SABD – Progetto 1

Progetto valido per il corso di Sistemi e Architetture per Big Data

Di Marco Luca
0333083

Di Totto Luca
0333084

***Abstract*—Questo documento si pone lo scopo di descrivere la progettazione e la realizzazione di un'applicazione per l'elaborazione di dati riguardanti misurazioni sui guasti verificatisi su hard disk, utilizzando un'architettura basata su container Docker. L'applicazione prevede l'uso di Apache NiFi per il prefiltraggio e la trasformazione dei dati, Hadoop HDFS per l'archiviazione, Apache Spark per l'elaborazione batch e Redis e Grafana per il salvataggio e la visualizzazione dei risultati.**

I. INTRODUZIONE

L'obiettivo del progetto è quello di analizzare ed eseguire query su un dataset riguardante i dati di monitoraggio S.M.A.R.T., esteso con alcuni attributi aggiuntivi. Quest'ultimo contiene eventi riguardanti circa duecento mila hard disk in un intervallo temporale di 23 giorni, per un totale di circa tre milioni di eventi.
Il documento ha lo scopo di descrivere nel dettaglio l'architettura utilizzata e le parti fondamentali: verranno prima descritti i componenti del sistema e i relativi dettagli implementativi; verranno poi analizzate in dettaglio le query; successivamente verranno analizzate le tempistiche e le prestazioni generali ed infine verranno discusse determinate considerazioni sul sistema in oggetto.

II. ARCHITETTURA

L'architettura complessiva del sistema è composta da diversi framework, ognuno con caratteristiche ed obiettivi differenti. Il sistema viene interamente containerizzato sfruttando la virtualizzazione di sistema offerta da Docker.
Ognuno dei componenti viene ospitato all'interno di uno o più container Docker e vengono interconnessi tra loro attraverso l'utilizzo di una unica rete virtuale.
Si analizzano di seguito i vari componenti del sistema in oggetto:

- NiFi (apache/nifi:latest)
- Hadoop File System (apache/hadoop:3)
- Apache Spark (apache/spark:latest – estesa con librerie di Spark)
- Redis (redis:latest)
- Grafana (grafana/grafana:latest)

Si veda *Figura 4: Architettura del sistema*.

A. NiFi

Apache NiFi è un framework che permette di effettuare ingestione e prefiltraggio dei dati, da e verso sorgenti differenti, andando a formare il flusso di processamento.
Tramite la User Interface permette di specificare quali azioni deve compiere sui dati tramite l'utilizzo di blocchi chiamati 'processori'.
Nel progetto viene sfruttata la capacità di processamento di NiFi, prima che i dati vengano immessi nel sistema per essere poi processati: nello specifico, dal dataset fornito vengono eliminate tutte le colonne non necessarie nell'elaborazione delle query così da rendere l'insieme dei dati più leggero e semplificare le operazioni di processamento.

1) Implementazione

Il flusso di operazioni svolte da NiFi viene avviato dalla User Interface e prevede prima il recupero da HDFS del file CSV contenente i dati da analizzare, successivamente il filtraggio di questi e la conversione in un file in formato Parquet che verrà infine salvato di nuovo su HDFS.
Di seguito troviamo l'analisi dei processori utilizzati nel flusso:

- GetHDFS: utilizzato per prendere il file CSV contenente i dati da una specifica directory di HDFS e successivamente generare il flowfile;
- QueryRecord: esegue query SQL su un flowfile di ingresso. Il risultato delle query diventa il flowfile di uscita. Viene utilizzato per effettuare il filtraggio sulle colonne del CSV e per la conversione dei dati in formato Parquet. Vengono definiti due processori distinti di questo tipo, che consentono di effettuare due tipologie di filtraggio differenti che danno vita a due file differenti che verranno usati successivamente nelle query, in modo da poter filtrare in maniera ancora più accurata il dataset ed evitare di effettuare l'elaborazione successiva considerando dati non utili.
- PutHDFS: si occupa di scrivere il flowfile che riceve in ingresso nell'Hadoop Distributed File System
- UpdateAttribute: utilizzato per mantenere il valore del tempo di inizio e di fine di processamento del flow di NiFi, utilizzato a fini di valutazione di prestazioni.

B. Hadoop File System

Il file system distribuito Hadoop viene utilizzato per effettuare l'archiviazione distribuita dei dati prima e dopo che questi siano stati filtrati da NiFi e prima che vengano processati da Spark.
Viene implementato tramite un container Docker che svolge il ruolo di namenode nell'architettura HDFS e due container che svolgono il ruolo di datanode, per simulare la distribuzione del dataset.

C. Apache Spark

Apache Spark è un framework open source per il calcolo distribuito. Viene utilizzato per l'esecuzione delle query sul dataset salvato in HDFS. Nell'implementazione di Spark in

un container Docker, viene utilizzata l'immagine di apache/spark estesa con l'aggiunta della libreria Python redis che fornisce le API per l'accesso e l'utilizzo di Redis.
Il framework viene implementato tramite la creazione di un nodo spark-master e tre nodi spark-worker. Come verrà analizzato più avanti, in termini prestazionali è risultato più conveniente utilizzare un solo nodo Worker.

1) Implementazione

Nell'implementazione di Spark, viene usato PySpark, ovvero un'API Python per Spark.
All'inizio di ogni file Python utile al processamento di una query, viene definita una sessione Spark, che rappresenta il punto di ingresso. Successivamente, in ogni query viene letto dall'HDFS il file in formato parquet scritto da NiFi e viene immagazzinato in un Dataframe. Le query vengono eseguite sfruttando le API dei Dataframe,

2) Salvataggio dei risultati

Per ogni query i risultati vengono salvati sia su Redis sia in formato CSV sul file system del container Docker (che vengono successivamente copiati nel file system della macchina host per poter salvare i risultati in maniera persistente).

3) Tempistiche

Per effettuare la misurazione delle tempistiche di esecuzione delle query, prima di eseguire qualsiasi operazione, i file Python prevedono il salvataggio dell'istante corrente in una variabile. Alla terminazione del processo di elaborazione dei dati, viene preso nuovamente il tempo corrente e viene calcolato il tempo effettivo impiegato nell'esecuzione della query.

D. Redis

Redis è un datastore key–value in-memory open source. Nel progetto considerato viene utilizzato per memorizzare i risultati delle tre query e per inviare i risultati delle query verso Grafana, per poter verificare i risultati anche graficamente.

E. Grafana

Grafana è un framework che offre visualizzazione e analisi interattiva di dati tramite grafici e diagrammi per agevolarne la comprensione e l'interpretazione. Viene implementato in un container tramite l'immagine grafana/grafana:latest. Nell'interfaccia web raggiungibile sulla porta 3000 vengono visualizzati i risultati di tutte le query, sfruttando come data source i dati scritti in Redis da Spark. È stata realizzata un'unica dashboard riportante la rappresentazione dei dati in diagrammi di tipo Bar Chart, Gauge e Bar Gauge.

III. QUERY

Di seguito vengono spiegate, approfondite e discusse le tre query realizzate, la loro implementazione e i risultati ottenuti.

A. Query 1

Per ogni giorno e per ogni vault (campo vault_id), calcolare il numero totale di fallimenti. Determinare i vault nei quali si verificano esattamente 2, 3 o 4 fallimenti.

1) DataFrame

Si utilizza il metodo `read.parquet()` per salvare in un dataframe il dataset filtrato in formato parquet.
Successivamente tramite il metodo `GroupBy()` vengono raggruppate tutte le tuple con stessa data, stesso vault_id e stesso valore nel campo failure, contando, per ogni raggruppamento, mediante il metodo `count()`, il numero di tuple presenti.
In questo modo si ottiene un dataframe in cui si hanno per ogni giorno e per ogni vault_id due raggruppamenti, uno con il campo fallimento uguale a uno ed uno con il campo fallimento uguale a zero.
A questo punto, tramite il metodo `filter()` si ottiene un dataframe contenente esclusivamente le tuple in cui il campo failure sia uguale a uno.
In conclusione, si utilizza nuovamente il metodo `filter()` per selezionare esclusivamente le tuple in cui il numero di fallimenti sia uguale a 2,3 o 4.

2) SparkSQL

Per la realizzazione della medesima query utilizzando SparkSQL, viene inizialmente utilizzato il metodo `createOrReplaceTempView()` per creare, a partire dal dataframe, una vista o tabella temporanea.
Quindi, viene implementata una prima query SQL utile per effettuare un filtraggio dei dati e, successivamente, una seconda query SQL in cui vengono raggruppate le tuple per data, vault_id e failure. A questo punto con la funzione `count(*)` vengono contate il numero di tuple per ogni aggregazione.

B. Query 2

Si calcoli la classifica dei primi dieci modelli di hard disk nei quali si verificano il maggior numero di fallimenti. Si mostrino il modello dell'hard disk e il numero di fallimenti verificatisi. Successivamente, si calcoli una classifica dei primi dieci vault nei quali si verificano il maggior numero di fallimenti. Si mostrino il numero di fallimenti e la lista senza ripetizioni dei modelli presenti in quel vault_id che presentano almeno un fallimento.

1) DataFrame

a) Parte 1

Anche in questo caso viene utilizzato il metodo `read.parquet()` per recuperare il dataset scritto da NiFi e successivamente, salvarlo in un dataframe.
Viene in primo luogo effettuato un filtraggio nel dataset su tutte le tuple aventi nella colonna failure il valore uno, così da considerare solo le registrazioni in cui si è verificato un fallimento.
In seguito, il dataset ottenuto viene raggruppato per modello e poi conteggiato, in modo da avere il numero di quanti fallimenti si sono verificati per ogni modello di hard disk.
Tramite il metodo `orderBy()` viene ordinato il dataset in maniera decrescente e successivamente con la funzione `limit(10)` vengono estratte le prime 10 tuple del dataset.

b) Parte 2

Per la seconda parte della query, a partire dal dataframe ottenuto dopo il filtraggio sul campo failure, viene effettuato un raggruppamento sui valori della colonna vault_id e un conteggio su questi. In questo modo si ottiene un dataframe

indicante il numero di fallimenti verificatisi per ogni vault_id. A questo punto, come nella prima parte, viene ordinato il dataframe per valore del conteggio in maniera decrescente e successivamente vengono estratti i primi dieci elementi del dataframe ottenuto.

2) SparkSQL

a) Parte 1

Nella realizzazione della query mediante SparkSQL viene effettuato dapprima un filtraggio della vista tramite semplice clausola `where` sul campo failure, per eliminare tutte le tuple che non registrano un fallimento. Successivamente, tramite la funzione `count(*)` in combinazione con `group by` sul campo model, vengono conteggiati il numero di tuple con fallimento presenti per ogni modello. Nella stessa query SQL, vengono ordinati in maniera decrescente i risultati con la clausola `order by` ed infine con la clausola limit vengono selezionati esclusivamente i primi dieci risultati.

b) Parte 2

Per l'implementazione della seconda parte della query, si riparte dalla vista ottenuta a seguito del filtraggio sulla colonna failure, viene eseguito un raggruppamento per vault_id e vengono selezionate la colonna vault_id e la colonna models (quest'ultima colonna viene estratta tramite la funzione `Collect_set`, che permette di restituire una matrice - o array nel nostro caso - composto dai valori univoci della colonna data in ingresso). In questo modo otteniamo un dataframe contenente la lista di hard disk differenti presenti in ciascun vault. Nell'ultima parte della query utilizziamo il dataframe appena ottenuto in `join` con la tabella ottenuta nella parte 1, in cui si hanno i vari vault con il numero di fallimenti verificatisi in maniera innestata, in modo da avere una tabella in cui si hanno i vault, il numero di fallimenti per il vault in questione e i modelli di hard disk che lo compongono.

C. Query 3

Si calcoli il minimo, massimo, 25, 50 e 75-esimo percentile e il massimo di ore operative per hard disk che presentano un fallimento e per hard disk che non presentano un fallimento. Si mostri anche il numero totale di eventi utilizzati per calcolare la statistica.

1) DataFrame

Dopo aver letto il file in formato parquet come nei casi precedenti, viene effettuato prima un raggruppamento dei dati nel dataframe in base al campo 'serial number' con il metodo `groupBy`. Per ogni gruppo, successivamente, tramite la funzione `agg(expr("max(date)"))` viene calcolata la data massima presente nelle tuple aggregate. Infine, questo campo viene rinominato per indicare che il valore ottenuto corrisponde all'ultima misurazione utile per quello specifico hard disk. Questa fase preliminare è necessaria per far si che i valori medi da calcolare successivamente siano basati sul numero di ore di funzionamento effettive di funzionamento che ogni modello ha raggiunto. Si realizza ora un nuovo dataframe generato dalla `inner join` del dataframe contenente l'ultima data valida per ogni modello con il dataframe di partenza. In questo modo si ottiene un dataframe contenente tutte le colonne iniziali e solo le tuple in cui si ha l'ultima misurazione valida per ogni modello di hard disk.
Si distinguono in due dataframe differenti le tuple relative alle misurazioni di modelli di hard disk che hanno subito un fallimento e quelle relative alle misurazioni che non presentano fallimenti tramite il metodo `filter()`. Su entrambi i dataframe ottenuti, si calcolano le statistiche richieste e il numero totale di eventi presenti.

2) SparkSQL

Nella versione con SparkSQL, dopo aver realizzato una prima vista a partire dal dataframe ottenuto leggendo il file parquet, mediante la clausola `group by` sul campo serial_number si ottengono, per ogni hard disk, la data più grande nel campo latest_detection_date.
Successivamente, viene creata una nuova vista dal dataframe ottenuto a seguito del processamento della query. Viene eseguita una nuova query in cui viene effettuata la join tra l'ultima tabella ottenuta e la tabella di partenza, così da ottenere per ogni serial_number le date di rilevamento più recenti. Da questa nuova query viene creata una nuova vista. Quest'ultima viene utilizzata per calcolare le ore di funzionamento per ciascun hard disk, mediante il campo s9_power_on_hours. Anche in questo caso, da questa query, viene creata una vista temporanea da cui vengono effettuate due query diverse per filtrare i dischi falliti e non falliti, mediante la clausola `where`. Dalle tabelle create vengono calcolati i valori massimi, minimi ed i percentili richiesti, mediante i metodi `MIN, MAX, PERCENTILE_APPROX`. Vengono, inoltre, contati gli eventi utilizzati per il calcolo delle medie utilizzando il metodo `count()`.

D. Analisi delle performance

Per l'analisi delle performance, le query vengono eseguite su una macchina con un processore Intel 13th Gen i7-1355U 1.7GHz con 16 GB di RAM DDR4 4267 MHz. Il tempo di pre-processamento, filtraggo ed ingestion dei dati presso NiFi richiede un tempo medio di 132 secondi. L'esecuzione delle query avviene in deploy-mode client, dove ogni worker presenta 2 core e 1G di memory. Il numero massimo di worker disponibili è pari a tre. Vengono valutate le prestazioni andando a modificare il numero di core a disposizione dei diversi executor. Le query vengono inviate verso Spark mediante il metodo `spark-submit`. Nella valutazione delle prestazioni delle query viene tenuto conto anche del tempo della scrittura dei risultati su Redis, effettuando un confronto delle tempistiche tra l'esecuzione senza scrittura e l'esecuzione con scrittura. In entrambi i casi viene incluso, nel tempo complessivo, anche il tempo di scrittura del CSV dei risultati finali delle query. I tempi di computazione delle query vengono recuperati sia mediante codice Python che tramite WebUI di Spark, per avere un confronto più preciso. La grandezza del dataset filtrato da NiFi e reso disponibile in HDFS utilizzato dalle query 1 e query 2 è di circa 45.8 MB, mentre il dataset filtrato utilizzato dalla query 3 è di circa 52.5 MB.

1) Query 1

Di seguito vengono riportati i tempi ottenuti dall'esecuzione della query 1 sia con le API dei Dataframe, sia con SparkSQL, e come questi variano al variare del numero di cores coinvolti:

	Dataframe		SQL	
	Senza Redis	Con Redis	Senza Redis	Con Redis
1 core	6,869	9,475	6,808	9,443
3 cores	9,389	12,484	9,202	12,833
6 cores	9,980	13,289	9,108	13,067

È possibile notare come nei casi di esecuzione sia con le API dei Dataframe, sia con SparkSQL, all'aumentare dei core, i tempi di completamento aumentano, sia senza considerare la scrittura verso Redis, sia considerandola. La causa di questo comportamento può essere ricondotta alla necessità di Spark di istanziare executor su più worker. La comunicazione distribuita e l'effort di setup dell'ambiente distribuito, in questo particolare caso di dataset molto ridotto, può portare a rallentamenti come illustrato in tabella. Troviamo un tempo di completamento minore nel caso di singolo core poiché viene istanziato un singolo executor su di un singolo worker e la computazione avviene localmente, senza necessità di distribuzione; quella con un singolo core si rivela essere il setup ottimo per questa specifica query con la grandezza del dataset ridotta.

2) Query 2

Di seguito vengono riportati i tempi ottenuti dall'esecuzione di entrambe le parti della query 2 sia con le API dei Dataframe, sia con SparkSQL, e come questi variano al variare del numero di cores coinvolti:

	Dataframe				SQL			
	1ª senza Redis	1ª con Redis	2ª senza Redis	2ª con Redis	1ª senza Redis	1ª con Redis	2ª senza Redis	2ª con Redis
1 core	7,59	9,68	11,99	13,91	6,90	9,03	9,86	12,66
3 cores	9,50	11,78	14,62	16,59	8,77	11,45	12,51	14,81
6 cores	9,72	12,64	15,12	17,38	9,05	11,41	12,67	15,14

È possibile notare come nei casi di esecuzione sia con le API dei Dataframe, sia con SparkSQL, anche in questo caso all'aumentare dei core, i tempi di completamento aumentano, sia senza considerare la scrittura verso Redis, sia considerandola. La causa di questo comportamento può essere ricondotta come nella precedente query, alla necessità di Spark di istanziare executor su più worker. La comunicazione distribuita e l'effort di setup dell'ambiente distribuito, con un dataset molto ridotto, può portare a rallentamenti come illustrato in tabella. Troviamo un tempo di completamento minore nel caso di singolo core poiché viene istanziato un singolo executor su di un singolo worker e la computazione avviene localmente, senza necessità di distribuzione; il setup con un singolo core si rivela essere l'ottimo anche in questo caso. In questa particolare query possiamo notare come la differenza di tempi di esecuzione tra tre core e sei core sia minima: questo è riconducibile al fatto che la differenza di prestazioni è minima, tenendo conto della dimensione del dataset.

3) Query 3

Di seguito vengono riportati i tempi ottenuti dall'esecuzione della query 3 sia con le API dei Dataframe, sia con SparkSQL, e come questi variano al variare del numero di cores coinvolti:

	Dataframe		SQL	
	Senza Redis	Con Redis	Senza Redis	Con Redis
1 core	54,920	75,703	52,938	72,489
3 cores	55,222	73,464	53,856	73,947
6 cores	52,354	73,140	50,085	68,044

È possibile notare come nei casi di esecuzione sia con le API dei Dataframe, sia con SparkSQL, all'aumentare del numero di core a 6, i tempi di completamento diminuiscono. In questo particolare caso, a differenza dei precedenti, troviamo un guadagno nel distribuire la query su diversi worker. La configurazione ottimale si rivela essere, quindi, quella con un numero maggiore di core. In questo caso l'effort di setup dell'ambiente distribuito è vantaggioso a causa della presenza di operazioni più onerose di questa query, a differenza dei casi precedenti.

4) Tabelle comparative e conclusioni

	Dataframe		SQL	
	Senza Redis	Con Redis	Senza Redis	Con Redis
Query 1	6,869	9,475	6,808	9,443
Query 2	11,99	13,91	9,86	12,66
Query 3	54,920	75,703	52,938	72,489

Nella tabella riportata sopra è possibile notare come variano i tempi richiesti dall'esecuzione delle varie query considerando il processamento con un singolo core.

	Dataframe		SQL	
	Senza Redis	Con Redis	Senza Redis	Con Redis
Query 1	9,980	13,289	9,108	13,067
Query 2	15,12	17,38	12,67	15,14
Query 3	52,354	73,140	50,085	68,044

Nella tabella sopra sono, invece, riportati i tempi richiesti per l'esecuzione delle tre query nel caso di coinvolgimento di 6 core. Dalle due tabelle riportate appare ancora più evidente quanto già evidenziato precedentemente, ovvero che nelle

prime due query, un aumento del numero di core ha un impatto negativo sui tempi computazionali richiesti. Nell'ultima, invece, la configurazione che prevede un numero di core utilizzati pari a sei, è quella che comporta un leggero miglioramento nelle prestazioni. La configurazione migliore, quindi, è quella non distribuita sui diversi worker per le prime due query, mentre per l'ultima, anche se per un minimo vantaggio, conviene distribuire l'esecuzione.

La rappresentazione grafica delle query, estratta da Grafana, viene inserita di seguito, unito ai DAG e all'architettura di NiFi. I DAG delle tre query prese in considerazione vengono aggiunte a fine documento, da figura 5 a figura 8: non rappresentano la versione ufficiale dei DAG di Spark poiché utilizzando i Dataframe, presentano ottimizzazioni e operazioni aggiuntive non utili a fini dello studio. Di conseguenza le figure rappresentano dei DAG custom che sintetizzano le operazioni delle query.

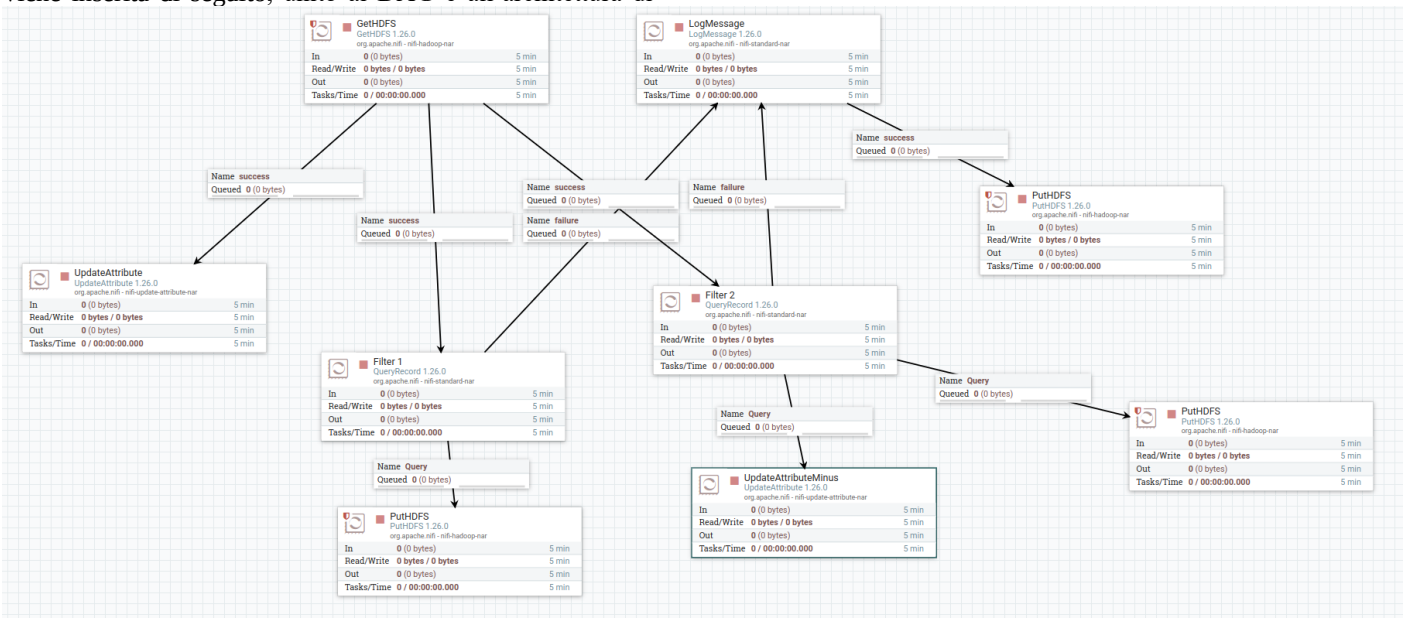

Figura 1: NiFi template

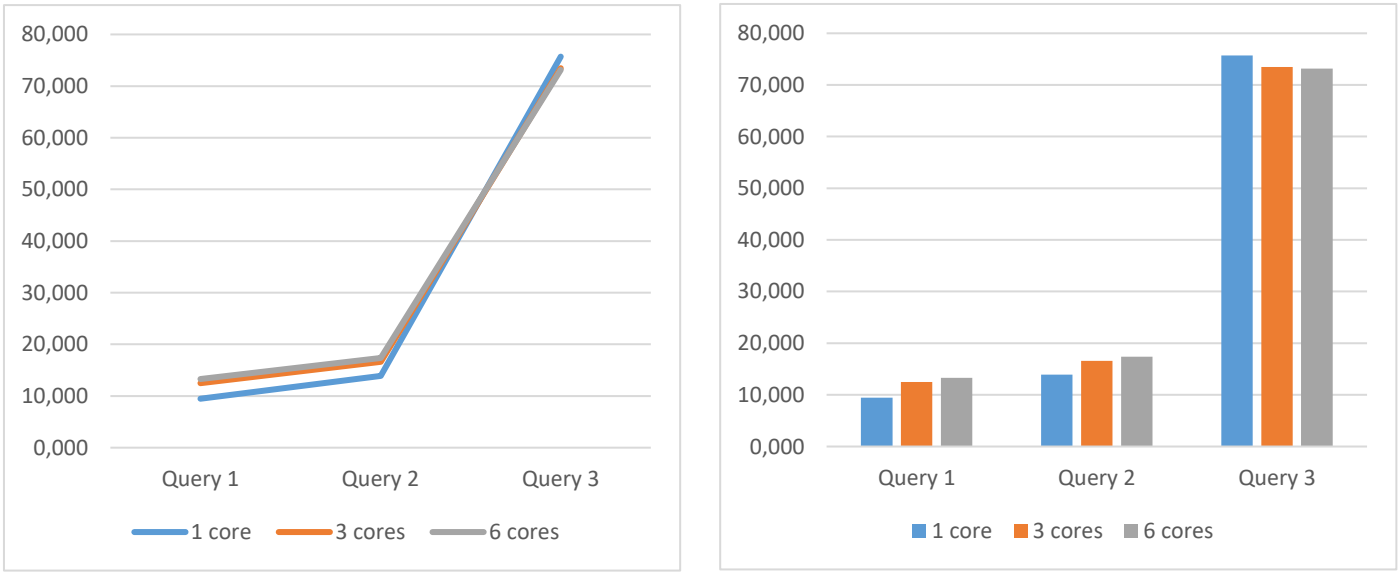

Figura 2: grafici comparativi query

Figura 3: Grafana dashboard

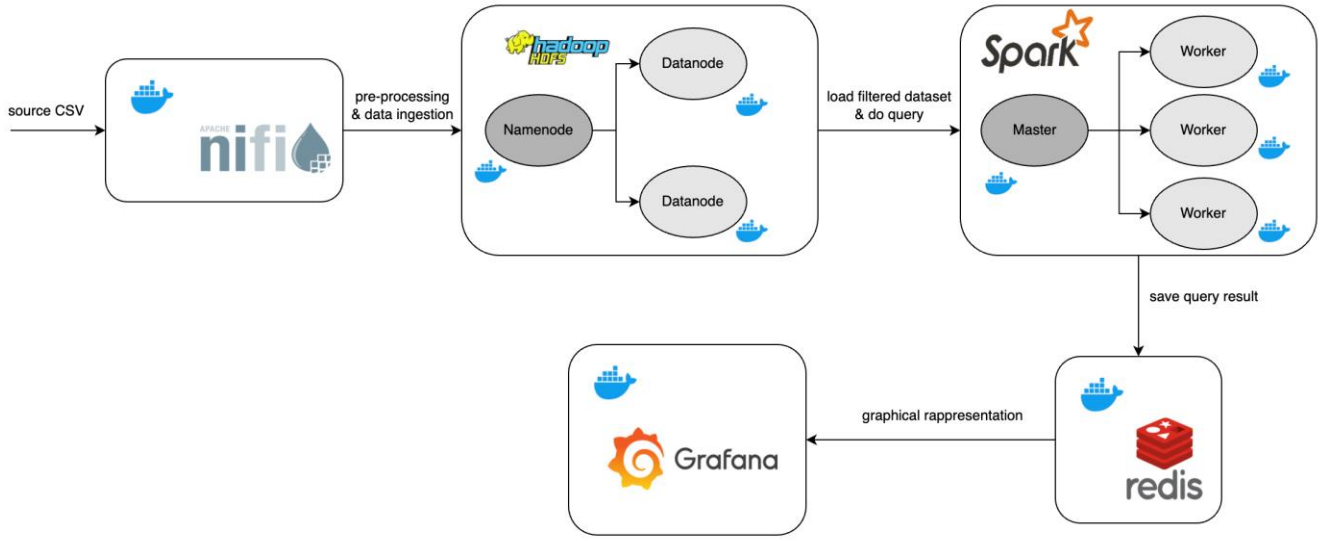

Figura 4: Architettura del sistema

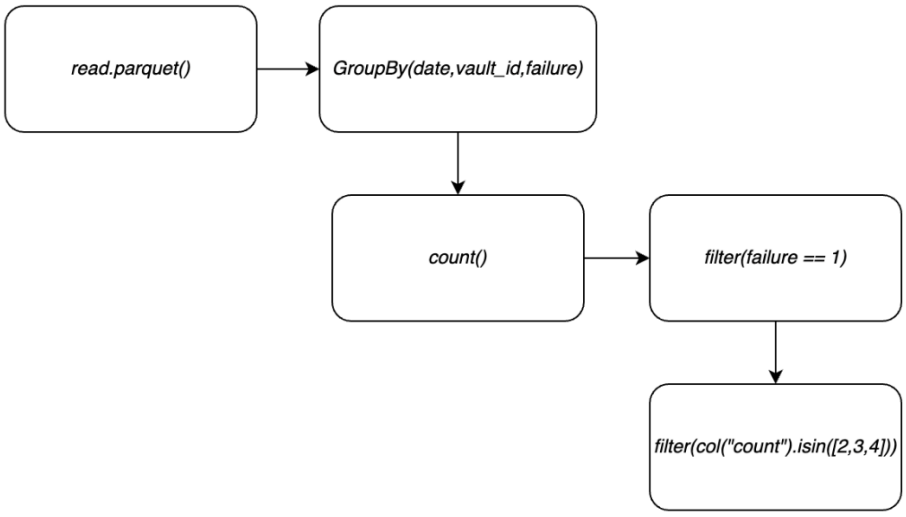

Figura 5: DAG riassuntivo query 1

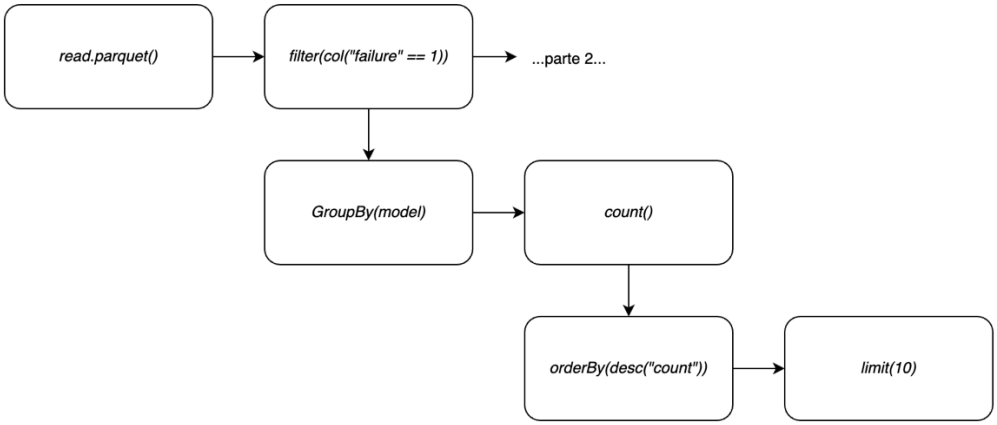

Figura 6: DAG riassuntivo query 2 parte 1

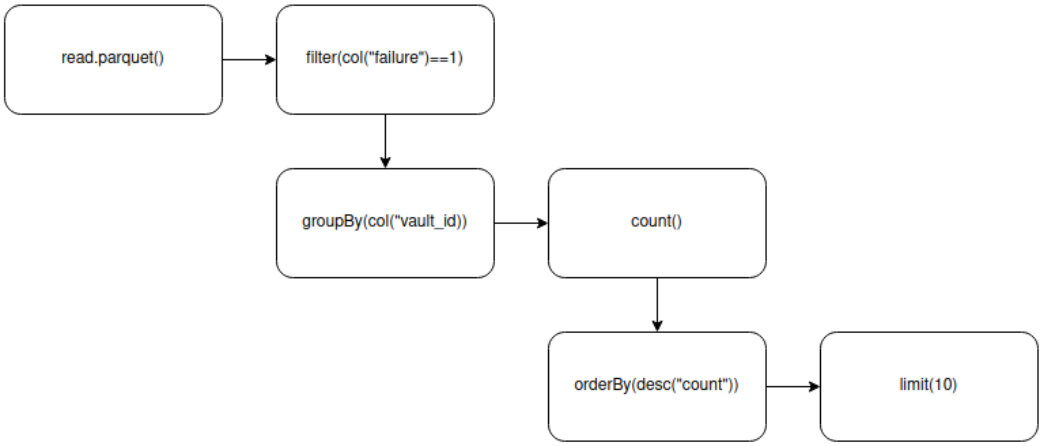

Figura 7: DAG riassuntivo query 2 parte 2

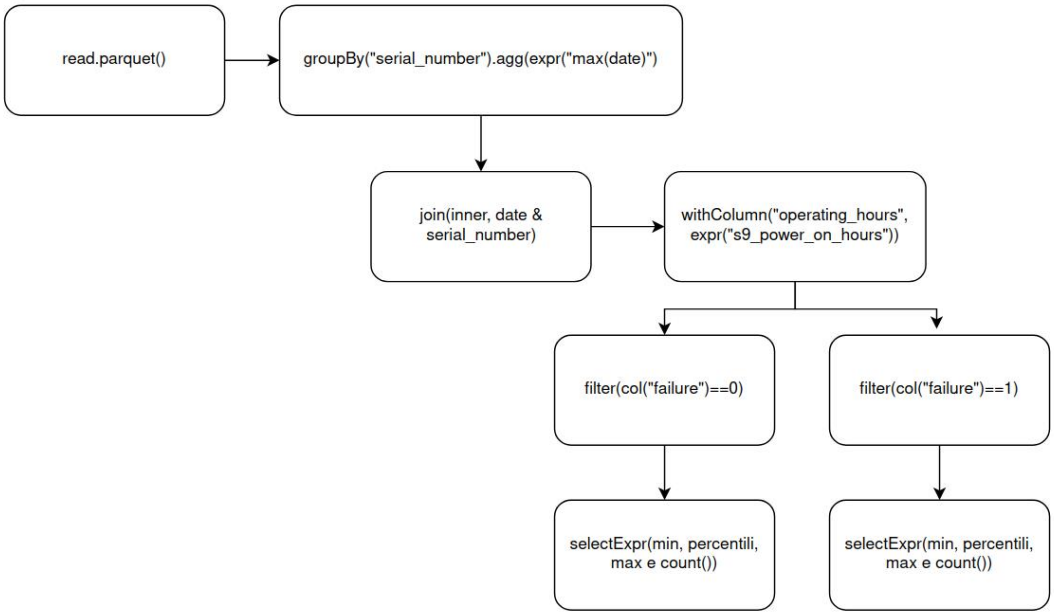

Figura 8: DAG riassuntivo query 3