Analisi di dati di monitoraggio con Apache Spark

Sistemi e Architetture per Big Data - Progetto 1

Alessandro Lioi
0333693

***Abstract*—Lo scopo di questo documento è quello di descrivere l'architettura della pipeline e l'implementazione di due query in Apache Spark su dati di telemetria di hard disk gestiti da Backblaze**

I. Introduzione

Il progetto consiste nello svolgimento di due query tramite batch processing di dati di telemetria forniti da Backblaze riguardanti circa 200k hard disk.

II. Architettura Pipeline

La pipeline (Figure 1) descrive il seguente percorso dei dati:

1) si caricano i dati su HDFS
2) si esegue pre-processing su Apache NiFi, salvando i risultati su HDFS
3) si esegue batch processing su Apache Spark (tramite API RDD e DataFrame)
4) si salvano i risultati su MongoDB o HDFS
5) si caricano i dati da MongoDB su Grafana per creare delle dashboard interattive

Ogni componente dell'architettura viene eseguito in un container Docker, con Docker Compose come orchestratore. Per HDFS si hanno 1 NameNode e 3 DataNode, mentre per Spark si hanno 1 master e 3 worker.
Le immagini usate per ogni container Docker sono quelle ufficiali (forniti direttamente da Docker o dallo sviluppatore del prodotto, come per il caso di Grafana o l'Apache Software Foundation per NiFi) ad eccezione dell'immagine di HDFS per il quale si è preferito usare l'immagine pre-configurata fornita dal professore Nardelli. È stato inoltre aggiunto un container per gestire tramite una Web-UI il data store MongoDB.

III. Data Ingestion e Pre-processing

Come framework per data ingestion e pre-processing è stato usato Apache NiFi In particolare, il flow definito tramite Web-UI (Figure 2) ha lo scopo di leggere l'intero dataset da HDFS nel formato CSV, eseguire pre-processing sui dati, convertire in vari formati e salvare il risultato nuovamente su HDFS.
Per poter svolgere questa fase di pre-processing, sono stati quindi usati i seguenti **processor**:

- **GetHDFS** per leggere il dataset originale da HDFS
- **QueryRecord** per poter selezionare le colonne di interesse per le query
- **ConvertRecord** per poter convertire da CSV in vari formati
- **UpdateAttribute** per poter dare nomi diversi ai vari file di output
- **PutHDFS** per poter salvare i dataset risultanti su HDFS

La query usata nel processor **QueryRecord** è la seguente:

```
SELECT SUBSTRING("date" FROM 0 FOR 11)
AS "date",
serial_number,model,failure,vault_id
FROM FLOWFILE
WHERE failure <> 'failure'
```

La query legge dal `FLOWFILE` (output del processore precedente, clausola `FROM`), filtra le righe che non sono di header (ogni 600k righe è presente l'header, clausola `WHERE`) e seleziona le colonne utilizzate dalle query (clausola `SELECT`). Nella selezione delle colonne inoltre si applica una trasformazione del campo `date`, ovvero si prende la sottostringa corrispondente al formato `YYYY-mm-DD` (granularità dei giorni). I formati in cui viene convertito il dataset sono: Apache Avro, Apache Parquet e CSV. Non è stato possibile convertire anche in Apache ORC, in quanto il processore di conversione non supporta ufficialmente questo formato. Per aggiungere il supporto ad Avro su Spark, è stato necessario aggiungere il pacchetto `org.apache.spark:spark-avro` nella sezione `packages` di `spark-submit`
Su HDFS vengono quindi salvati i dataset pre-preocessati nei formati precedenti e il dataset completo, senza pre-processing per poter effettuare la fase di pre-processamento direttamente su Apache Spark.
La fase di pre-processamento effettuata da Spark consiste nel filtrare le colonne necessarie per rispondere alle due query e convertire il formato della data in `YYYY-mm-DD`:

- si prende una sottostringa, nel caso di RDD
- si applica la funzione `to_date`, nel caso dei DataFrame

A seguito del processamento ogni tupla del dataset è nel seguente formato:

date	serial_number	model	failure	vault_id

A seguito di questo pre-processamento, non è necessario filtrare ulteriormente le righe del dataset poichè non risultano valori nulli o mancanti rilevanti per le query di interesse. Ciò è stato analizzato tramite l'esecuzione di `df.summary().show()`

IV. Query 1

La Query 1 richiede di calcolare il numero totale di fallimenti per ogni giorno, per ogni vault e determinare la lista dei vault con esattamente 4, 3 o 2 fallimenti

A. RDD

1) si mappa ogni riga in una tupla key-value con:
   - **key**: tupla `(date, vault_id)`
   - **value**: `failure`
2) si esegue una **reduceByKey**, ottenendo quindi una tupla `((date, vault_id), count)`
3) si filtrano le tuple in base al valore di `count`, come richiesto (quindi pari a 4, 3 o 2)
4) si esegue una **map** per "appiattire" la chiave, ottenendo quindi una tupla `(date, vault_id, count)`
5) si ordinano le tuple in modo decrescente per `count` e crescente per `date` e `vault_id`

DAG: Figure 3

B. DataFrame

1) si scartano le colonne `serial_number` e `model`, in quanto non servono per rispondere alla query
2) si raggruppa, tramite **groupBy**, in base a `(date, vault_id)`
3) si esegue un'aggregazione con funzione di somma rispetto alla colonna `failure`, tramite **agg(sum)**, salvando la colonna risultante con l'alias `count`
4) si filtrano i dati in base al valore di `count`, come richiesto dalla query
5) si ordinano i risultati con lo stesso odinamento descritto nella sezione RDD

V. Query 2 Ranking 1

Per la prima parte della seconda query è necessario identificare i 10 modelli con il maggior numero di fallimenti

A. RDD

1) si esegue una **map** per avere una tupla `(model, failure)`
2) si esegue una **reduceByKey**, ottenendo quindi `(model, failures_count)`
3) si ordinano i risultati per `failures_count` decrescente
4) si trasforma l'RDD risultante in DataFRame, limitando i risultatu a 10 per ottenere la classifica richiesta

Viene effettuata la trasformazione in DataFrame per poter assegnare i nomi alle colonne, facilitando la successiva scrittura su HDFS o MongoDB. DAG: Figure 4

B. DataFrame

1) si scartano le colonne non necessarie, quindi `date`, `serial_number` e `vault_id`
2) si esegue una **groupBy** per `model`
3) si esegue una aggregazione con funzione di somma, tramite **agg(sum)** sulla colonna `failure`, salvando la colonna risultante con l'alias `failures_count`
4) si ordinano i risultati nello stesso modo del caso di RDD
5) si limita la dimensione a 10

VI. Query 2 Ranking 2

Per la seconda parte invece è necessario identificare i 10 vault con il maggior numero di fallimenti e i modelli, appartenenti a questi vault, con almeno un fallimento

A. RDD

Si deve calcolare il numero di failures per ogni vault:

1) si esegue una **map** per ottenere la tupla `(vault_id, model)`
2) si esegue quindi una **reduceByKey** con funzione di somma

Si identifica poi la lista di modelli associati ad ogni vault:

1) si esegue una **filter** sulle tuple con `failure` maggiore di 0
2) si esegue una **map** per ottenere la tupla `(vault_id, model)`
3) si esegue una **groupByKey** per ottenere la tupla `(vault_id, list_of_models)`
4) si esegue una **mapValues** con funzione `set` per rimuovere le ripetizioni nella lista dei modelli

Si può quindi ottenere la classifica eseguendo una **join**.
Per poter salvare correttamente il risultato in formato CSV, si esegue una **map** eseguendo la funzione di **join** sulla lista dei modelli con separatore ",".
Si esegue infine una **sortBy** per ordinare per `failures_count` decrescenti e si limitano i risultati a 10
DAG: Figure 5

B. DataFrame

In maniera analoga al caso di RDD, si calcolano il numero di failures per ogni vault:

1) si scartano le colonne non necessarie, quindi `date`, `serial_number` e `model`
2) si esegue una **groupBy** sulla colonna `vault_id`
3) si esegue una aggregazione con funzione di somma, tramite **agg(sum)**, sulla colonna `failure`

Si calcolano quindi i modelli associati ad ogni vault:

1) si scartano le colonne non utilizzate: `date` e `serial_number`
2) si filtrano i modelli con almeno una failure
3) si raggruppano in base a `vault_id`, tramite **groupBy**
4) si esegue un'aggregazione con funzione `collect_set`

Per ottenere la classifica si esegue una left-**join** in base a `vault_id` e si ordinano con `failures_count` decrescente.
Come per il caso degli RDD, si concatena la lista di modelli tramite `concat_ws` con separatore "," e si limitano i risultati a 10

VII. Salvataggio Risultati

L'applicazione permette di salvare il risultato delle query su HDFS o sul data store NoSQL basato su documenti MongoDB. Il salvataggio su HDFS avviene nel formato CSV, con una copia salvata sul filesystem locale.

Per quanto riguarda MongoDB la scrittura dei risultati viene effettuata da Spark tramite il connettore ufficiale sul database `spark`. Ogni query viene salvata nella propria collezione in cui ogni documento rappresenta una riga del CSV di output. Per caricare il connettore di MongoDB, è stato aggiunto il pacchetto `org.mongodb.spark:mongo-spark-connector` come `package` aggiuntivo del comando `spark-submit`

VIII. Grafana

Si utilizza Grafana come strumento per poter visualizzare in maniera interattiva i risultati delle query. Grafana è stato connesso a MongoDB tramite un plugin sviluppato dalla community (poichè il plugin ufficiale è riservato solo agli utenti con licenza enterprise). Una volta connesso MongoDB a Grafana, sono state create 3 dashboard, una per query. A causa di una limitazione del plugin utilizzato, è stato necessario definire delle query di tipo aggregazione per interfacciarsi con MongoDB (`db.collection.aggregate()`). La configurazione dei datasource e delle dashboard avviene in maniera automatica tramite il sistema di provisioning fornito da Grafana.

A. Query 1

La dashboard riguardante la Query 1 (Figure 6) presenta una time-series che mostra sulle ascisse i giorni e sulle ordinate il numero di failure per ogni vault.
Tramite la definizione di una variabile nella dashboard, è possibile filtrare in base al Vault ID. Essendo una time-series, Grafana mette a disposizione la possibilità di estendere o ridurre la dimensione della finestra temporale visualizzata.
Query Variabile: Listing 1
Query Dashboard: Listing 2

B. Query 2 Ranking 1

La seconda dashboard (Figure 7) presenta invece un bar chart con il nome del modello sulle ascisse e il numero di failure sulle ordinate. Come per la Query 1, è stata definita una variable per poter filtrare in base al modello.
Query Variabile: Listing 3
Query Dashboard: Listing 4

C. Query 2 Ranking 2

Anche per la seconda classifica della Query 2 (Figure 8) si ha un bar chart con il Vault ID sulle ascisse e il numero di failures sulle ordinate. In questo caso sono state definite due variabili per filtrare in maniera interattiva: vault ID e nome del modello.
Query Variabile Vault ID: Listing 1
Query Variabile Modelli: Listing 5
Query Dashboard: Listing 6

IX. Performance

L'analisi delle performance è stata guidata dalla volontà di identificare l'API e il formato di storage più efficiente al variare del numero di executors. Per poter decidere a priori il numero di executors, è stato necessario passare dei parametri di configurazione aggiuntivi a `spark-submit`:

- `spark.executor.cores=1`: si limita ad 1 core per executors
- `spark.cores.max=`: si imposta dinamicamente con quanti core eseguire il job di Spark

Il formato (Figure 9) con i tempi di caricamento più brevi risulta essere il CSV pre-processato, nonostante il formato con peso minore risulta essere Parquet.
Per quanto riguarda il confronto sui tempo di esecuzione (Figure 10 e Figure 11), l'API con le performance migliori è DataFrame con il formato Parquet, probabilmente grazie al lavoro svolto dal Catalyst Optimizer. È interessante notare come all'aumentare del numero di executors, non si hanno performance migliori. Ciò potrebbe essere causato dall'overhead aggiuntivo dato dalla necessità di gestire i vari tasks e di scambiare dati tra più executors.
Nonostante ciò le query eseguite con gli RDD mostrano delle performance migliori all'aumentare del numero di executors e che, probabilmente, aumentandone il numero, si potrebbero raggiungere le performance dei DataFrame. Ciò potrebbe essere causato da un partizionamento migliore del dataset sui vari executors, che limita il numero di operazioni di comunicazione tra di essi.

X. Riferimenti e Manuale d'Uso

Il codice sorgente è disponibile su GitHub
L'esecuzione dell'intera pipeline si divide in due passi:

- esecuzione di `./scripts/setup.sh`: script in Bash che:
  - avvia i container di Docker tramite Docker Compose
  - formatta e avvia HDFS
  - crea le cartelle necessarie su HDFS, assegnando i permessi giusti per poter essere accedute da NiFi e Spark
  - copia il dataset dal filesystem locale su HDFS
  - avvia il master e i workers di Spark
  - esegue il flow di NiFi, avviando uno script scritto in Python che automatizza il caricamento e l'esecuzione del flow
- esecuzione di `./scripts/run.sh` che esegue `spark-submit` e accetta 3 possibili sotto-comandi:
  - **save**: esegue entrambe le query nella configurazione con le performance migliori con l'obiettivo di salvare l'output su HDFS o MongoDB (specificato con un parametro aggiuntivo)
  - **analysis**: esegue tutte le possibili configurazioni, stampando su schermo i risultati; accetta il numero di executors come parametro aggiuntivo
  - **check**: esegue una configurazione specifica su entrambe le API per controllare l'uguaglianza delle implementazioni

  Il dataset non è stato caricato sul repository di GitHub ma ci si aspetta che sia disponibile nella cartella `data` con il nome `dataset.csv`

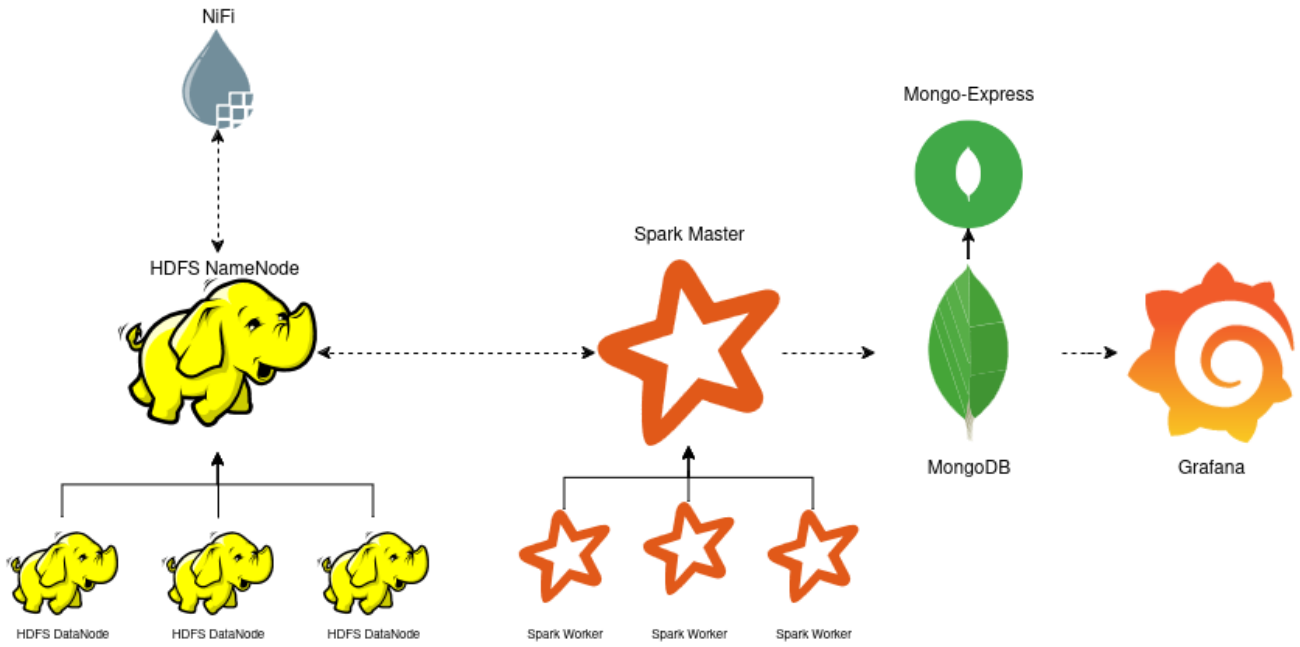

Fig. 1. Architettura

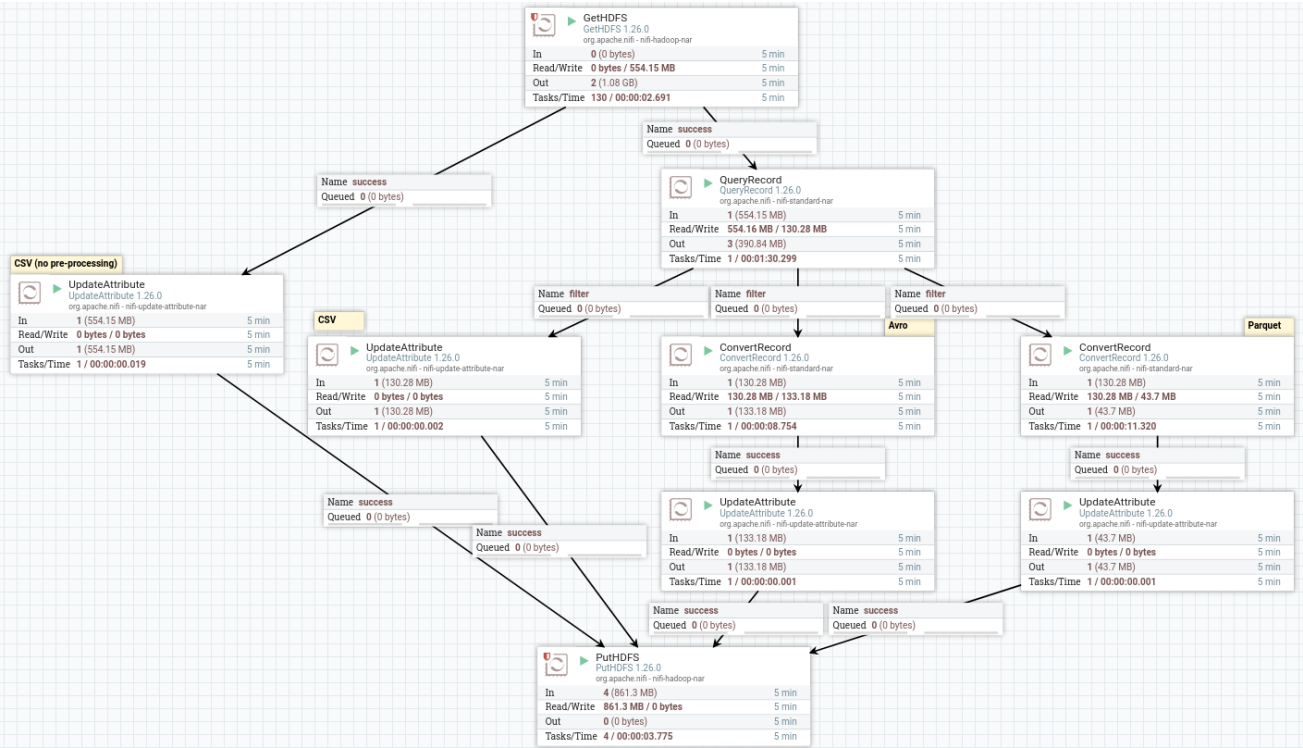

Fig. 2. NiFi Pipeline

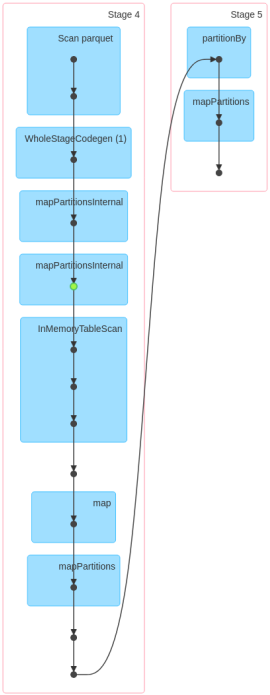

Fig. 3. DAG Query 1

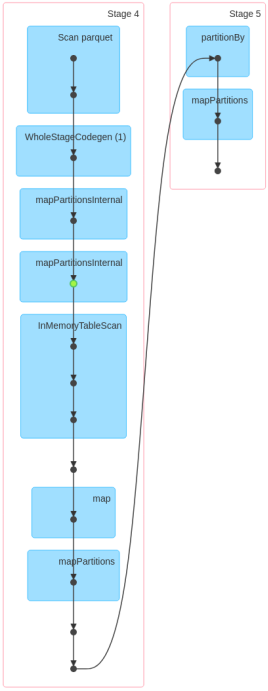

Fig. 4. DAG Query 2 Ranking 1

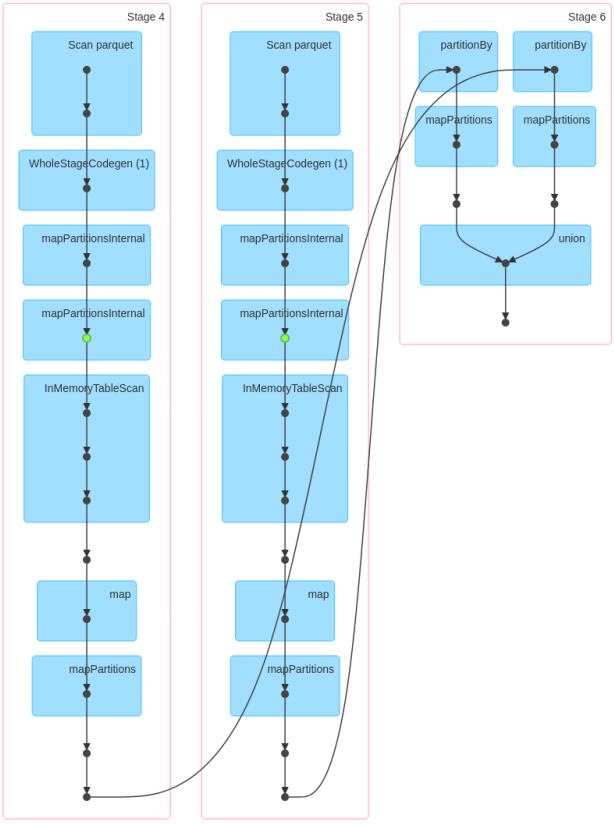

Fig. 5. DAG Query 2 Ranking 2

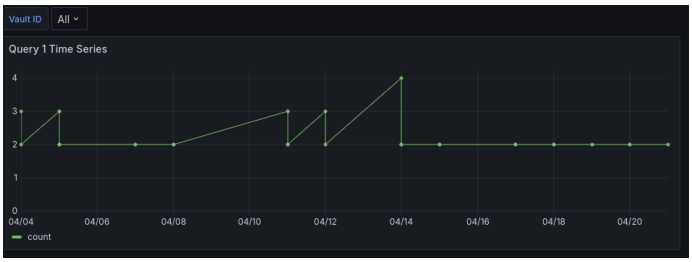

Fig. 6. Grafana Dashboard Query 1

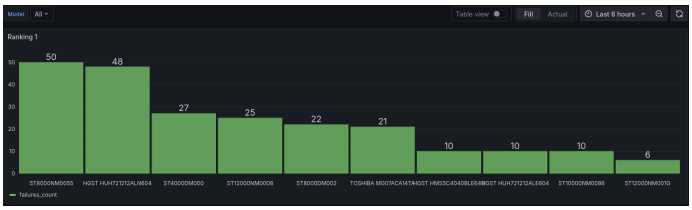

Fig. 7. Grafana Dashboard Query 2 Ranking 1

Fig. 8. Grafana Dashboard Query 2 Ranking 2

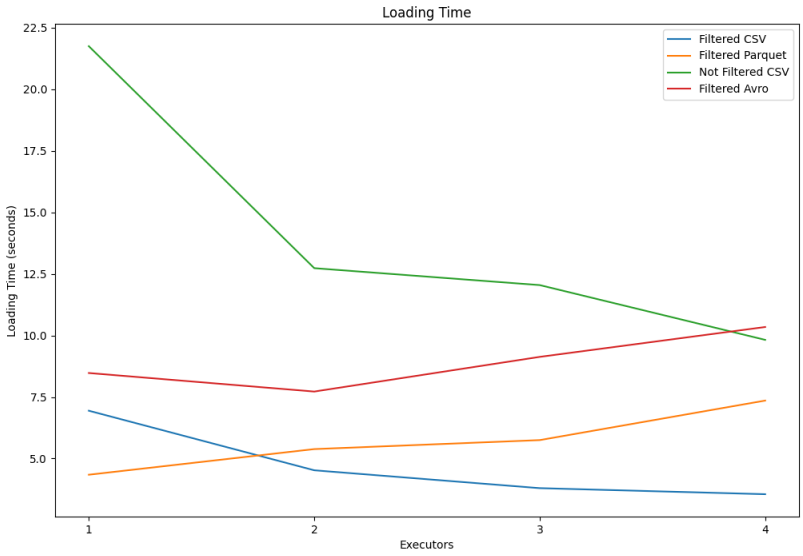

Fig. 9. Loading Time File Formats

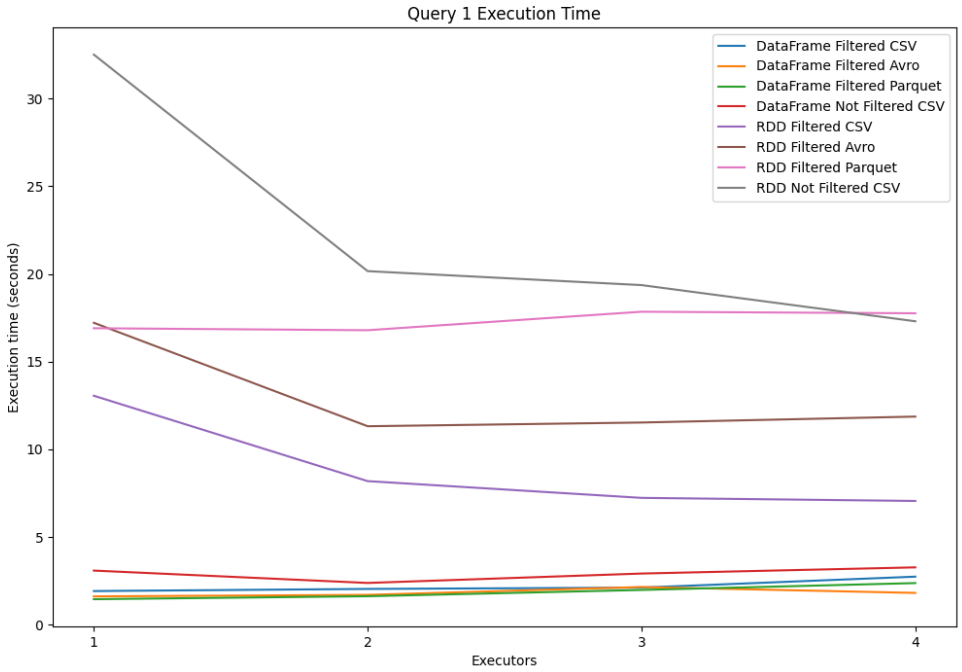

Fig. 10. Execution Time Query 1

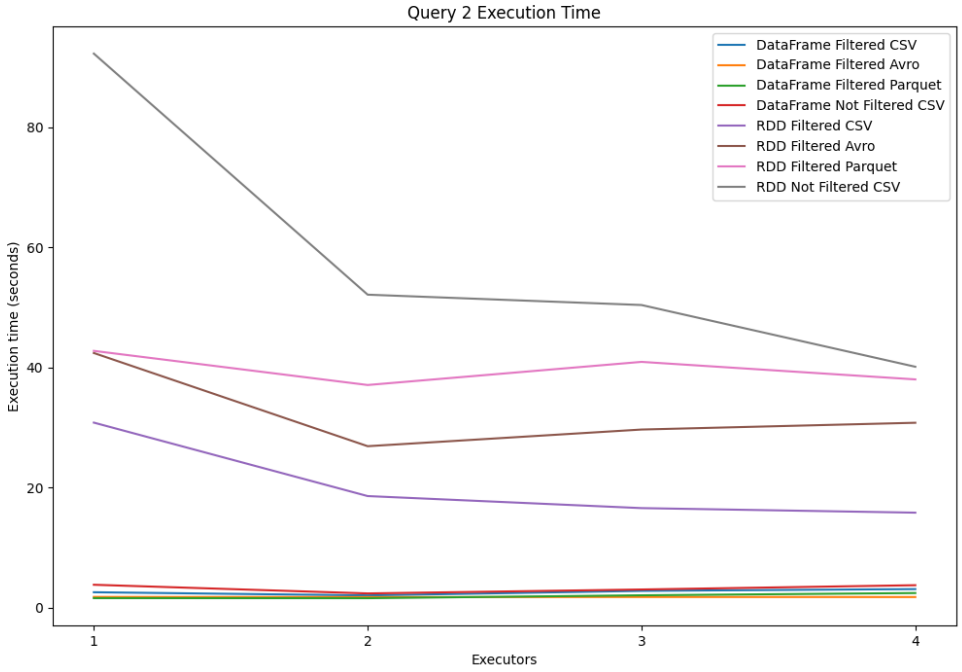

Fig. 11. Execution Time Query 2

```
[
  {
    "$group": { "_id": "$vault_id" }
  },
  {
    "$project": { "_id": "$_id" }
  }
]
```

Listing 1. Grafana Query 1 Variable

```
[
  {
    "$match": {
      "vault_id": { "$in": [${vaults:csv}] }
    }
  },
  {
    "$project": {
      "_id": 0,
      "count": 1,
      "date": 1,
      "vault_id": 1
    }
  },
  {
    "$project": { "count": 1, "date": 1 }
  }
]
```

Listing 2. Grafana Query 1

```
[
  {
    "$group": { "_id": "$model" }
  },
  {
    "$project": { "_id": "$_id" }
  }
]
```

Listing 3. Grafana Query 2 Ranking 1 Variable

```
[
  {
    "$match": {
      "model": { "$in": ${models:json} }
    }
  },
  {
    "$project": {
      "_id": 0,
      "failures_count": 1,
      "model": 1
    }
  }
]
```

Listing 4. Grafana Query 2 Ranking 1

```
[
  {
    "$addFields": {
      "list_of_models": {
        "$split": ["$list_of_models", ","]
      }
    }
  },
  {
    "$unwind": "$list_of_models"
  },
  {
    "$group": { "_id": "$list_of_models" }
  },
  {
    "$project": { "_id": 1 }
  }
]
```

Listing 5. Grafana Query 2 Ranking 2 Variable Model

```
[
  {
    "$addFields": {
      "list_of_models": {
        "$split": [ "$list_of_models", "," ]
      }
    }
  },
  {
    "$match": {
      "vault_id": { "$in": [${vaults:csv}] },
      "list_of_models": { "$in": ${models:json} }
    }
  },
  {
    "$project": {
      "_id": 0,
      "vault_id": 1,
      "failures_count": 1,
      "list_of_models": 1
    }
  }
]
```

Listing 6. Grafana Query 2 Ranking 2