Analisi di dati di monitoraggio con Apache Flink

Sistemi e Architetture per Big Data - Progetto 2

Alessandro Lioi
0333693

***Abstract*—Lo scopo di questo documento è quello di descrivere l'architettura della pipeline e l'implementazione di due query in Apache Flink e Spark Structured Streamingsu dati di telemetria di hard disk gestiti da Backblaze**

I. Introduzione

Il progetto consiste nello svolgimento di due query tramite stream processing di dati di telemetria forniti da Backblaze riguardanti circa 200k hard disk.

II. Architettura Pipeline

La pipeline (Figure 1) descrive il seguento percorso dei dati:

1) si esegue uno script in Python che legge il dataset ed esegue il replay scrivendo le tuple su un topic di Kafka `original`
2) si esgue un flow di NiFi che effettua pre-processing dei dati: scrive sul data store Mongo le tuple non valide, e quindi scartate; mentre scrive su un altro topic di Kafka, `filtered`, le tuple da prendere in considerazione per l'analisi
3) si esegue tramite Flink o Spark Structured Streaming un'applicazione in grado di calcolare le query di interesse, prendendo i dati dal topic `filtered` di Kafka
4) si salvano i dati in output sul filesystem locale per quanto riguarda Flink e su Mongo/console per Spark

Ogni componente dell'architettura viene eseguito in un container Docker, usando le immagini ufficiali per ogni servizio, orchestrato tramite Docker Compose.
I container di Flink o Spark vengono eseguiti in maniera differenziata tramite l'uso dei profili.
Per la raccolta delle metriche e la visualizzazione delle performance, si è optato per l'uso di Prometheus e Grafana.

III. Producer e Apache NiFi

A. Producer

Il producer è uno script scritto in Python che inizializza il flow di NiFi, per automatizzare completamente la pipeline, ed esegue il replay del dataset.
L'inizializzazione di NiFi consiste nell'importare e avviare un template definito precedentemente tramite Web UI.
Il replay dell'intero dataset viene eseguito in circa 40 minuti e si è optato per inviare su Kafka gli eventi di una singola giornata in modo tale che questi siano equamente spaziati tra di loro.
Per poter avviare la computazione dell'ultima finestra richiesta (riguardante l'intero dataset), alla fine del replay si invia una tupla che ha come campo *data* un valore successivo all'ultimo giorno di misurazione considerato

B. Apache NiFi

Il flow di NiFi (Figure 2) ha lo scopo di leggere le tuple inviate dal producer di Kafka, filtrare le tuple con un valore di `s194_temperature_celsius` assente, che vengono inviate su MongoDB e inoltrare su un altro topic le tuple valide per il processamento.
Per poter svolgere queste funzioni sono stati utilizzati i seguenti **processor**:

- **ConsumeKafkaRecord**: leggere le tuple dal topic `original` in formato CSV e convertirle in JSON, per facilità di manipolazione su NiFi
- **SplitJson**: dividere l'array di tuple lette dal processor precedente in Flowfile diversi, in modo tale che ogni tupla corrisponda ad un Flowfile differente
- **EvaluateJsonPath**: aggiungere il valore del campo `s194_temperature_celsius` come attributo del Flowfile
- **RouteOnAttribute**: dividere il flusso dei dati in base a se il valore di temperatura è valido o meno
- **PublishKafkaRecord**: pubblicare sul topic `filtered` di Kafka le tuple valide, convertendole nuovamente in CSV
- **PutMongoRecord**: scrivere su MongoDB le tuple non valide, in modo da poterle analizzare successivamente

IV. Apache Flink

Per lo sviluppo dell'applicazione in Apache Flink, si è optato di utilizzare Scala come linguaggio di programmazione in quanto è un linguaggio fortemente tipizzato, al contrario di Python e che permette di utilizzare l'API esposta per Java, poichè viene compilato in un jar che viene eseguito sulla JVM, ma risulta essere meno verboso rispetto a Java.
Il jar viene compilato direttamente dal container del `jobmanager` in una fase di build iniziale.
L'applicazione definisce quindi come **source** dell'input per entrambe le query un `KafkaSource` definito con uno schema di deserializzazione custom che prende dalla tupla in CSV solo i campi necessari per la computazione.
Poichè si devono eseguire delle operazioni su un dataset storico, riprodotto ad una velocità maggiore rispetto alla generazione originaria, è stato necessario creare un sistema di **watermark** custom che assegna i timestamp direttamente dalla tupla, convertendo il campo *date* in millisecondi.

A seguito dell'esecuzione delle query, si ha come **sink** un sink custom che scrive l'output di ogni finestra su un singolo file.

A. Query 1

La query chiede di calcolare per i vault con ID compreso tra 1000 e 1020, il numero di eventi, il valore medio e la deviazione standard della temperatura su 3 finestre temporali (1 giorno, 3 giorno e l'intero dataset).
Per poter rispondere a questa query si esegue:

1) si selezionano, tramite una `filter`, le tuple con `vault_id` in $[1000, 1020]$
2) si crea un `KeyedStream`, tramite la funzione `keyBy` in base al `vault_id`
3) si crea una finestra di tipo **tumbling** di 1, 3 e 23 giorni, con un offset di 0, 2 e 13 giorni; questo è stato necessario poichè Flink crea le finestre a partire dall'epoch (01/01/1970) e per finestre con durata maggiore della giornata, è necessario aggiungere un offset affinchè la finestra cominci nel primo giorno di misurazione
4) si esegue un'aggregazione, con `aggregate`, che implementa l'algoritmo online di Welford per il calcolo del numero di eventi e della media e deviazione standard della temperatura
5) si esegue una `map` che converte l'aggregato ottenuto in una stringa per poterla salvare su file

Il DAG corrispondente a questa query è il seguente: Figure 3

B. Query 2

La query chiede di calcolare una classifica dei 10 vault con il maggior numero di fallimenti e la lista di HDD (identificati dalla tupla `serial_number, model`) sulle 3 finestre precedenti
Per questa query si esegue:

- si filtrano i vault che hanno subito una failure
- come prima si crea un `KeyedStream` in base al `vault_id` e si crea una finestra **tumbling** di 1, 3 e 23 con gli offset precedenti
- si esegue un'aggregazione che calcola il numero di failures e salva in un `Set` la tupla che identifica un HDD
- si raggruppa nuovamente in una finestra, senza chiave della stessa durata della finestra precedente, in modo tale da raggruppare tutti gli aggregati per poter creare la classifica
- si esegue una `process` che permette di avere a disposizione tutte le risultanti dalla computazione precedente
  - si crea quindi la classifica, ordinando i vault per numero di failures decrescente e prendendo i primi 10 elementi
  - si converte la classifica identificata in una stringa, per poter essere salvata su file come CSV

Il DAG corrispondente a questa query è il seguente: Figure 4

V. Spark Structured Streaming

Come per Flink, l'applicazione è scritta in Scala e il jar viene compilato dal container del `spark-master` in una fase di build.

Si utilizza come `readStream` Kafka, leggendo la tupla come stringa e convertendola in una `Row` prendendo solo i campi necessari per entrambe le query e convertendo la stringa *data* in timestamp
Come per il framework precedente, è necessario aggiungere un watermark per poter gestire correttamente le tuple che potrebbero arrivare fuori ordine ma, a differenza di Flink, non potendo assegnare un timestamp, è necessario usare il tempo di riproduzione: in questo caso è stato impostato a 3 minuti che è un tempo poco maggiore rispetto al replay di una giornata (dimensione minima della finestra).
Affinchè il job di Spark venga eseguito in maniera continuativa, è stato necessario aggiungere un trigger, impostato sempre a 3 minuti, sul Dataframe risultante dall'esecuzione delle query.

A. Query 1

- si filtrano le tuple con `vault_id` in $[1000, 1020]$, tramite la funzione `filter`
- si selezionano solo le colonne di interesse, quindi `vault_id` e `s194_temperature_celsius`
- si raggruppa per:
  - finestra in base al timestamp di 1, 3 o 23 giorni con tempo di sliding uguale alla durata della finestra, in modo tale da avere una finestra tumbling e un offset di 0, 2 e 13 come per il caso di Flink
  - `vault_id`
- si esegue quindi una aggregazione, tramite `agg` in cui si calcolano le statistiche necessarie, tramite le funzioni `count`, `avg` e `stddev`
- si selezionano quindi le colonne necessarie, riconvertendo il timestamp nel formato di input

L'output per questa query viene salvata su MongoDB, in modalità *append*, tramite il connettore apposito di Spark.
Il DAG corrispondente a questa query è il seguente: Figure 5

B. Query 2

- si selezionano le tuple con una `failure`
- si selezionano le colonne necessarie per il calcolo della query
- come prima, si raggruppa per la finestra di interesse e il `vault_id`
- si esegue una aggregazione:
  - si conta il numero di `failure`, tramite `count`
  - si crea la lista di HDD, tramite `collect_set` formato da una stringa del tipo `serial_number,model`
  - si ordina per failure decrescenti
  - si limitano i valori a 10, per ottenere la classifica desiderata
  - per poter scrivere l'output come richiesto dalla query:
    - si raggruppa per timestamp
    - si esegue un'aggregazione per mettere insieme i 10 vault della classifica in una lista

    - si esegue infine una selezione per prendere il timestamp e la classifica, in modo tale che sia nell'output richiesto

Il risultato di questa query, a differenza di quella precedente, viene stampato sul terminale in modalità *complete*. Ciò è causato da una limitazione di Spark Structured Streaming che non permette di effettuare un'ordinamento su un Dataframe in real-time se quest'ultimo viene salvato in modalità *append*.
Il DAG corrispondente a questa query è il seguente: Figure 6

VI. Metriche

Per entrambi i framework, le metriche sono state aggregate tramite Prometheus, abilitando i reporter tramite file di configurazione. Prometheus è configurato con due job di tipo *scrape* che vengono eseguiti ogni 10 secondi. Per poter visualizzare in maniera grafica le metriche raccolte, è stato utilizzato Grafana definendo due dashboard, una per framework. Grafana è stato automatizzato interamente tramite il sistema di provisioning (ad eccezione per le metriche riguardanti la seconda query di Flink il cui nome viene rigenerato ogni volta ed è quindi necessario selezionare il nuovo `task name` nei vari pannelli)

A. Flink

La dashboard relativa a Flink mostrano la latenza e il throughput delle due query. Queste due metriche sono state calcolate tramite la definizione di due metriche custom calcolate nel seguente modo:

- latenza:
  - nella fase di deserializzazione della tupla da Kafka, si aggiunge il timestamp corrispondente all'ingestion time nel sistema
  - nelle aggregazioni si prende come timestamp il minimo tra tutti i timestamp
  - prima di entrare nel `sink`, le tuple risultati passano per una `map` *vuota* in cui viene aggiornata la metrica riportata su Prometheus come la differenza tra il tempo attuale e l'ultimo ingestion time dell'aggregazione

  Si ha quindi una *end-to-end latency* che rappresenta il tempo necessario per ottenere l'output a partire dall'ultima tupla di interesse per la finestra
- throughput:
  - nelle aggregazioni si tiene traccia del numero di tuple utilizzate per calcolare le query richieste: per la prima query, si utilizza il valore `count` richiesto; per la seconda query, si aggiunge nelle varie aggregazioni un contatore
  - come per la latenza, prima di entrare nel `sink`, nella `map` *vuota* si aggiorna la metrica con il valore riportato dalla tupla risultante (questa metrica è effettivamente il numero di tuple processate)

I grafici quindi mostrano come varia la latenza nel tempo (il valore viene diviso per 1000, in modo tale da avere i secondi come unità di misura) e il *rate* del throughput in una finestra temporale di 2 minuti (corrispondente ad una giornata; in questo modo si ha il throughput effettivo).

1) Query 1: La latenza (Figure 7) si assesta sotto il secondo: questo risultato è atteso in quanto le operazioni da fare non richiedono una computazione eccessiva.
Per quanto riguarda il throughput (Figure 8), come ci può attendere rispetto a come viene calcolata e aggiornata la metrica, si possono notare i picchi allo scadere delle finestre.

B. Query 2

La latenza (Figure 9) risulta essere molto maggiore rispetto alla prima query. Ciò è dovuto alla necessità di dover eseguire una ulteriore finestra per poter calcolare la classifica.
Per il throughput (Figure 10) si possono fare le stesse considerazioni della prima query.

C. Spark Structured Streaming

Per quanto riguarda Spark sono state utilizzate le metriche messe a disposizione direttamente dal framework.
Per il throughput, è stato utilizzata come metrica il `processingRate`.
Per entrambe le metriche e per entrambe le query, si può facilmente notare come il sistema aggiorni questi valori allo scadere del tempo di *trigger*, corrispondente alla fine dell'esecuzione del microbatch.
Di particolare nota è la necessità di dover attendere, alla fine della riproduzione del dataset, lo scadere dell'ultimo trigger per poter ottenere i risultati. Rispetto a Flink, non si ha quindi una vera esecuzione in real-time ma si accumulano le tuple per poi essere eseguite in micro-batch
Query 1: Latenza 11, Processing Rate 12
Query 2: Latenza 13, Processing Rate 14

VII. Riferimenti e Manuale d'Uso

Il codice sorgente è disponibile su GitHub
La computazzione può essere avviata tramite lo script `./scripts/run.sh` che:

- avvia il cluster con il profilo passato come argomento allo script
- crea il topic `filtered` su Kafka; il topic `original` viene creato automaticamente da NiFi
- avvia la query passata come argomento su Flink o Spark

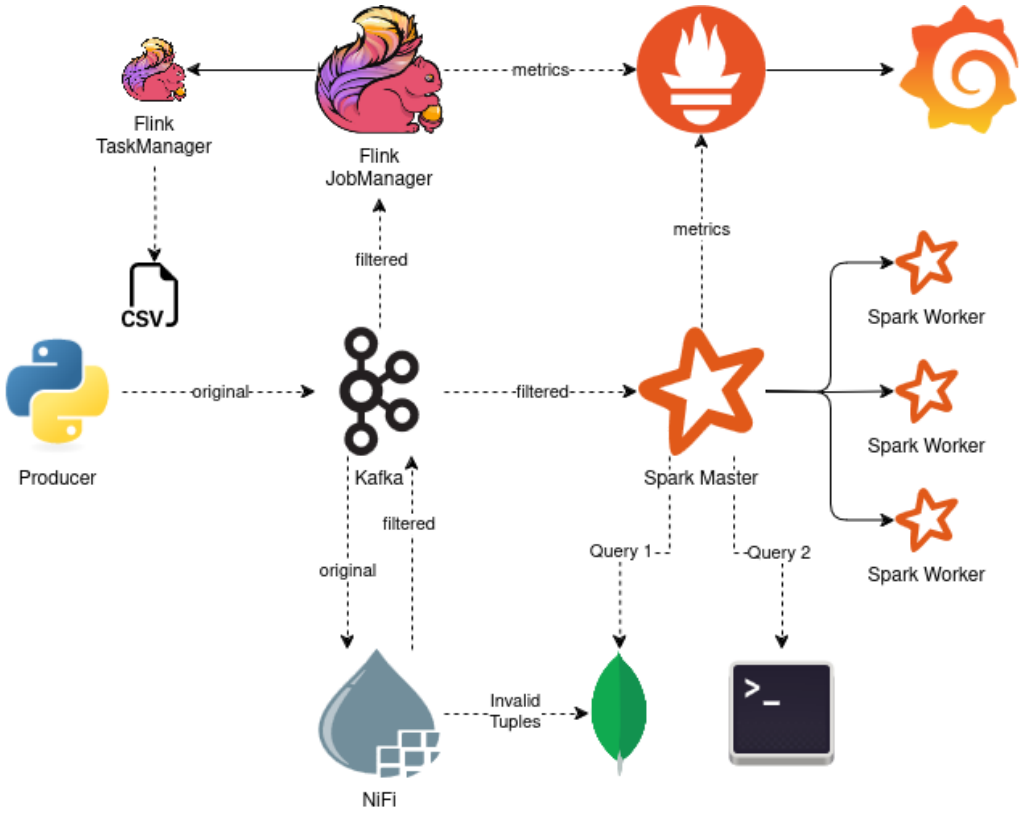

Fig. 1. Architecture

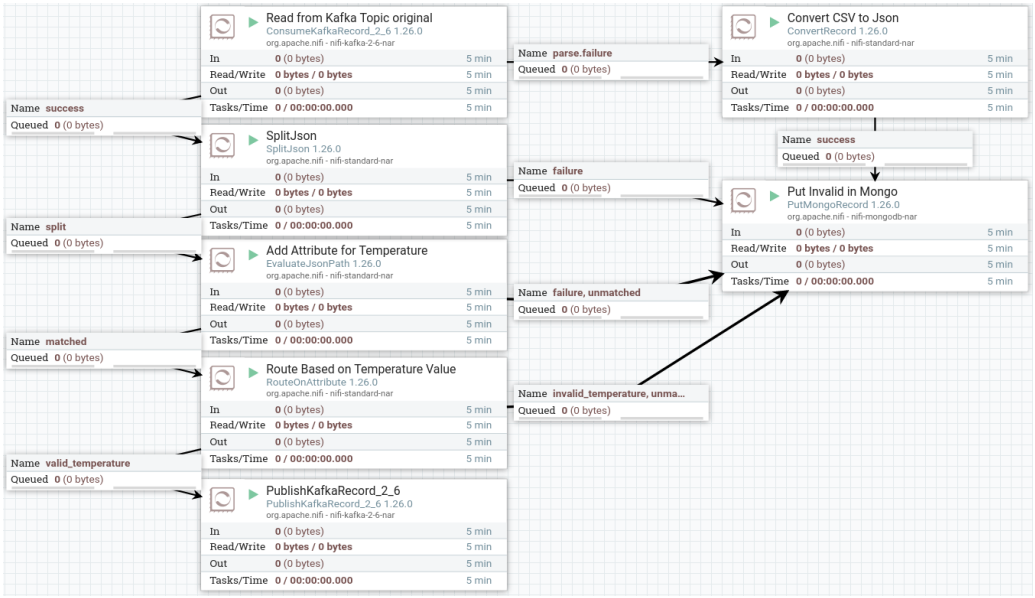

Fig. 2. NiFi Flow

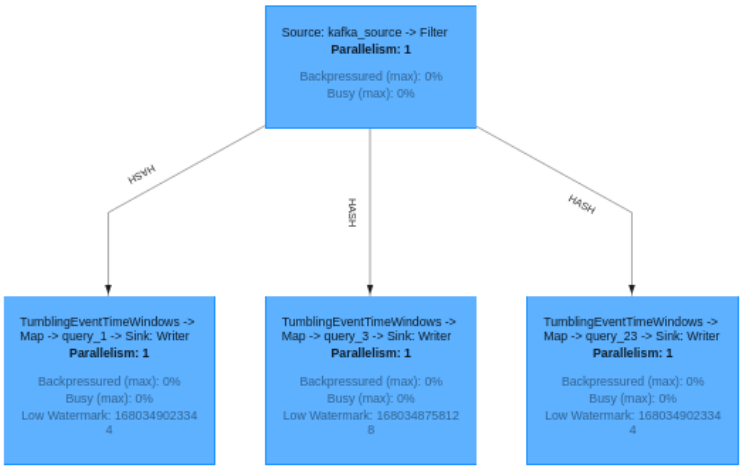

Fig. 3. Flink DAG Query 1

Fig. 4. Flink DAG Query 2

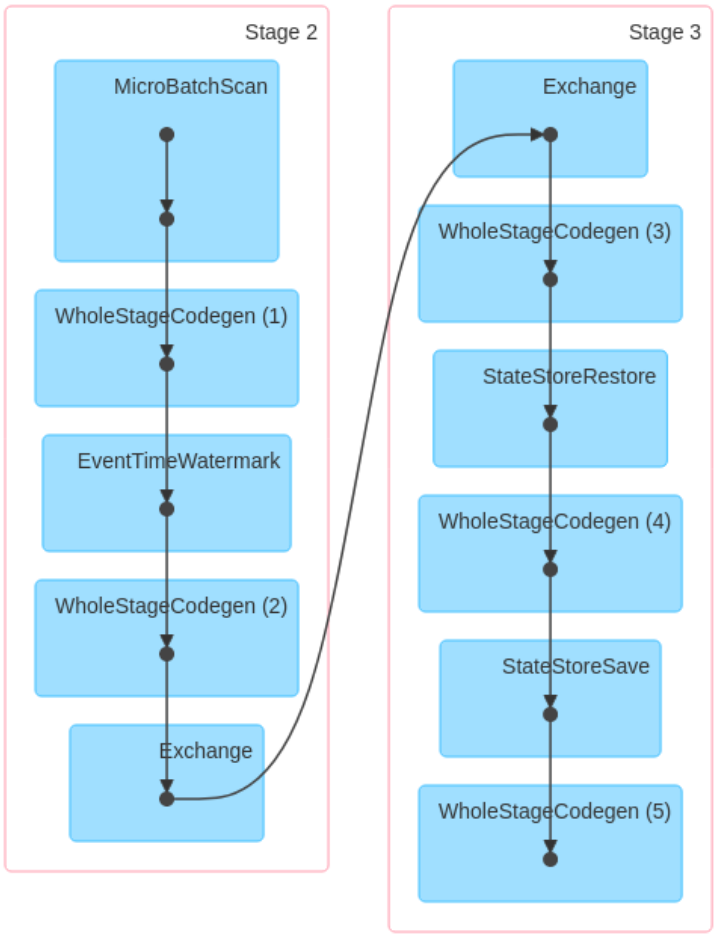

Fig. 5. Spark DAG Query 1

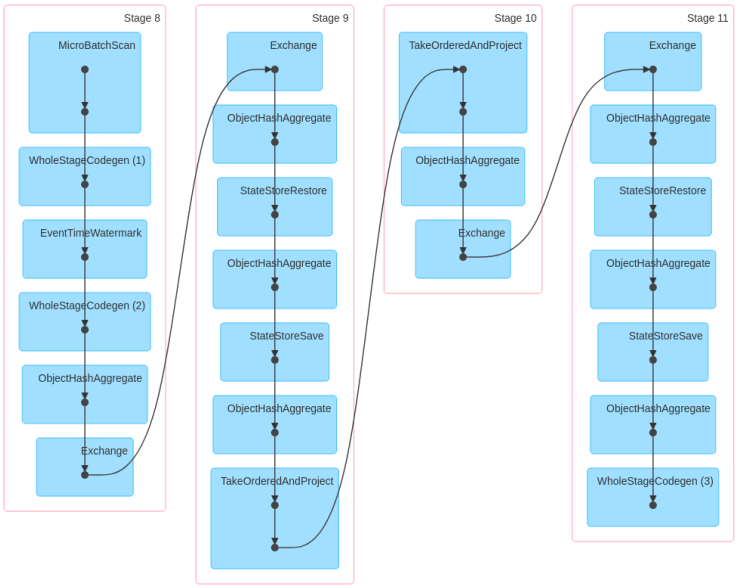

Fig. 6. Spark DAG Query 2

Fig. 7. Flink Latency Query 1

Fig. 8. Flink Throughput Query 1

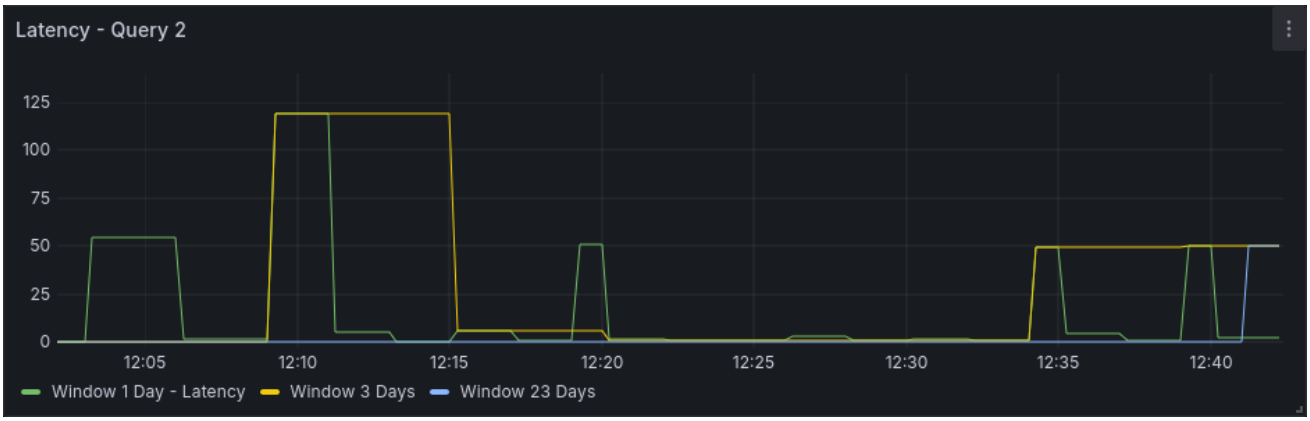

Fig. 9. Flink Latency Query 2

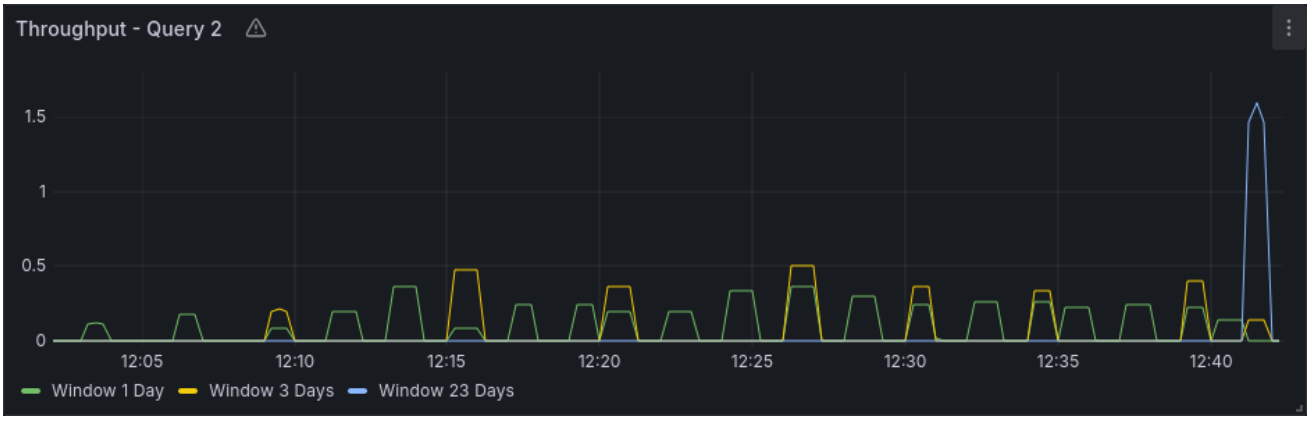

Fig. 10. Flink Throughput Query 2

Fig. 11. Spark Latency Query 1

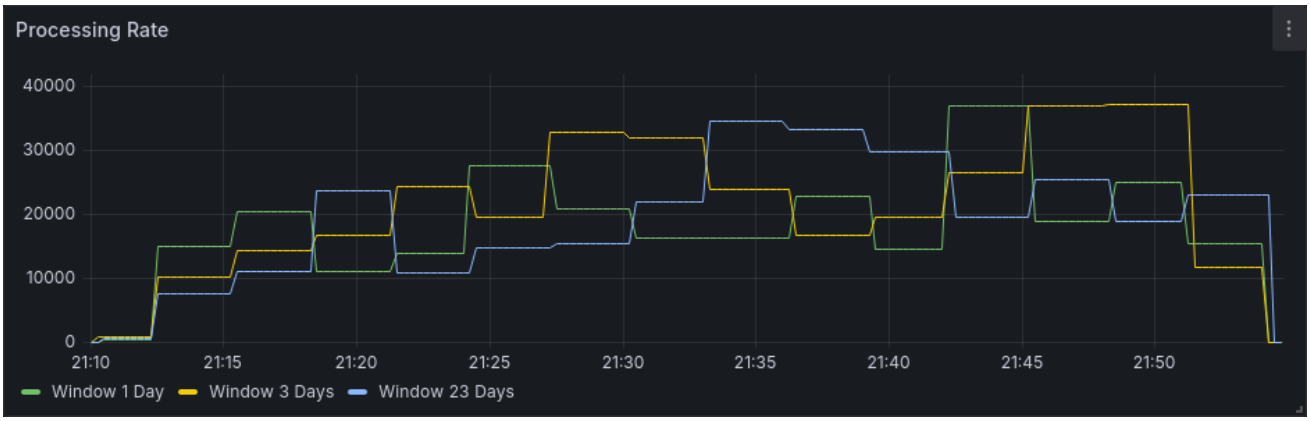

Fig. 12. Spark Processing Rate Query 1

Fig. 13. Spark Latency Query 2

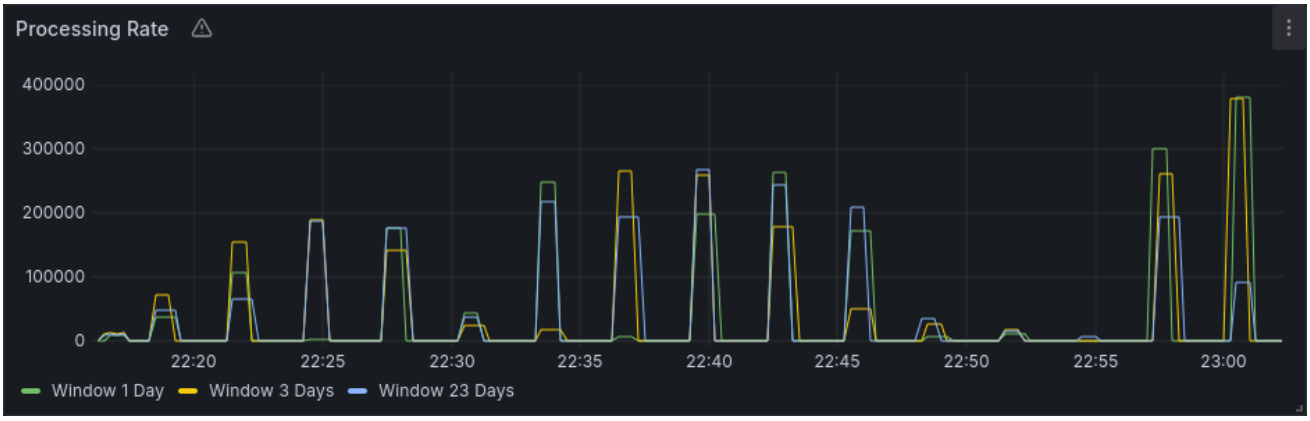

Fig. 14. Spark Processing Rate Query 2