Hard disk failure data analysis

Luca Falasca
0334722
luca.falasca@students.uniroma2.eu

Matteo Conti
0323728
matteo.conti97@students.uniroma2.eu

***Abstract*—Questo studio presenta l'implementazione di una pipeline per il batch processing di dati di grandi dimensioni, finalizzata all'analisi dei fallimenti dei dischi rigidi. La pipeline, containerizzata e gestita tramite Docker Compose, utilizza Apache NiFi per l'ingestion dei dati, HDFS per lo storage, Spark per il processamento, MongoDB per l'archiviazione dei risultati e Grafana per la visualizzazione di essi. Il dataset analizzato, una versione ridotta di quello del Grand Challenge della conferenza ACM DEBS 2024, contiene informazioni su data di misurazione, identificativo del disco rigido, modello, ore di accensione e fallimenti. Per analizzare i dati e confrontare le prestazioni dei formati di file CSV e Parquet sono state eseguite tre query. L'analisi ha evidenziato che il formato Parquet, grazie alla sua struttura a colonne e compressa, risulta generalmente più efficiente nelle query di aggregazione rispetto al formato CSV. Tuttavia, eccezioni sono state rilevate nella query 2. I risultati delle query e le prestazioni sono stati visualizzati tramite dashboard in Grafana, offrendo una visione dettagliata dei fallimenti dei dischi rigidi e delle performance delle diverse query.**

I. Introduzione

A. Descrizione del problema

Il problema da affrontare consiste nell'eseguire un processamento batch di un dataset mediante l'uso di una pipeline basata su framework per Big Data. Nello specifico, l'obiettivo è analizzare un dataset contenente informazioni relative ai fallimenti dei dischi rigidi attraverso l'esecuzione di 3 query.

B. Obiettivi

Gli obiettivi del progetto sono quelli di eseguire le query proposte e di valutare le prestazioni di esse a partire da due formati di file, CSV e Parquet. Il motivo della scelta di Parquet è perché essendo un formato basato su colonne, risulta essere più efficiente per le query di aggregazione e quindi in fase di progettazione analizzando le query ci è sembrato più adeguato utilizzarlo rispetto ad altri formati per righe.

C. Dataset

Il dataset fornito è una versione ridotta di quello presentato nel Grand Challenge della conferenza ACM DEBS 2024. Delle numerose colonne presenti nel dataset, ne verranno selezionate solamente cinque per l'esecuzione delle query, in particolare:

- *date*: data della misurazione
- *serial_number*: identificativo del disco rigido
- *failure*: indica se il disco rigido ha avuto una failure o meno
- *model*: modello del disco rigido
- *vault_id*: identificativo del gruppo di storage server
- *s9_power_on_hours*: ore di accensione del disco rigido

Esplorando il dataset si può notare un'ambiguità nella colonna *s9_power_on_hours*, essa contiene dei valori 0. Questi valori possono essere interpretati in due modi:

- Il dato è un errore di misurazione e quindi va eliminato
- Il disco rigido è stato acceso per meno di un'ora al momento della misurazione

Date le seguenti considerazioni si è deciso di non eliminare i valori 0 e considerarli quindi validi. Questo andrà ad influenzare i risultati della query 3, in particolare questo sarà molto evidente per quanto riguarda il minimo delle ore di accensione. La colonna *s9_power_on_hours* contiene anche diversi valori nulli, le righe corrispondenti a tali valori verranno eliminate in fase di preprocessamento. Inoltre, la colonna *vault_id* contiene elementi il cui valore non è un intero bensì la stringa "vault_id", anche questa volta le righe corrispondenti a tali valori verranno eliminate in fase di preprocessamento.

II. Pipeline

Per la gestione dei dati relativi ai guasti dei dischi rigidi, è stata implementata una pipeline (Fig. 1) il cui deploy è stato fatto tramite docker compose. La pipeline è composta da diverse componenti, ognuna delle quali svolge un compito specifico, in particolare sono stati utilizzati Apache NiFi per la data ingestion, HDFS per il data storage, Apache Spark per il data processing, MongoDB per l'analytical data storage e Grafana per la visualizzazione dei risultati. Di seguito verrano illustrati i dettagli di ciascuna componente.

Fig. 1. Pipeline ad alto livello

A. Data Ingestion

Per la data ingestion è stato utilizzato il framework Apache NiFi, il quale permette di creare dei flussi di dati a partire da diversi tipi di sorgente, definire delle operazioni di trasformazione su di essi ed andare a memorizzare il risultato delle trasformazioni in sink di vario genere. Nel nostro caso NiFi svolge le seguenti operazioni:

1) Riceve, tramite HTTP, il file CSV contenente i dati relativi ai guasti dei dischi rigidi.
2) Esegue una query SQL per selezionare solamente le cinque colonne di interesse del CSV ed effettuare una

pulizia dei dati, rimuovendo righe contenenti valori nulli o invalidi.

3) Effettua l'assegnazione esplicita dello schema ai dati, se questo non viene fatto NiFi non riconosce correttamente il tipo di alcune colonne del dataset, interpretandole come delle strutture.
4) Scrive i dati su HDFS in formato Parquet e CSV.

Per implementare il flusso NiFi (Fig. 2) sono stati utilizzati diversi processors e controller services, in particolare:

- *ListenHTTP*: per ricevere il file CSV tramite HTTP
- *QueryRecord*: per eseguire la query SQL ed effettuare l'assegnazione dello schema
- *CSVReader*: per specificare a QueryRecord come è fatto il flusso che riceve in input
- *CSVSetWriter*: per specificare a QueryRecord come deve essere il flusso che genera in output
- *AvroSchemaRegistry*: per specificare a CSVReader ed a CSVSetWriter lo schema dei dati
- *PutHDFS*: per scrivere i dati su HDFS
- *PutParquet*: per scrivere i dati su HDFS in formato Parquet
- *UpdateAttribute*: per specificare che nome dare al flusso e di conseguenza il nome del file che verrà salvato su HDFS

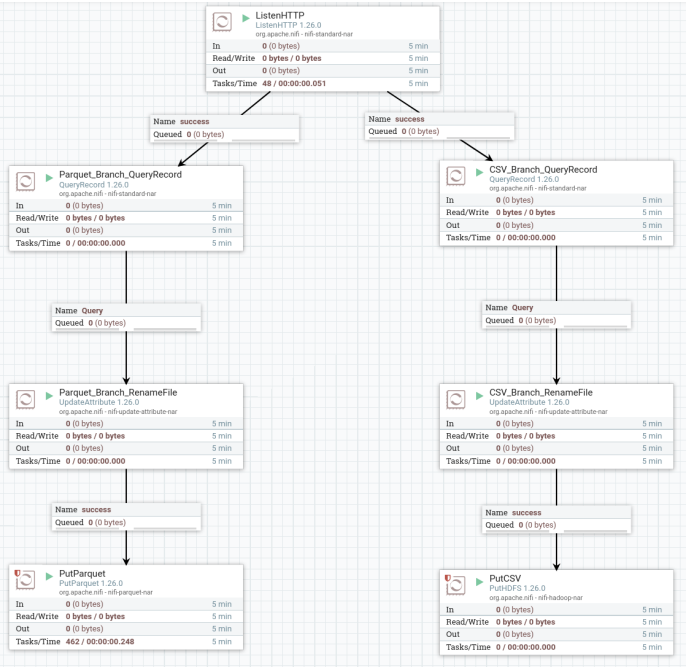

Fig. 2. Nifi flow

B. Data Storage

Per memorizzare i dati a a seguito del preprocessamento effettuato da parte di NiFi è stato utilizzato HDFS, il quale permette di memorizzare grandi quantità di dati su un cluster di nodi. Nel nostro caso il cluster è composto da:

- 1 Namenode
- 2 Datanode

Non è stata utilizzata una struttura particolare per l'organizzazione dei file all'interno dell'HDFS, essi sono stati memorizzati tutti nella root directory.

C. Data Processing

Per il processamento dei dati, e quindi l'esecuzione delle query, è stato utilizzato il framework Apache Spark, in particolare utilizzando la libreria PySpark. È stato utilizzato un cluster composto da uno spark master e quattro spark worker (Fig. 3). Per implementare il cluster è stata utilizzata l'immagine bitami/spark nella versione 3.5.1. Il dataset, filtrato, è stato caricato dall'HDFS ed i risultati delle query sono stati salvati in MongoDB utilizzando il connettore spark-mongodb. Le query sono state eseguite sia a partire dal file in formato CSV che dal file in formato Parquet per valutare le differenze prestazionali tra un formato basato su colonne e uno basato su righe. Per effettuare le query sono state utilizzate le API per i Dataframe e non quelle per gli RDD. Inoltre dato che le query sono formate principalmente da trasformazioni, l'azione che scatenerebbe la lazy evaluation, e quindi l'effettiva esecuzione della query, sarebbe la scrittura del risultato su MongoDB; tuttavia la durata della scrittura dipende dalla grandezza del risultato della query, il quale è abbastanza eterogeneo, abbiamo quindi deciso di unificare la metodologia di acquisizione delle misure di prestazioni inserendo un'azione di show() prima della scrittura su MongoDB.

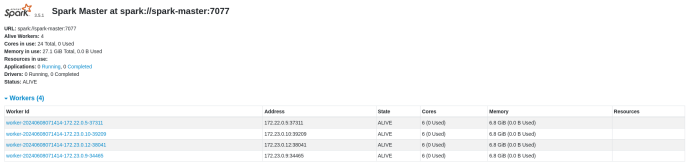

Fig. 3. Spark cluster

1) Query 1: La query 1 prevede di calcolare per ogni giorno, per ogni vault (campo *vault_id*), il numero totale di fallimenti, in particolare occorre determinare la lista di vault che hanno subito esattamente 4, 3 e 2 fallimenti.

Per prima cosa è stato fatto il drop delle colonne inutili per la query, che in questo caso sono *serial_number*, *model* e *s9_power_on_hours*. Successivamente è stato fatto il group by per *date* e *vault_id* e si è aggretato sulla somma di *failure* (essendo i fallimenti rappresentati dai valori 0 e 1 la loro somma rappresenta il numero di fallimenti). Infine è stato effettuato un filtraggio che seleziona solo i vault che hanno subito un numero di fallimenti ≥ 2 e ≤ 4. (Fig. 4)

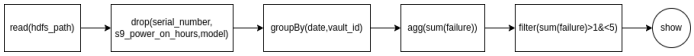

Fig. 4. DAG di alto livello della query 1

2) Query 2: La query 2 è composta di due parti:

- Calcolare la classifica dei 10 modelli di hard disk che hanno subito il maggior numero di fallimenti, riportando

il modello di hard disk e il numero totale di fallimenti subiti dagli hard disk di quello specifico modello.
- Calcolare la classifica dei 10 vault che hanno registrato il maggior numero di fallimenti riportando, per ogni vault, il numero di fallimenti e la lista (senza ripetizioni) dei modelli di hark disk soggetti ad almeno un fallimento.

Anche in questo caso la prima cosa che viene fatta è il drop delle colonne inutili per la query, che in questo caso sono *serial_number*, *s9_power_on_hours* e *date*. Successivamente viene fatto un cache() per mantenere in memoria il dataframe ed evitare di doverlo costruire più volte, in quanto esso verrà utilizzato per entrambe le parti della query. Per la prima parte della query è stato fatto un group by per *model* aggregando sulla somma di *failure*, successivamente è stato fatto un ordinamento in ordine decrescente ed infine sono stati selezionati i primi 10 elementi.
Per la seconda parte della query vengono generati due dataframe, il primo facendo una group by su *vault_id* ed aggregando sulla somma di *failure*, mentre il secondo filtrando i fallimenti pari a 1, facendo una group by su *vault_id* ed aggregando i *model* in un set tramite collect_set(). Successivamente i due dataframe vengono uniti tramite una join sul campo *vault_id*. Il dataframe ottenuto viene ordinato in modo decrescente sul campo *failure* ed infine vengono selezionati i primi 10 elementi. (Fig. 5)

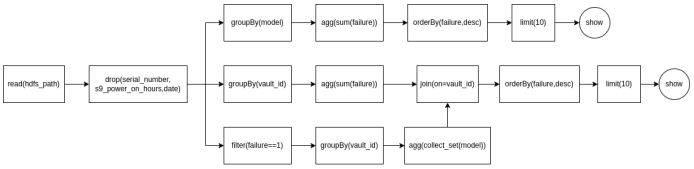

Fig. 5. DAG di alto livello della query 2

3) Query 3: La query 3 prevede di calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo delle ore di funzionamento (campo *s9_power_on_hours*) degli hark disk che hanno subito fallimenti e degli hard disk che non hanno subito fallimenti, indicando anche il numero totale di eventi utilizzati per il calcolo delle statistiche. Dato che il campo *s9_power_on_hours* è un campo cumulativo, per ottenere il tempo di funzionamento è necessario guardare la data più recente disponibile.
Anche in questo caso per prima cosa è stato fatto il drop delle colonne non necessarie che sono *model* e *vault_id*. Successivamente è stato fatto un cache() per mantenere in memoria il dataframe ed evitare di doverlo costruire più volte, in quanto esso verrà utilizzato per due operazioni diverse. A questo punto dal dataframe originale ne vengono generati altri due, uno contenenete solamente dischi che hanno subito fallimenti e l'altro con solamente dischi che non hanno subito fallimenti; questo viene fatto tramite una filter() sul campo *failure*. Per entrambi i dataframe viene fatto un group by su *serial_number* e successivamente viene prelevato il valore più recente di *s9_power_on_hours* applicando alla colonna *date* la funzione first(), la quale ne preleva la prima riga; questo è stato possibile in quanto il dataset è ordinato per data evitando di dover usare la funzione max() che avrebbe avuto un impatto maggiore sulle performance. Prima di eseguire il calcolo dei quantili è stato effettuato il drop delle colonne *serial_number* e *date* in modo da tenere solamente il dato relativo delle ore di funzionamento per le failure 0 e 1 ed allegerire, in questo modo, le successive elaborazioni. Infine per il calcolo effettivo dei quantili è stata utilizzata la funzione approxQuantile() che permette di calcolare in modo efficiente i quantili, con grado di approssimazione configurabile; tramite essa è stato possibile calcolare anche il minimo ed il massimo che corrispondono rispettivamente al quanitle 0 e 100. Per generare il risultato della query è stato creato un nuovo dataframe contenente, per entrambi i dataframe precedenti (con e senza fallimenti), l'output di approxQuantile() e la count() delle righe utilizzate per il calcolo dei quantili.

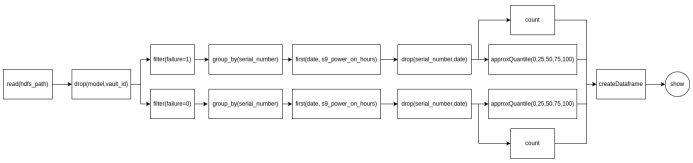

Fig. 6. DAG di alto livello della query 3

D. Analytical Data Storage

I risultati delle query sono stati memorizzati in un database a documento, in particolare MongoDB. Per organizzare i dati, è stata creata una collezione per ciascuna query (Fig. 7), in cui ogni documento rappresenta una riga dei risultati ottenuti dalla query stessa. Inoltre, è stata creata un'altra collezione che contiene le performance, in particolare vengono riportati il timestamp dell'esperimento, la query, il formato del dataset su cui la query è stata eseguita ed il tempo di esecuzione della query stessa.

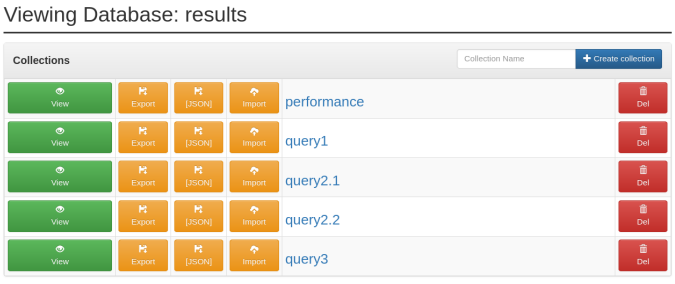

Fig. 7. Schema del database MongoDB.

E. Visualizzazione

Per la visualizzazione dei risultati è stato utilizzato il framework Grafana, il quale permette di creare dashboard personalizzate a partire da diversi tipi di sorgenti di dati. In particolare, è stata creata una dashboard in cui sono presenti vari pannelli che visualizzano alcuni aspetti dei risultati delle query e delle performance della loro esecuzione (Fig. 8). In particolare sono stati sviluppati i seguenti pannelli:

- Un pannello che mostra i top 10 vault che hanno subito il maggior numero di fallimenti con il relativo numero di fallimenti. Derivante dalla query 2.
- Un pannello che mostra i top 10 modelli di hard disk che hanno subito il maggior numero di fallimenti con il valore. Derivante dalla query 2.
- Un pannello che mostra la ripartizione percentuale di vault id che hanno subito 2, 3 e 4 fallimenti. Derivante dalla query 1.
- Un pannello che mostra i vari quantili delle ore di funzionamento per i dischi che hanno subito fallimenti e per quelli che non hanno subito fallimenti. In modo da poter visualizzare come sono distribuiti i dati. Derivante dalla query 3.
- Un pannello che mostra il count dei fallimenti e dei non fallimenti totali su cui sono stati calcolati i quantili. Derivante dalla query 3.
- Un pannello che mostra i tempi di esecuzione delle query sui due formati di file, CSV e Parquet.

Fig. 8. Dashboard di Grafana.

F. Analisi delle prestazioni

1) Configurazione ambiente:

Risorse hardware

```
Processors: AMD Ryzen™ 5 7530U with Radeon™ Graphics × 12
Memory: 16.0 GiB of RAM
Graphics Processor: AMD Radeon™ Graphics
```

Risorse Docker

```
Processors: 6 core
Memory: 8 GiB of RAM
```

2) Risultati: L'analisi delle prestazioni è stata effettuata per ciascuna query, in particolare sono stati valutati i tempi di esecuzione delle query sui due formati di file, CSV e Parquet, e sono stati confrontati i risultati ottenuti. Come detto in precedenza ci si aspetta che il formato Parquet sia più efficiente rispetto al formato CSV, in quanto essendo un formato basato su colonne, dovrebbe avere un vantaggio per le query di aggregazione. Tuttavia questo vantaggio sembra esserci solo per la query 1 e 3. Infatti nella query 2 i tempi di esecuzione sono minori utilizzando il formato CSV (Fig. 9). Questo comportamento potrebbe essere dovuto al fatto che nella query 2 viene eseguita una operazione di join tra due dataframe (Fig. 5), e quindi il formato Parquet, che è basato su colonne, potrebbe essere meno efficiente rispetto al formato CSV, che è basato su righe, per questa operazione.

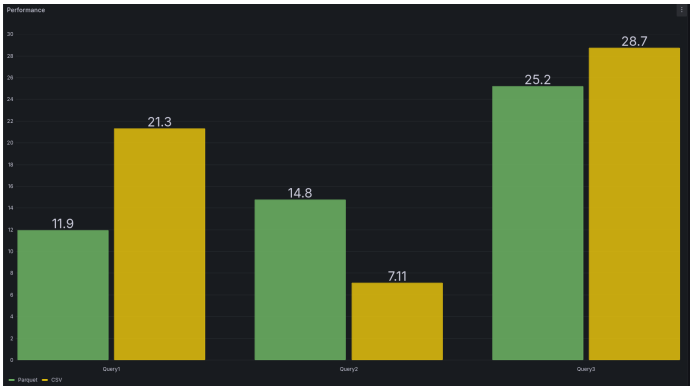

Fig. 9. Pannello delle performance di Grafana.

III. Riferimenti

- Codice sorgente github