Sistemi e Architetture per i Big Data

Presentazione primo progetto

~ Andrea Cantarini

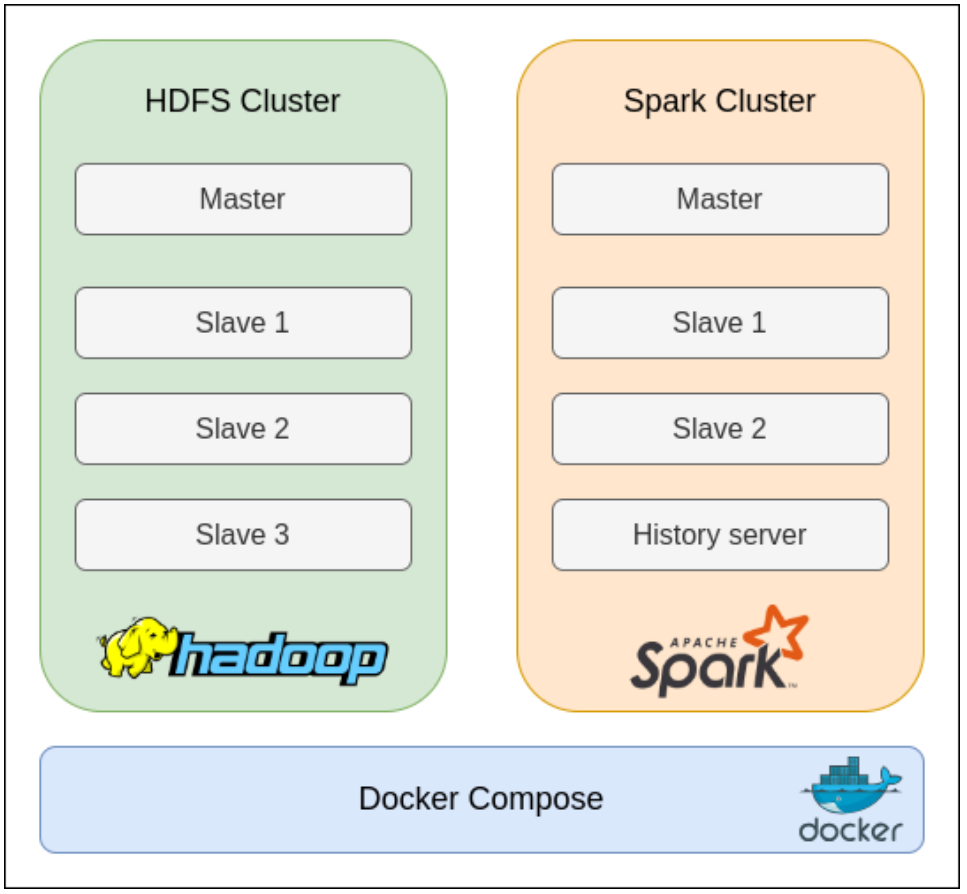
Traccia

- È richiesto l’utilizzo del framework Apache Spark per eseguire delle interrogazioni su un dataset contenente eventi relativi a circa 200k hard-disk

Traccia

- 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.
- 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 hard disk soggetti ad almeno un fallimento.

Architettura



Query 1

```
// Discard row without failures
.filter(row -> row._4() != 0)

// Map each row in a ((timestamp, vault_id), 1) tuple
.mapToPair(row -> new Tuple2<>(new Tuple2<>(row._1(), row._5()), 1L))

// Sum failures for each (timestamp, vault_id) couple
.reduceByKey(Long::sum)

// Take only entries with 2, 3 or 4 failures
.filter(row -> row._2() >= 2 && row._2() <= 4)

// Map the result to a (timestamp, vault_id, failures_count) tuple
.map(row -> new Tuple3<>(row._1()._1(), row._1()._2(), row._2()))

// Sort the result by timestamp
.sortBy(Tuple3::_1, true, 1)

// Collect the results
.collect();
```

Query 2 - 1

```
// Discard row without failures
.filter(row -> row._4() != 0)

// Map each row to (model, 1) pair
.mapToPair(row -> new Tuple2<>(row._3(), 1L))

// Sum failures count by model
.reduceByKey(Long::sum)

// Swap columns to take ordered results
.mapToPair(row -> new Tuple2<>(row._2(), row._1()))

// Sort by failures count
.sortByKey(false)

// Take TOP_N (10) models with more failures
.take(TOP_N);
```

Query 2 - 2

```
// Discard row without failures
.filter(row -> row._4() != 0)

// Map each row to a (vault_id, (1, model)) pair
.mapToPair(row -> new Tuple2<>(row._5(), new Tuple2<>(1L, new
                    ArrayList<>(Collections.singletonList(row._3()))))

// Sum failures count and concatenates models (unique) names
.reduceByKey((Function2<Tuple2<Long, ArrayList<String>>, Tuple2<Long,
        ArrayList<String>>, Tuple2<Long, ArrayList<String>>>) (t1, t2) -> {
                t1._2().addAll(t2._2());
                return new Tuple2<>(t1._1() + t2._1(),
                        new ArrayList<>(new HashSet<>(t1._2())));
            })

// Map each row to a (failures_count, (vault_id, models_list)) pair
.mapToPair(row -> {
        StringBuilder sb = new StringBuilder();
        row._2()._2().forEach(model -> sb.append(model).append(" "));
        new Tuple2<>(row._2()._1(), new Tuple2<>(row._1(), sb.toString()));
})

// Sorting, mapping and taking top 10
```

Data ingestion

```
// Load data from HDFS and take only the first five columns
JavaRDD<String> fileContent =sparkSession
       .read()
       .textFile(FILE_PATH)
       .javaRDD();

JavaRDD<Tuple5<String, String, String, Long, Long>> value =
    fileContent
        .map(CsvParser::parseCSV)
        .filter(Objects::nonNull)
        .map(x -> new Tuple5<> (
           x.getTimestamp(),
           x.getSerialNumber(),
           x.getModel(),
           x.getFailure(),
           x.getVault_id()
       )
);
```

Scrittura dei risultati

- Si sfruttano i risultati di ogni query, ottenuti sottoforma di List, per creare un oggetto Dataset e scrivere tramite esso su HDFS
- Esempio per la prima query

```
Dataset<Tuple3<String, Long, Long>> query1Dataset =
    sparkSession.createDataset(converted, Encoders.tuple(Encoders.STRING(),
        Encoders.LONG(), Encoders.LONG()));

query1Dataset.repartition(1)
                .withColumnRenamed("_1", "DD-MM-YYYY")
                .withColumnRenamed("_2", "vault_id")
                .withColumnRenamed("_3", "failures_count")
                .write()
                .option("header", true)
                .format("com.databricks.spark.csv")
                .save("hdfs://master:54310/out/out_1.csv");
```

Automatizzazione

- Vengono forniti degli script per avviare e fermare l’architettura e alcuni servizi, nonché caricare il dataset su hdfs ed eseguire l’applicazione
- Esempio

```
./Scripts/run.sh        // Avvia l’architettura
```

```
./deploy.sh             // Avvia l’applicazione
```

Criticità

- Non è stato possibile misurare i tempi di processamento delle singole query, ma solo dell'applicazione completa (~22s dalla Spark Web UI)
- Mancando un livello di data injection, in questo tempo è incluso il tempo di lettura e scrittura dei dati da e verso HDFS
- RDD rende il codice complesso e di facile lettura e manutenzione, pertanto si può optare per una soluzione che sfrutti la libreria Spark SQL, migliorando così anche i tempi di processamento

Grazie per l’attenzione