Analisi di dati di monitoraggio con Apache Spark

1st Valerio Brauzi
TorVergata
Rome, Italy
0333768

2nd Marina Calcaura
TorVergata
Rome, Italy
0334044

***Sommario*—Questo documento si pone lo scopo di descrivere le caratteristiche e lo stack architetturale utilizzato per rispondere a 3 query assegnate nel secondo progetto del corso di "Sistemi e Architetture per Big Data" della facoltà di Ingegneria Informatica magistrale dell'universita di Roma Tor Vergata (a.a 2024/25), giustificando le scelte progettuali prese nella costruzione dell'applicazione.**

I. Introduzione

Il progetto svolto consiste nello svolgimento di tre query su dati elaborati tramite processamento straming, relativi a rilevazioni telemetriche di circa 200k hard disk nei data center gestiti da Backblaze.
Il lavoro è stato diviso in quattro fasi:

- Progettazione dell'architettura
- Simulazione arrivo dati in streaming
- Analisi e processing dei dati
- Analisi delle performance

Nome Framework	Utilizzo
Apache Flink	Processamento streaming
Apache Kafka	Pubblicazione records di input e output
Apache Zookeeper	Coordinazione Kafka Cluster
Docker	Sviluppo distribuito su container
Docker Compose	Orchestratore in cluster locale

Tabella I
Framework utilizzati e relativi utilizzi

II. Architettura

A. Overview

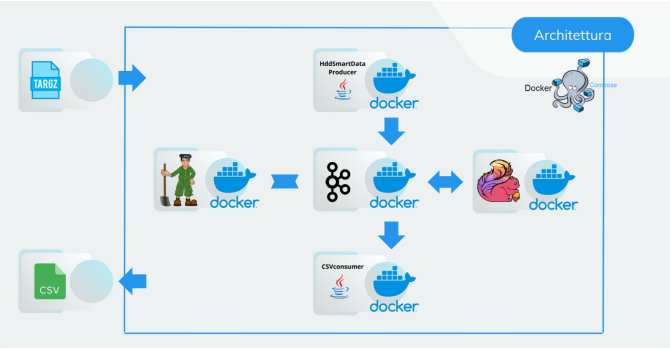

Figura 1. Architettura Applicazione

L'applicazione sviluppata è strutturata come segue: nella fase di **Simulazione** il dataset di partenza viene gestito in modo da simulare un arrivo streaming di rilevazioni riguardanti hard-disk. Nella fase di **processing** invece tali dati vengono processati in tempo reale al fine di rispondere alle query (descritte nel dettaglio più avanti): i vari risultati vengono scritti da flink su vari topic kafka e vengono conseguentemete esportati in formato csv. In contemporanea, vengono prese le **performance** in tempo reale sui record processati da flink e mostrate in formato csv.

III. Simulazione

La prima sfida nello sviluppo dell'applicazione è stato, per l'appunto, simulare un arrivo streaming di dati invece già tutti disponibili. I dati vengono inizialmente recuperati dal file compresso in formato tar.gz che viene copiato in un applicazione containerizzata chiamata **Producer** (publisher Kafka), per venire poi decompresso (si noti che il file csv è già ordinato per timestamp). Dopodiche il Publisher pubblica ogni record nel topic di Kafka **input-data** con un intervallo di tempo tra l'invio di un record e il successivo proporzionale alla differenza tra i relativi timestamps, in modo da simulare l'arrivo dei dati in tempo reale a partire dalle rilevazioni. Per far si che anche il tempo sia simulato, e che quindi il tempo di esecuzione dell'applicativo sia sostenibile, la differenza temporale viene ridotta di un fattore di velocizzazione (5.000.000), chiamato **speedup**.
Si noti come al fine di non snaturare i dati, che vengono collezionati alla mezzanotte di ogni giorno, il tempo di arrivo nei singoli giorni non è stato randomizzato. A causa del meccanismo dei watermark utilizzati da flink, la finitezza del dataset non permette un corretto svolgimento delle query: per aggirare il problema sono state inserite 4 tuple con date successive alla massima disponibile nel topic di input.

IV. Processing

A. Overview

La fase di processamento dei dati è stata effettuata utilizzando **Flink** come framework di streaming-processing: in particolare, ogni query è stata scritta in una classe Java distinta e viene eseguita sfruttando un container che ospita il nodo **Job manager** che si avvale di un **Task manager** per l'esecuzione delle query a livello operativo. Per ogni query il job Flink consuma direttamente i record dal topic interessato di kafka; ogni query inoltre utilizza delle Tumbling Windows basate su Event Time ogni:

- giorno

- 3 giorni
- tutto il dataset (23 giorni)

. I risultati di ogni query vengono pubblicati su dei topic kafka unici per ogni query e per ogni finestra, per essere resi disponibili all'utenza in un secondo momento.

B. Query 1

La prima query richiede di **calcolare il numero di eventi, il valor medio e la deviazione standard della temperatura misurata delgi hard disk facenti parte dei vault con identificativo compreso tra 1000 e 1020**.
In prima istanza, tramite un **consumer kafka** vengono estratti i dati grezzi dallo stesso e trasformati in oggetti **Record** tramite la funzione *parseRecord*. Questa funzione effettua operazioni di parsing sui dati di input e li converte in un formato strutturato: si noti come questa fase includa i filtri per prendere solo i record in cui il vaultId ha i valori desiderati. Si noti inoltre come la classe record utilizzata comprenda solo i campi di interesse, ovvero *vaultId, temperature e timestamp.* In questa fase viene impostata anche la strategia di **Watermarking** da utilizzare *(WatermarkStrategy.<Query1Record>forBoundedOutOfOrderness(Duration.ofSeconds(30))*. In questo modo si riesce a gestire eventuali record fuori ordine con un ritardo massimo di 30 secondi.
Per calcolare il valore delle statistiche si applica un' **AggregateFunction** (che trova la sua implementazione in *TemperatureStatisticsAggregator*) congiunta a una **ProcessWindow** (*WindowResultFunction*): la prima fa perno su un'implementazione dell'algoritmo di Welford, per calcolare i valori di interesse senza aver bisogno di memorizzarli; la WindowFunction invece serve per collezionare i dati interessanti in un oggetto di tipo tupla, permettendone un'agevole lettura.
I risultati della query, come accennato, vengono infine pubblicati sul topic designato tramite un sink chiamato *FlinkKafkaProducer*.

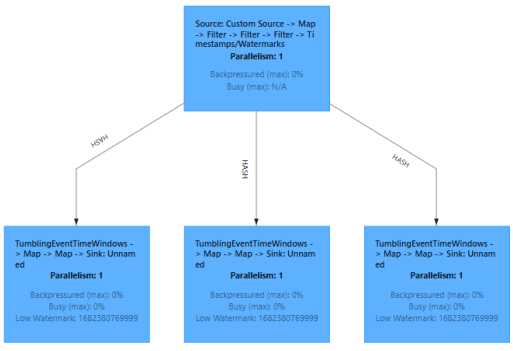

Figura 2. Dag della UI del job manager di flink per la query1

Chiaramente le computazioni per tutte le finestre vengono eseguite in parallelo, nella natura di un'applicazione streaming.

C. Query 2

La seconda query chiede di **calcolare la classifica aggiornata in tempo reale dei 10 vault che registrano il piu alto numero di fallimenti nella stessa giornata, riportando per ogni vaultId il numero di fallimenti, il modello e numero seriale degli hard disk guasti**.
I passi iniziali sono analoghi all'implementazione precedente, dalla manipolazione dei dati in input fino alla definizione delle strategie di Watermarking; si noti come nelle creazione dei dataStream, stavolta le filter garantiscono che si prendano in esame solo i record che hanno *failure = 1*.
I record, raggruppati per vaultId, vengono ora aggregati nelle finestre temporali già descritte; l'aggregatore tiene traccia delle seguenti informazioni: *vaultId, numberOfFailures, List<DiskFailure>* (lista di oggetti che contengono dettagli sui fallimenti dei dischi). Tramite un accumulatore è possibile aggiornare questi parametri in relazione ai record sottoposti a processamento: alla fine della finestra temporale, l'accumulatore contiene il numero totale di fallimenti e i dettagli dei dischi per quel vaultId.
Per il calcolo dei Top 10 Vault si utilizza una finestra globale (**windowAll**) per raccogliere tutti i risultati aggregati di una determinata finestra temporale. I isultati vengono ordinati in base al numero di fallimenti (in ordine decrescente). Vengono selezionati i primi 10 record dalla lista ordinata. Nelle stesse modalità precedenti i risultati vengono pubblicati sui topic designati.

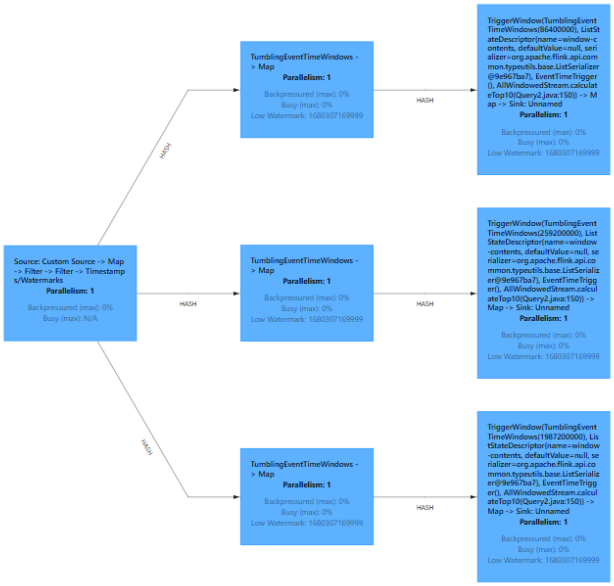

Figura 3. Dag della UI del job manager di flink per la query2

Chiaramente le computazioni per tutte le finestre vengono eseguite in parallelo, nella natura di un'applicazione streaming.

D. Query 3

Infine, la terza query richiede di **Calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo delle ore di funzionamento degli hark disk per i vault con identificativo tra 1090 (compreso) e 1120 (compreso)**.
I percentili devono essere calcolati in tempo reale, senza ordinare tutti i valori e possibilmente senza accumularli.
Per rispettare quanto richiesto (ovvero di calcolare le statistiche in real time e senza accumularle) si è scelto di usare la classe **TDigest**, che raggruppa dati in centroidi senza accumularli. Ogni centroide rappresenta un gruppo di valori vicini e tiene traccia del conteggio dei valori e della loro somma: interrogando la distribuzione dei centroidi è possibile ricavare i quantili approssimati.
Come per le prime 2 query, i passi iniziali sono analoghi all'implementazione precedente, dalla manipolazione dei dati in input fino alla definizione delle strategie di Watermarking; si noti come nelle creazione dei dataStream, stavolta le filter garantiscono che si prendano in esame solo i record che hanno vaultId compreso tra 1090 e 1120.
Dato che il valore di powerOnHours è cumulativo nel dataset, l'aggregatore *LatestPowerOnHoursAggregator* ha il compito di mantenere il record powerOnHours più aggiornato per ciascun disco, identificato univocamente dal *serialNumber*, all'interno di una finestra temporale definita.
I risultati vengono emessi attraverso le stesse modalità delle prime 2 query.

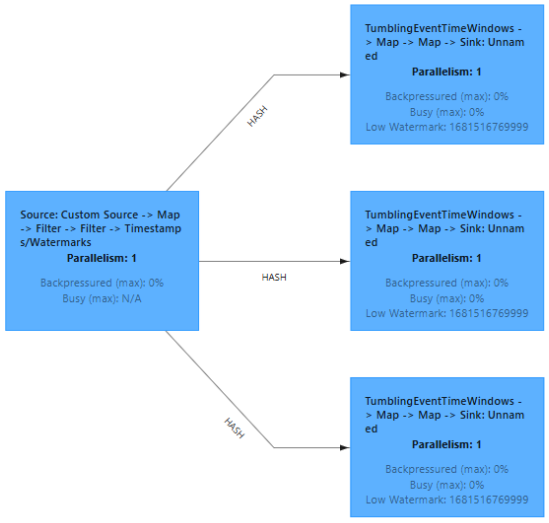

Figura 4. Dag della UI del job manager di flink per la query3

E. Visualizzazione risultati

Parallelamente all'esecuzione delle query, i risultati delle stesse risultano visualizzabili in tempo reale su file csv appositi (suddivisi per query e per finestra), visibile all'utenza tramite l'utilizzo dei volumi Docker. La scrittura su tali file è completamente affidata ad un Consumer kafka che legge dai topic specifici per finestre e query e li esporta sui file csv designati: sebbene flink sia in grado di scrivere su csv, questa scelta alleggerisce le operazioni del taskmanager, fornendo un grado di disaccoppiamento apprezzabile. I risutati sono consultabili a questo link . Nell'implementazione è stata posta attenzione nella dinamicità delle scritture: il consumer definisce l'header del csv basandosi sul nome del topic da cui sta leggendo.

V. Analisi delle performance

Le metriche scelte per valutare le prestazioni dell'applicazione sono **Throughput e Latenza**. Esse vengono misurate a runtime mediante la classe *MetricRichMapFunction*: ogni job flink chiama la classe attraverso una map, che aggiorna internamente il contatore delle tuple processate. Viene dunque calcolato il tempo trascorso dall'inizio dell'elaborazione: questo valore è ottenuto sottraendo il timestamp di inizio, memorizzato nella variabile *start*, dal timestamp corrente.
Il throughput rappresenta il numero di tuple elaborate per secondo. È calcolato dividendo il numero totale di tuple elaborate (counter) per il tempo trascorso in secondi. Il tempo trascorso in secondi è ottenuto dividendo elapsedMillis (tempo corrente - start) per 1000. La latenza rappresenta il tempo medio impiegato per elaborare ciascuna tupla. È calcolata dividendo il tempo totale trascorso (elapsedMillis) per il numero di tuple elaborate (counter).
In questo modo si perviene a delle metriche di valutazione quanto più affidabili possibile, relative al singolo job e divise in base alle finestre dello stesso.
Di seguito vengono riportati i tali metriche graficate in un istogramma. Il trend del throughput e della latenza rispetta le attese: per le finestre **1 day** la frequenza di uscita dei risultati e molto più elevata rispetto a **3 days** e **all time**.

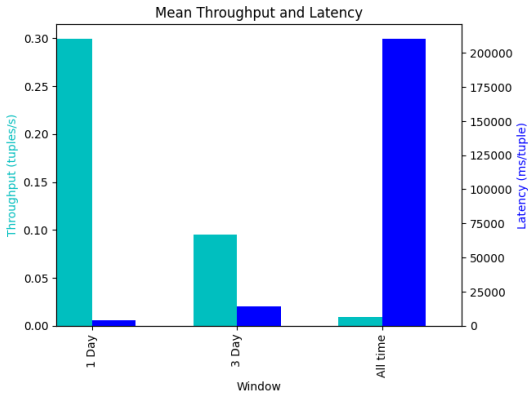

Figura 5. Metriche query 1

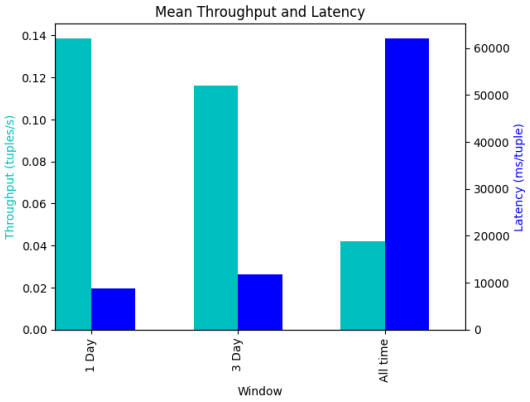

Figura 6. Metriche query 2

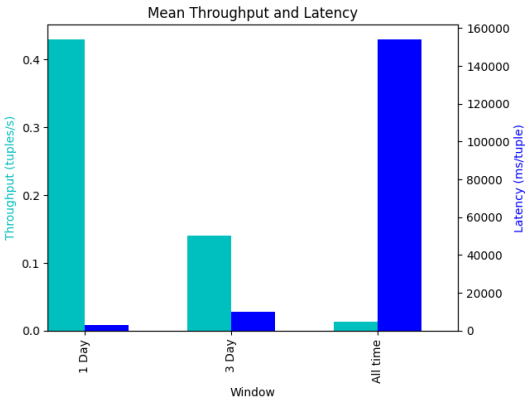

Figura 7. Metriche query 3

Di seguito la configurazione della macchina su cui sono state eseguite le run:

- **Processore**: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 core, 8 processori logici
- **Memoria fisica installata (RAM)**: 16,0 GB
- **SO**: Microsoft Windows 11 Pro