SABD – Progetto 1

Esecuzione di Query Spark su dataset Blackblaze – S.M.A.R.T

Pieraldo Santurro
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
santurro.pieraldo99@gmail.com

ABSTRACT

L'obiettivo del progetto è quello di analizzare dei dati forniti da Blackblaze relativi alla telemetria di circa 200k hard disk in contenuti in alcuni data center. Per gli scopi del progetto si utilizza una versione ridotta del dataset indicato nel Grand Challenge della conferenza ACM DEBS 2024. Il dataset, inizialmente fornito in formato csv, viene convertito in formato Parquet, filtrato e caricato, tutto tramite Nifi, in un cluster HDFS utilizzato come storage per mantenere sia il dataset originario che il risultato delle query. I risultati vengono prodotti tramite Apache Spark, con utilizzo di PySpark con API Dataframes e Spark SQL API, caricati in formato csv su HDFS e scaricati, sempre tramite Nifi, su File System Locale. Opzionalmente, durante l'esecuzione del programma principale dell'applicazione è possibile caricare i risultati su MongoDB e visualizzarli tramite MongoDB Compass.

1. INTRODUZIONE

Preventivamente all'esecuzione del progetto è necessario effettuare alcuni step:

- Se la directory del progetto viene scaricata da GitHub, è necessario inserire nella directory *dataset*, il dataset originario in formato csv e rinominato *'dataset.csv'*.
- Se la directory del progetto viene ottenuta con altri modi, verificare se è presente il file *'dataset.csv'* all'interno della cartella *'dataset'*. Nel caso non ci sia, agire come nel punto sopra.

Per ulteriori informazioni è possibile consultare il *README.md* file all'interno della directory principale del progetto. Per inizializzare il progetto aprire un terminale nella cartella principale ed eseguire il comando '*./start_project.sh*'. Questo eseguibile è responsabile del flusso totale dell'applicazione ed andrà chiuso solamente una volta terminata l'esecuzione. Le query che andranno eseguite sono:

1. Per ogni giorno, per ogni vault (si faccia riferimento al campo vault id), calcolare il numero totale di fallimenti. Determinare la lista di vault che hanno subito esattamente 4, 3 e 2 fallimenti.
2. Calcolare la classifica dei 10 modelli di hard disk che hanno subito il maggior numero di fallimenti. Laclassifica deve riportare il modello di hard disk e il numero totale di fallimenti subiti dagli hard disk di quello specifico modello. In seguito, calcolare una seconda classifica dei 10 vault che hanno registrato il maggior numero di fallimenti. Per ogni vault, riportare il numero di fallimenti e la lista (senza ripetizioni) di modelli di hark disk soggetti ad almeno un fallimento.
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 che hanno subito fallimenti e degli hard disk che non hanno subito fallimenti. Si presti attenzione, il campo s9 power on hours riporta un valore cumulativo, pertanto le statistiche richieste dalla query devono far riferimento all'ultimo giorno utile di rilevazione per ogni specifico hard disk (si consideri l'uso del campo serial number). Nell'output indicare anche il numero totale di eventi utilizzati per il calcolo delle statistiche.

Il documento è strutturato secondo punti relativi alla descrizione di componenti e frameworks utilizzati nell'applicazione, alla realizzazione delle query, all'analisi delle prestazioni ed infine alcune considerazioni finali sul sistema.

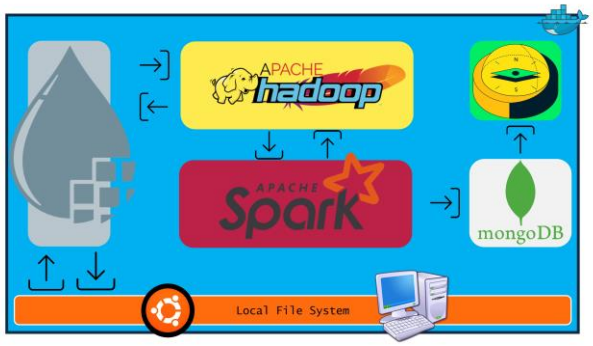

Figura 1- Architettura di Sistema

Durante l'esecuzione dello script principale verranno aperti più terminali per il controllo dei flussi, in particolare il terminale di esecuzione del compose file non va assolutamente chiuso fino alla fine dell'utilizzo del progetto. Inoltre, verranno richiesti i permessi di accesso massimi per la cartella nifi, dato che altrimenti avrebbe problemi nel comunicare con HDFS.

2. ARCHITETTURA E IMPLEMENTAZIONE

Si utilizzano diversi framework che verranno fatti comunicare tra loro al fine di ottenere i risultati richiesti. L'esecuzione principale è sul filesystem locale governata e gestita dalla virtualizzazione in ambiente docker dei container che andremo a creare. Ogni framework viene quindi hostato su un container docker ed è capace di comunicare con gli altri container grazie alla rete virtuale sostenuta da docker compose. I componenti architetturali del sistema sono:

- Nifi (apache/nifi:latest)
- Hadoop File System: Immagini 'Master' e 'Worker' create ad hoc per questa applicazione basata sull'immagine docker ufficiale matnar/hadoop, modificando la versione alla 3.4.0 e, nell'immagine Master, modificando il file di bootstrap per fare in modo che il container HDFS venga acceso all'inizializzazione del container. Viene creata preventivamente una cartella 'dataset' all'interno di HDFS per favorire l'inserimento successivo tramite Nifi.
- Apache Spark (bitnami/spark:latest)
- MongoDB (mongo:latest)

2.1 Nifi

Apache Nifi è un framework di data ingestion che permette di recuperare ed aggregare i dati provienienti da diverse fonti per caricarli all'interno dei vari frameworks che formeranno la pipeline di processamento. La particolarità di Nifi risiede nella sua capacità di poter effettuare del pre-processamento dei dati prima dell'ingestion. Queste funzionalità verranno utilizzate nell'applicazione sia nel caricamento del dataset su HDFS, effettuando una conversione in Parquet ed un filtraggio delle colonne superflue, sia download dei risultati da HDFS al file system locale, andando a rinominare i risultati e salvandoli in apposite cartelle. Il flow file di Nifi deve essere manualmente caricato, come indicato nello script principale del progetto, solo dopo che verrà copiato il dataset sorgente nel container '*input_data*' di Nifi stesso, operazione gestita anch'essa dallo script principale del programma. A quel punto è possibile caricare il flow file in formato .xml che si trova nella cartella nifi. Il flow file è diviso in due parti:

1. Nella prima parte c'è il raggruppamento relativo all'Upload dei file dalla cartella 'input_data' del nostro container Nifi sul file system locale all'HDFS nella cartella 'dataset'. Come già detto verrà effettuata una conversione di formato, con conseguente cambio di nominazione del file in 'dataset.parquet' per facilitare la lettura del programma, ed un filtraggio delle colonne superflue. Per il corretto funzionamento è necessario abilitare le proprietà di conversione di uno dei processori dell'upload. Una volta che l'abilitazione è stata effettuata, sarà possibile far partire il flow iniziale di upload su HDFS. Alla fine del processo, su HDFS sarà presente, nella cartella 'dataset' il file 'dataset.parquet' che avrà una dimensione di circa 50Mb.
2. Nella seconda parte c'è il raggruppamento relativo al Download dei risultati da HDFS al file system locale tramite il volume di Nifi 'output_data'. Per il corretto funzionamento è fortemente consigliato far partire i raggruppamenti di processori uno alla volta al fine di non appesantire il processo totale. Il flow file per il resto è abbastanza esemplificativo, con i raggruppamenti che indicano quale risultato andiamo a scaricare. La struttura dei raggruppamenti dei processori è abbastanza simile. In ognuno di essi viene scaricato dalla cartella HDFS corretta il risultato di una delle tre quary, il risultato viene poi rinominato, ed infine salvato sul file system locale.

Una volta terminata la fase di download sarà possibile visualizzare i dati ottenuti in formato csv all'interno della cartella 'nifi/output_data/'.

2.2 Hadoop File System

Si usa HDFS per due scopi nell'architettura:

- Mantenere il dataset filtrate e modificato da Nifi
- Scrittura dei risultati delle query eseguite con Apache Spark

La costruzione dell'HDFS è stata effettuata modificando l'immagine ufficiale Docker di *matnar/hadoop*, utilizzando due build diverse per il nodo master e per i nodi worker. La scelta è stata effettuata in modo tale da poter agevolmente inizializzare l'intero cluster direttamente alla creazione del container docker grazie alla modifica del file di bootstrap del nodo master. Il suddetto file è stato modificato con comandi HDFS per fare in modo di inizializzare il cluster durante l'invocazione da parte del Dockerfile del nodo Master, creando una cartella '*dataset'* nella quale verrà inserito da Nifi il file *dataset.parquet* da passare poi a Spark. Un'altra modifica, fondamentale nella build del Dockerfile sia dei nodi worker che dei nodi master, è stata quella della versione di Hadoop, passando alla più recente 3.4.0

2.3 Apache Spark

Per l'esecuzione delle Query sul dataset si utilizza Apache Spark con le API di Pyspark; quindi, il progetto è stato sviluppato utilizzando l'ambiente Python, fornendo una maggiore chiarezza nella progettazione e nell'esecuzione delle query. Le API utilizzate per la risoluzione sono quelle dei Dataframes[1] e quelle di Spark SQL. L'architettura è composta da un nodo master e tre

nodi worker, inizializzati e virtualizzati tramite docker compose secondo l'immagine ufficiale docker *bitnami/latest*.

2.3.1 Implementazione

Per tutte e tre le query richieste, sia nel caso di utilizzo di Dataset che nell'utilizzo di SQL, si inizializza il codice copiando, tramite comando docker, nel container master il codice relativo all'esecuzione delle query utilizzando un volume temporaneo, per poi essere sottomesso a Spark ed eseguito. Per tutte le query è possibile modificare prima dell'esecuzione il numero di core a disposizione di Spark, cambiando il valore di *NUM_COR* nello script *'Scripts/spark_exec-sh'*. Il salvataggio dei risultati avverrà su una cartella di HDFS creata da Spark al momento del salvataggio, diversa a seconda della query in esecuzione. Il file verrà salvato in formato csv.

2.4 MongoDB

Per il salvataggio dei risultati delle query su un datastore NoSQL si utilizza MongoDB. Sempre tramite docker, utilizziamo MongoDB in single node utilizzando l'immagine *mongo*. Il caricamento dei risultati avviene mediante uno script separato che utilizza sempre Apache Spark ed i suoi connettori a MongoDB, leggendo i dati risultanti scritti su HDFS nelle relative cartelle direttamente su MongoDB. L'esecuzione di tale scrittura, come già detto, è resa opzionale nello script generale del progetto, al fine di appesantire il meno possibile l'esperienza dell'utente utilizzatore. Per la visualizzazione delle query su MongoDB si è utilizzato MongoDB Compass, una desktop interface sviluppata da Mongo di facile utilizzo che ci consente di connetterci alla porta di MongoDB e, oltre alla semplice visualizzazione, ci permette di interagire e modificare i dataset risultanti.

3. QUERY

Vediamo di seguito le scelte effettuate durante la progettazione delle query. Vedremo inoltre il DAG risultante di ciascuna query con il relativo numero di jobs, ed un'analisi sui tempi di esecuzione, nella sezione Appendice è possibile vedere un'astrazione dei DAG generati dal codice. È possibile modificare prima dell'esecuzione delle query, nel relativo codice di ciascuna query, i valori *print_intermediate = True* e *visualize_dag = True* per rispettivamente permettere la visualizzazione intermedia dei vari passi delle query e permettere la visualizzazione del DAG tramite la porta 4040 di Spark. Le misuarazioni temporali sono fatte dividendo il tempo necessario al quering del dataset e del salvataggio dei risultati dal tempo necessario alle operazioni sulle query; per semplicità, verranno riportate nelle apposite sezioni i dati medi di misurazione di Apache Spark, visualizzabili liberamente sulla porta 8080, in quanto strettamente vicini ai dati misurati nel codice. L'analisi seguente terrà in maggiore considerazione l'implementazione in dataframes API, dato che per la realizzazione delle query in SQL si è effettuata una sempice traduzione della logica degli script in python. I DAG verranno analizzati nelle sezioni seguenti tenendo conto del fatto che Spark, tramite il caching delle operazioni e l'utilizzo di funzioni che ottimizzano le operazioni a tempo di esecuzione, saranno abbastanza difficili da inquadrare in una logica chiara. L'analisi dei tempi di esecuzione è stata effettuata, sia per le query in dataframes che per le controparti in SQL, per un valore di uno, due e tre cores a disposizione di Spark, nella sezione Appendice è possibile visualizzare i grafici relativi all'analisi temporale effettuata.

3.1 Query 1

Per ogni giorno, per ogni vault (si faccia riferimento al campo vault id), calcolare il numero totale di fallimenti. Determinare la lista di vault che hanno subito esattamente 4, 3 e 2 fallimenti.

3.1.1 Implementazione Dataframes

Si inizializza la sessione Spark, nominando l'applicazione per fare in modo che sia facilmente riconoscibile durante l'eventuale visualizzazione sulla Web UI di Spark. Si inizializza un contatore di tempo, che utilizzeremo per l'analisi temporale sull'interrogazione alla Query. Si esegue la prima azione del programma, creando il primo dataframe leggendo il dataset dall'HDFS. Si trasforma il primo membro delle colonne *failure* e *vault_id* in interi per continuare il trattamento dei dati, utilizzando un *cast* con un'operazione *withColumn*. Questo problema è nato probabilmente dalla conversione in Parquet fatta da Nifi ed è ricorrente nell'utilizzazione di conversione effettuata in questo modo, come vedremo anche in seguito. Si è deciso poi di utilizzare una funzione di un'altra libreria chiamata *datetime* al solo fine di scrivere i risultati della query in un formato "*dd-MM-yyyy"*. Avviene poi il conteggio dei fallimenti per ogni giorno e per ogni vault, utilizzando il dataframe *failures_df* e creando un nuovo dataframe *failures_count_df*, utilizzando metodi *groupBy* e *count*. Viene poi effettuata una *filter* per selezionare solamente i vaults con 4,3 e 2 fallimenti. Viene effettuato il salvataggio dei dati su HDFS, nella cartella '*results1/*', convertendo il dataframe risultante *sorted_df*, dataframe ottenuto ordinando per data il dataframe risultante dalle operazioni di conteggio e filtraggio precendenti, in formato csv utilizzando il metodo *coaelsce(1)* per fare in modo che il file non venga partizionato al momento del salvataggio. Avviene poi la visualizzazione dei tempi impiegati dal progetto per essere eseguito. È presente la linea di codice opzionale per evitare la chiusura della spark session e visualizzare il DAG risultante sulla porta 4040. Viene infine chiusa la spark session.

3.1.2 Implementazione SQL

Una volta prelevato il dataset in formato Parquet verrà effettuata un'operazione di SELECT dove preleveremo le colonne con valore di *failure* pari ad uno, sempre castando il relativo valore ad intero dati i problemi della traduzione in Parquet. Come nella Query in dataframes, verrà formattato il dator relativo alla data in modo che sia visualizzato nel formato *dd-MM-yyy*; si eseguirà poi un'altra operazione di SELECT per contare i fallimenti per ogni giorno e per ogni vault. Il risultato verrà poi ordinato e salvato su HDFS.

3.1.3 DAG - Dataframes

Ottengo dal mio programma 3 Jobs, come ci aspettiamo dal numero di azioni effettuate all'interno della query spark, con un totale di 8 stage dove alcuni di questi vengono skippati da Spark. Leggendo da varie fonti[2], è perfettamente normale avere una visualizzazione di DAG di questo tipo quando si utilizzando Dataframes, dato che, essendo un'astrazione da RDD, non è detto che i metodi che utilizziamo e che sembrano trasformazioni non invochino in realtà azioni RDD all'interno delle funzioni. In figura è stato inserito il DAG relativo all'ultimo job eseguito; gli stages skippati in grigio sono gli stage già eseguiti da alti jobs, i quali risultati vengono tenuti in cache da Spark per non effettuare ulteriori chiamate. Avere stages skippati viene considerata un'operazione benigna effettuata da Spark, dato che migliora i tempi di esecuzione. Nel nostro caso i due stages skippati sono quelli già eseguiti dai Jobs precedenti, quindi Spark non ha un ritardo di esecuzione nel considerarli nuovi stages per poi skipparli nel job finale.

3.1.4 Tempi di esecuzione

Per un totale di cinque esecuzioni abbiamo:

- Un core:
  - Dodici secondi di media per dataframes
  - Venti secondi di media per SQL
- Due core:
  - Quindici secondi di media per dataframes
  - Ventisei secondi di media per SQL
- Tre core:
  - Venti secondi di media per dataframes
  - Trentadue secondi per SQL

3.1.5 Analisi dei tempi

Dalla documentazione, è perfettamente normale aspettarsi un valore più elevato nella controparte SQL, così come lo sarebbe nel caso in cui ci fossero tempi sovrapponibili, dato che Dataframes API e SQL API in Spark sfruttano lo stesso engine che esegue le dovute operazioni di ottimizzazione. Per quanto riguarda il maggior tempo impiegato dall'esecuzione con più core, e quindi anche più memoria a disposizione, il motivo potrebbe essere che la gestione del parallelismo delle operazioni su più worker nel nostro cluster introduca un ritardo dovuto proprio alla presenza di più core di quanti effettivamente ne servano: il numero di jobs generato dalla nostra query è infatti molto ridotto rispetto alle possibilità di Spark, di conseguenza un numero minore di core, a differenza di quanto ci si possa aspettare, introduce un ritardo nell'esecuzione del programma[3].

3.1.6 Risultati ottenuti

Sia per Dataframes che per SQL abbiamo:

date,vault_id,failure_count
04-04-2023,1066,2
04-04-2023,1113,3
05-04-2023,1010,2
05-04-2023,1014,2
05-04-2023,1032,3
05-04-2023,1041,2
05-04-2023,1044,2
05-04-2023,1093,3
07-04-2023,1033,2
07-04-2023,1133,2
08-04-2023,1124,2
08-04-2023,1400,2
11-04-2023,1036,2
11-04-2023,1040,3
11-04-2023,1090,3
11-04-2023,1092,2
11-04-2023,1093,2
12-04-2023,1096,2
12-04-2023,1113,3
14-04-2023,1120,4
14-04-2023,1124,2
14-04-2023,1128,2
14-04-2023,1161,2
15-04-2023,1113,2
15-04-2023,1118,2
17-04-2023,1053,2
17-04-2023,1066,2
17-04-2023,1113,2
18-04-2023,1033,2
18-04-2023,1113,2
18-04-2023,1118,2
19-04-2023,1091,2
20-04-2023,1113,2
20-04-2023,1132,2
21-04-2023,1127,2

Figura 2-Risultati Query 1

3.2 Query 2

Calcolare la classifica dei 10 modelli di hard disk che hanno subito il maggior numero di fallimenti. La classifica deve riportare il modello di hard disk e il numero totale di fallimenti subiti dagli hard disk di quello specifico modello. In seguito, calcolare una seconda classifica dei 10 vault che hanno registrato il maggior numero di fallimenti. Per ogni vault, riportare il numero di fallimenti e la lista (senza ripetizioni) di modelli di hark disk soggetti ad almeno un fallimento

3.2.1 Implementazione Dataframes

Lo script di esecuzione delle due query richieste come risultato è racchiuso tutto nello stesso eseguibile python, quindi all'interno del programma si eseguiranno fondamentalmente due query che avranno risultati diversi che andranno scritti uno ciascuno in una apposita cartella HDFS. Ci si riferirà di seguito a query 1 per indicare la prima query della seconda richiesta e query 2 per la seconda query della seconda richeista. Si inizializza la sessione Spark rinominando l'applicazione. Si legge il dato Parquet passato in input. Come nella query 1 si fa una trasformazione delle colonne *failure* e *vault_id* in interi per migliorare l'esecuzione del programma. Poi avviene l'inizio di esecuzione per la query1, calcolando per il dataframes *failures_count* il numero totale di fallimenti per ogni modello di Hard Disk, utilizzando i metodi *groupBy* per raggruppare per modello e *agg* per aggregare i risultati del numero di fallimenti al dataframe. Si utilizza nel dataframe successivo *sorted_failures* il metodo *orderBy* con l'aggiunta di altri metodi per riordinare il dataframe in ordine discendente e prendere i primi 10 valori. In seguito, si scrivono i risultati per la query1 su HDFS nella cartella '*results2.1/*'. Viene catturato infine il tempo di esecuzione per la query 1. Per la query

2 si inizia filtrando in un dataframes solo le colonne con valore di *failures* pari ad 1, quindi solo gli Hard Disk che hanno registrato un fallimento. A questo punto in un nuovo dataframe *vault_failures* viene effettuata una *groupBy* per vault_id e vengono aggregati utilizzando metodi come *count e collect_set* i valori, rinominati per semplicità nell'output, di *failure e model*. A questo punto avviene una conversione in comma separated strings che ci permette di avere nello stesso campo di colonna più modelli. Anche in questo caso poi avviene un ordinamento discendente dei primi 10 valori ottenuti ed una conseguente conversione in csv e scrittura su HDFS nella cartella '*results2.2/*'. Avviene poi la visualizzazione delle tempistiche a schermo e la chiusura della spark session. Anche qui è possibile de-commentare le due linee di codice per visualizzare agevolmente il DAG.

3.2.2 Implementazione SQL

Verrà prelevato il dataset in formato parquet dalla sorgente HDFS e verranno effettuate delle operazioni di SELECT, per la prima query, dove si selezioneranno gli hard disk con il maggior numero di fallimenti, e altre operazioni di SELECT per la seconda query dove verranno selezionati gli hard disk con il maggior numero di fallimenti per poi selezionarli in base ai modelli unici. Alla fine di ogni operazione su una delle due query avverrà la scrittura su HDFS.

3.2.3 DAG - Dataframes

Vengono generati 4 jobs nella nostra esecuzione, dove due riguardano la prima query e due la seconda, anche qui valgono le considerazioni fatte nella Query 1.

3.2.4 Tempi di esecuzione

Per un totale di cinque esecuzioni abbiamo:

- Un core:
  - Tredici secondi di media per dataframes
  - Ventuno secondi di media per SQL
- Due core:
  - Diciassette secondi di media per dataframes
  - Ventisei secondi di media per SQL
- Tre core:
  - Venti secondi di media per dataframes
  - Trentuno secondi per SQL

3.2.5 Analisi dei tempi

Come per la Query 1, sono risultati perfettamente in linea con l'esecuzione di Spark, così come il numero necessario di cores.

3.1.6 Risultati ottenuti

Sia per Dataframes che per SQL abbiamo:

model,failures_count
ST8000NM0055,50
HGST HUH721212ALN604,48
ST4000DM000,27
ST12000NM0008,25
ST8000DM002,22
TOSHIBA MG07ACA14TA,21
ST10000NM0086,10
HGST HMS5C4040BLE640,10
HGST HUH721212ALE604,10
WDC WUH721414ALE6L4,6

Figura 3 - Risultato Query 2.1

vault_id,total_failures,unique_models
1113,15,[HGST HUH721212ALN604]
1120,10,[HGST HUH721212ALN604]
1093,9,[ST10000NM0086]
1053,8,"[TOSHIBA MQ01ABF050M, ST8000NM0055]"
1118,8,[HGST HUH721212ALN604]
1032,7,[ST8000DM002]
1090,7,"[TOSHIBA MQ01ABF050, ST8000NM0055, WDC WD5000LPVX]"
1066,6,[TOSHIBA MG07ACA14TA]
1124,6,"[HGST HUH721212ALE604, HGST HUH721212ALN604]"
1055,6,[ST8000NM0055]

Figura 4 - Risultato Query 2.2

3.3 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 che hanno subito fallimenti e degli hard disk che non hanno subito fallimenti. Si presti attenzione, il campo s9 power on hours riporta un valore cumulativo, pertanto, le statistiche richieste dalla query devono far riferimento all'ultimo giorno utile di rilevazione per ogni specifico hard disk (si consideri l'uso del campo serial number). Nell'output indicare anche il numero totale di eventi utilizzati per il calcolo delle statistiche.

3.3.1 Implementazione - Dataframes

Per questa query utilizziamo delle operazioni su singolo dataframe per fare in modo di filtrare il dataset ottenendo solamente le ultime osservazioni effettuate sui dischi, vediamo nel dettaglio più avanti. Iniziamo facendo partire il contatore temporale e creando una spark session, modificando il nome anche di questa. Prendiamo il dato in formato parquet dall'HDFS e come fatto nelle prime due query facciamo un cast ad intero del campo *failure*. Viene creata una finestra di ripartizione[4], ripartizionando i dati per *serial_number* e *date* in ordine discendente:

```
# Define a partition window by serial_number ordered by date descendingc
window_spec = Window.partitionBy("serial_number").orderBy(col("date").desc())
```

Figura 5

Tramite l'aggiunta di una colonna *rank*, teniamo traccia dell'ultima osservazione effettuata per ciascun hard disk, utilizzando la finestra di osservazione creata precedentemente:

```
# Add a column with row number for each partition
data_with_rank = data.withColumn("rank", row_number().over(window_spec))

# Filter to keep only rows with rank 1 (latest observation for each hard disk)
latest_data = data_with_rank.filter(col("rank") == 1).drop("rank")
```

Figura 6

A questo punto, filtriamo il dataset in modo tale da tenere il record con la dicitura *rank=1*, cioè l'ultima osservazione effettuata, nel dataframe *failure_data*; in seguito filtriamo il dataframe in due dataframes differenti, uno per i *failures=1* e *failures=0*, in modo da poterli utilizzare per le operazioni successive. Adesso verrà creata una funzione per calcolare i dati statistici di un dataframe, utilizzando *percentile_approx*[5]. Il valore di *s9_power_on_hours* reale ha presentato lo stesso problema del valore di failure nelle precedenti query, di conseguenza il valore utilizzato è solo il '*member0*', cioè il valore vero dell'intero utilizzabile nell'esecuzione del programma. Viene creata una lista contenente i valori statistici cercati. A questo punto viene creato da zero un nuovo dataframe che contiene ciò che abbiamo trovato, per poi scriverlo su HDFS con le modalità utilizzate anche dalle altre query. Anche sono presenti le due linee di codice utilizzabili per visualizzare il DAG a runtime. Vengono stampate le statistiche necessarie e poi viene chiusa la spark session.

3.3.2 Implementazione - SQL

Come per la controparte in Dataframes, viene prelevato il dataset in formato Parquet e vengono eseguite operazioni di SELECT per aggiungere una colonna che tenga traccia dell'ultima misurazione effettuata e un ordinamento per data. Dopo aver effetttuato delle *filter* verranno calcolate le statistiche con la stessa funzione utilizzata per Dataframes. Il risultato verrà poi scritto ed aggregato nella stessa query e salvato su HDFS.

3.3.3 DAG - Dataframes

Dal DAG si vedono le operazioni effettuate preliminarmente alla creazione dell'ultimo dataframe. Data la struttura del programma, si vede che gli stages 4 e 5 e gli stages 10 e 11 sono molto probabilmente le stesse operazioni effettuate prima del filtraggio dei dati, che viene diviso per *failure = 1* e *failures = 0*, poi c'è la creazione del nuovo dataframe che verrà scritto su HDFS.

3.3.4 Tempi di esecuzione

Per un totale di cinque esecuzioni per i dataframes e tre esecuzioni per SQL abbiamo:

- Un core:
  - Trentadue secondi di media per dataframes
  - Circa un minuto di media per SQL
- Due core:
  - Trentacinque secondi di media per dataframes
  - Circa un minuto di media per SQL
- Tre core:
  - Quaranta secondi di media per dataframes
  - Circa un minuto di media per SQL

3.3.5 Analisi dei tempi

Come per le Query precedenti anche qui il numero giusto di core risulta essere uno; tuttavia i tempi per tutti i casi di differenza nel numero di core risultano essere più appiattiti rispetto alle Query precedenti, questo ci suggerisce che la vera potenza di Spark nell'esecuzione di operazioni in parallelo viene utilizzata per programmi che coinvolgono un alto numero di operazioni su dataset estesi, come ci si aspetterebbe da un framework del genere.

3.1.6 Risultati ottenuti

Sia per Dataframes che per SQL abbiamo

```
failure,min,25th_percentile,50th_percentile,75th_percentile,max,count
1,522.0,26898.0,38669.0,51965.0,71608.0,249
0,0.0,15119.0,22649.0,42061.0,87702.0,242661
```

Figura 7 - Risultato Query 3

4. CONCLUSIONI

Le principali difficoltà riscontrate sono state nella creazione e nella comunicazione dei vari framework utilizzati, in particolare per alcuni di loro non esistono vere e proprie community e le documentazioni sono spesso scarse. Il progetto potrebbe essere migliorato eventualmente aggiungendo un framework come Grafana per la graficazione dei risultati delle query; riscrivere le query in RDD per un confronto migliore dei risultati e dell'analisi temporale ed eventualmente confrontare il risultato dei DAG (Operazione effettuata solamente per la query 1 e non inserita nel progetto, solo per confronto); automatizzazione completa del setup di Apache Nifi.

5. RIFERIMENTI

[1] https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/core_classes.html

[2] https://stackoverflow.com/questions/68522035/why-is-spark-creating-multiple-jobs-for-one-action

https://community.databricks.com/t5/community-discussions/more-than-expected-number-of-jobs-created-in-databricks/td-p/54478

https://stackoverflow.com/questions/34580662/what-does-stage-skipped-mean-in-apache-spark-web-ui

[3] https://medium.com/analytics-vidhya/how-no-of-cores-and-amount-of-memory-of-the-executors-can-impact-the-performance-of-the-spark-276cf58900a3l

[4] https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-window.html

[5] https://spark.apache.org/docs/latest/api/sql/index.html#percentile_approx

6. APPENDICE: DAGs e Grafici temporali

6.1 DAG Query 1

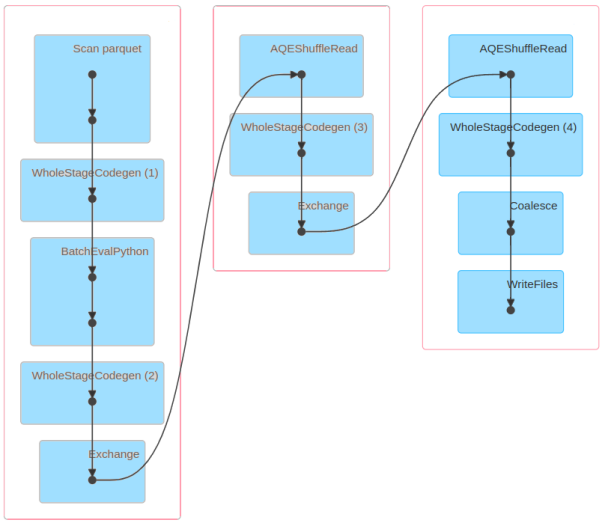

Figura 8 - DAG Query 1

6.2 DAG Query 2

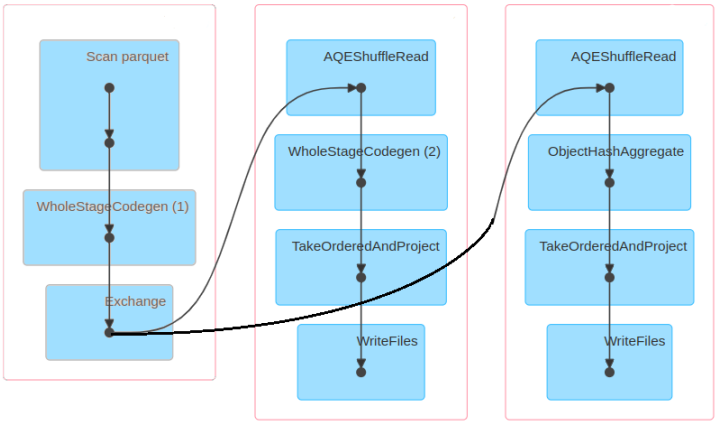

Figura 9 - DAG Query 2

6.3 DAG Query 3

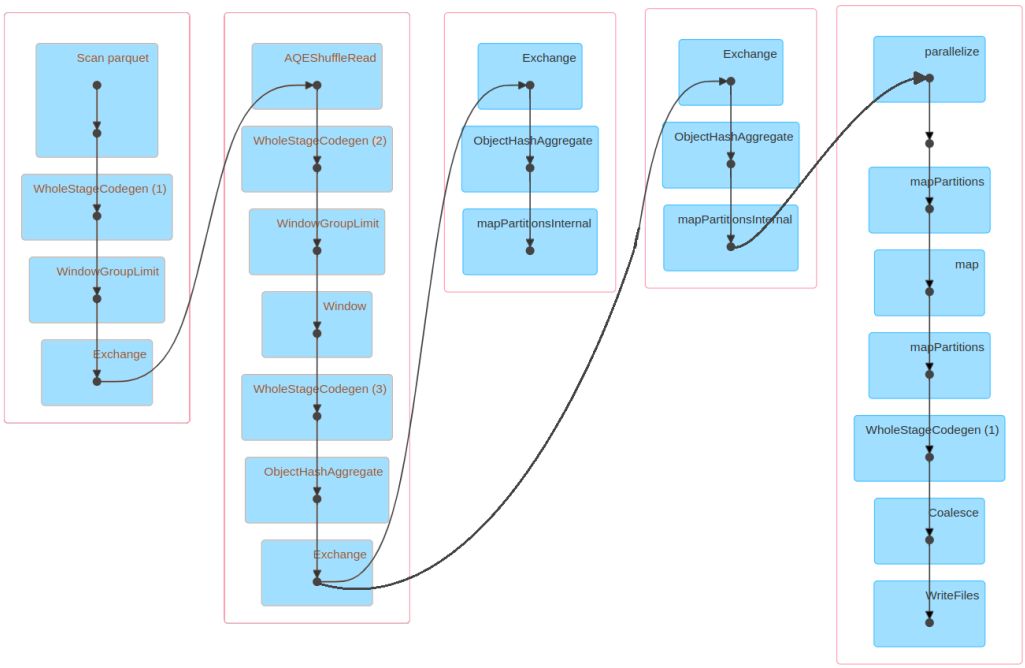

Figure 10 - DAG Query 3

6.4 Query Dataframes Diagramma Temporale

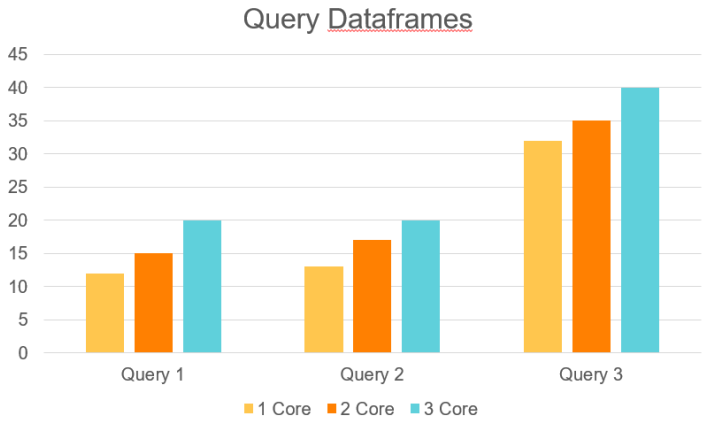

Figure 11 - Diagramma Temporale Query Dataframes

6.5 Query SQL Diagramma Temporale

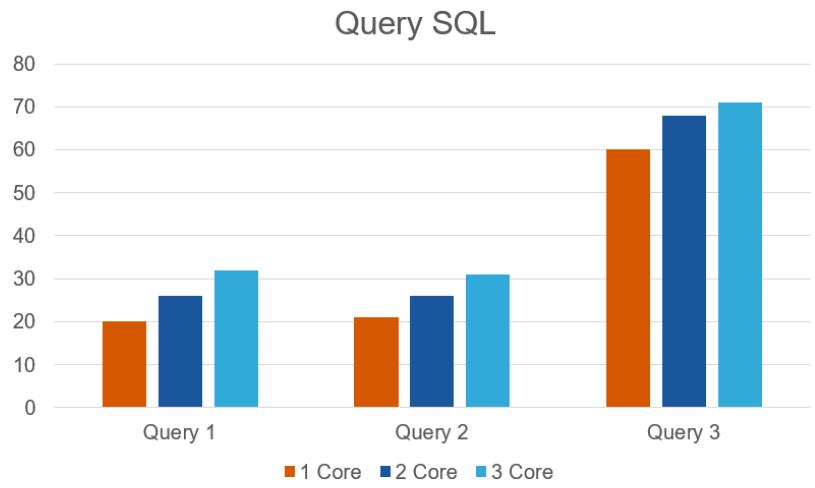

Figure 12 - Query SQL Diagramma Temporale

6.6 Confronto temporale tra Query

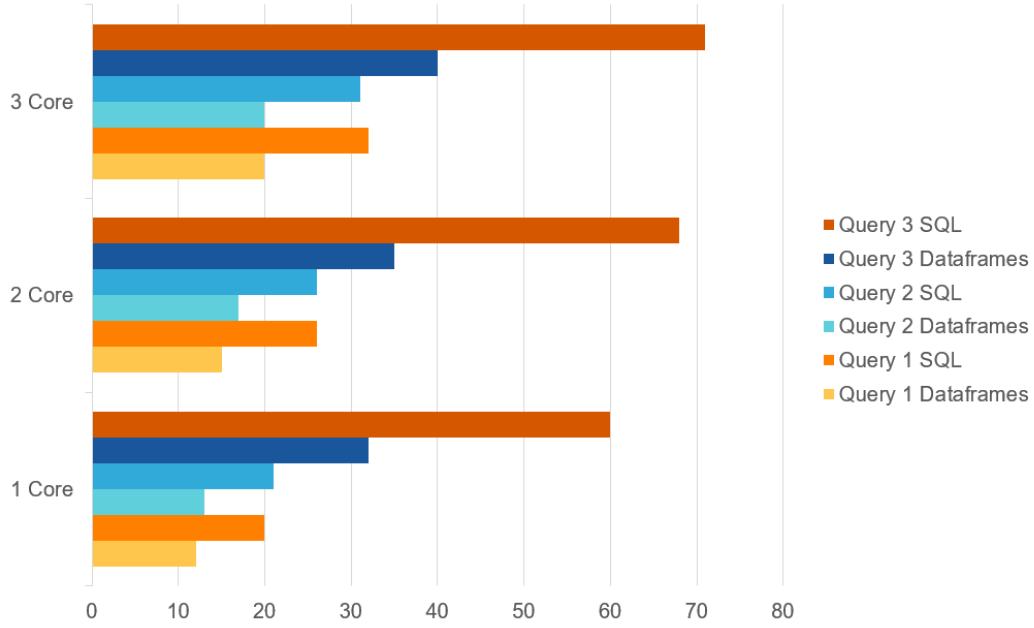

Figure 13 - Confronto temporale tra Query