Systems and Architectures for Big Data - A.Y. 2023/24 Project 2: Real-time Analysis of Disk Monitoring Events with Apache Flink

Giulio Appetito
University of Rome Tor Vergata
Rome, Italy
0321669

***Abstract*—The aim of the document is to illustrate the ideas behind the development of a stream processing pipeline and the reasoning on some architectural and analytical aspects.**

I. Introduction

This project analyzes telemetry data from around 200,000 hard disks managed by Backblaze using Apache Flink for stream processing. A reduced version of the dataset from the ACM DEBS 2024 Grand Challenge is used. The dataset contains approximately 3 million events, each reporting the S.M.A.R.T. status of a specific hard disk on a given day. The project simulates data flow with accelerated timelines, maintaining coherence between time intervals. This report outlines the system architecture, implementation, and performance evaluation.

II. System architecture

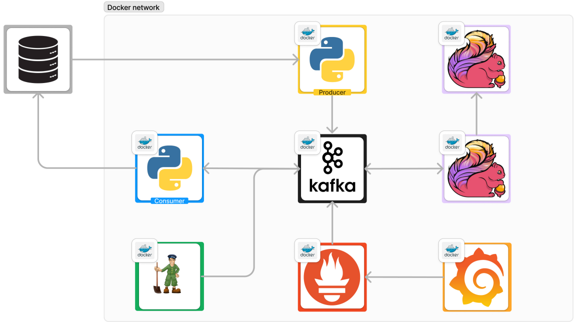

Fig. 1. System architecture

The system architecture is deployed on a single node using Docker Compose. A bridge network is adopted, in order to facilitate the communication between the various containers. The architecture is composed by the following main components:

- ***Producer:*** This node replaces actual devices who would produce the events, by replaying the original dataset on a Kafka topic. The producer is coded in Python, and it exploits *kafka-python* library in order to communicate with Kafka.
- ***Apache Kafka:*** Publish/subscribe platform Apache kafka was used for the Messaging layer. The records are written on a topic (*hdd_events*) that is automatically created when the first tuple is attempted. Similarly, Kafka was used for writing query results, creating a different topic for each query and for each time window considered in the processing.
- ***Apache Zookeeper:*** Zookeeper is essential for Apache Kafka's operation. Zookeeper handles controller election, ensuring there is always one active controller to manage partition leaders and handle node failures. It also maintains topic configurations, manages access control lists (ACLs) for security, and monitors the status of active brokers in the cluster for efficient coordination and management of Kafka.
- ***Apache Flink:*** Apache Flink was used as a framework for stream data processing. The code of Query execution was written in python using python's pyflink library. Being the kafka connector not provided in the library, the jar is downloaded from the web using the wget command when creating the image container via Dockerfile, and this jar is added to the env when the Flink execution environment is created. The system architecture comprises a single TaskManager and a single JobManager.
- ***Consumer:*** Used to read query results from Kafka and save them in a csv file. The consumer registers on the Kafka topics corresponding to the individual queries and window results: when a tuple is inserted the consumer receives it and the saves to a query-and-window-specific csv file with the same name as the source topic. The consumer is also coded in Python, using the *kafka-python* library to communicate with Kafka.
- ***Prometheus:*** Prometheus collects and stores metrics about throughput and latency, allowing detailed analysis and querying.
- ***Grafana:*** Grafana provides a visual interface to display the data collected from Prometheus, making it easier to track system performance.

The decision to introduce an additional consumer node, rather than having Flink write the query results directly to files, was driven by the desire to better distribute responsibilities among the system components and decouple them as much as possible. In this way, Kafka serves as an intermediary element that connects the various components of the system, enabling greater modularity and flexibility. This architecture facilitates scalability and maintenance, as each component can be developed, deployed, and updated independently of the others. Additionally, using Kafka as an intermediary allows for more robust and reliable data management between Flink and the consumer node.

III. Dataset replay

The *Producer* node is responsible for the dataset replay: it reads a local file containing the dataset, and outputs the records in temporal order, according to the reported timestamp. It simulates the data flow by appropriately accelerating the time scale between the tuples, preserving consistency between time intervals. The scaling factor variable is contained in the $.env$ file, and is set to default to $3600 * 24$; by doing so, the producer sleeps for a second between each day's tuples, and emits the tuples within a day in a single burst. This is due to the dataset not reporting the actual time of the events, but just the day, so one cannot infer the time interval between tuples within a single day; a solution to this is waiting for a randomized time between each tuple (in the code, a random uniform time between (0, 0.05)s); although this can be more more compliant with a realistic scenario, it can be avoided for the sake of reducing execution time just by passing the parameter False to the Producer script.

Producer node filters only the required fields within the record:

- date
- serial_number
- model
- failure
- vault_id
- s9_power_on_hours
- s194_temperature_celsius

This node does a minimal preprocessing on the data by converting each *csv* row from the file to a JSON object before publishing the record on Kafka; after this transformation, it sends these produced tuples to Kafka on a topic called hdd_events, in order for the consumers (Flink jobs) to read from it.

In order to trigger the last windows in the queries, producer node outputs a last dummy record, whose date is set to seven days after the last day (timestamp $2023-04-30T00:00:00.000000$).

IV. Queries

For the queries implementation, DataStream Flink API in Python was used. This section aims at explaining how the queries were implemented.

A. Query 1

Q1. *For vaults (vault id field) with an identifier between 1000 and 1020, calculate the number of events, the average value and the standard deviation of the temperature measured on its hard disks (field temperature s194 Celsius). Pay attention to the possible presence of events that do not have assigned a value for the temperature field. For calculating the deviation standard, an online algorithm is used, such as the Welford algorithm. Calculate the time windows query:*

- *1 day (event time)*
- *3 days (event time)*
- *from the beginning of the dataset*

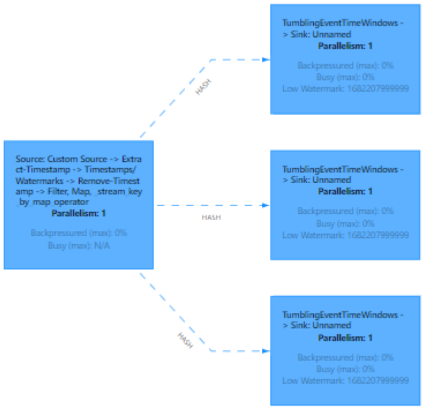
Source: Custom Source -> Extract-Timestamp -> Timestamps/Watermarks -> Remove-Timestamp -> Filter, Map, _stream_key_by_map_operator
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
HASH
HASH
HASH
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999

Fig. 2. Query 1 DAG

The implementation of query 1 followed these steps:

1) ***Source.*** The source stream is obtained from a *KafkaConsumer*, that reads the tuples within the *hdd_events* topic on Kafka, the one the producer writes on.
2) ***Timestamp and watermark strategy assignment.*** A custom TimestampAssigner was used, which converts the *date* field of the records to milliseconds, in order for Flink to use it, and the watermark strategy was set to *for_monotonous_timestamps()*. This choice was due to records being emitted in temporal order based on timestamps. This strategy assumes that timestamps increase monotonically, meaning each subsequent record has a timestamp equal to or greater than the previous one.
3) ***Filter.*** The records are filtered to retain only the ones with *vault_id* between 1000 and 1020.
4) ***Map.*** A map function is applied to select only the required fields.
5) ***Key_By***. A *key_by* is applied in order to key by *vault_id* field, obtaining a keyed stream.
6) ***Windows.*** On each keyed stream, a *TumblingEventTimeWindows* was applied, varying the width as requested by the query. To consider the range from the beginning of the dataset, a window of 23 days was used.
7) ***AggregateFunction***. An *AggregateFunction* is applied to each window, which computes the count, mean and

standard deviation in real-time as a new tuple arrives. This function exploits the Welford algorithm to do so, implemented by the two functions *update()* and *finalize()*. For a new value, *update()* function computes the new count, new mean, and the new M2 (squared distance from the mean), while *finalize()* function retrieves the mean, variance and sample variance from an aggregate. This is the schema for the AggregateFunction:

- it creates an accumulator in the form (count, mean, M2)
- in the *add* function, it uses the *update()* function in the Welford algorithm to update the accumulator with the incoming value of temperature
- in the *get_result()*, it just returns the computed accumulator
- in the *merge()* function, it merges two accumulators, always by means of the Welford algorithm; spefically, it uses the *Chan's method*, a generalization of the Welford algorithm to combine arbitrary sets.

8) ***ProcessWindowFunction***. A *ProcessWindowFunction* is then applied to each window in order to obtain the final result: it applies the *finalize()* function of the Welford implementation, computes the square root of the variance in order to obtain standard deviation, and adds the timestamp from the beginning of the window, as requested by the query, returning a Row in the form $(ts, vault\_id, count, mean, stddev)$
9) ***Sink.*** The result of the query is then routed to a specific sink for each window lenght, which is a *KafkaProducer* that sends the results rows to a specific Kafka topic in order for the consumer to read them.

B. Query 2

Q2. *Calculate the updated real-time ranking of the 10 vaults that record the highest number of failures on the same day. For each vault, report the number of failures and the model and number serial number of failed hard disks. Calculate the time windows query:*

- *1 day (event time)*
- *3 days (event time)*
- *from the beginning of the dataset*

The implementation of query 1 followed these steps:

1) ***Source.*** The source stream is obtained from a *KafkaConsumer*, that reads the tuples within the *hdd_events* topic on Kafka, the one the producer writes on.
2) ***Timestamp and watermark strategy assignment.*** A custom TimestampAssigner was used, which converts the *date* field of the records to milliseconds, in order for Flink to use it, and the watermark strategy was set to *for_monotonous_timestamps()*. Again, this choice was due to records being emitted in temporal order based on timestamps. This strategy assumes that timestamps increase monotonically, meaning each subsequent record has a timestamp equal to or greater than the previous one.

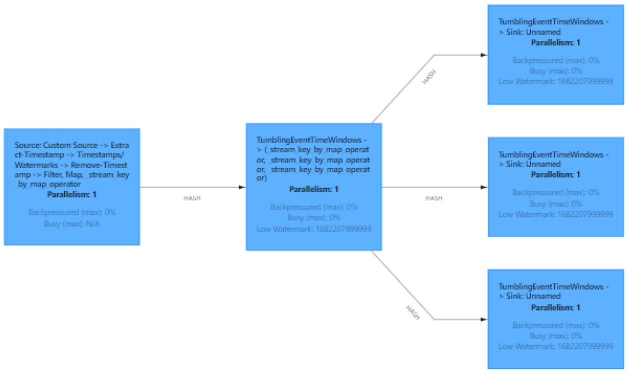
Source: Custom Source -> Extract-Timestamp -> Timestamps/Watermarks -> Remove-Timestamp -> Filter, Map, _stream_key_by_map_operator
Parallelism: 1
Backpressured (max): 0%
Busy (max): N/A
HASH
TumblingEventTimeWindows -> (_stream_key_by_map_operator, _stream_key_by_map_operator, _stream_key_by_map_operator)
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999
HASH
HASH
HASH
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999
TumblingEventTimeWindows -> Sink: Unnamed
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1682207999999

Fig. 3. Query 2 DAG

3) ***Filter.*** The records are filtered to retain only the ones with *failure* set to $True$ (so, the ones for which there was a failure).
4) ***Map.*** A *Map* is then applied to retain only the necessary fields of each row, obtaining a tuple in the form $(vault\_id, 1, [model, serial\_number])$, where 1 stands for a single failure.
5) ***key_by.*** A *key_by* is applied to the stream in order to key by vault_id, in order to compute the number of failures and the list of models and serial numbers within each day.
6) ***Window.*** A *TumblingEventTimeWindows* of 1 day is then applied to the stream, to compute the above stated values.
7) ***ReduceFunction.*** A *ReduceFunction* is applied within each window(*VaultFailuresPerDayReduceFunction*). This function receives two tuples, and sums up the failures_count values, concatenates the lists of failed_disks and returns a tuple in the form $(vault\_id, failures\_count, failed\_disks)$.
8) ***Window.*** Then, a *TumblingEventTimeWindows*, respectively of lenght 1,3 and 23 days (as requested by the query) are applied to the stream through a *window_all* invocation, in order to consider all the keyed streams to compute the rankings.
9) ***AggregateFunction.*** Within each window, an *AggregateFunction* (*FailuresAggregateFunction*) is applied in order to compute the ranking by failures count. This function:
   - Creates an accumulator in the form of a list (*create_accumulator*)
   - At each incoming tuple in the window, in the *add* function, it checks if a record for the incoming vault_id is already stored in the accumulator;
     - if so, it checks if the incoming tuple has a bigger value for the *failures_count* field, and in this case it updates the entry associated with the vault_id;
     - else, it just adds the new entry if the vault is not already stored.

  After adding the incoming tuple, it sorts the accumulator *ranking* by *failures_count* and returns just the first 10 elements.
  The choice to retain only the maximum by failures count for each vault_id was to avoid having possible duplicate entries for the same vault_id in the ranking.
  - In the *merge* function, it combines two different accumulators (*ranking*), and again when doing so, it keeps just one entry for each vault, the one with the maximum failure count, by using a dictionary. Afterwards, it converts the dictionary back to a list, sorts it and returns the first ten elements.
  - In the *get_result* function, then, it just returns the accumulator (i.e., the ranking).

10) ***ProcessAllWindowFunction.*** Finally, a ProcessAllWindowFunction is applied, in order to apply the timestamp from the beginning of the window, by returning a tuple in the form $(window.start, ranking)$, since it had access to context information.
11) ***Map.*** Finally, a *map* is applied to the result in order to convert the tuples to rows, in order to be compliant with the serialization schema.
12) ***Sink***. Finally, the result of the query is routed to a specific sink for each window lenght, with each sink being a KafkaProducer that sends the resulting rows to a specific Kafka topic, in order for the consumer to read them.

V. Results

The simulations have been executed on a single node using *Docker Compose*. Flink was configured with the following parameters:

- *Parallelism* : 1
- *Number of TaskManagers* : 1
- *Number of JobManagers* : 1
- *taskmanager.numberOfTaskSlots*: 6
- *jobmanager.memory.process.size*: 1024m
- *taskmanager.memory.process.size*: 1024m

The node the simulation was executed on has the following features:

- *CPU*: 13th Gen Intel(R) Core(TM) i7-13700H, 2400 Mhz, 14 core, 20 logic processors
- *RAM*: 16gb

The evaluation considered two different metrics:

- ***Throughput***
- ***Latency***

Three distinct jobs were executed, each configured with a different time window: 1 day, 3 days, and 23 days, by means of the *env.execute_async()* function in Flink. The producer node was run with the 'fast' mode (meaning no time between tuples within the same day was considered, so these tuples are sent in a one-day burst, only time between different days is considered), for the sake of the simulation execution time. For this experiment, *Latency Tracking* was activated in Flink, by setting *env.get_config().set_latency_tracking_interval(10).*

A. Latency

The objective was to evaluate how the size of the time window affects the latency of data processing.
To measure latency, the metric `flink_taskmanager_job_latency_source_id _operator_id_operator_subtask_index_latency` was used, which records the latency at the level of individual operators and subtasks within the Flink jobs, given a source_id and an operator_id. So, this metric is not considering actual end-to-end latency, but latency at the operator granularity.
Data was collected using Prometheus and aggregated and visualized with Grafana. The aggregation was done by taking the maximum of the latency over the dimension *job.name*, ensuring that the average latency for each job was represented, and by filtering taking the 0.999 quantile. The choice of taking the maximum is due to the fact that each value reported by the metric measures the latency from the source to a specific operator; so, if there's a record reporting the value of latency from source to sink, it is clearly comprehensive of all of the previous other values (regarding the previous operators), so taking the maximum value can be a quite good estimate of the end-to-end latency of the job, and it is also representative of possible bottlenecks in the DAG.
As stated above, for latency LatencyTracking was activated. LatencyMarkers in Flink measure the time it takes for the markers to travel from each source operator to each downstream operator. As LatencyMarkers bypass user functions in operators, the measured latencies do not reflect the entire end-to-end latency but only a part of it, reflecting queuing delays but not the time spent within operators like window buffers. Also, latency markers can not overtake regular user records, thus if records are queuing up in front of an operator, it will add to the latency tracked by the marker.
This justifies what can be seen in the charts for latency for both jobs Q1 and Q2, expecially in the last section of Q1's chart: we can observe that larger time windows tend to have lower latency. This is because Flink's LatencyTracker measures the travel time of latency markers through the system, bypassing operator functions and primarily reflecting queueing delays. When dealing with larger time windows, operators need to accumulate more tuples before processing them, which can reduce the queueing time per operator unless there is excessive tuple accumulation.

In fact, we see that queries with global and 3-day windows tend to have lower latency compared to the daily windows of the same query. This can be attributed to resource distribution on the single node and the operators preparing the tuples for processing. Moreover, the latency spikes observed can be due to resource contention and high computational load moments.

B. Throughput

The objective was to assess how the size of the time window impacts the throughput of the jobs. Throughput was measured using the metric *flink_taskmanager_job_task_operator_numRecordsOutPerSecond*,

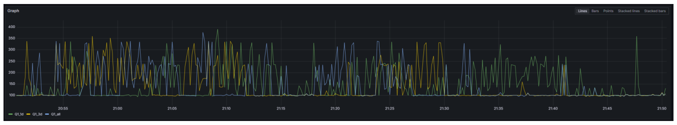

Fig. 4. Latency for Q1 Job

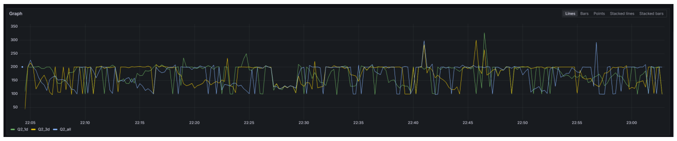

Fig. 5. Latency for Q2 Job

which tracks the number of records processed per second by the task operators within the Flink jobs.

Data was gathered through Prometheus and subsequently aggregated and visualized using Grafana. The aggregation process involved averaging the throughput based on the *job.name*, in order to represent each job's operators average throughput.

Q1. From the plot, it is clear that all three window configurations—1 day, 3 days, and 23 days—show similar throughput trends, stabilizing around 500 records/second after an initial ramp-up period. This initial phase represents the time taken for the system to reach a steady state of processing.

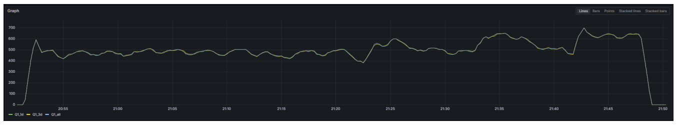

Fig. 6. Throughput for Q1 Job

Q2. An analogous trend is observed for Q2 Job's throughput, whose steady state value is slightly smaller than Q1 Job's one. This is probably due to the increased complexity in Q2 Job's operators, which include sortings and list/dictionary iterations that may cause a reduction in average throughput.

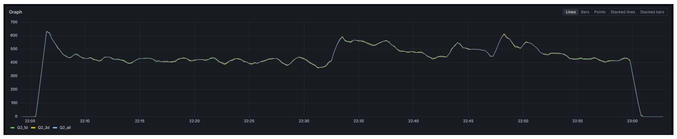

Fig. 7. Throughput for Q2 Job

The impact of window size on throughput appears to be minimal for both queries. Even though the observation that all the three window configurations have quite overlapping mean values for throughput may be counterintuitive (since one may expect the throughput to decrease with larger windows), this is mainly due to the nature of the chosen metric for throughput: in fact, the metric *flink_taskmanager_job_task_operator_numRecordsOutPerSecond* measures the number of records processed per second by individual task operators. So, this metric focuses on the rate at which records are processed by each operator, rather than the overall job throughput. Since operators process records continuously as they arrive, the throughput per operator is not affected by different window sizes.

As for latency, the throughput spikes observed can be due to resource contention and high computational load moments.