Sistemi e Architetture per i Big Data

Anastasia Brinati
Macroarea di Ingegneria
Università di Roma "Tor Vergata"
Roma, Italia
anastasia.brinati@alumni.uniroma2.eu

Andrea Cantarini
Macroarea di Ingegneria
Università di Roma "Tor Vergata"
Roma, Italia
andrea.cantarini@alumni.uniroma2.eu

***Sommario*—L'obiettivo di questo documento è illustrare le idee dietro lo sviluppo di una pipeline di stream processing e le motivazioni di alcune scelte architetturali e di analisi.**

***Index Terms*—stream data processing, NiFi, Flink, Kafka**

I. Introduzione

La presente relazione descrive la progettazione della pipeline di stream processing, su una versione ridotta del dataset di dati reali di telemetria per fallimenti di hard drive, fornito da Backblaze per la DEBS 2024 GC, usando Apache Flink come framework per il processamento, per rispondere ad alcune queries; descrive l'architettura distribuita su cui fa affidamento la pipeline, ed infine analisi e considerazioni sui risultati richiesti.

II. Architettura

L'architettura del sistema è emulata usando Docker Compose su un singolo nodo. Una bridge network consente la comunicazione fra i containers, per assicurare un flusso di dati coordinato attraverso tutte le diverse componenti del sistema.

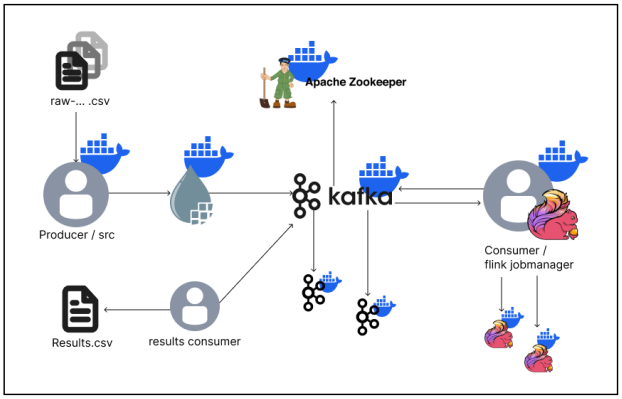

Figura 1. System architecture

Il sistema è strutturato in tre macroaree principali di interesse, che possono essere eseguite separatamente:

- **Producer**.
  Si occupa dell'acquisizione dei dati in tempo reale; legge i dati da un file *.csv* locale riga per riga e ne simula l'arrivo in base ai timestamp di ogni record; ogni tupla è poi inviata ad intervalli simulati verso NiFi, che si occupa dell'ingestione e dell'elaborazione successiva;
- **Backbone**.
  E' il cluster di container che fa da scheletro e sostiene il flusso di dati, srotolando l'ingestione tramite due framework per acquisizione e messaggistica, qui sotto descritti.
  Apache Nifi: gestisce un nififlow per ascoltare record provenienti dal producer per poi pubblicarli presso un Kafka broker sul topic *sabd*;
  Apache Kafka: cluster di 2 broker che gestisce lo streaming distribuito consentendo la pubblicazione, l'abbonamento, la memorizzazione e l'elaborazione dei dati in tempo reale;
  Apache Zookeeper: servizio di coordinamento distribuito che gestisce la configurazione e la sincronizzazione del cluster Kafka.
- **Consumer**.
  Cluster Flink che ha il ruolo di consumer verso il topic *sabd* pubblicato su Kafka, è costituito di un jobmanager e due taskmanager. Parte di quest'area dell'architettura è anche il container che si occupa di consumare i risultati prodotti dal jobmanager Flink e riportarli su file.

III. Data Pipeline

La pipeline di processamento inizia con l'ingestion delle tuple tramite Apache NiFi, che ne assicura la pubblicazione presso un broker di Apache Kafka, trasferendo interamente gli eventi come record , senza effettuare alcun tipo di pre-processing. Proprio quest'ultima fase infatti è delegata ad Apache Flink: mentre i dati vengono pubblicati, il consumer legge dal topic di interesse per effettuare pre-processing e processing.
Gli utenti che si occupano del lato consumer hanno la possibilità di scegliere quale query eseguire e con quale finestra temporale. Infine, i risultati di ogni query vengono salvati localmente nell'apposita cartella *Results*.

A. Simulazione dello streaming in arrivo

Al fine di considerare uno scenario realistico di stream processing, si è optato per l'implementazione di un Producer che leggesse riga per riga dal file *.csv* passato in locale e temporizzasse lo streaming tramite un semplice meccanismo di *sleep()*, affinché fosse possibile uno speed up con un fattore costante di scala che simboleggia il trascorrere di un giorno.

B. Ingestion

Il Producer inoltra singolarmente ogni evento come payload di richieste POST verso Nifi, il quale espone, tramite un processor di tipo *ListenHTTP*, una porta su cui ricevere. Ad ogni richiesta ricevuta, Nifi sposta il contenuto verso il processor successivo di tipo *PublishKafka_1_0*, il cui compito è pubblicare sul topic *sabd* i record che man mano vengono ricevuti.

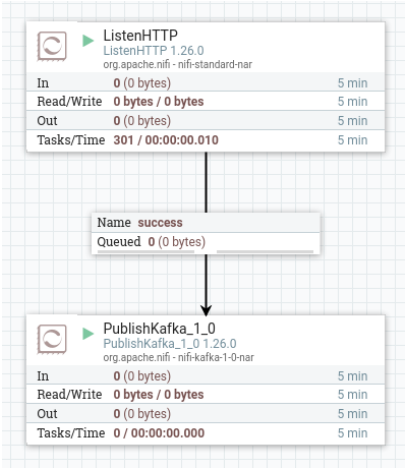

Figura 2. Nififlow for Data Ingestion

C. Pre-processing

Sono state implementati alcuni meccanismi di pre-processamento dei dati:

- all'interno del dataset l'intestazione è ripetuta alcune volte, pertanto il producer è in grado di riconoscerla per non inviarla al cluster Flink.
- durante l'esecuzione della query 1, il consumer Flink non processa le tuple di eventi che non rappresentano fallimenti all'interno del dataset, al fine di ottenere una maggiore efficienza.

D. Processing

L'API Python di Flink mette a disposizione due API per il DSP: Table API e DataStream API. Nel presente progetto è stato fatto uso di quest'ultima.
La DataStream API Python di Flink [1] consente di scrivere programmi Python per eseguire trasformazioni su stream di dati. È possibile definire finestre di vario tipo (tumbling, sliding, ...) in base a diversi parametri (numero di eventi, event time, ...). Inoltre, vi è una distinzione tra finestre keyed e non-keyed, con la differenza che le prime suddividono lo stream in più stream logici in base a una chiave, sui quali poi eseguire le trasformazioni. Le finestre keyed sono state utilizzate nell'implementazione delle query.
Un importante strumento messo a disposizione dalla DataStream API sulle finestre keyed sono le AggregateFunction, una versione generalizzata della classica operazione di reduce che consente di definire il tipo di dato in ingresso, in uscita e un accumulatore. È particolarmente utile quando si ha a che fare con dati cumulativi, ad esempio media o varianza. Estendendo la classe di base *AggregateFunction* e implementando i rispettivi metodi astratti, è possibile creare un accumulatore, aggiungere dei valori, ottenere i risultati finali oppure eseguire il merge di due accumulatori. Infine, le *ProcessWindowFunction* forniscono più flessibilità rispetto alle altre funzioni sulle finestre offerte dall'API, a discapito di un maggior consumo di risorse dovuto alla bufferizzazione interna degli elementi della finestra finché questa non è pronta al processamento. Per questo motivo, il suo utilizzo è stato sempre accompagnato da quello di una AggregateFunction. Le ProcessWindowFunction sono particolarmente utili poiché forniscono accesso anche ad informazioni di contesto della finestra, ad esempio lo stato o il tempo di processamento.

Il codice implementato include la definizione di funzioni di elaborazione personalizzate (ad esempio *ParseJsonArrayFunction()* per deserializzare i record dello stream in arrivo dalla sorgente e *MyTimestampAssigner()* per assegnare timestamp agli elementi del flusso basandosi sul vero timestamp degli eventi), la configurazione delle sorgenti e dei sink e la logica di aggregazione e filtraggio per ogni specifica query. Il processamento è triggerato da *env.execute()* per ciascuna query e window.

- Sorgente: la sorgente dei dati nel nostro caso è il **FlinkKafkaConsumer** con i seguenti principali dettagli di configurazione: lettura dal topic Kafka *sabd* e schema di deserializzazione SimpleStringSchema(). La sorgente è aggiunta allo StreamExecutionEnvironment per creare un oggetto DataStream, che rappresenta i dati provenienti da Kafka.

- Parsing: la prima operazione che viene eseguite sul flusso in ingresso è parsing, tramite una map che effettua la conversione delle stringhe JSON in tuple.

- **Query 1**: *Per i vault (campo vault id) con identificativo compreso tra 1000 e 1020, calcolare il numero di eventi, il valor medio e la deviazione standard della temperatura misurata sui suoi hard disk (campo s194 temperature celsius).*
  Q1. Sono stati fatti i seguenti passaggi:
  Prima di tutto viene utilizzata la funzione `map` per trasformare ogni tupla del parsed stream in una nuova tupla contenente solo gli elementi di interesse, ovvero il timestamp, il vault_id e s194_temperature_celsius. Successivamente, vengono assegnati timestamp e watermark agli elementi del flusso con *watermark_strategy* inizializzata in precedenza con for_monotonous_timestamps e con il timestamp assigner personalizzato, fondamentale per la gestione delle finestre temporali basate sull'event time. Il passaggio successivo prevede il filtraggio tramite `filter` per mantenere solo le tuple il cui campo

vault_id è compreso fra 1000 e 1020. Segue la `key_by` per raggruppare lo stream in stream logici in base alla chiave, composta dal timestamp e dal vault_id. In termini tecnici, qui si fa uso - come nelle altre query - di keyed streams. Viene dunque applicata la finestra temporale di tipo tumbling con la dimensione indicata in input al consumer all'avvio del programma. Giunti alla fase di aggregazione, si combinano gli elementi della finestra, con la funzione di aggregazione personalizzata `Query1AggregateFunction` per calcolare count, mean e la squared distance dalla media. La funzione di aggregazione fa uso dell'algoritmo online di Welford [2] per il calcolo della media e la varianza, che consente di aggiornare tali valori all'arrivo di ogni tupla con un minore spreco di risorse a discapito di un'accuratezza minore.

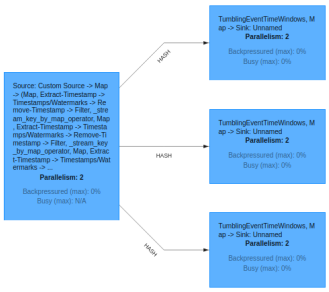

Figura 3. Query 1 DAG

- **Query 2**: *Calcolare la classifica aggiornata in tempo reale dei 10 vault che registrano il più alto numero di fallimenti nella stessa giornata. Per ogni vault, riportare il numero di fallimenti ed il modello e numero seriale degli hard disk guasti.*

Q2. L'esecuzione della seconda query è composta di due passaggi fondamentali:

1) conteggio dei fallimenti in ogni vault per ogni giorno;
2) composizione della classifica.

Per ogni passaggio è stata usata una *AggregateFunction* apposita. In particolare, nel primo caso:

- si estraggono, con `map`, dalle tuple del dataset i campi di interesse, ossia timestamp, vault_id, modello, numero di serie e flag failure
- si assegna l'event time in base al timestamp
- si filtrano le tuple con `filter` per elaborare soltanto quella che rappresentano un guasto, ossia quelle con il flag di failure impostato a 1 (nota: nonostante sia un valore booleano, questo campo è interpretato come intero per facilitare il conteggio)
- si trasforma lo stream in keyed stream tramite `key_by`, con la chiave composta da timestamp e vault_id, con una finestra basata sull'event time di dimensione specificata dall'utente all'avvio del programma
- tramite una `AggregateFunction`, si contano i fallimenti in ogni vault per ogni giorno separatamente. L'accumulator tiene traccia anche delle coppie (modello, vault_id) univoche. Ciò significa che se, ad esempio, un hard disk fallisce molteplici volte nell'arco di una giornata, il suo modello e il suo numero seriale compariranno una sola volta nella lista, mentre il numero di guasti sarà comunque aggiornato.
- l'aggregate function restituisce una tupla composta dal numero totale di fallimenti nel vault in quello specifico giorno e una stringa contenente le coppie univoche (modello, vault_id) di ogni hard disk fallito
- tramite una `ProcessWindowFunction` apposita, ai risultati del punto precedente vengono aggiunti il timestamp e il vault_id, ossia gli elementi della chiave, restituendo la tupla intera

Nella seconda parte:

- si indicizza, tramite `key_by` il datastream ottenuto dal passaggio precedente in base al timestamp e si utilizza una finestra come prima
- tramite un'ulteriore AggregateFunction, si accumulano tutte le tuple dei fallimenti nei singoli vault, si ordinano in base al numero di fallimenti e si combinano per creare la classifica dei primi 10 vault con più fallimenti nel formato di dati richiesto dalla traccia. La lista potrebbe non essere formata da 10 elementi, pertanto gli elementi finali vengono impostati a valori di base (0 per i vault_id e stringa vuota per la lista dei modelli e dei numeri di serie)

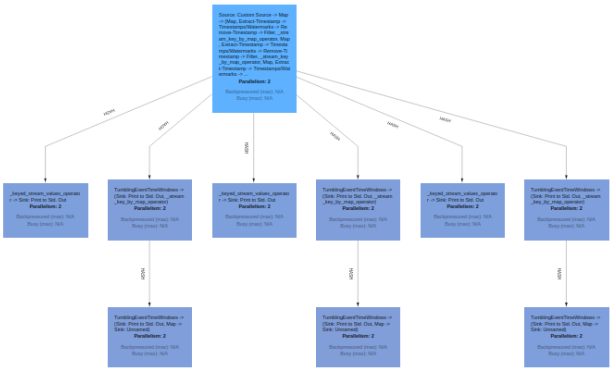

Figura 4. Query 2 DAG

- **Query 3**: *Calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo delle ore di funzionamento (campo s9 power on hours) degli hark disk per i vault con identificativo tra 1090 (compreso) e 1120 (compreso). Si presti attenzione, il campo s9 power on hours riporta un valore cumulativo, pertanto le statistiche richieste dalla query devono far riferimento all'ultimo valore utile di rilevazione per ogni specifico hard disk (si consideri l'uso*

del campo serial number).

Q3. Sono stati adottati i seguenti passaggi:

- si calcola il totale delle ore di lavoro per le query per le tuple con vault_id compreso tra 1090 e 1120, sfruttando il numero di serie per identificare univocamente gli hard disk all'interno di uno stesso vault;
- si applica, con una finestra di dimensione impostata dall'utente e basata sull'event time, indicizzando in base al timestamp, una `AggregateFunction` che applica l'algoritmo del T-digest per il calcolo dei percentili. Il T-digest [3] è una struttura dati che consente di calcolare valori cumulativi di distribuzioni con l'utilizzo di poche risorse, a discapito di una minore accuratezza nel calcolo. Come implementazione, è stata utilizzata una libreria Python già esistente [4]. La funzione restituisce anche il minimo e il massimo dei valori incontrati.

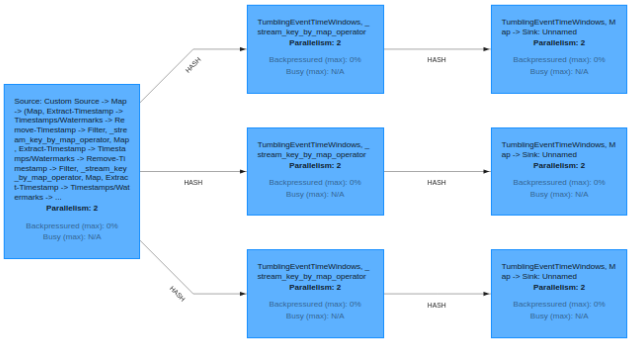

Figura 5. Query 3 DAG

- **Sink**: la destinazione dei dati è nuovamente Kafka, utilizzando stavolta il **FlinkKafkaProducer** per pubblicare i dati trasformati e aggregati in un topic Kafka 'q**i**_windowsize'. Al fine di riportare i risultati nella forma richiesta da ogni specifica query, è stato applicato un ultimo operatore allo stream, che tramite una *map* effettua la serializzazione dell'output delle aggregate functions nel Flink Type ROW_NAMED, in modo da riportare per ciascuna tupla anche il nome della colonna relativo a ciascun campo. Quest'ultimo passaggio ha facilitato la serializzazione tramite *JsonRowSerializationSchema* ed ha permesso di rendere la scrittura su *.csv* dei risultati indipendente dal tipo di query analizzata e priva di elementi hard coded, in quanto ogni record è pubblicato come un dizionario json con keys proprio le colonne indicate in fase di serializzazione.

E. Results

Una volta che i risultati sono pubblicati su un determinato topic Kafka, o durante la loro pubblicazione, è possibile lanciare il results_consumer, al fine di trasferire su file *.csv* i vari record.

IV. Analisi dei risultati

Come metriche di efficienza delle operazioni, sono state utilizzati il throughput (numero di tuple processate per unità di tempo) e latenza (tempo necessario al processamento di una tupla).

A. Custom_maps

Al fine di implementare la misurazione delle metriche di cui sopra, è stata utilizzata una `MapFunction` apposita che conta il numero di tuple e tiene traccia del tempo trascorso dall'inizio del processamento, emettendo coppie (throughput, latenza).

B. Graphs

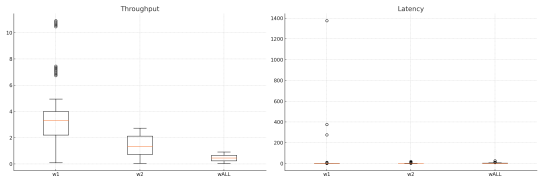

Figura 6. Query 1

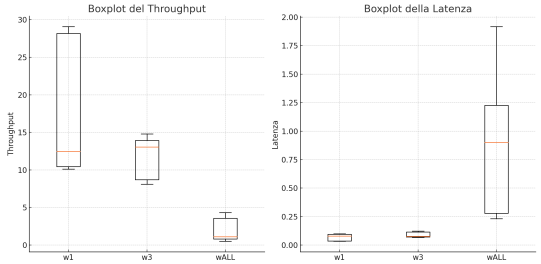

Figura 7. Query 2

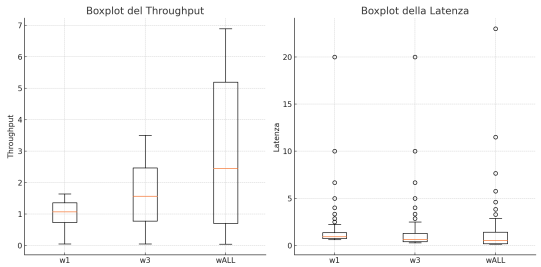

Figura 8. Query 3

C. Criticità riscontrate

Sfortunatamente i limiti computazionali dei mezzi a nostra disposizione si sono rivelati un grave svantaggio sia per la costruzione dell'architettura sia per la gestione dell'enorme quantità di dati, rallentando di molto l'esecuzione delle query, nonché il testing delle stesse. A causa di queste limitazioni la maggior parte dei grafici sono realizzati andando a considerare le metriche raccolte su una porzione ulteriormente ridotta del

dataset, al fine di favorire un confronto comunquesia verosimile con le prestazioni attualmente ottenute dall'implementazione delle query. Siamo inoltre consapevoli che l'utilizzo di Nifi come ulteriore framework per rendere la gestione dell'acquisizione ed ingestion più complessa è risultato in un collo di bottiglia di peso non trascurabile sulle tempistiche esecutive.

RIFERIMENTI BIBLIOGRAFICI

[1] https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/overview/
[2] https://en.m.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford́s_online_algorithm
[3] The t-digest: Efficient estimates of distributions (https://doi.org/10.1016/j.simpa.2020.100049)
[4] https://github.com/CamDavidsonPilon/tdigest