Architettura per il processamento distribuito di Big Data

Usando Apache Spark e Apache Hadoop

Andrea Paci
Università degli studi di Roma
Tor Vergata
Roma
andrea.paci1998@gmail.com

Alessandro Amici
Università degli studi di Roma
Tor Vergata
Roma
a.amici@outlook.it

ABSTRACT

La seguente trattazione mira a presentare l'architettura e la logica di processamento sviluppate per l'elaborazione di Big Data all'interno di un cluster.

Verrà innanzitutto presentata l'architettura, descrivendo brevemente ogni singolo componente, per poi entrare nel dettaglio e presentare tutto il sistema motivando le varie scelte implementative.

Infine, verranno presentati con una breve analisi statistica i risultati ottenuti in seguito al processamento dei dati di input. A questo seguiranno due brevi paragrafi riguardanti i possibili sviluppi futuri e le difficoltà riscontrate durante lo sviluppo.

SCOPO DEL PROGETTO

Il focus principale del progetto è quello di sviluppare un'architettura per effettuare **batch processing** per grandi moli di dati. In particolare, il sistema ha il compito di rispondere ad alcuni quesiti riguardanti le *vaccinazioni in Italia per il COVID19* al fine di ricavare dati statistici ed informazioni utili per descrivere la campagna di vaccinazione in Italia.

I dati riguardanti le vaccinazioni si distribuiscono temporalmente tra il 27 dicembre 2020 ed il 31 maggio 2021; la fonte è *Opendata Vaccini*[1]. Nonostante la mole di dati in questo caso non giustifichi l'impiego di framework atti all'elaborazione e lo storage di Big Data, le scelte progettuali e le considerazioni fatte durante lo sviluppo non tengono conto di questo fattore ma simulano un contesto dove è necessario avere questo tipo di infrastruttura.

I quesiti a cui è necessario rispondere sono 3:

1. Computare per ogni regione e per ogni mese solare quante vaccinazioni vengono somministrate mediamente ogni giorno in un qualsiasi centro vaccinale di una specifica regione.
2. Per ogni mese solare e per ogni fascia di età, effettuare una classifica delle prime 5 regioni che si stimi effettuino più somministrazioni per le donne il primo giorno del mese successivo.
3. Per ogni regione, stimare il numero di somministrazioni totali al 1 giugno 2021 e, attraverso l'impiego di algoritmi di Machine Learning, effettuare il clustering delle regioni basandosi sulla percentuale di popolazione vaccinata per ognuna di queste.

Questi 3 quesiti, insieme a possibili altri, possono fornire un' analisi quantitativa sull'andamento della pandemia, infatti tutta l'architettura non è limitata alla risposta di quest'ultimi ma è possibile estenderla per estrapolare altri dati.

ARCHITETTURA

L'architettura utilizzata per la risposta dei 3 quesiti comprende un insieme di framework e tecnologie orientate ai Big Data, interconnessi opportunamente tra loro.

L'architettura è composta da 3 componenti principali:

- **Apache Hadoop,** con all'interno **Apache HBase,** per lo *storage* ed il *processamento* distribuito

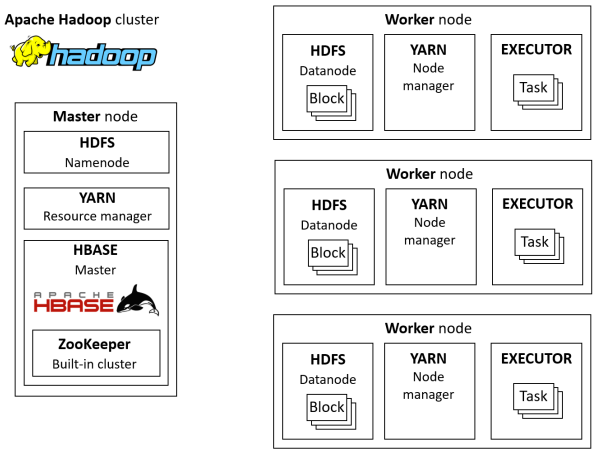


Figura 1: Schematizzazione del cluster Apache Hadoop

- **Apache NiFi** per la *data ingestion* e per il *preprocessamento* del *dataset*.

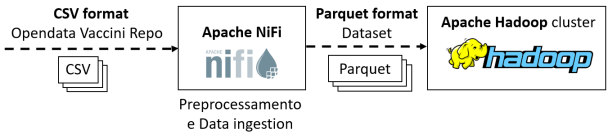


Figura 2: Schematizzazione dell'impiego di Apache Nifi

- **Apache Spark** come framework per il *batch processing* in un cluster.

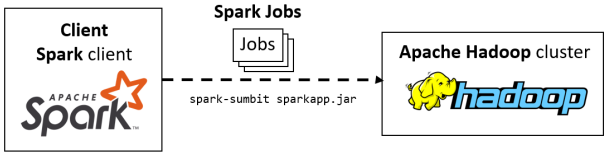


Figura 3: Schematizzazione dell'impiego di Spark

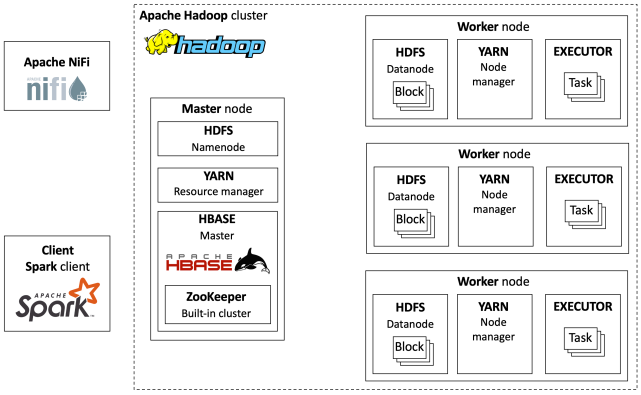


Figure 4: Schematizzazione complessiva del sistema

1 Apache Hadoop

Apache Hadoop è un framework che grazie all'impiego di molteplici sottosistemi permette il processamento in parallelo di dataset di grandi dimensioni. In questa trattazione verranno presentati solamente i due componenti principali utilizzati attivamente:

- **Apache Hadoop Distributed File System (HDFS)** è un sistema di storage persistente distribuito con architettura **Master/Worker** con supporto alla **Avaliability** e **Partition tolerance**. Con HDFS è quindi possibile memorizzare dataset di dimensioni importanti, nelle unità basilari chiamati **Blocchi**, garantendo un throughput elevato e **fault tolerance** grazie alla replicazione dei dati ed alla presenza di **Namenode secondari**. In particolare, nel cluster in questione è stata impostata una replicazione di 2 unità e la presenza di un singolo Namenode secondario. Nell'architettura sono previsti 3 woker node, dove ognuno di essi ha al suo interno il **Datanode HDFS** corrispettivo per lo storage e il processamento di richieste.

  HDFS viene usato principalmente come **datasource** per il **batch processing**, quindi memorizza i file contenti i dataset, e come **output** dove vengono archiviati i risultati del processamento. Oltre a questo, HDFS viene utilizzato come storage per HBase.

- **Apache Yet Another Resource Negotiator (YARN)** è una tecnologia per il **resource management** e **job scheduling** utilizzata all'interno dell'ecosistema Apache Hadoop. Gestisce quindi, in termini di *numero di core* e *memoria*, le risorse che i singoli worker node devono allocare, inoltre definisce lo *scheduling* e *l'esecuzione* dei Job o i Task che devono essere processati. Fornisce anche attività di *monitoring* dei Jobs. La versatilità di YARN risiede nel fatto che la sua **architettura master/worker** permette di distribuire la computazione tra più nodi, gestendone l'intrinseco parallelismo che ne consegue, ed inoltre permette l'utilizzo di una moltitudine di framework diversi per il processamento, tra cui **Apache MapReduce** e **Apache Spark**. Come per HDFS, ogni worker node dell'architettura ha un suo **Yarn Node Manager** ed il **Yarn Resource Manager** riside nel nodo master insieme al Namenode di HDFS.

  YARN viene quindi utilizzato come resource manager e job scheduler per il framework di processamento impiegato nell' architettura: **Apache Spark**

- **Apache HBase** è il database non relazionale column-based a *chiave-valore* distribuito e scalabile di Hadoop, pensato appositamente per il datastore di grandi moli di dati. HBase è particolarmente prestante nei contesti dove sono necessarie operazioni di *real-time random read/write*. HBase monta al suo interno un cluster Apache ZooKeeper come servizio di sincronizzazione distribuita. Viene usato con un singolo Master e Region server, ma la sua architettura permette l'espansione a più Region Server senza nessun setup aggiuntivo, portando ad una maggiore scalabilità orizzontale.

  HBase viene usato come una ulteriore destinazione di **output** per il risultato dei Job eseguiti.

La scelta di **YARN** è da ricercare nella sua integrazione nativa in Apache Hadoop e nel voler sperimentare la sua capacità di interfacciarsi con framework di processamento diversi da **Apache MapReduce,** come per esempio **Apache Spark**. Inoltre, l'impiego di un worker node sia come **HDFS Datanode** e sia come **Task Executor** per Yarn, permette di ridurre i tempi di latenza per il fetching dei dati ed avere un alto throughput.

La scelta di usare **HBase** risiede nelle sue performance altamente competitive rispetto alla concorrenza, il supporto nativo ai cluster Apache Hadoop, le numerose API per interfacciarsi con il datastore e infine il supporto fornito da altri framework/tecnologie all'interazione con HBase mediante letture e scritture.

2 Apache NiFi

Apache NiFi è un framework per **data ingestion** e **data trasformation**. Supporta lo sviluppo del *data routing* attraverso **Grafi diretti aciclici (DAG)**. NiFi è particolarmente adatto a questo sistema poiché si interfaccia molto bene con svariati sistemi di data storage (tra cui HDFS), è adatto a sistemi di batch processing ed infine permette di gestire dei flussi più o meno complessi a seconda delle necessità. Supporta molti **formati di dati** (CSV, Json, Plain, …) e permette **trasformazioni** articolate per pulire o ridimensionare il dataset.

In questo sistema, NiFi è stato utilizzato per recuperare dalle **data sources** (la repository Github di Opendata) i dataset delle vaccinazioni in formato **CSV**, convertirli in formato **Apache Parquet**, rimuovere le colonne non desiderate dai dataset e salvare in maniera persistente il risultato in HDFS per renderlo disponibile al processamento eseguito con Spark.

Rimuovendo le colonne si va ad alleggerire di molto il carico a cui è sottoposto successivamente il sistema in fase di processamento, sia per quanto riguarda l'esecuzione dei Job che per quanto riguarda il fetching dei dati, e nel caso di dataset di dimensioni notevoli, permette di risparmiare una quantità non indifferente di spazio di storage. Sperimentalmente, è stato misurato un miglioramento di circa del 70% nel tempo di fetching dei dati dopo l'eliminazione di colonne superflue.

Apache Parquet è stato scelto come formato dei dati per la sua natura **column oriented** che permette la selezione e l'eliminazione di colonne dal dataset molto più rapida rispetto alla sua controparte row-based. Inoltre, data la natura del progetto, è stato ritenuto il più adeguato poiché favorisce notevolmente le letture rispetto alle scritture. L'analisi del dataset richiede molteplici raggruppamenti e aggregazione di colonne, rendendo Parquet ancora una volta la scelta appropriata per questo tipo di carico.

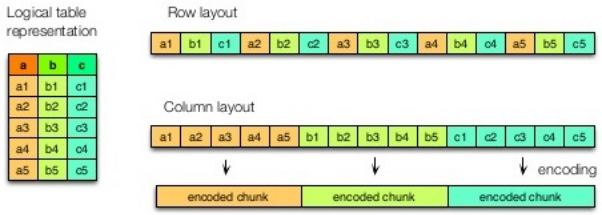


Figura 5**:** Differenza sullo storing dei dati **row-based** contro **column-based**.

3 Apache Spark

Apache Spark è un engine analitico per il processamento di dati a larga scala che segue il paradigma **"map reduce**", e permette l'esecuzione di Job in maniera completamente distribuita e parallelizzata. Contrariamente alla sua controparte **Apache MapReduce,** Spark offre una semplicità maggiore nella definizione del flusso di operazioni, permettendo allo sviluppatore di interfacciarsi con un paradigma di computazione più familiare.

Le sue **performance** migliorano di due ordini di grandezza quelle che offre invece MapReduce. Nonostante Spark offre anche i servizi di resource management e job scheduling di Yarn, è possibile comunque usare quest'ultimo per l'adempimento di questi due task, e l'integrazione tra questi due sistemi risulta molto semplice: Spark al suo interno prevede, tra le possibili opzioni, l'uso di cluster Yarn, limitando gli step per interconnettere i due sistemi ad una semplice configurazione di parametri.

La dimensione del salto di prestazioni rispetto a MapReduce è da ricercare nel modo in cui vengono salvati i risultati di ogni fase di map reduce. Mentre MapReduce salva ogni risultato in maniera persistente, Spark mantiene i dati in-memory, e vengono resi persistenti solo nel momento in cui lo sviluppatore lo ritiene più appropriato.

Spark espone anche delle API di alto livello, tra cui **Spark SQL** che abilita lo sviluppatore alla scrittura di query in formato **SQL-like** (SQL 99). Da ciò ne consegue che la curva di apprendimento di questo paradigma di programmazione si abbassi sensibilmente.

Una ulteriore API di alto livello è **Spark MLlib** per operazioni di Machine Learning e analisi statica sui dati. Supporta i principali algoritmi di ML e Regressione unendoli ai principi di parallelismo del processamento

Entrambe le API descritte sopra sono state utilizzate all'interno dell'applicativo.

Per le motivazioni appena descritte Spark è il candidato ideale per questo sistema.

L'esecuzione di un'applicazione Spark può avvenire in due modi:

- **Cluster mode**: il Driver Program di Spark è interamente trasferito nel Cluster Manager, riducendo la latenza nella comunicazione Driver-Worker
- **Client mode**: il Driver Program viene istanziato localmente, e nel caso il nodo dove è presente il Driver Program è esterno al cluster, i tempi di latenza possono portare ad un degrado sostanziale delle performance.

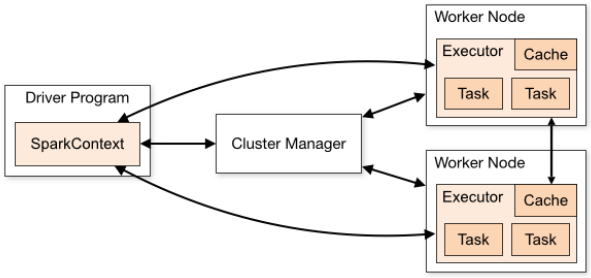


Figura 6: Schematizzazione di un cluster Spark

Si è preferito usare la **client mode** esclusivamente per motivi di *debug* e per permettere di tenere traccia dell'esecuzione dei Job tramite il log di console. Cambiando un semplice parametro di esecuzione è possibile eseguire Spark in **cluster mode**.

DEPLOYMENT

Il deployment è stato effettuato usando la **containerizzazione** dei **nodi** del sistema con **Docker,** simulando abbastanza fedelmente un cluster di macchine distinte tra loro. Ogni nodo dell'architettura presentata precedentemente è un container docker all'interno di una Docker Network. L'istanziazione dei container è effettuata mediante dei Dockerfile, dove principalmente vengono impostati i parametri di configurazione per una corretta comunicazione tra i nodi.

Il numero di Worker è di default impostato a 3, ma è possibile aumentare o ridurre questo numero a seconda delle necessità.

Tutta la fase di deployment è automatizzata dagli script presenti nel sorgente, i quali compilano l'applicativo e fanno il download dei vari package necessari. Per rendere l'architettura più portatile possibile sono stati ridotti al minimo il numero di dipendenze con la macchina host, dove l'unica dipendenza rimasta è la presenza di Docker, il resto delle operazioni di configurazione e compilazione vengono processate all'interno dei container.

Alla chiusura delle istanze docker, viene pulito l'ambiente interrompendo ed eliminando tutti i container del cluster. E' necessaria una minima interazione con l'utente per soli scopi di sincronizzazione: i nodi del **Cluster** (HDFS, Yarn e HBase) devono essere istanziati e *in running* prima che il **nodo NiFi** operi la data ingestion; l'ultimo nodo da istanziare sarà il client **Spark** dopo che i dati vengono caricati con successo su HDFS.

Il deploy su Docker, a differenza di un deploy sul Cloud con servizi come AWS, permette una maggiore personalizzazione del cluster e un maggior controllo sui singoli nodi. Inoltre, con un deploy manuale è possibile sperimentare la configurazione ed il setup delle varie tecnologie, anche al fine di comprenderne meglio il loro funzionamento.

L'utilizzo di Docker su uno stesso nodo rende la latenza di comunicazione tra i vari nodi prossima a 0, ma i singoli nodi, condividendo uno stesso hardware, hanno accesso ad un insieme ristrette di risorse computazionali.

Le tecnologie ed i framework utilizzati sono:

- **Apache Spark** e **Spark MLlib** v. 3.1.1
- **Apache** Common Math
- **Apache Hadoop** v. 3.2.2
- **Apache HBase** v. 1.4.13 con ZooKeeper Built-in
- **Apache NiFi** v. 1.13.2
- **Apache Parquet**
- **Java** 1.8 (Sia come JDK che come JRE nei nodi)
- **Docker**
- **Grafana**

QUERY

L'applicativo sviluppato utilizzando Spark risponde ai 3 quesiti presentati precedentemente. Per effettuare le query sono stati utilizzati sia **Spark Core** sia **Spark SQL**, dove quest'ultimo si è mostrato molto più prestante rispetto al suo corrispettivo e anche più semplice da utilizzare data la sua natura SQL-Like, molto più familiare.

Prima di eseguire le query, vengono caricati i dati da HDFS sottoforma di **Dataset**, per poi essere convertito, nel caso di query Spark Core, al suo corrispettivo RDD.

Alla fine di ogni query, il risultato viene salvato in HDFS ed in HBase.

Di seguito vengono presantati in modo riassuntivo i passaggi delle singole query.

1 Query 1

Come passaggio preliminare, viene effettuato un **sorting del dataset** basato sulla data.

Successivamente, la colonna 'data_somministrazione' viene convertita dal formato 'yyyy-mm-dd' al formato 'yyyy-mm' dato che il giorno per questa query è ininfluente, ed inoltre con la data in questo formato è possibile raggruppare i dati mese per mese.

A seguire, vengono sommati, tramite funzioni di **reduce**, il **numero di centri totali per regione** ed il **numero di vaccinazioni totali per ogni regione e per ogni mese**.

Come ultimo passaggio, con una **map** viene calcolato il rapporto:

*totale_vacc/(num_centri * giorni_mese)*

Dove:

- *totale_vac*: numero totale vaccinazioni in una regione specifica e in un mese specifico
- *num_centri:* numero centri di vaccinazione in quella specifica regione
- *giorni_mese:* numero di giorni in uno specifico mese

Alla fine di questo processamento, viene riportato il numero medio di vaccinazioni effettuato in un centro qualsiasi in ogni regione e per ogni mese.

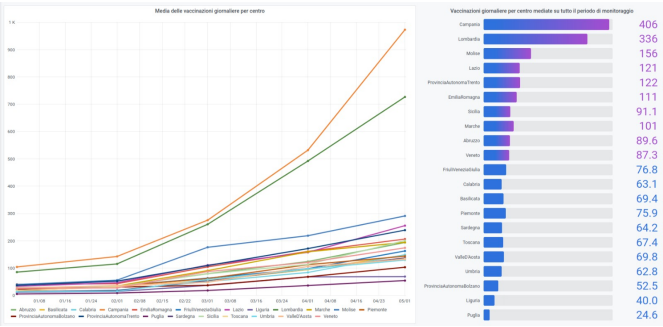


Figura 7: Esempio di output per la prima query con Garafana

2 Query 2

La seconda query prevede di effettuare una stima usando la **regressione,** la quale è stata implementata con la libreria **Simple Regression** di Apache Common Math. Inoltre, per quanto riguarda la query su **Spark SQL** è stato necessario registrate due **User-Defined Functions (UDFs)** per la stima mediante regressione lineare e per estrapolare la *top 5* di regioni con la stima più alta.

I passaggi sono i seguenti:

- Una **reduce** iniziale per calcolare la somma di tutte le vaccinazioni in un giorno tra le varie tipologie di vaccino (Pfizer, AstraZeneca, …)
- Una **groupby** e una **flatmap** per raggruppare tutti i giorni di un determinato mese per ogni regione con il relativo numero di somministrazioni.
- Una **filter** per rimuovere le casistiche di meno di due giorni di campagna vaccinale in un mese
- Una **map** dove viene applicata la regressione
- Infine, un'ultima **map** dove viene computata la top 5

La query corrispettiva in SQL non differisce di molto nei passaggi.

mese	fascia_anagrafica	leaderboard
2021-Mar	12-19	Lazio, 6; Puglia, 6; Sicilia, 5; Toscana, 5; Veneto, 5;
2021-Mar	20-29	Toscana, 444; Puglia, 363; Veneto, 334; Piemonte, 283; Lombardia, 271;
2021-Mar	30-39	Toscana, 911; Campania, 564; Puglia, 545; Piemonte, 393; Lombardia, 388;
2021-Mar	40-49	Toscana, 1424; Campania, 1150; Puglia, 919; Lazio, 880; Sicilia, 678;
2021-Mar	50-59	Campania, 1867; Lazio, 1159; Toscana, 1069; Puglia, 975; Sicilia, 622;
2021-Mar	60-69	Campania, 960; Puglia, 441; Lazio, 286; Toscana, 236; Veneto, 163;
2021-Mar	70-79	Lombardia, 133; Lazio, 91; Sardegna, 54; Toscana, 54; Sicilia, 49;

Figura 8: Esempio di output della Query 2 in formato *csv* sottoforma di *classifica*.

3 Query 3

La Query 3 prevedeva di fare clustering delle regioni a seconda della stima di popolazione vaccinata. Per fare ciò sono stati impiegati gli algoritmi di **KMeans e Bisecting KMeans**, implementti nella libreria **Spark MLlib**. Questi algoritmi, come è possibile vedere dalla dashboard di Spark, sfruttano il parallelismo della computazione.

I passaggi sono i seguenti:

- Una **groupby** iniziale per separare le vaccinazioni basandosi sulla regione
- Una **map** per applicare la regressione al fine di predire per ogni regione le vaccinazioni del giorno 1 giugno
- Una **join** per unire i valori appena predetti con quelli dell'RDD iniziale
- Una **reduceByKey** per calcolare il totale delle vaccinazioni per ogni regione
- Una **map** per computare la percentuale stimata di persone vaccinate in ogni regione (*vacc_tot_regione/popol_regione)*
- L'applicazione degli algoritmi di clustering KMeans e BisectingKMeans per classificare le regioni in clusters (da 2 fino a 5 clusters)

numero_cluster	WSSSE	[cluster]Regione
2	51.82	[0]ABR-[0]BAS-[0]CAM-[0]EMR-[0]FVG-[0]LAZ-[0]LIG-[0]LOM-[0]MAR-[(
3	24.87	[0]ABR-[0]BAS-[0]CAM-[0]EMR-[0]FVG-[0]LAZ-[0]LOM-[0]MAR-[0]PAB-[
4	14.00	[0]ABR-[0]BAS-[0]CAM-[0]EMR-[0]FVG-[0]LAZ-[0]LOM-[0]MAR-[0]PAB-[
5	6.97	[0]ABR-[0]BAS-[0]EMR-[0]FVG-[0]LOM-[0]MAR-[0]PAB-[0]PUG-[0]UMB·

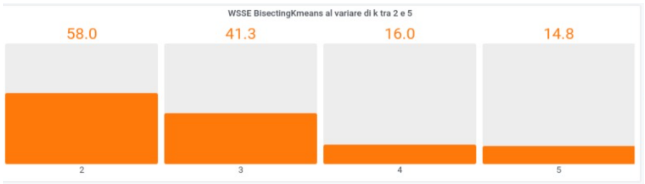
Figura 9: esempio di output della Query 3

Il clustering è di tipo **monodimensionale**, infatti l'unico valore su cui l'algoritmo si basa è il *rapporto:*

totale_vacc_stimate/popolazione

Il quale, per ogni regione, si attesta tra circa il 50% ed il 65%.

Per valutare la **qualità** del **clustering**, sia per KMeans che per Bisecting KMeans è stata presa in considerazione la metrica **WSSSE (Within Set Sum of Squared Error)**, che non è altro che la somma della distanza al quadrato di ogni singolo punto al suo centroide. Questo valore, generalmente, scende al crescere di K, poiché aumentando il numero di cluster, e quindi il numero di centroidi, mediamente la distanza tra un punto ed il suo centroide di riferimento decresce. Inoltre, i dati in questione non presentano **outliers** particolarmente importanti, perciò la WSSSE non è influenzata da questo fattore.



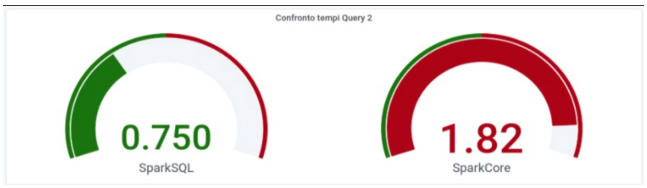
L'algoritmo di KMeans e la sua variante sono fortemente soggetti all'*inizializzazione* dei centroidi, perciò più run dell'algoritmo possono portare a risultati diversi, quindi i risultati presentati possono non essere pienamente rappresentativi.

Si può dire che la qualità del clustering aumenti all'aumentare del numero di cluster, ma in un contesto dove si vuole raggruppare dei punti all'interno di categorie, la scelta migliore è quella di decidere a priori in quante categorie si vogliono classificare i vari punti ed operare di conseguenza.

ANALISI DEI TEMPI

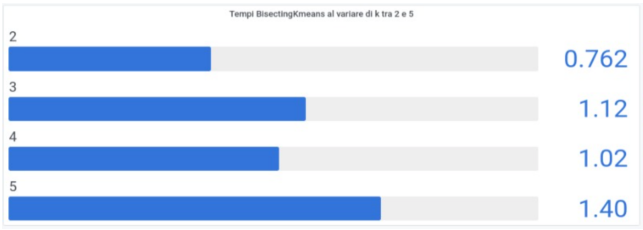
Per valutare sperimentalmente il rapporto tra le performance di **Spark Core** e **Spark SQL** sono stati prese in esame **100 run**, e il tempo di esecuzione calcolato come la media tra questi. Le **run** sono state fatte su una **macchina virtuale Ubuntu 20.04** dove è stato emulato il cluster con Docker.

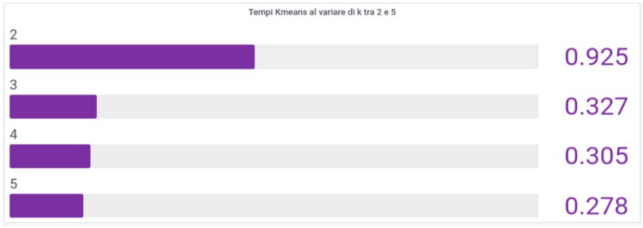
Dai tempi sono stati esclusi le operazioni che le due query avevano in comune e sono stati esclusi anche i tempi relativi alla persistenza dell'output.



Come già esposto precedentemente, SparkSQL presenta dei tempi molto più competitvi rispetto a Spark Core, rendolo la scelta ottimale.

Oltre a valutare la qualità del cluster, sono stati valutati anche i tempi di processamento dei due algoritmi utilizzati al variare del numero dei cluster. Data la natura monodimensionale e la vicinanza dei punti l'uno con l'altro, si è ritenuto sufficiente effettuare **20 iterazioni dell'algoritmo**. Anche in questo caso, i tempi sono computati come la media tra 100 run della query.





Mentre KMeans riduce il tempo di esecuzione al crescere di K, Bisecting KMeans al contrario, ha tempi più lunghi al crescere di K. Questo accade per la natura gerarchica dell'algoritmo di Bisecting KMeans, il quale ha bisogno di step ulteriori con K più grandi

Tutte le query hanno mostrato circa un 10%-15% di aumento dei tempi di esecuzione nel passaggio tra l'esecuzione in locale all'esecuzione nel cluster. Questo aumento è dovuto dal fatto che il parallelismo, implementato con container, aggiunge solo overhead e non fornisce vantaggi in termini prestazionali. Inoltre, le ridotte dimensioni del dataset non permettono di sfruttare a pieno il parallelismo a discapito di un overhead che diventa sempre più piccolo al crescere del dataset.

SVILUPPI FUTURI

Durante lo sviluppo, sono state pensate alcune soluzioni che potrebbero essere implementate a lungo termine:

- Automatizzazione completa del setup di Apache NiFi con aggiunta di **scalabilità orizzontale**
- Aggiungere più **Region Server** per HBase
- Provare sperimentalmente la differenza nell'uso di **Spark Standalone** contro **Spark w/ Yarn**.

DIFFICOLTA' RISCONTRATE

Le principali difficoltà riscontrate sono state relative al setup del cluster, sia per le documentazioni non particolarmente esplicative in alcuni casi e sia per le forti discrepanze tra una versione ed un'altra dello stesso framework.

L'utlizzo invece di Spark con il paradigma map reduce non si è rivelato particolramente ostico

NOTE FINALI

E' possibile consultare la repository:

https://github.com/andreapaci/SABD

per ulteriori approfondimenti sul sorgente dell'applicativo ed il setup del cluster.

Sono anche presenti le istruzioni su come istanziare l'intero cluster ed alcune note aggiuntive.

Nella cartella *Results* sono presenti i risultati delle query in formato *csv.*

Al link di seguito:

https://snapshot.raintank.io/dashboard/snapshot/3EpFd4eW2fKRzxL7vKdqdM5b60Y0OEFi

e' possibile consultare la Dashboard di Grafana

REFERENCES

[1] Presidenza del Consiglio dei Ministri, Github, https://github.com/italia/covid19-opendata-vaccini