APACHE SPARK STREAMING PER

IL MONITORAGGIO DI TWEETS

UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Matteo D'Eramo (Mat. 327994)

Alessandro Noto (Mat. 328598)

INTRODUZIONE

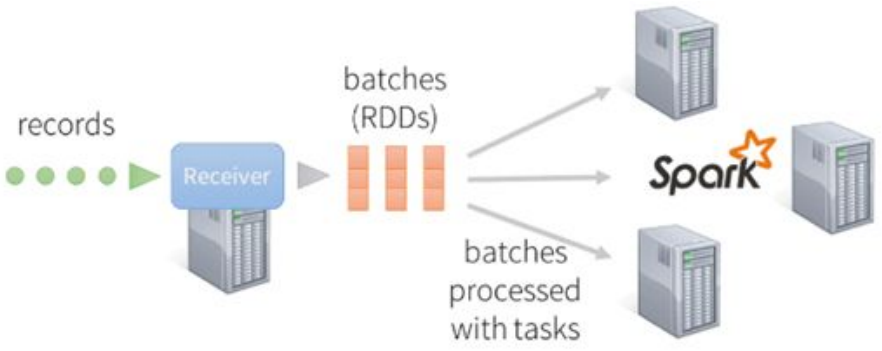
Uno degli aspetti più importanti per l'analisi di dati riguarda la rappresentazione visuale; poter dare una rappresentazione visuale chiara ed esaustiva dei dati permette sicuramente di poter analizzare in modo semplice le relazioni che intercorrono tra di essi, o poterne trarre conclusioni che possono essere determinanti per prendere decisioni riguardo un determinato aspetto.

In questa applicazione ci siamo posti il problema di poter dare una rappresentazione visuale in tempo reale di Tweets; nello specifico si intende monitorare, per un certo numero di Tweets, l'andamento dei like e retweet di ognuno di essi. Per realizzare un'applicazione di questo tipo è necessario utilizzare quindi un *software* che permetta di compiere operazioni in tempo reale su un flusso continuo di informazioni.

Il software in questione è *Apache Spark*, il quale è un motore di esecuzione *in-memory* per il calcolo distribuito che permette di eseguire di per sé operazioni ad una velocità molto superiore rispetto ad altri motori di calcolo distribuito, come per esempio *Hadoop*. I vantaggi di Spark però risiedono anche nella possibilità di utilizzare più linguaggi di programmazione e soprattutto nella presenza di vari moduli e librerie utilizzabili per il *processing* di dati in contesti differenti, come *processing* di grafi (*GraphX*) o per *machine learning* (*Mlib*); tra queste vi è anche *Spark Streaming*, che permette l'analisi *real time* di stream di dati. *Spark Streaming* consiste in un'estensione delle API di Spark ed il quale "*consente un'elaborazione del flusso scalabile, ad alta velocità e fault-tolerant di flussi di dati in tempo reale*". I dati possono essere inviati a partire da fonti eterogenee come piattaforme di *stream-processing* (*Kafka*), servizi di *storage* (come S3) o *socket TCP*. Il meccanismo alla base di *spark Streaming* è molto semplice: esso riceve flussi di dati di input in tempo reale e divide i dati in *batch* (divisi in base ad un certo intervallo di tempo), che vengono quindi elaborati dal motore Spark per generare il flusso finale come mostrato in figura:

Quello che otteniamo è uno "stream discretizzato" e per ogni *batch* verranno eseguite determinate operazioni. *Spark* fa al caso nostro non solo per le caratteristiche appena viste ma anche perché consente di eseguire il calcolo in maniera distribuita su un *cluster*; questo porta ovviamente dei vantaggi

considerevoli in termini di scalabilità in quanto l'esecuzione delle operazioni per ogni batch può essere distribuita su esecutori presenti in differenti nodi del cluster e permettendo quindi potenzialmente di gestire grandi quantità di dati in ingresso senza dover disporre di una memoria primaria di dimensione esagerata:



L'applicazione si basa su 3 fasi:

1. Un server riceverà per mezzo di APIs i valori di likes e retweets relativi agli ultimi 30 *tweet* di un profilo scelto dall'utente; successivamente li inoltrerà sotto forma di stringa ad *spark Streaming* tramite *socket TCP.*

2. Il motore di esecuzione *Apache Spark Streaming* riceverà i dati sotto forma di stringa e andrà ad effettuare uno *splitting* in modo da ottenere le singole stringhe relative ai dati dei singoli *tweet*. Le stringhe verranno poi inviate tramite protocollo UDP ad un altro server.

3. Il server di arrivo processerà le stringhe in modo che ne sia data una rappresentazione grafica (in JavaScript).

Per il progetto è stato utilizzato il linguaggio di programmazione *Python* sia per il primo server che per il programma eseguito da *Spark Streaming*; la scelta nel primo caso è stata dettata dalla facilità di utilizzo delle API di Twitter, mentre nel secondo caso è stato scelto perché supportato da *Apache Spark* e perché le API di *Spark* in *Python* sono utilizzabili in maniera molto facile e intuitiva. Per quanto riguarda il server di arrivo, questo è stato realizzato mediante *node.js*, il quale permette la creazione di server in cui è integrato l'utilizzo di JavaScript.

Apache Spark è utilizzabile mediante differenti *cluster manager*; in questo caso utilizziamo YARN, il quale è un cluster manager che abbiamo ampiamente analizzato a lezione e per cui possiamo fare delle considerazioni più specifiche dal punto di vista di un ipotetico cluster in cui il programma verrebbe eseguito. Per quanto riguarda lo *storage*, non è stata implementata alcuna funzionalità riguardo la memorizzazione dei dati quindi per il cluster è sufficiente la presenza di un file system distribuito.

DATAFLOW E TECNOLOGIE UTILIZZATE

Come accennato prima il flusso di dati si sviluppa in tre fasi. Nella prima fase il flusso di dati è generato a partire dall'utilizzo delle API nel primo server. Le API utilizzate sono quelle della libreria *Tweepy* la quale permette l'utilizzo delle API di Twitter e rende più semplice l'implementazione. Dopo essersi autenticati mediante degli *access token*, ottenuti registrando l'applicazione nel *twitter developers*, siamo in grado di poter utilizzare le API. Tali API consentono di ottenere i dati di al più 200 tweets pubblicati da un determinato utente; nel nostro caso abbiamo considerato, per motivi di chiarezza del grafico, solamente gli ultimi 30 tweets. A partire dai dati ottenuti sotto forma di lista, andiamo ad estrarre i dati dei campi di nostro interesse (*created_at*, *favorite_count*, *retweet_count*).

```
tweet_dates = [str(tweet.created_at.strftime("%d-%m-%y?%H:%M")) for tweet in tweets]
tweet_likes = [str(tweet.favorite_count) for tweet in tweets]
tweet_retweet = [str(tweet.retweet_count) for tweet in tweets]
tot = np.stack([tweet_dates,tweet_likes,tweet_retweet],axis=1);
```

I dati filtrati vengono poi combinati in un unico array bidimensionale, trasformato in seguito in una stringa unica nel seguente formato:

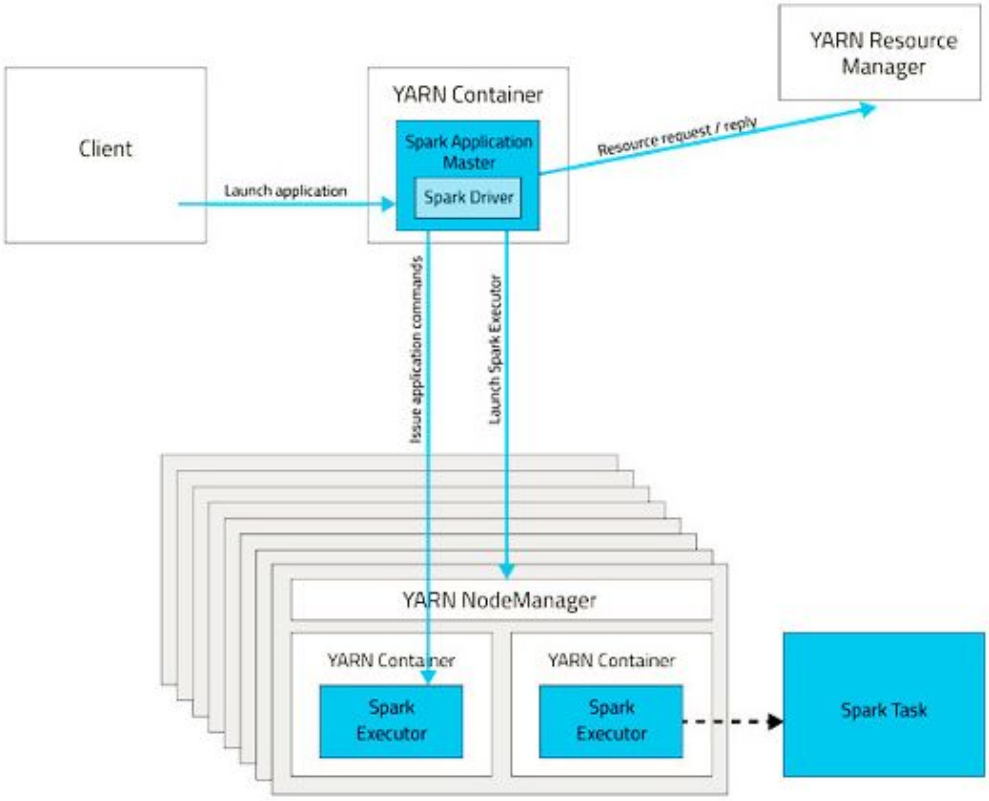
…./08-01-21?15:36,5688,197/08-01-21?02:15,62604,4350/08-01-21?01:35,193250,8959/….

Alla base del funzionamento di qualsiasi applicazione in Spark vi è uno *SparkContext*, il quale rappresenta l'*entry point* principale per le funzionalità di Spark; questo è l'oggetto che permette di connetterci con il *cluster manager* e ottenere gli esecutori per l'esecuzione dei vari *task*. In *Spark Streaming* otteniamo a partire dallo *SparkContext* un oggetto *StreamingContext* il quale funge da elemento base per eseguire un qualsiasi programma di questo tipo. Uno degli attributi di questo oggetto riguarda l'intervallo di tempo in cui andremo a "tagliare" lo streaming in input; nel nostro caso utilizzeremo un intervallo di due secondi. Da notare che la scelta dell'intervallo di tempo influenza le performance del cluster e la quantità di dati contenuta in ogni *batch*.

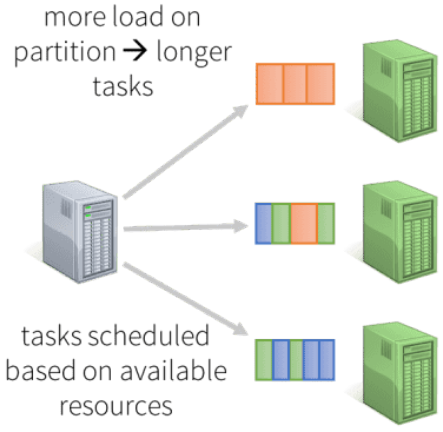
A questo punto si deve definire una sorgente di input creando un *discretized stream* (da ora in poi lo chiameremo *DStream*); il server si basa su protocollo TCP, pertanto utilizziamo una funzione (*socketTextStream*) che permette di connetterci al server per ottenere i dati di input.

```
ssc = StreamingContext(sc, 2)
lines = ssc.socketTextStream("localhost", 8000)
```

Come già detto in precedenza andremo ad eseguire il programma mediante l'utilizzo del *cluster manager* YARN in modalità *cluster*: in questa modalità lo *SparkDriver* contenente lo *SparkContext* è eseguito come *application Master* di YARN e il processo di acquisizione degli esecutori da parte dello *SparkDriver* avviene mediante il *resource manager*, come mostrato in figura.

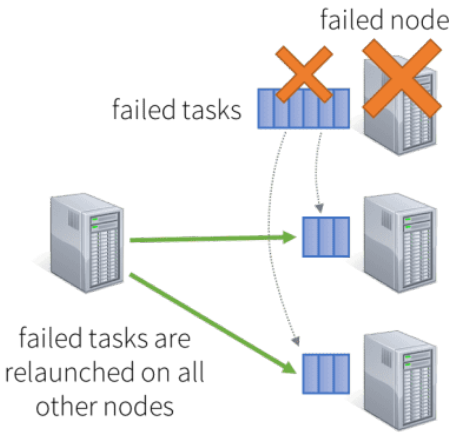


Secondo questa modalità andiamo ad eseguire il programma in un *cluster* composto da un solo nodo, ovvero la nostra macchina virtuale.



Nel nostro progetto abbiamo considerato l'invio di dati relativi a 30 *tweets* e intuitivamente avremmo un tempo di processamento maggiore nel caso in cui aumentassero i tweet. Da notare che *Spark* sfrutta un meccanismo di *load balancing* per bilanciare i *task* in base alla tempo necessario per la loro esecuzione; questo evita che si creino dei nodi che saranno dei colli di bottiglia per l'esecuzione.

Inoltre in caso di malfunzionamento di un nodo, i *task* assegnati ad esso saranno "rilanciati" in parallelo sugli altri nodi operanti, come mostrato in figura:



Spark permette di definire delle proprietà che vanno a regolare l'esecuzione del programma; tra queste, di particolare rilevanza ci sono quelle relative alla memoria assegnata allo *SparkDriver* e agli esecutori e al numero di questi ultimi.
Per quanto riguarda il consumo di memoria di esecutori e *driver* questo dipende da fattori eterogenei come ad esempio la dimensione dei dati per ogni *batch* o il tipo di trasformazioni che vengono utilizzate (per esempio, nel caso utilizzassimo trasformazioni veloci come *narrow transformation* oppure lente come le *wide transformation*). Nella nostra applicazione abbiamo sfruttato il valore di default per la memoria di driver ed esecutori, corrispondente ad 1 GB; tali valori per il nostro caso d'uso sono più che sufficienti ma nell'ipotesi in cui i dati in input siano di dimensione di gran lunga maggiori, le proprietà andrebbero impostate diversamente (2 GB, 4 GB, …).
E' ovvio che, a fronte di una richiesta di memoria primaria (RAM) maggiore, l'opzione migliore consiste nell'aggiungere nodi al *cluster* aumentando gli esecutori dell'applicazione, che nel nostro caso sono 2:

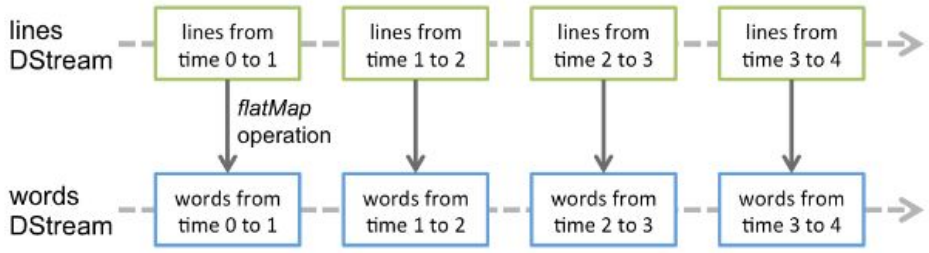
Executors

Show 20 entries

Executor ID ▲	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
driver	10.0.2.15:32961	Active	0	39.3 KiB / 366.3 MiB	0.0 B	0	0	0	0	0
1	bigdata-VirtualBox:37059	Active	0	27.1 KiB / 366.3 MiB	0.0 B	1	1	0	33	34
2	bigdata-VirtualBox:44999	Active	0	12.2 KiB / 366.3 MiB	0.0 B	1	0	0	85	85

Riprendendo il discorso relativo alle operazioni svolte dal programma, a partire dal *DStream* ottenuto avremo una RDD per ogni *batch* a cui potrò applicare delle

specifiche operazioni come se fossero considerate singolarmente, in modo da filtrarle o ottenerne di nuove, come mostrato in figura:

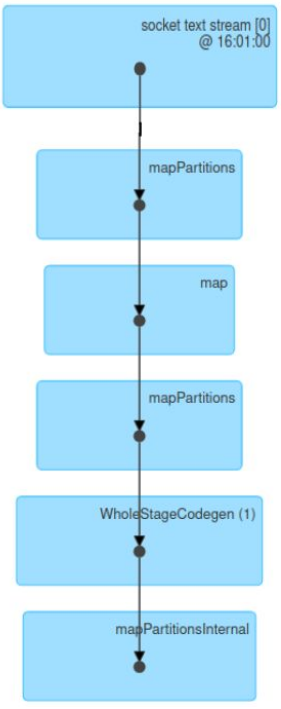


Nel nostro caso utilizziamo sul *DStream* una funzione *flatMap()* il quale restituisce un nuovo *DStream* in cui ogni elemento è sottoposto a una funzione (passata come attributo di *flatMap*); la funzione che passiamo permette di *splittare* la stringa contenuta nella RDD di un qualsiasi intervallo in un insieme di RDD contenenti le singole stringhe.

```
values = lines.flatMap(lambda x:str(x).split('/'))
```

A questo punto applichiamo al nuovo *DStream* ottenuto una funzione *foreachRDD* il quale è un operatore di output che a sua volta applica una funzione (passata sempre come parametro) ad ogni RDD del *DStream*. La funzione in questione determina il modo in cui le RDD saranno processate e successivamente inviate al server: inizialmente andiamo a creare un *dataframe* dal *DStream* mediante un oggetto *SparkSession* (definito come un *singleton*); questa conversione ci permette di manipolare i dati sfruttando le caratteristiche dei *dataframe*.

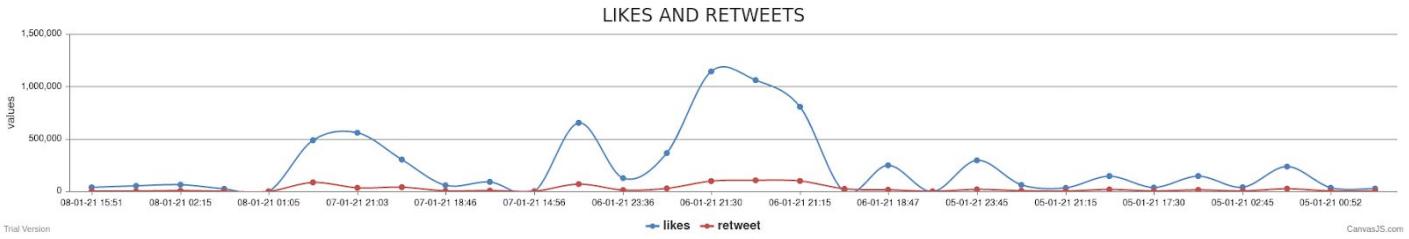
```
rowRdd = rdd.map(lambda w: Row(value=w))

DataFrame = spark.createDataFrame(rowRdd)
```



A questo punto possiamo inviare i dati al server di arrivo mediante l'utilizzo di una funzione *send_df*: questa funzione prende in ingresso il data frame, un oggetto socket e la coppia indirizzo-porta (*"127.0.0.1",3001*) del server. Il *dataframe* viene convertito in un array che verrà poi utilizzato per inviare una ad una le stringhe contenenti i valori dei vari tweets.

All'inizio dell'esecuzione dell'applicazione è fornito un *tracking*URL dove è possibile monitorare informazioni relative al programma in esecuzione, come ad esempio il DAG (*Direct Acyclic Graph*) delle operazioni e statistiche relative alle esecuzioni del programma (*input Rate*, *scheduling delay*, *processing time*…).

I dati vengono inviati tramite protocollo UDP al server implementato in *Node.js* il quale sfrutta UDP per la ricezione dei messaggi e HTTP per la visualizzazione dei dati in una pagina web. In particolare abbiamo utilizzato il modulo *express* per la realizzazione dell'interfaccia web (all'indirizzo *localhost:3000/dashboard*) ed il modulo *node-cache* per la memorizzazione e l'aggiornamento dei dati ricevuti. Connettendosi alla pagina web precedentemente indicata viene visualizzata la rappresentazione grafica realizzata tramite l'API di *canvasJS*.



CASI D'USO

L'applicazione può essere utilizzata per analizzare l'andamento in tempo reale dei likes e dei retweets relativi ai tweets pubblicati da un determinato utente.
Supponiamo di essere un'azienda che vuole analizzare l'andamento delle proprie pubblicazioni in Twitter; l'analisi visuale delle interazioni dei tweet del suo profilo potrà dare una misura dell'esposizione mediatica di essa nel social network, il quale è un aspetto sempre più importante per la crescita e la visibilità delle aziende.
E' possibile testare l'applicazione inserendo il nome utente di un profilo a piacimento (sono indicati alcuni esempi nel file README.txt).

LIMITI E POSSIBILI ESTENSIONI

L'applicazione realizzata presenta alcuni limiti su diversi aspetti:

- Una ovvia limitazione riguarda il numero di tweets che possono essere analizzati, in quanto le APIs di Twitter consentono di acquisire soltanto i dati relativi alle ultime 200 pubblicazioni.

- L'applicazione è fortemente "*profilo-dipendente*", in quanto l'analisi sarà differente in relazione alla frequenza di pubblicazione di un profilo rispetto ad un altro: ad esempio il profilo Twitter della Repubblica condivide oltre 50

tweets al giorno mentre esistono profili inattivi con 10 pubblicazioni nell'arco di molti anni.

- In un ipotetico funzionamento continuativo dell'applicazione 24/7, considerare gli ultimi 30 tweets non permette l'analisi a lungo termine di tutti i tweets pubblicati dall'avvio dell'applicazione. Per ovviare a questo sarebbe necessaria l'implementazione di un sistema di *storage* che permetta il salvataggio secondo fissati intervalli di tempo dei dati relativi ai tweets.

- *Spark* può essere spento solo manualmente, in un modo non propriamente corretto; questo aspetto non è stato approfondito per mancanza di tempo.

Per quanto riguarda i possibili sviluppi futuri:

- Implementazione di un sistema di *storage distribuito* che permetta il *backup* dei dati relativi ai tweets ogni intervallo di tempo; un possibile database distribuito adatto a questo tipo di compito potrebbe essere *MongoDB*, in quanto è un *database* orientato all'aggregato in cui per ogni tweet si potrebbe salvare documenti strutturati come in questo esempio:

```
{
  _id: "data e orario di pubblicazione",
  likes: "n° like",
  retweets: "n° retweet"
}
```

  Inoltre *Node.js* e *MongoDB* sono fortemente compatibili in quanto entrambi utilizzano *JavaScript*.

- Realizzazione di un programma tramite *Spark* che aggiorna automaticamente i valori presenti sul *database* relativi ai tweets meno recenti.

- Rendere possibile l'analisi dei dati di più utenti, anche nello stesso grafico, per confrontarli direttamente.