Università degli studi di Salerno

Dipartimento di Informatica

Laurea Magistrale in Informatica

Sistemi Operativi Avanzati

anno 2020−2021

Query su Tweet Covid−19

Professore
Giuseppe Cattaneo

Studenti

Mario Cetrangolo

Manlio Santonastaso

Indice

1 Introduzione

Questo elaborato si focalizza sul problema dei Big Data. La domanda cruciale: "Perché abbiamo bisogno di un nuovo motore e modello di programmazione per l'analisi dei dati."
Come molte tendenze dell'informatica, dovuto ai cambiamenti nei fattori economici che sono alla base delle applicazioni informatiche e hardware. Per la maggior parte della loro storia, i computer sono diventati più veloci ogni anno grazie all'aumento della velocità del processore: i nuovi processori ogni anno potrebbero eseguire più istruzioni al secondo rispetto all'anno precedente. Come conseguenza a ciò, anche le applicazioni sono diventate automaticamente più veloci ogni anno, senza bisogno di modifiche per il loro codice.
Questa tendenza ha portato a un ampio e consolidato ecosistema di applicazioni che si accumulano oltre tempo, la maggior parte dei quali sono stati progettati per funzionare solo su un singolo processore. Queste applicazioni hanno guidato la tendenza di velocità del processore migliorando la scalabilità fino a calcoli e volumi di dati sempre più grandi col tempo.
Sfortunatamente, questa tendenza nell'hardware si era interrotta intorno al 2005: a causa dei limiti della dissipazione del calore, gli sviluppatori di hardware hanno smesso di rendere più veloci i singoli processori e hanno aggiunto più core paralleli per CPU che funzionano tutti alla stessa velocità. Questo cambiamento ha significato che improvvisamente le applicazioni dovevano essere modificate per aggiungere parallelismo al fine di funzionare più velocemente, il che ha posto le basi per nuovi modelli di programmazione come Apache Spark.
Lo scopo di questo progetto è di dimostrare come può essere elaborata e analizzata una grande quantità di dati attraverso l'utilizzo di un sistema distribuito. In particolare, l'obiettivo è quello di effettuare un confronto tra eseguire query su un database relazionale (MySQL) e una query utilizzando la tecnologia Apache Spark cioè dataframe. E si è notato diversi vantaggi tra cui analizzare una quantità di dati molto grande GB (nel nostro caso) ma è possibile operare anche su PB di dati, un altro vantaggio sostanziale è il tempo di esecuzione di una query e vedremo nel capitolo 3 in cui è la differenza, infine un ultimo vantaggio è che un DataFrame ha ulteriori metadati grazie al suo formato tabellare, che consente a Spark di eseguire determinate ottimizzazioni sulla query finalizzata.
L'elaborato inizialmente parlerà dell'importanza dei Big Data perché dato l'elevata mole di dati che abbiamo a disposizione vi è la necessità di utilizzare tecnologie, paradigmi e strumenti adeguati a essere elaborati in modo efficace. In seguito, parlerà di quali strumenti di programmazione sono stati utilizzati per sviluppare l'applicazione e infine nell'ultimo capitolo vedremo i confronti tra il calcolo sequenziale e il calcolo distribuito.

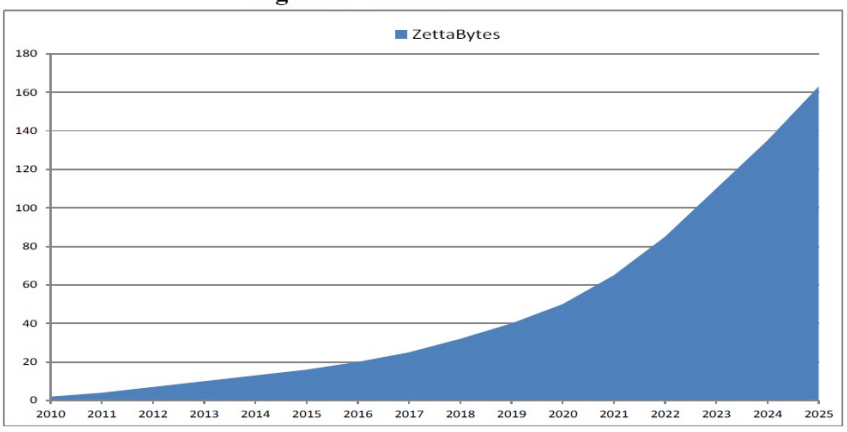
Capitolo 2

2 Big Data

2.1 Che cos'è Big Data

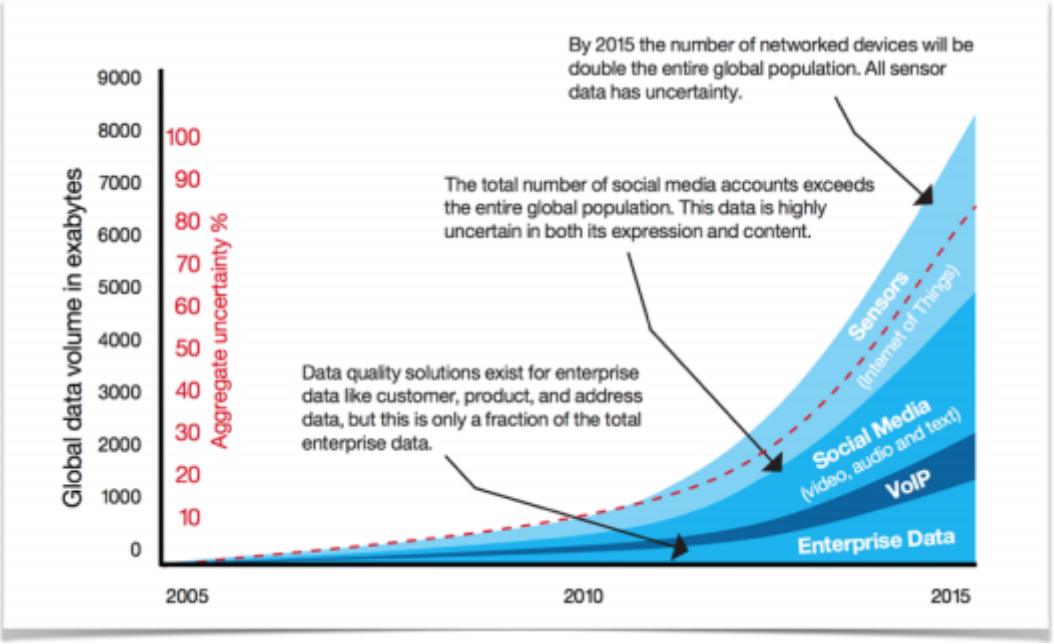
Il termine "big data" si riferisce a dati informatici così grandi, veloci o complessi, difficili o impossibili da elaborare con i metodi tradizionali. O'Reilly Media ha utilizzato esplicitamente il termine "Big Data" per riferirsi a grandi insiemi di dati che è quasi impossibile gestire ed elaborare utilizzando i tradizionali strumenti di business intelligence. Le fonti di dati sono tantissime e in costante aumento e pertanto ciò che caratterizza i big data non è solo la quantità.

Figura 1 – La crescita dei dati



Fonte: elaborazione AGCM in base ai dati forniti nel rapporto tecnico IDC[8]

I big data sono caratterizzati anche dalla complessità riconducibile alla loro varietà, come si può vedere nella figura sottostante.



Infine, i big data si basano su tre concetti:
Volume: le organizzazioni e le aziende raccolgono e raccolgono dati diversi da diverse fonti, che includono transazioni e dati aziendali, dati dai social media, dati di accesso, nonché informazioni dal sensore e dati da macchina a macchina. In precedenza, questa memorizzazione dei dati sarebbe stata un problema, ma a causa dell'avvento di nuove tecnologie per la gestione di dati estesi con strumenti come Apache Spark, Hadoop, il carico di enormi dati è diminuito.

Velocità: i dati vengono ora trasmessi a una velocità eccezionale, che deve essere gestita adeguatamente. Sensori, smart metering, dati utente e tag RFID stanno aumentando la necessità di affrontare un'inondazione di dati quasi in tempo reale.

Varietà: i rilasci di dati da vari sistemi hanno tipi e formati diversi. Si va da dati numerici strutturati a non strutturati di database tradizionali a documenti non numerici o di testo, e−mail, audio e video, dati di stock ticker, dati di accesso, dati crittografati di Blockchain o persino transazioni finanziarie.

2.2 Big Data: Analisi Descrittiva

Big data analytics fa riferimento al processo che include la raccolta e l'analisi dei big data per effettuare analisi descrittive fanno esattamente ciò che il nome suggerisce, sintetizzano o descrivono i dati grezzi e creano qualcosa che è interpretabile dagli

esseri umani. Nello specifico vengono analizzati gli eventi passati, dove per eventi passati ci si riferisce a qualsiasi punto del tempo in cui si è verificato un evento, che sia un minuto fa o un mese fa. Le analisi descrittive sono utili in quanto consentono alle organizzazioni di imparare dai comportamenti passati e di aiutarli a capire come potrebbero influenzare i risultati futuri. Nel nostro caso di studio analizziamo il momento pandemico Covid−19 e come gli esseri umani hanno reagito a questa situazione interrogando attraverso delle query i tweet pubblicati dagli utenti che utilizzano il social network Twitter.

2.3 Query

Query letteralmente significa interrogazione, richiesta, domanda. Generalmente è una definizione che è associata ai database. In un database le informazioni sono organizzate in una struttura logica che permette di accedere con facilità ad ogni dato. Il modo per accedere a questi dati è la query. La query viene scritta in un linguaggio di interrogazione. Ne esistono decine ed il più famoso prende il nome di SQL. Come tutti i linguaggi, l'SQL ha una sintassi e delle regole. Tramite queste regole è possibile ricercare fra i dati, applicando dei filtri ed ordinando i dati a piacimento

2.4 Twitter

Twitter è un servizio di notizie e microblogging su cui gli utenti postano sulla propria pagina personale e interagiscono tra loro tramite messaggi chiamati "tweet", che consistono in messaggi di testo di lunghezza massima di 280 caratteri. Il servizio vanta grande popolarità in tutto il mondo, nel 2012 ha raggiunto i 500 milioni di iscritti a partire dal 2016, contava oltre 319 milioni di utenti attivi mensilmente.
Twitter è considerato uno dei social network più adatti per raccogliere pareri, considerazioni, reazioni e altre informazioni grazie ad una delle sue caratteristiche chiave, cioè l'immediatezza nella comunicazione e nell'espressione dei propri pensieri tramite tweet. Gli utenti di Twitter, infatti, tendono ad esprimere liberamente le proprie opinioni, rendendolo una fonte ideale di dati utilizzabili per ottenere informazioni riguardanti una grande varietà di argomenti per esempio nel nostro caso la pandemia Covid−19.

Capitolo 3

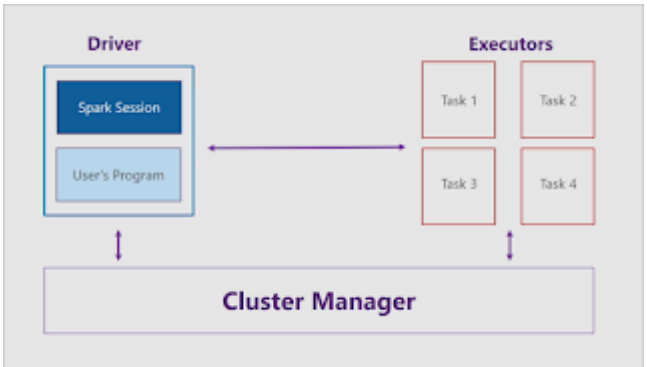
3 Strumenti Utilizzati

3.1 Apache Spark

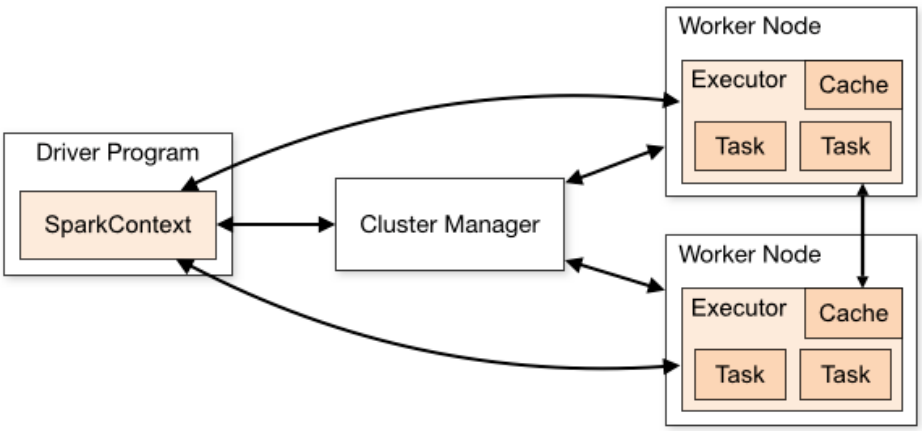
Apache Spark è un Framework di elaborazione parallela open source che supporta elaborazione in memoria per migliorare le prestazioni delle applicazioni che analizzano BIG DATA. Le soluzioni per Big Data sono progettate per gestire dati troppo grandi o complessi nei tradizionali database. Spark è in grado di elaborare queste grandi quantità di dati in memoria, nel modo più veloce possibile rispetto alle alternative basate su disco. Infatti, grandi giganti Tech come Netflix, Yahoo ed Alibaba sono alcuni che hanno utilizzato su vasta scala, per eseguire una elaborazione dati più veloce. La sua caratteristica più importante è il cluster computing in memoria che è responsabile di aumentare la velocità di elaborazione dei dati. Un cluster è un gruppo di computer collegati e coordinati tra di loro per elaborare e analizzare dati. In questo modo si aumenta la velocità utilizzando risorse di molti processori come computer collegati tra loro per le sue analisi, per avere una soluzione scalabile. Apache inoltre è anche tollerante ai guasti, perché utilizza dei Resilient Distributed Dataset (RDD), ovvero una struttura dati che può
contenere qualsiasi oggetto Python, Java o Scala, comprese le classi definite dagli utenti.

3.2 Architettura Apache Spark

L'architettura Apache Spark è composta da 3 componenti principali: Driver, Executor e Cluster Manager che collaborano fra di loro, seguendo un'architettura master slave, dove master è il driver, gli slave sono executor.



Driver: è costituito dal programma, un'app console C# e una sessione di Spark che prende il programma e lo divide in attività più piccole gestite dagli esecutori. Executors: ogni executor o nodo di lavoro, riceve un'attività dal driver ed esegue tale attività. Gli executor si trovano in un'entità nota come cluster.
Executors: ogni executor o nodo di lavoro, riceve un'attività dal driver ed esegue tale attività. Gli executor si trovano in un'entità nota come cluster
Cluster Manager: comunica con il driver e gli executor per comunicare alcune operazioni tra cui allocare delle risorse, assegnare task ed esecuzione dei task.
Un'altra caratteristica di Apache Spark è quella di supportare linguaggi di programmazione come Scala, Python, Java, SQL, R, Linguaggi .NET (C#/F#). Inoltre, supporta diverse API.
In particolare, applicazioni Spark sono gestite come serie indipendenti di processi su cluster, coordinate dall'oggetto SparkContext nel programma principale (driver). E sono eseguiti su un cluster, dove SparkContext può connettersi a diversi tipi di gestore di cluster o Cluster Manager (autonomo di Spark, Mesos, YARN) che allocano altre risorse tra le applicazioni. Una volta connesso, Spark acquisisce gli esecutori sui nodi del cluster, dei processi che eseguono calcoli e archiviano i dati per applicazione. Dopodiché invia il codice dell'applicazione (JAR o Python file a SparkContext) agli executors.
Infine, Spark Context invia le attività agli executors per esecuzione. Che sono responsabili dell'esecuzione effettiva del lavoro datogli. Ognuno di essi deve eseguire il codice e riportare lo stato del calcolo al nodo del driver. (Modalità è detta Cluster Mode).



Spark dispone anche di un'atra modalità definita Client Mode. In questo modo il programma SparkContext e il Driver vengono eseguiti all'esterno del cluster, un laptop. Questa modalità è funzionale nel caso in cui non si vuole usare un cluster, ma si voglia eseguire tutto sul proprio computer per effettuare un test alla nostra applicazione.

3.2.1 PySpark

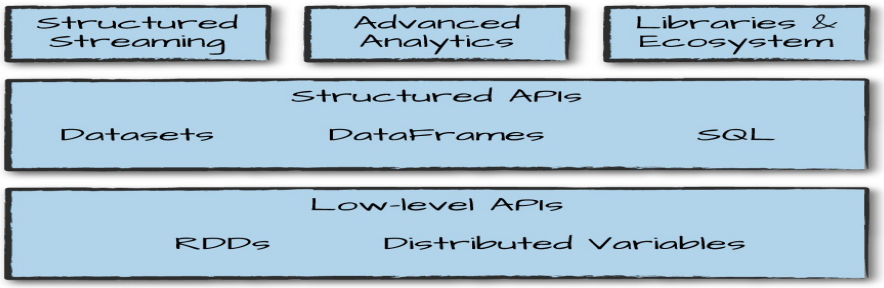
Apache Spark è scritto nel linguaggio di programmazione Scala, PySpark è stato rilasciato per supportare la collaborazione di Apache Spark e Python, in realtà è un'API Python per Spark. Inoltre, PySpark, ti aiuta a interfacciarti con i set di dati distribuiti resilienti (RDD). Ciò è stato ottenuto sfruttando la libreria Py4j, libreria popolare che è integrata all'interno di PySpark e consente a Python di interfacciarsi dinamicamente con gli oggetti JVM. Non solo ti consente di scrivere applicazioni Spark utilizzando le API Python, ma fornisce anche la shell PySpark per analizzare in modo interattivo i tuoi dati in un ambiente distribuito. PySpark supporta la maggior parte delle funzionalità di Spark come Spark SQL, DataFrame, Streaming, MLib (Machine Learning) e Spark Core. Inoltre, ci sono varie librerie esterne che sono anche compatibili:
PySpark SQL: libreria per applicare analisi di tipo SQL su un'enorme quantità di dati strutturati o semi−strutturati. Con la possibilità di utilizzare anche le query SQL direttamente. PySpark SQL ha introdotto DataFrame, una rappresentazione tabulare di dati strutturati simili a quella di una tabella di un sistema di gestione di database relazionali.
MLib è un wrapper per PySpark ed è la libreria di machine learning (ML) di Spark. Questa libreria utilizza la tecnica del parallelismo dei dati per archiviare e lavorare con i dati. L'API di apprendimento automatico fornita dalla libreria MLib è abbastanza facile da usare. MLib supporta molti algoritmi di apprendimento automatico per la classificazione, la regressione, il clustering, il filtraggio collaborativo, la riduzione della dimensione e le primitive di ottimizzazione sottostanti.
GraphFrames è una libreria per l'elaborazione di grafici che fornisce un set di API per eseguire l'analisi dei grafici in modo efficiente, utilizzando il core PySpark e PySpark SQL. È ottimizzato per il calcolo distribuito veloce.

3.3 PySpark SQL Dataframe

L'API Spark DataFrame consente all'utente di eseguire l'elaborazione di dati strutturati paralleli e distribuiti sui dati di input. Un dataframe è una raccolta distribuita di insieme di dati a un insieme di colonne denominate. Le righe possono avere una varietà di formati di dati (eterogenei), mentre una colonna può avere dati dello stesso tipo di dati (omogenei). È simile a una tabella in un database relazionale e ha un aspetto simile. Può essere derivato da un dataset che può essere costituito da file di testo delimitati, file Parquet e ORC, CSV, RDBMS Table, Hive Table, RDD. Ciò aiuta Spark a ottimizzare il piano di esecuzione su queste query, potendo gestire anche Petabyte di dati. Inoltre, può anche essere costruito da formati semi−strutturati come JSON e XML.

Le caratteristiche principali dei dataframe troviamo:

- Accesso unificato ai dati
- Capacità di gestire dati strutturati e semi−strutturati
- Supporta un'ampia varietà di origini dati
- Funzionalità abbondanti per la manipolazione e aggregazione dei dati
- Supporta più linguaggi come Python, Java, R e Scala



3.4 Hadoop Yarn

Acronimo per Yet Another Resource Negotiator, YARN è un framework per la gestione di risorse per il calcolo distribuito e si occupa di assegnare le risorse disponibili alle varie applicazioni nella maniera più efficace possibile.
In un'architettura distribuita YARN si pone tra l'HDFS e il framework utilizzato per eseguire le applicazioni. In questo caso l'utilizzo di Apache Spark con YARN fa sì che ogni esecutore Spark venga eseguito come un container YARN.
YARN può assegnare risorse dinamicamente in caso di necessità, in modo da migliorare l'utilizzo delle stesse e le performance dell'applicazione. YARN decentralizza l'esecuzione e il monitoraggio dei processi andando ad assegnare a diverse componenti i vari compiti necessari. Un ResourceManager globale si occupa di accettare le sottomissioni dei job, li schedula e alloca loro risorse. Per ogni nodo è presente un NodeManager che si occupa del suo monitoraggio e comunica le informazioni al ResourceManager. Infine, per ogni applicazione viene creato un ApplicationMaster che le permette di ottenere risorse, esso inoltre lavora insieme al NodeManager per eseguire e monitorare i vari task.

3.5 HDFS

L'Hadoop Distributed File System (HDFS) è un file system distribuito, scalabile e portabile scritto in Java per il framework Hadoop, che fornisce la possibilità di immagazzinare dati in modo distribuito. Basato sul Google File System, l'HDFS è un file system con struttura a blocchi in cui i singoli file sono memorizzati come blocchi di grandezza fissata e in cui blocchi dello stesso file non sono necessariamente nella stessa macchina.
Uno dei grandi vantaggi dell'HDFS è la possibilità di sfruttare la data locality, i blocchi di dati vengono infatti assegnati in modo tale che ogni nodo del cluster lavori su dati direttamente presenti nella propria memoria locale, permettendo di velocizzare tutti i processi di elaborazione. L'architettura è basata su un sistema master−slave in cui un master node, detto NameNode, gestisce il namespace e l'accesso degli slave ai file, mentre i molteplici slave corrispondono ognuno ad un DataNode, uno per ogni nodo del
cluster. La tolleranza ai guasti è fornita grazie alla replicazione del NameNode in primis, oltre alla possibilità di impostare un parametro di replicazione anche per i singoli blocchi di file in base all'affidabilità voluta direttamente all'interno di un file di configurazione.

3.6 Python

Python è un linguaggio di programmazione moderno, dalla sintassi semplice e potente che ne facilita l'apprendimento. Gli ambiti di applicazione di questo linguaggio di programmazione sono svariati: sviluppo di siti o applicazioni Web e desktop, realizzazione di interfacce grafiche, amministrazione di sistema, calcolo scientifico e numerico, database, giochi, grafica 3D. Python fu ideato da Guido van Rossum, programmatore olandese attualmente operativo in Dropbox, all'inizio degli anni novanta, il nome fu scelto per la passione dello stesso inventore verso i Monty Python e per la loro serie televisiva Monty Python's Flying Circus.
I punti di forza di python sono molteplici tra cui:

Open Source: Python è completamente gratuito ed è possibile usarlo e distribuirlo senza restrizioni di copyright. Nonostante sia open source, da oltre 25 anni Python ha una comunità molto attiva, e riceve costantemente miglioramenti che lo mantengono aggiornato e al passo coi tempi.

Multi−paradigma: Python è un linguaggio multi−paradigma, che supporta sia la programmazione procedurale (che fa uso delle funzioni), sia la programmazione ad oggetti (includendo funzionalità come l'ereditarietà singola e multipla, l'overloading degli operatori, e il duck typing). Inoltre, supporta anche diversi elementi della programmazione funzionale (come iteratori e generatori).

Portabile: Python è un linguaggio portabile sviluppato in ANSI C. È possibile usarlo su diverse piattaforme come: Unix, Linux, Windows, DOS, Macintosh, Sistemi Real Time, OS/2, cellulari Android e iOS. Ciò è possibile perché si tratta di un linguaggio interpretato, quindi lo stesso codice può essere eseguito su qualsiasi piattaforma purché abbia l'interprete Python installato.
Ci sono anche altri punti di forza come facilità d'uso, librerie ampie, performance e infine gestisce automaticamente la memoria (Garbage collection).

3.7 Covid-19 Tweet Dataset

Il repository contiene una raccolta continua di tweet associati al nuovo coronavirus COVID−19 dal 22 gennaio 2020. Al 05/11/2021 sono stati raccolti un totale di 1.736.919.773 tweet. I tweet vengono raccolti utilizzando gli argomenti di tendenza di Twitter e le parole chiave selezionate. Inoltre, i tweet di Chen et al. (2020) è stato utilizzato per integrare il set di dati idratando i tweet non duplicati. Questi tweet sono solo un esempio di tutti i tweet generati forniti da Twitter e non rappresentano l'intera popolazione di tweet in un dato momento.
Il set di dati è organizzato per ora (UTC), mese e per tabelle. La descrizione di tutte le caratteristiche in tutte e cinque le tabelle è fornita di seguito. Ad esempio, il percorso "./Summary_Details/2020_01/2020_01_22_00_Summary_Details.csv" contiene tutti i dettagli di riepilogo della raccolta dei tweet del 22 gennaio alle 00:00 ora UTC. Nella figura sottostante è possibile notare l'intera struttura del dataset.

Features Description

Table	Feature Name	Description
Primary key	Tweet_ID	Integer representation of the tweets unique identifier
1.Summary_Details	Language	When present, indicates a BCP47 language identifier corresponding to the machine-detected language of the Tweet text
	Geolocation_cordinate	Indicates whether or not the geographic location of the tweet was reported
	RT	Indicates if the tweet is a retweet (YES) or original tweet (NO)
	Likes	Number of likes for the tweet
	Retweets	Number of times the tweet was retweeted
	Country	When present, indicates a list of uppercase two-letter country codes from which the tweet comes
	Date_Created	UTC date and time the tweet was created
2.Summary_Hastag	Hashtag	Hashtag (\#) present in the tweet
3.Summary_Mentions	Mentions	Mention (@) present in the tweet
4.Summary_Sentiment	Sentiment_Label	Most probable tweet sentiment (neutral, positive, negative)
	Logits_Neutral	Non-normalized prediction for neutral sentiment
	Logits_Positive	Non-normalized prediction for positive sentiment
	Logits_Negative	Non-normalized prediction for negative sentiment

5.Summary_NER	NER_text	Text stating a named entity recognized by the NER algorithm
	Start_Pos	Initial character position within the tweet of the NER_text
	End_Pos	End character position within the tweet of the NER_text
	NER_Label Prob	Label and probability of the named entity recognized by the NER algorithm
6.Summary_Sentiment_ES	Sentiment_Label	Most probable tweet sentiment (neutral, positive, negative)
	Probability_pos	Probability of the tweets sentiment being positive (\<=0.33 is negative, \>0.33 OR \<0.66 is neutral, else positve)
6.Summary_NER_ES	NER_text	Text stating a named entity recognized by the NER algorithm
	Start_Pos	Initial character position within the tweet of the NER_text
	End_Pos	End character position within the tweet of the NER_text
	NER_Label Prob	Label and probability of the named entity recognized by the NER algorithm

Capitolo 4

4 Dettagli implementativi

La soluzione proposta al problema di Query su Big Data è lo sviluppo di un'applicazione di calcolo distribuito, utilizzando Dataframe per contenere i dati in una struttura tabellare composta da righe e colonne. Questa API ci consente di poter gestire dataframe suddivisi su migliaia di computer, in modo tale che se i dati hanno dimensioni molto elevati per essere inseriti su di una macchina o semplicemente ci vorrebbe troppo tempo per eseguire calcolo (query nel nostro caso) su una macchina. Tuttavia, Spark ha diverse interfacce per il linguaggio di programmazione che si utilizza R, Python. Esistono diverse astrazioni in Spark: Dataset, DataFrame, tabelle SQL, e RDD. Tutte queste astrazioni rappresentano le raccolte di dati distribuiti. Il più semplice ed efficienti sono i DataFrame, disponibili in tutte le lingue.

4.1 Transformation

In Spark, le strutture dati principali sono immutabili, il che significa che non possono essere modificate dopo che sono state create. Il concetto molto contorto, se non puoi cambiarlo come si dovrebbe usare, e qui che Spark ci agevola istruendo un DataFrame per modificarlo. Queste istruzioni sono chiamate Transformation. Per esempio, trovare tutti i numeri pari nel nostro DataFrame.

```
# in Python
divisBy2 = myRange.where("numero % 2 = 0")
```

Le Transformation sono il fulcro del modo in cui si esprime la logica di business utilizzando Spark. Abbiamo due tipi di Transformation: narrow dependencies, wide dependencies.
Le Transformation costituite da narrow dependencies sono quelle per le quali ogni partizione dell'input contribuirà a una sola partizione di output. Nella precedente frammento di codice, l'istruzione where specifica una stretta dipendenza, dove solo una partizione
contribuisce al massimo a una partizione di output, come si può vedere nella Figura 2−4.

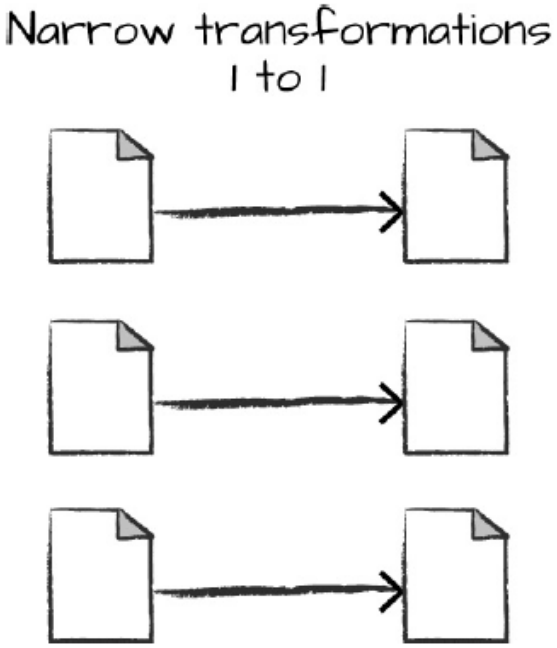


Figure 2-4. A narrow dependency

Una wide dependencies invece avrà partizioni di input che contribuiscono a molte partizioni di output. Questo definito come shuffle per cui Spark esegue un vero e proprio swap delle partizioni nel cluster.
In particolare, per le narrow Transformation, sono eseguite automaticamente operazione definita pipelining, il che significa che se specifichiamo più filtri su DataFrames, verranno tutti eseguiti in memoria. Questo non vale per shuffle, quando eseguiamo, Spark scrive i risultati sul disco, Wide transformation sono illustrate in Figura 2−5.

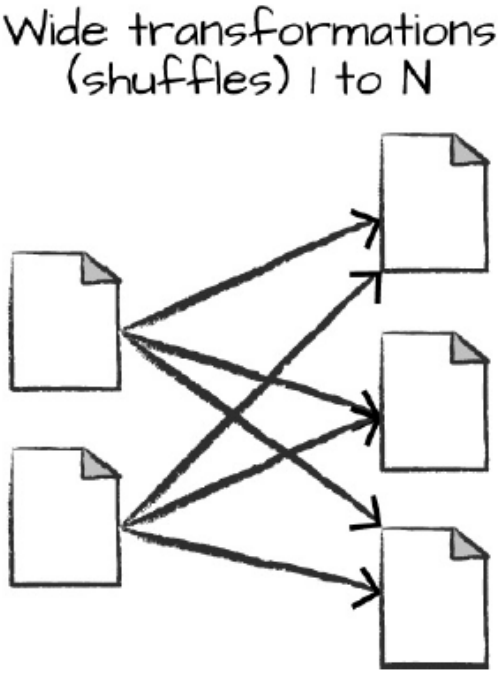


Figure 2-5. A wide dependency

Le Transformation sono semplicemente modi per specificare diverse serie di manipolazioni su dati. Questo conduce in un argomento chiamato Lazy Evaluation.

4.2 Lazy Evaluation

Significa che Spark aspetta fino all'ultimo momento prima di eseguire il grafico di istruzioni di calcolo. In Spark, invece di modificare i dati immediatamente quando esprimi qualche operazione, costruisci un piano di trasformazioni che vorresti applicare al tuo dati di origine. Aspettando fino all'ultimo minuto per eseguire il codice, Spark compila questo piano da le tue trasformazioni DataFrame grezze in un piano fisico semplificato che funzionerà nel modo più efficiente possibile in tutto il cluster. Ciò offre immensi vantaggi perché Spark può ottimizzare l'intero flusso di dati da un capo all'altro. Un esempio di questo è qualcosa chiamato pushdown predicato su DataFrame. Se creiamo un lavoro Spark di grandi dimensioni ma alla fine specifichiamo un filtro che ci richiede solo di recuperare una riga dai nostri dati di origine, il modo più efficiente per eseguirlo è accedere al singolo registrare di cui abbiamo bisogno. Spark lo ottimizzerà effettivamente per noi spingendo il filtro verso il basso automaticamente.

4.3 Actions

Le Transformation ci consentono di costruire un nostro piano di trasformazione logico. Un'azione indica a Spark di calcolare un risultato da una serie di Transformation. L'azione più semplice che si può intraprendere è count, che ci restituisce il numero totale di record nel DataFrame:

```
divisBy2.count()
```

Esistono tre tipi di actions:

- Azioni per visualizzare i dati nella console
- Azioni per raccogliere dati su oggetti nativi nella rispettiva lingua
- Azioni da scrivere sulle origini dati di output

Nello specificare action, si avvia un job Spark che esegue la nostra filter Transformation (narrow Transformation), oppure un'aggregazione (wide Transformation).

4.4 Fase Preliminare

Il progetto è stato realizzato in Python, le librerie utilizzate fanno parte tutte del pacchetto Pyspark versione 3.1.4, in particolare si è usufruito di pyspark.sql.Dataframe, che è stato illustrato nel paragrafo 2.1.3, e pyspark.sql.functions che consiste di operazioni su set di dati che sono principalmente correlate al calcolo dell'analisi.
Come illustrato in precedenza Spark possiede un API SQL con molte funzioni integrate che vengono incontro nelle operazioni SQL. Alcune più importati sono Count, avg, collect_list, first, mean, max, variance, sum. In ogni modo queste libreria function ci permette anche di eseguire delle proprie funzioni realizzate, definite dall'utente.

4.5 File di input

I file di input necessari al funzionamento dell'applicazione sono divi in tre file che hanno schemi di intestazione diversi. In entrambi i casi i file sono in formato csv, ampiamente supportato sia da Spark che dalle sue librerie.
I file csv sono stati raccolti da un ente di terze parti reperibili a questo indirizzo, (`https://github.com/lopezbec/COVID19_Tweets_Dataset`). Il dataset raccolto essendo suddiviso organizzativo in directory per anno 2020, 2021, che a loro volta

erano raggruppati per mesi dove vi si trovano per ogni mese un migliaio di file csv che strutturano i dati raccolti ora per ora.
Prima di tutto sono stati uniti questi piccoli file di pochi MB, tramite linea di comando:

find /path/to/directory/ −name ∗.csv −exec cat {} + > merged.file

Nome_Dataset	Dimensione
Summary_details	79,03 GB
Summary_hastag	16,61 GB
Summary_sentiment	67 GB

È stato necessario mantenere suddivisi i tre dataset, per il loro differente header schema, questo ci ha permesso di realizzare in fase di calcolo ed estrazioni di statistiche e query dettagliate in modo tale da effettuare delle join tra i dataset per mezzo della key "Tweet_ID" che identifica ciascun dato raccolto in modo univoco all'interno del set di dati.

4.6 Fase di analisi

Come primo passo vengono prelevati dall'HDFS e caricati in memoria, i dataset descritti nei precedenti paragrafi 2.3 e 3.5, convertiti sotto forma di Dataframes, per permettere la distribuzione del carico di lavoro sull'intero cluster da parte di Apache Spark. Dopodiché vengono eseguite ciascuna query realizzata, catturando ciascun tempo di esecuzione, per poi salvare i risultati in una struttura dati temporanea, che alla fine dell'esecuzione verranno salvati sull'HDFS.
Il risultato finale sarà quindi un file in formato csv che conterrà le varie informazioni che sono state estrapolate dalle query realizzate.

Capitolo 5

5 Implementazione

4.1 Query Dataset Summary Details

Nel seguente dataset Summary Details (79,03 GB), sono stati estratti informazioni dal punto di vista statistico dalle seguenti query mostrate in seguito.

Query topLang raggruppa le righe che hanno lo stesso linguaggio contando quanti tweet sono stati descritti con un determinata lingua e tutti le righe tramite il metodo

collect vengono memorizzate in una struttura dati in cui verranno memorizzati tutti i risultati delle altre query per essere salvati su un file csv alla fine dell'esecuzione.

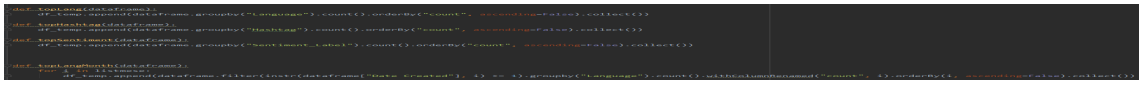
È stata realizzata una versione della query TopLang nel linguaggio di programmazione Scala per effettuare test di confronto tra i diversi linguaggi di programmazione.

```scala
object TopLang {

  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().config("spark.master","yarn").getOrCreate()
    print("Create Spark Session")
    val pathcsv = args(0)
    print(pathcsv)

    var dfTopLang = spark.read.option("header","true").csv(pathcsv)
    val df = dfTopLang.groupBy( col1 = "Language").count()

    df.write.format( source = "csv").option("header","false").option("path","/user/soa/cetrangolo_santonastaso/toplang.csv").save()

    spark.stop()

  }

}
```

Inoltre, per esecuzione sulla macchina locale è stata realizzata la versione che esegue la query TopLang su un DBMS MySQL, per effettuare confronti con ambiente distribuito.

```python
mycursor = mydb.cursor()
mycursor.execute("SELECT COUNT(Language) FROM summarydetails GROUP BY Language ORDER BY COUNT(Language) DESC ")
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

La Query topLangMonth è simile alla query precedente ma mantiene il conteggio della lingua per ogni mese.



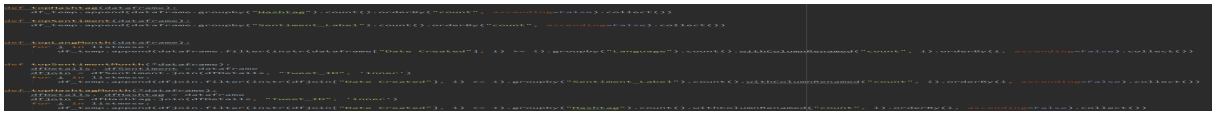
La query avgTweetsMonth calcola la media dei Tweet e Retweet avvenuti per ogni mese. La query totTweets_Retweet_Month invece calcola la somma dei Tweet e Retweet per ogni mese.

La query summaryMonth calcola il tasso di apprezzamento per ogni mese effettuando una somma di Retweets e Likes.

5.1 Query Dataset Summary Hashtag

Nel seguente dataset Summary Hashtag (16,61 GB), sono stati estrapolati altre informazioni dalle query mostrate in seguito.
La query topHashtag calcola gli hashtag utilizzati maggiormente per ogni tweet ordinandoli in modo decrescente.

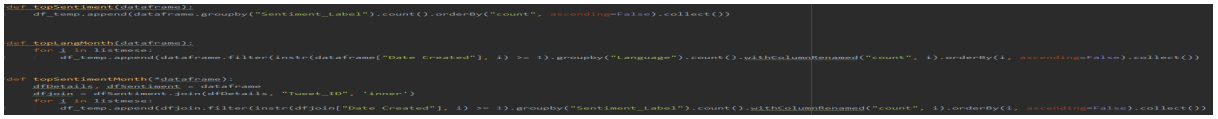
La query topHashtagMonth è simile alla query precedente calcolando gli hashtag più utilizzati raggruppando per ogni mese e ciò è stato realizzato tramite la join con il dataset Summary Details utilizzando la primary key Tweet_id.



5.2 Query Dataset Summary Sentiment

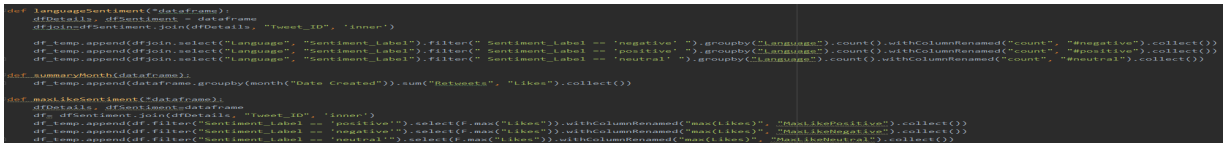
Nel seguente dataset Summary Sentiment (67 GB), sono stati ricavati altre informazioni dalle query mostrate in seguito.
La query topSentiment restituisce il conteggio dei vari sentimenti (positive, neutral, negative calcolati in base alla percentuale dei commenti, like, emoticon e retweet) di tutti i tweet effettuati.

La query topSentimentMonth è simile alla query precedente restituendo il conteggio dei vari sentimenti per ogni mese e ciò è stato realizzato tramite la join con il dataset Summary Details utilizzando la primary key Tweet_id.



La query languageSentiment per ogni sentimento (positive, neutral, negative) raggruppa per lingua (tramite la join tra il dataset Sentiment e Details), specificando il numero di tweet che hanno determinata lingua con quel sentimento

La query maxLikeSentiment estrapola per ogni sentimento il tweet che ottenuto il maggior like utilizzando sempre la join tra il dataset Details e Sentiment.



5.3 Dataset Join

Nel seguente paragrafo descriveremo delle query ottenute tra le join di tutte e tre dataset: SummaryDetails (79,03 GB), SummaryHashtag (16 GB), SummarySentiment (67 GB).
La query topAllDay calcola le varie query topLang, topHashtag e topSentiment nei giorni significativamente delicati per la pandemia Covid−19 e questi giorni sono 22 gennaio 2020 (Primi casi di Covid nel mondo), 21 febbraio 2020 (Primo caso di covid in Italia) 11 marzo 2020 (OMS dichiara il coronavirus pandemia), 28 ottobre 2020 (seconda ondata di covid), 14 dicembre 2020 (Inizio di campagna vaccinale in Inghilterra).

```
def topAllDay(*dataframe ):
    dfDetails,dfSentiment,dfHastag  = dataframe
    sj = dfHastag.join(dfSentiment, "Tweet_ID", 'inner')
    dfJoin = sj.join(dfDetails, "Tweet_ID", 'inner')
    topAllDay1(dfJoin)
    topAllDay2(dfJoin)
    topAllDay3(dfJoin)

def topAllDay1(dfJoin):
    for i in listgiorni:
        try:
            df = dfJoin.filter(instr(dfJoin["Date Created"], i) >= 1)
            topLang(df)
        except:
            # print("Giorno saltato: " + i)
            continue

def topAllDay2(dfJoin):
    for i in listgiorni:
        try:
            df = dfJoin.filter(instr(dfJoin["Date Created"], i) >= 1)
            topSentiment(df)
        except:
            # print("Giorno saltato: " + i)
            continue
```

```
def topAllDay3(dfJoin):
    for i in listgiorni:
        try:
            df = dfJoin.filter(instr(dfJoin["Date Created"], i) >= 1)
            topHashtag(df)
        except:
            # print("Giorno saltato: " + i)
            continue
```

La query maxHashtagSentiment calcola per ogni mese l'hashtag più utilizzato e su quest'ultimo il numero di tweet rispetto alla percentuale del sentimento.

```
def maxHashtagSentiment(*dataframe):
    dfHastag, dfSentiment, dfDetails=dataframe
    sj=dfHastag.join(dfSentiment,"Tweet_ID",'inner')
    dfJoin=sj.join(dfDetails,"Tweet_ID",'inner')

    for i in listmese:

        try:
            df = dfJoin.filter(instr(dfJoin["Date Created"], i) >= 1).groupby("Hashtag").count().withColumnRenamed(
                "count", i).orderBy(i, ascending=False).limit(1)
            maxHashtag=df.head()["Hashtag"]
            df_temp.append(dfJoin.filter("Sentiment_Label == 'positive'").where("Hashtag == '" + maxHashtag + "'").agg(F.count("Tweet_ID")).
                   withColumnRenamed("count(Tweet_ID)", "Positive_max_Hashtag_" + i).collect())
            df_temp.append(dfJoin.filter("Sentiment_Label == 'negative'").where("Hashtag =='" + maxHashtag + "'").agg(F.count("Tweet_ID")).
                   withColumnRenamed("count(Tweet_ID)", "Negative_max_Hashtag_" + i).collect())
            df_temp.append(dfJoin.filter("Sentiment_Label == 'neutral'").where("Hashtag == '" + maxHashtag + "'").agg(F.count("Tweet_ID")).
                   withColumnRenamed("count(Tweet_ID)", "Neutral_max_Hashtag_" + i).collect())

        except :
            #print("Mese saltato: "+i)
            continue
```

Capitolo 6

6 Test e valutazioni

I test sono stati effettuati utilizzando il cluster messo a disposizione dall'Università degli Studi di Salerno. L'accesso al cluster veniva effettuato attraverso il protocollo ssh grazie ad una chiave di sicurezza. Sono stati necessari diversi test per valutare le prestazioni dell'applicazione e le eventuali modifiche da apportare al codice.

6.1 Configurazione Cluster

- **Numero nodi:** 8
- **Hardware:** 8 CPU 2.70 Ghz – 31.42 GB RAM
- **Total CPUs:** 72
- **Sistema operativo:** Ubuntu 16.04
- **Ambiente software:**
  - Apache Spark version 2.3.3
  - Scala version 2.11.12
  - Java version 1.8.0
  - Python version 2.7

6.2 Configurazione Software

- **Spark master:** YARN
- **DFS replication: 2**
- **DFS blocksize:** 128 Mb
- **YARN nodemanager resource**
  - **Memory:** 32 GB
  - **Cpu−vcores:** 8

6.3 Configurazione esecuzione distribuita

Apache Spark prevede una serie di parametri addizionali che possono essere configurati per esecuzione delle applicazioni, per tale motivo sono stati necessari una prima serie di test per trovare la configurazione ottimale.
I parametri in questione sono rappresentati dal numero di esecutori da eseguire e il numero di core e la quantità di memoria assegnata ad ognuno di essi.
Partendo dal numero di core per esecutore si è preceduto a calcolare le restanti impostazioni. Tutti i calcoli sono effettuati ragionando per difetto e lasciando sempre una parte di risorse libere per l'overhead di esecuzione dei servizi di Apache Spark e dell'HDFS. Ciò è stato necessario in quanto utilizzando l'intero pool di risorse si sono riscontrati una serie di problemi.

6.4 Test configurazione Apache Spark 1 e Local

Il primo test ha come obiettivo di mettere a confronto tempi di esecuzione dell'applicazione su 10 GB file di input, eseguendo delle specifiche query mostrate nel capitolo precedente. Raccolti in forma tabellare si ha a confronto i test effettuati su una macchina locale, con 8 GB di RAM, 1 CPU, 4 cores. Il dataset è stato caricato e convertito in un Database su MySQL per poter realizzare i test in locale. Sulla macchina in locale si è riscontrato **Break Even** già a partire da un input pari 5 GB dovuto alle capacità ridotte della RAM. Infatti, il tempo di esecuzione come possiamo vedere su 10 GB risulta quattro volte rispetto le altre, dovuto alla saturazione delle risorse (RAM), perdendo tempo nell'andare a salvare risultati temporaneamente in memoria non volatile e ricaricare in RAM la successiva porzione di input per riprendere esecuzione.
Invece la query lanciata sul Cluster (python, scala) con una configurazione di 19 GB RAM, 1 CPUs, 1 core executor, hanno tempi di esecuzione con forte correlazione

tra di loro. Come sarà mostrato più avanti i test effettuati con applicativo Scala impiegano di gran lunga minor tempo.

Query	Tempo esecuzione	Dataset	Ambiente esecutivo	Linguaggio di programmazione
TopLang	1m 5s	summaryDetails	Cluster	python
TopLang	4m1s	summaryDetails	Locale	python
TopLang	40 s	summaryDetails	Cluster	scala

6.5 Test configurazione Apache Spark 2

Il secondo test aveva come obiettivo la saturazione delle risorse del cluster andando utilizzare il maggior numero di esecutori. Riservando un core di ogni nodo per il daemon di YARN, il cluster ha disposizione 7 core $*$ 8 nodi per un totale di 56 nodi. Assegnando 1 core per executor si potranno quindi istanziare 56 executor. Su ogni nodo saranno quindi in esecuzione 56 / 8 executor, per un risultato di 7 executor.
Avendo per ogni nodo 32 GB di memoria si ottiene un valore di 32 / 7 executor, ossia 4,57 GB di memoria per executor. Da questo risultato bisogna escludere il 7% all'incirca rappresentante l'overhead dell'heap per ottenere circa 27 GB di memoria per executor. Riassumendo questa configurazione presenta 56 executor con 1 core e 4 GB di memoria, per un tempo di esecuzione totale di 3 ore 26 minuti.

6.6 Test configurazione Apache Spark 3

Il terzo test ha come obiettivo l'ottimizzazione delle risorse, andando ad utilizzare core 6 per executor, valore che risulta il range superiore di quello indicato come buon compromesso per il throughput dell'HDFS. Riservando un core di ogni nodo per il deamon di YARN il cluster ha a disposizione 7 $*$ 8 nodi per un totale di 56 core.
Assegnando 6 core per executor si potranno quindi istanziare 56 / 6 executor, per un totale 9 executor. Su ogni nodo saranno quindi in esecuzione 9 su 8 executor per un totale di 1 executor circa per nodo.
Avendo ogni nodo 32 GB di memoria ogni executor potrà avere 32 GB di memoria. Da questo valore bisogna sempre escludere il $7-10\%$ rappresentante l'overhead dell'heap per ottenere circa 27 GB di memoria per executor. Riassumendo questa configurazione presenta 48 executor con 6 core e 27 GB di memoria, per un tempo totale di 4 ore 18 minuti.

6.7 Test configurazione Apache Spark 4

Il quarto test, anch'esso con obiettivo di ottimizzare le risorse utilizzate, assegnando 3 core executor, valore che risulta il range inferiore di quello indicato come buon compromesso per il throughput dell'HDFS. Riservando sempre un core di ogni nodo per il daemon YARN il cluster ha disposizione 56 core.
Assegnando 3 core per executor si potranno quindi istanziare 56 / 3 executor, per un totale di 18 executor. Su ogni nodo saranno quindi in esecuzione 18 / 8 executor, per un risultato di 2 executor per nodo. Di conseguenza ogni nodo avrà la metà della memoria 16 GB, da questo valore bisogna escludere il 7% rappresentante l'overhead dell'heap per ottenere all'incirca 13 GB di memoria per executor. Riassumendo questa configurazione presenta 18 executor con 3 core e 13 GB di memoria, per un tempo di esecuzione totale di 3 ore 58 minuti.

6.8 Test configurazione Apache Spark 5

Il quinto test, anch'esso con obiettivo di ottimizzare le risorse utilizzate utilizzando la stessa configurazione del precedente 6 core executor per nodo, 27 GB RAM. Dai test effettuati si è concluso che è la configurazione che permette di ottenere il miglior risultato in termini di tempo di esecuzione. Di seguito si trovano i tempi di esecuzione raccolti di ciascuna query dell'applicazione. Da notare il tempo di esecuzione della query TopLang eseguita su dataset di 79 GB sia in python che in scala. Tempo esecuzione query in Scala è di 1 minuto 06 secondi, rispetto ai 3 minuti di quella realizzata in Python.

Query	Tempo di esecuzione	Dataset	Linguaggio Di Programmazione
TopHashtag	3 m 50 s	Summary Hashtag	Python
TopSentiment	2 m 42 s	Summary Sentiment	Python
TopLang	3 m	Summary Details	Python
SummaryMonth	3 m 20 s	Summary Details	Python
LanguageSentiment	14 m 27 s	Details,Sentiment	Python
MaxLikeSentiment	14 m 42 s	Details,Sentiment	Python
TopHashtagMonth	15 m 32 s	Details,Hashtag	Python
TopLangMonth	16 m 33 s	Summary Details	Python
TotTweetsRetweetMonth	17 m 12 s	Summary Details	Python
AvgTweetsMonth	17 m 58 s	Summary Details	Python
TopSentimentMonth	47 m 59 s	Details,Sentiment	Python
TopAllDay	20 m 25 s	Details,Hashtag,Sentiment	Python
MaxHashtagSentiment	45 m 33 s	Details,Hashtag,Sentiment	Python

7 Conclusioni

Il lavoro presentato ha voluto dimostrare come determinati problemi possano essere risolti in maniera più efficace ed efficiente attraverso l'utilizzo di sistemi distribuiti. Nonostante la difficoltà naturale nella gestione di sistemi di questo tipo, grazie alla tecnologia fornita da Apache Spark le operazioni necessarie risultano ampiamente semplificate e in alcuni casi del tutto automatizzate o gestite dal software stesso. Non si voglia negare la necessità di conoscere i paradigmi caratterizzanti le architetture distribuite, Apache Spark infatti richiede comunque una serie di configurazioni iniziali che vanno ritoccate in corso d'opera, ma si può affermare che questo strumento porti una serie di vantaggi importanti a cui difficilmente un programmatore può rinunciare.
Molti aspetti legati al problema affrontato hanno trovato una rapida soluzione grazie al sistema distribuito, tra cui uno dei problemi principale è la memoria RAM disponibile nei singoli nodi del cluster perché non è abbastanza sufficiente per poter memorizzare i dataset citati nell'elaborato dovuto alla loro dimensione elevata, in più un altro problema è legato alla notevole differenza riguardante ai tempi di esecuzione tra un ambiente locale e un ambiente distribuito.
Un altro aspetto importante è il linguaggio di programmazione, perché nel capitolo 5 si può notare una notevole differenza nei tempi di esecuzione di una query implementata in Scala e Python, perché Scala è un linguaggio ibrido, funzionale e orientato agli oggetti, ed è tipizzato staticamente e durante l'esecuzione utilizza la Java Virtual Machine (JVM) che può sfruttare le librerie Java esistenti aumentando notevolmente le funzionalità disponibili. In più ci sono diversi motivi perché preferire Scala rispetto gli altri linguaggi.

Innanzitutto, Scala è open source e compatibile; infatti, una volta compilato con il suo compilatore, il codice diventa normalissimo bytecode per la JVM che può usare librerie Java ed essere perfettamente integrato in applicazioni Java esistenti, senza referenziare classloader particolari perché' non viene interpretato a run−time, come Groovy o Clojure.
Le applicazioni moderne necessitano sempre più di nascere scalabili per tanti motivi; Scala (**SCAlable LAnguage**) permette e anzi favorisce la scrittura di un codice che, come può essere eseguito in un'applicazione console di un PC, può essere eseguito in un cluster. Come sappiamo, in Java non è semplice rendere un'applicazione o una libreria veramente scalabili: ciò richiede uno sforzo di design non indifferente e il risultato può essere un codice sì scalabile ma spesso di difficile gestione, lettura e manutenzione.
Il terzo motivo è infatti è che Scala è conciso e di alto livello. Ciò permette non solo di realizzare un DSL in modo molto semplice, ma anche di risolvere alcuni problemi tipici della maggioranza dei linguaggi.

Un altro motivo che è stato notato nei nostri test è la differenza di performance tra Scala e Python è l'HDFS (Hadoop Distributed File System) perché Python interagisce molto male con i servizi Hadoop, quindi gli sviluppatori devono utilizzare

librerie di terze parti (come hadoopy), mentre Scala interagisce direttamente con Hadoop tramite l'API nativa di Hadoop in Java.
Si è costato dai nostri test che le query che hanno un tempo di esecuzione abbastanza elevato sono quelle riguardanti ad ottenere risultati statistici caratterizzazione mensili e determinate date. Questo è dovuto al fatto nell'effettuare il matching per ogni riga contenuta nel dataset.
Concludendo i test hanno dimostrato come l'approccio utilizzato risulti di valido e scalabile, rendendo soddisfacente l'intero progetto.

Bibliografia

[1] Spark The Definitve Guide (Big Data Processing made simple) O'Reilly Bill Chambers & Matei Zaharia First Edition 2018.

https://cloud.google.com/bigquery/

https://www.geeksforgeeks.org/google-cloud-platform-introduction-to-bigquery/

"About Twitter's APIs", help.twitter.com/en/rules−and−policies/twitter−api

How−to: Tune Your Apache Spark Jobs (Spark−Tuning.pdf)

Dataset utilizzati:
1. https://github.com/lopezbec/COVID19_Tweets_Dataset

Documentazione utilizzati per implementazione:
https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html
https://sqlandhadoop.com/pyspark-convert-sql-to-dataframe
https://towardsdatascience.com/pyspark-and-sparksql-basics-6cb4bf967e53