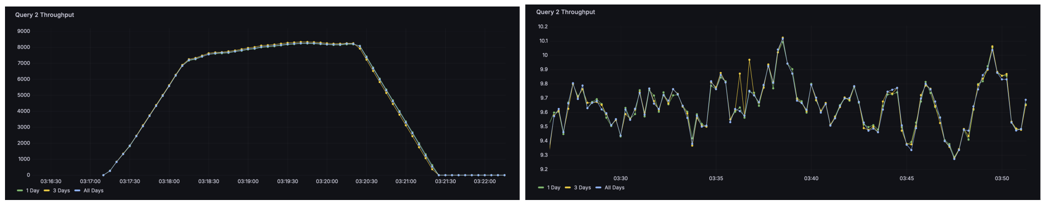

Fig. 5. Query 2 Throughput

- **Query2**
  Query 2 has the same behaviour of the last query in terms of comparison between the two production modes, and this is of course due to the same reasons. However, there is a slight difference between the previous query in terms of the metric evaluation.
  - **High arrival rate:** The upper bound of the throughput is slightly lower than query one, this is probably attributed to the sorting operation during the vault failures ranking accumulation.
  - **Uniformly distributed arrivals:** Given a more realistic arrival rate, the metric oscillates between 9.3, and 10.1 *numRecordsOut/s*, and with respect to query 1 is slightly inferior. This is probably attributed to the filtering out of more tuples in some operators.

## 5.2 Latency Analysis

The analysis was performed only for the configuration with high arrival rates, to stress the system and evaluate maximum latencies among operators to eventually identify a possible bottleneck in such conditions. In general, even if it could seem the opposite, is possible to note how latencies of larger time windows tends to be lower: this is based on how the Flink's LatencyTracker works; it can "bypass" the windows and operators, unless these operators have queued tuples; when the window is made up of large time windows, a lot of time will be spent waiting for the accumulation of tuples in the windows, and the trackers can move ahead without issues.

- **Query 1:**
  Overall, given the high arrival rate and the maximum latencies, the query performs well, although there are some latency spikes, particularly for the first two smaller windows. These spikes can likely be attributed to bursts of tuples within the same day, which overload the operators associated with the smaller time frame windows. As soon as these windows are disposed, they are immediately stressed again to process a new window, causing an overload on the operators. In fact, this is observed initially for the 1-day window and subsequently for the 3-day window. Additionally, a smaller spike is noticeable in the last window, probably due to its execution

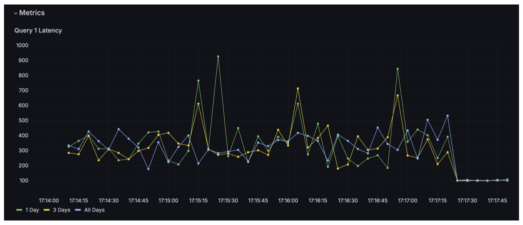

Fig. 6. Query 1 Latency

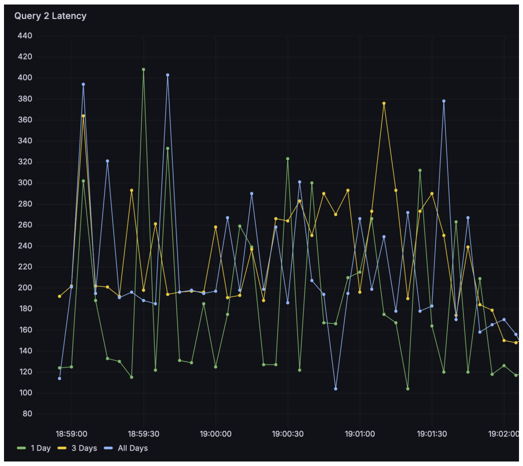

Fig. 7. Query 2 Latency

and slight queuing within its operators, causing latency trackers to stop. Additionally, the fact that operators are not parallelized within multiple tasks, and that the system resources comes from a single machine, certainly impacts such metric.

- **Query 2:**
  For Query 2, the spikes in latency are not as pronounced as in Query 1, likely due to the nature of the query, which filters out all vaults that did not report a failure, thus reducing the load on subsequent operators. In this case, the general fact that larger windows exhibit lower latency is not observed because all time slots initially rely on a 1-day window assigner to group all vault failures within a day. This makes the latencies appear more independent of the window size.