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 primo 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 batch, 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
- Preprocessing dei dati
- Analisi e processing dei dati
- Postprocessing dei dati
- Analisi delle performance

Nome Framework	Utilizzo
Apache Spark	Processamento batch
Nifi	Data ingestion e preprocessing
HDFS	Storage su file system distribuito
Redis	Salvataggio dei risultati su DB noSQL
Grafana	Visualizzazione risultati query
Docker	Sviluppo distribuito su container
Docker Compose	Orchestratore in cluster locale

Tabella I
Framework utilizzati e relativi utilizzi

II. Architettura

A. Overview

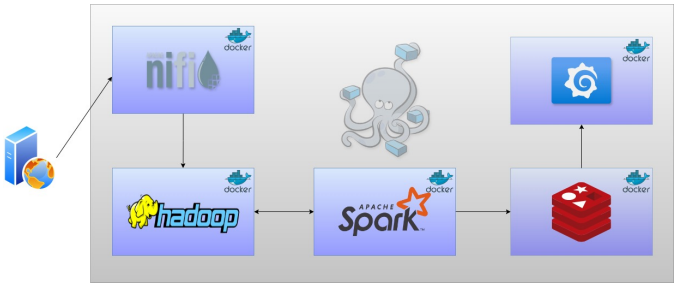

Figura 1. Architettura Applicazione

L'applicazione sviluppata è strutturata in tre fasi principali: **pre-processamento** con **NiFi**, **processamento** con **Spark** e **post-processamento** con **Redis**. I **dataset iniziali** vengono recuperati tramite **NiFi** e salvati su **HDFS**. Al termine del pre-processamento, i file di **output** sono salvati su **HDFS** in formato **CSV** e **Parquet**. Nella fase di processamento **Spark** legge i parquet prodotti nella fase precedente ed esegue le query avvalendosi dei worker specificati: i risultati delle interrogazioni sono salvati su **HDFS** e resi disponibili in locale, oltre ad essere salvati su **Redis**. I risultati di Redis possono essere visualizzati consultando una dashboard custom tramite **Grafana**. Tutti i framework sono stati eseguiti su container **Docker** orchestrati su una singola macchina tramite **docker-compose**.

III. Pre-processing

Per i task di **data-acquisition** e **data-injestion** è stato utilizzato il framework NiFi: questo framework ha permesso di automatizzare e ottimizzare la gestione dei dati dalla fase di download fino alla preparazione per l'analisi.

A. Creazione del Template per il Download e la Conversione dei Dati

Nel processo di creazione del template su Apache NiFi "DownloadDataToHDFS", come prima operazione è stato automatizzato il download dei dati da una fonte remota (link fornito sulla traccia) utilizzando il processore *GetHTTP*. Il formato dei dati scaricati è .tar.gz: viene dunque decompresso tramite il processore *CompressContent* e i suoi contenuti vengono estratti utilizzando il processore *UnpackContent*. I file estratti vengono quindi convertiti in formato CSV utilizzando il processore *ConvertRecord*. Infine, i dati convertiti in formato CSV vengono salvati su HDFS tramite il processore *PutHDFS*. Questo garantisce la persistenza dei dati anche in caso di interruzione della connessione di rete, consentendo il recupero del dataset senza la necessità di ripetere l'intero processo di download e conversione, migliorando così l'efficienza complessiva del sistema.

B. Creazione dei Template per la Pulizia e il Filtraggio dei Dati:

Sono stati creati tre template aggiuntivi ("templateQuery1", "templateQuery2", "templateQuery3") su Apache NiFi per eseguire operazioni di pulizia e filtraggio dei dati ottenuti dalla fase appena descritta, generando così tre dataset distinti ottimizzati per le specifiche query richieste. Questi utilizzano una serie di processori, tra cui *QueryRecord*, sia per estrarre

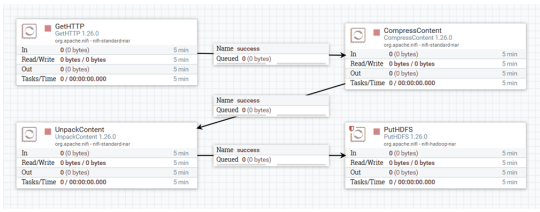

Figura 2. Template "DownloadDataToHDFS"

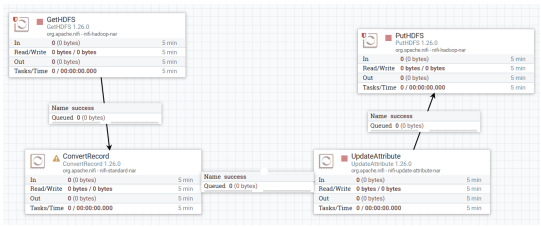

Figura 4. Template CSVtoParquet

colonne specifiche utilizzando istruzioni *SQL* che per escludere i dati nulli.
Per garantire la disponibilità immediata dei dataset per l'analisi con Apache Spark, i dati filtrati vengono salvati in formato CSV su HDFS utilizzando il processore *PutHDFS*. Questo assicura che i dati siano prontamente accessibili e utilizzabili per l'analisi successiva. Inoltre, grazie al processore ConvertRecord, il file viene salvato anche in formato Parquet per garantire prestazioni ottimali durante l'analisi in Apache Spark. Questa scelta è stata fatta considerando le migliori prestazioni offerte dal formato Parquet rispetto al formato CSV in contesti di analisi dei big data.
Si noti come tutto il pre processamento viene saltato qualora i dati necessari alla fase successiva fossero già disponibili su HDFS, evitando un impiego di risorse e di tempo che risulterebbe eccessivo e assolutamente inutile.
Qui si riportano un esempio di visualizzazionl del template utilizzati provvisto da Nifi.

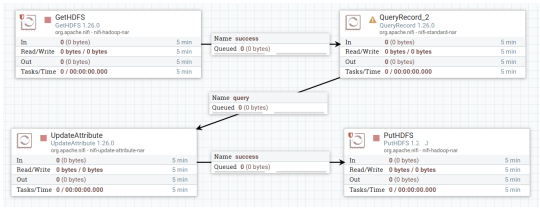

Figura 3. Template Query2

C. Conversione in parquet

Ad ogni template che genera un csv come output, corrisponde un template che, preso tale csv, lo converte in formato *parquet*: in questo modo le letture delle query sono poco più rapide di quanto lo sarebbero leggendo semplicemente l'output dei primi template: questo è dovuto al fatto che i dataset preprocessati risultano comunque di dimensioni ridotte, permettendo dunque una lettura abbastanza veloce. Si è comunque scelto di penalizzare leggermente i tempi di preprocessamento (fase che viene eseguita una sola volta alla prima invocazione per ciascuna query), al fine di risparmiare tempo e risorse nella fase di processamento (che invece è, banalmente, obbligatoria). In figura 4 si mostra uno dei template per attuare tale conversione.

D. Automatizzazione

Tutto il processo di acquisizione e pre-elaborazione dei dati è stato automatizzato utilizzando le API REST di Apache NiFi.
Inizialmente, il flusso di lavoro è stato avviato importando il template "DownloadDataToHDFS", il quale scarica il dataset nel sistema di file distribuito HDFS. Questo è stato fatto inviando una richiesta POST per istanziare il template, ottenendo così l'Id del gruppo di processori creato. Successivamente, sono stati recuperati i servizi di controllo associati al gruppo di processori mediante una richiesta GET. Questa operazione è cruciale per assicurare che i flussi di dati NiFi siano correttamente configurati, monitorati e gestiti. Dopo aver recuperato e abilitato i controller services, è stato avviato il gruppo di processori impostando lo stato su running attraverso una richiesta PUT.
Al termine dell'elaborazione del flusso, il template è stato fermato utilizzando di nuovo una richiesta PUT per modificare lo stato del gruppo e disabilitare i servizi di controllo. Successivamente, sono state recuperate tutte le connessioni e inviate richieste POST per eliminare i file di flusso presenti in ciascuna coda.
Dopo aver pulito l'ambiente, il template è stato rimosso inviando una richiesta DELETE all'API di NiFi. Questo processo di rimozione di tutti i componenti e la pulizia dell'ambiente è essenziale per preparare NiFi all'integrazione di altri template necessari per l'elaborazione e il filtraggio dei dati scaricati.
L'automatizzazione dei flussi relativi agli altri template avviene nello stesso modo descritto precedentemente.

IV. Processing

A. Overview

La fase di processamento dei dati è stata effettuata utilizzando Spark come framework di batch-processing: in particolare, ogni query è stata scritta in una classe Java distinta e viene eseguita sfruttando un container che ospita il nodo Master e n altri container che ospitano altrettanti nodi Worker (n è configurabile al momento della creazione della rete). Si noti come all'utilizzo degli RDD sia stato preferito quello dei Dataframe: disponendo di un dataset ben strutturato è stato infatti possibile usufruire delle ottimizzazioni software concesse dalla seconda soluzione, che ha reso la scelta architetturale quasi obbligata.

B. Query 1

La prima query richiede di **determinare la lista di vault che hanno subito esattamente 4, 3 e 2 fallimenti per ogni giorno**.
Dalla SparkSession si legge il file *parquet* designato per la query1 da Hadoop creando un Dataframe che dispone di tutte

e sole le colonne del dataset interessanti.
I dati di tipo *"DataTime"* e *"Boolean"* vengono opportunamente convertiti per essere manipolati e viene creato un successivo Dataframe tramite un'operazione di *groupBy* fatta sulle colonne date e vaultId, ove i valori della colonna failure (precedentemente resi numerici) vengono sommati. Infine, il Dataframe viene filtrato per ottenere esclusivamente i vault che nel giorno *DD-MM-YYYY* hanno subito 2,3 o 4 fallimenti. Il risultato viene dunque ordinato per data e vaultId e la data formattata come richiesto; il Dataframe viene in ultimo scritto su un file formato *CSV* e caricato sia su *HDFS* che su *Redis* (specificando le opportune chiavi).

C. Query 2

La seconda query chiede di **calcolare la classifica dei 10 modelli di hard disk che hanno subito il maggior numero di fallimenti**; **inoltre si calcoli una seconda classifica dei 10 vault che hanno registrato il maggior numero di fallimenti, indicando il modello degli hard disk che hanno subito i fallimenti per ogni valut**.
Come per la precedente, viene letto il file *parquet* designato e creato il conseguente Dataframe e la colonna *failure* viene convertita in valore intero. Nella prima parte della query lo stesso viene manipolato tramite una groupBy fatta sulla colonna *model* ove i valori di failure vengono sommati. Tramite una successiva filter (e una order) viene creata la prima classifica richiesta.
Nella seconda parte query invece vengono selezionate dal dataframe iniziale solo le righe degli hard disk che hanno subito un fallimento tramite una *filter*; tramite una *groupBy* per *vaultId* viene calcolato il numero totale di fallimenti per ogni vault, inoltre, con un'altra groupBy, vengono raccolti i modelli distinti di hard disk soggetti ad almeno un fallimento. I due dataframe risultanti da tali operazioni vengono uniti con una *join* sulla colonna *vaultId*: il dataframe risulante viene dunque ordinato in ordine decrescente e troncato con una *filter*, pervenendo alla classifica desiderata.
I Dataframe delle due classifiche vengono scritti su HDFS e Redis in modalità analoghe a quelle precedentemente illustrate.

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 (campo s9PowerOnHours) degli hark disk che hanno subito fallimenti e degli hard disk che non hanno subito fallimenti, riportando nell output anche il numero totale di eventi utilizzati per il calcolo delle statistiche**.
Come per le precedenti, viene letto il file *parquet* designato e creato il conseguente Dataframe e la colonna *failure* viene convertita in valore intero, mentre la colonna *date* viene convertita in *formato date*.
In seconda istanza, tramite una *groupBy* per *serialNumber* vengono calcolate le date più recenti relative alle rilevazioni per ogni hard disk, allo scopo di tenere solo le righe più aggiornate per ciascun dispositivo (sfruttando il fatto che il campo *s9PowerOnHours* sia cumulativo).
Successivamente viene generato un nuovo dataframe riportante tutti e soli gli hard-disk che hanno subito un fallimento in tutta la storia del dataset: questo dataframe ed il precedente vengono uniti tramite una *leftJoin* di modo che il risultato sia una struttura riportante tutti i dispositivi divisi in failed e notFailed con la colonna s9PowerOnHours aggiornata all'ultimo valore disponibile.
Su questo nuovo Dataframe vengono calcolate le statistiche richieste per i due insiemi di hard disk *failed* e *notFailed* attraverso una funzione creata ad hoc. In ultimo, i due dataframe risultanti vengono uniti attraverso una *union* ed i risultati vengono scritti su *HDFS* e *Redis*.

V. Post-Processing e Visualizzazione

A. Redis

Il framework **Redis** è stato utilizzato come cache affinchè il framework di visualizzazione **Grafana** possa sperimentare una latenza quanto più bassa possibile. Per ogni query sono stati salvati i dati in delle HashMap con diverse chiavi a seconda della circostanza e con i campi che riportano le restanti informazioni.

B. Grafana

I risultati delle varie query vengono resi disponibili agli utenti tramite una facile visualizzazione, utilizzando le funzioni del framework *Grafana*. Anche in questa circostanza la priorità massima è stata data all'automatizzazione: le query per prendere i dati da redis e visualizzarli in maniera precisa su grafana sono state scritte utilizzando le operazioni concesse dallo stesso, facendo inoltre utilizzo di trasformazioni per creare una view di facile comprensione. Le varie visualizzazioni, unite, formano una dashboard, già disponibile all'utenza nel momento dell'avvio di Grafana: si noti che in caso il dataset subisse variazioni non sarebbe necessario cambiare alcunché, in quanto tutte le query sono dinamiche.
Alla sezione VII alcuni esempi di visualizzazione.

VI. Performance

Le performance del modello sono state misurate prendendo come punto di partenza l'istante subito successivo alla creazione della spark session e come istante finale quello successivo alla creazione del dataframe di output, escludendo quindi le scritture e la chiusura della sessione. Di seguito si riporta un semplice grafico relativo alle performance per ogni query in terine di millisecondi al variare del numero di spark workers.

I risultati rispecchiano le attese: le query hanno performance migliori con un solo worker: questo è dovuto al fatto che il dataset non è molto grande e le interrogazioni non sono molto onerose, di conseguenza si paga il delay di comunicazione tra i nodi più di quanto si guadagni in termini di scaling sulle risorse.
Facendo dunque riferimento al grafico, si suggerisce una

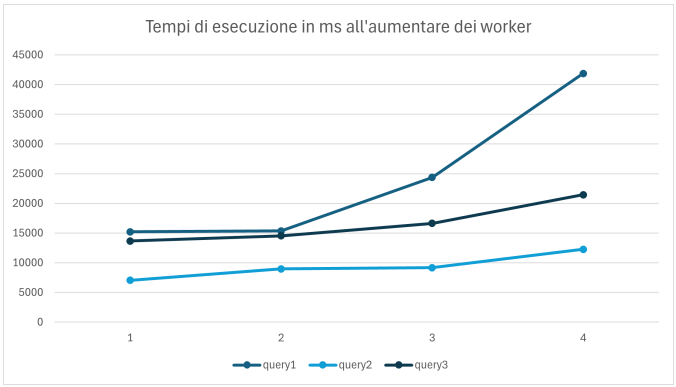

Figura 5. Performance al variare dei nodi worker

configurazione con uno spark worker, in modo da non gravare eccessivamente sulle performance.
Per completezza si riportano le tabelle utilizzate per creare il grafico in Excel (si noti che ogni misurazione presente è la media di 5 misurazioni per ogni query e per ogni numero di worker).

#nodi worker	ms q1	ms q2	ms q3
1	15218	7035	13653
2	15347	8974	14526
3	24361	9174	16636
4	41873	12263	21440

Tabella II
PERFORMANCE IN MS

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

VII. VISUALIZZAZIONI GRAFANA

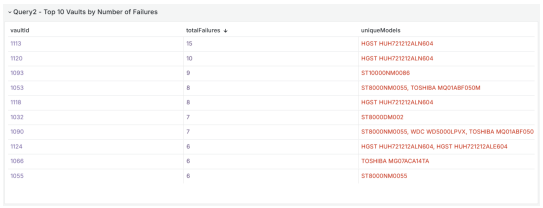

Figura 6. Risultato query 2.2

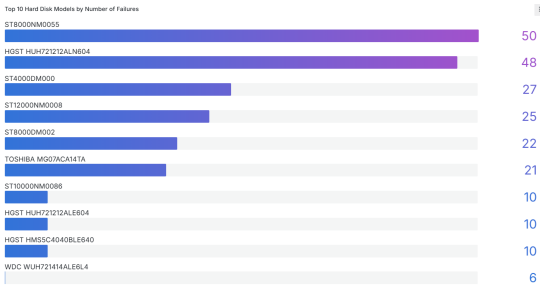

Figura 7. Risultato query 2.1

Figura 8. Risultato query 3