Report progetto di Big Data: Classifica delle città sulla base della differenza tra la gravità media degli incidenti dell'ultimo anno e quella dell'anno precedente

Luca Ragazzi - Mat. 0000897452

July 6, 2020

Contents

1 Introduzione

1.1 Il Dataset

Il dataset, reperibile al link `https://www.kaggle.com/sobhanmoosavi/us-accidents` e scaricabile al link `https://www.kaggle.com/sobhanmoosavi/us-accidents/download`, raccoglie i dati degli incidenti stradali avvenuti dal 2015 al 2019 in 49 stati diversi degli Stati Uniti. I dati sono stati raccolti utilizzando diversi provider, tra cui due API che forniscono dati sugli eventi di traffico in streaming. Queste API trasmettono gli eventi sul traffico catturati da diverse entità, come i dipartimenti di trasporto, le forze dell'ordine, le telecamere del traffico e i sensori del traffico all'interno delle reti stradali.

1.1.1 Descrizione dei dati

Il dataset presenta un unico file in formato CSV in cui sono presenti circa 3 milioni di record e occupa poco più di 1 GB di memoria. Il file è composto da 49 colonne, ma per l'analisi sono sufficienti solo le seguenti 3:

- **Severity**: definisce la gravità dell'incidente, un numero compreso tra 1 e 4, dove 1 indica il minor impatto sul traffico;
- **Start_Time**: indica la data e l'ora di inizio dell'incidente nel fuso orario locale;
- **City**: mostra la città in cui è avvenuto l'incidente.

1.2 Preparazione dei dati

In fase di pre-processing è stato rimosso l'header contenente le informazioni relative a ciascuna colonna del file CSV. Inoltre, sono stati analizzati i dati relativi alle colonne di interesse ed è risultato che non vi sono dati mancanti.

Il dataset è stato memorizzato su HDFS ed è raggiungibile al seguente path: `hdfs:/user/aravaglia/exam_lr/accidents.csv`.

2 Job

2.1 Descrizione

L'analisi che il progetto intende sviluppare sarà realizzata sia in MapReduce che in Spark, ed è la seguente:

"Stilare la classifica delle prime 100 città sulla base della differenza tra la gravità media degli incidenti dell'ultimo anno e quella dell'anno precedente".

2.2 Implementazione MapReduce

Siccome l'analisi da implementare non è fattibile con un solo job map-reduce, la soluzione è stata quella di creare 2 job concatenati. Il secondo job prenderà in input l'output del primo job map-reduce e continuerà l'analisi.

2.2.1 Job #1

Il primo job ha l'obiettivo di ottenere per ogni città e anno la relativa media di gravità di incidenti stradali.

Nello specifico, il mapper ha il compito di leggere correttamente il file CSV in input e mappare la gravità dell'incidente avvenuto in un determinato anno e città.

MAP - key: *(city, year)*, value: *(severity, 1)*

Con il reducer è stata calcolata la media dei valori di gravità relativi ad ogni anno e città.

REDUCE - key: *(city, year)*, value: *(severityAvg)*

Di seguito un estratto dell'output del primo job map-reduce:

```
2018,Whiteford  2.8
2018,Whitestown 3.05
2018,Willis     2.7857142857142856
2018,Windham    2.3859649122807016
2018,Winter Garden      2.1535433070866143
2018,Wixom      2.7285714285714286
2018,Woodford   2.5806451612903225
2018,Woodgate   2.0
2018,Wynnewood  2.019230769230769
2018,Zamora     2.7142857142857144
2018,Zellwood   2.0
2019,Abbottstown        2.0
2019,Acampo     2.035971223021583
2019,Adelphi    2.0
2019,Akeley     2.0
2019,Alabaster  2.6835443037974684
```

Figure 1: Output del primo job map-reduce del tipo: *(city_year, severityAvg)*.

2.2.2 Job #2

Il secondo job legge in input l'output del primo job map-reduce.

Il mapper è delegato a mappare, per ogni città, l'anno e la gravità media di incidenti.

MAP - key: *(city)*, value: *(year, severityAvg)*

Il reducer controlla che per ogni città siano presenti i valori di gravità media relativi agli ultimi 2 anni. Se sono presenti ne calcola la differenza, altrimenti non considera la città nel risultato finale.

Per ottenere in output la classifica delle prime 100 città con la differenza più alta del valore medio di gravità è stato utilizzato il metodo di MapReduce *"cleanup()"*, che viene eseguito solo al termine del ciclo di vita del reducer. In questo modo è stata precedentemente popolata una mappa contenente le città e i relativi valori di differenza di gravità, e solo alla fine è stata ordinata per valore e sono state estrapolate le prime 100 coppie.

REDUCE - key: *(city)*, value: *(diffSeverityAvg)*

Questo rappresenta l'output del secondo job map-reduce e quindi il risultato finale dell'analisi. Di seguito un estratto:

```
Bridgeboro      2
Haddam  2
Upatoi  2
Rydal   2
Peachland       2
Waveland        2
Archbold        2
Mc Alisterville 2
Orderville      1.963
Mount Horeb     1.8333
Guyton  1.8
Dillsboro       1.8
Kenna   1.75
Dyer    1.75
Keystone        1.7143
Meadville       1.7128
```

Figure 2: Output del secondo job map-reduce del tipo: *(city, diffSeverityAvg)*.

2.2.3 Considerazioni sulle performance

Per quanto riguarda il primo job è stato utilizzato un Combiner dopo la fase di Map, e prima della fase di Shuffle e Sort, per abilitare una pre-aggregazione dei dati, per cui riducendo la quantità di dati intermedi e il traffico di rete. L'utilizzo del Combiner è possibile solo quando la funzione di Reduce è associativa e commutativa. Siccome il reducer era incaricato a calcolare la media, è stato applicato un "work-around" per poter utilizzare il Combiner anche in questa situazione, perché di solito la media rappresenta un esempio di funzione non associativa.

Quando si esegue l'applicazione è possibile configurare da riga di comando il numero di task di reduce del primo job con il quarto parametro. Per il secondo job, invece, è stato impostato un unico task di reduce. Bisogna comunque considerare che avere troppi task può comportare un peggioramento delle performance a causa di un eccessivo overhead dovuto all'allocazione delle risorse e la gestione dei thread.

É possibile analizzare le statistiche del primo job su YARN al link `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1583679666662_4024`. Di seguito un estratto con le informazioni relative all'esecuzione del primo job:

```
        FILE: Number of bytes read=556466
        FILE: Number of bytes written=5696325
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=1124766807
        HDFS: Number of bytes written=477854
        HDFS: Number of read operations=87
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=40
Job Counters
        Launched map tasks=9
        Launched reduce tasks=20
        Data-local map tasks=5
        Rack-local map tasks=4
        Total time spent by all maps in occupied slots (ms)=1983744
        Total time spent by all reduces in occupied slots (ms)=158669
        Total time spent by all map tasks (ms)=1983744
        Total time spent by all reduce tasks (ms)=158669
        Total vcore-milliseconds taken by all map tasks=1983744
        Total vcore-milliseconds taken by all reduce tasks=158669
        Total megabyte-milliseconds taken by all map tasks=2031353856
        Total megabyte-milliseconds taken by all reduce tasks=162477056
Map-Reduce Framework
        Map input records=2974335
        Map output records=1846245
        Map output bytes=45765318
        Map output materialized bytes=861172
        Input split bytes=1242
        Combine input records=1846245
        Combine output records=55386
        Reduce input groups=18771
        Reduce shuffle bytes=861172
        Reduce input records=55386
        Reduce output records=18771
        Spilled Records=110772
        Shuffled Maps =180
        Failed Shuffles=0
        Merged Map outputs=180
        GC time elapsed (ms)=5471
        CPU time spent (ms)=2012400
        Physical memory (bytes) snapshot=8635662336
        Virtual memory (bytes) snapshot=46193086464
        Total committed heap usage (bytes)=11006377984
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=1124765565
File Output Format Counters
        Bytes Written=477854
```

Figure 3: Informazioni sul primo job map-reduce.

Le statistiche del secondo job sono analizzabili su YARN al link `http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1583679666662_4028`. Di seguito un estratto con le informazioni relative all'esecuzione del secondo job:

```
        FILE: Number of bytes read=195308
        FILE: Number of bytes written=3556332
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=480814
        HDFS: Number of bytes written=1273
        HDFS: Number of read operations=63
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
Job Counters
        Launched map tasks=20
        Launched reduce tasks=1
        Data-local map tasks=14
        Rack-local map tasks=6
        Total time spent by all maps in occupied slots (ms)=158717
        Total time spent by all reduces in occupied slots (ms)=7066
        Total time spent by all map tasks (ms)=158717
        Total time spent by all reduce tasks (ms)=7066
        Total vcore-milliseconds taken by all map tasks=158717
        Total vcore-milliseconds taken by all reduce tasks=7066
        Total megabyte-milliseconds taken by all map tasks=162526208
        Total megabyte-milliseconds taken by all reduce tasks=7235584
Map-Reduce Framework
        Map input records=18771
        Map output records=18771
        Map output bytes=435739
        Map output materialized bytes=268525
        Input split bytes=2960
        Combine input records=0
        Combine output records=0
        Reduce input groups=10780
        Reduce shuffle bytes=268525
        Reduce input records=18771
        Reduce output records=100
        Spilled Records=37542
        Shuffled Maps =20
        Failed Shuffles=0
        Merged Map outputs=20
        GC time elapsed (ms)=1064
        CPU time spent (ms)=34700
        Physical memory (bytes) snapshot=9467277312
        Virtual memory (bytes) snapshot=33219416064
        Total committed heap usage (bytes)=11291590656
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=477854
File Output Format Counters
        Bytes Written=1273
```

Figure 4: Informazioni sul secondo job map-reduce.

2.3 Implementazione Spark

In Spark si può implementare l'analisi in un unico job. Questo è possibile perché Spark considera come job una serie di "trasformazioni" sulla struttura dati distribuita dell'RDD che terminano con una "azione", che permette di computare effettivamente il job, per la proprietà di "lazyness", e memorizzare il risultato in maniera persistente su HDFS.

2.3.1 Job

Inizialmente viene creato un RDD dal file CSV in input e vengono estratte le colonne di interesse, cioè la città, l'anno e la gravità dell'incidente.

La prima trasformazione è una "map", che definisce come chiave l'anno e la città e come valore la gravità, ed è seguita da una "filter", per filtrare solo i record associati agli ultimi due anni.

MAP - key: *(city, year)*, value: *(severity)*

FILTER - *year = 2019* OR *year = 2018*

Segue una fase di aggregazione con "aggregateByKey" seguita da una "map" per calcolare la gravità media per ogni record.

AGGREGATE_BY_KEY - key: *(city, year)*, value: *(severity, count)*

MAP - key: *(city, year)*, value: *(severityAvg)*

Successivamente viene eseguito un ordinamento decrescente delle chiavi, per avere prima i record relativi all'ultimo anno, una "map" per ottenere coppie chiave-valore del tipo: *(città, severityAvg)* e un raggruppamento per chiave con "groupByKey".

SORT_BY_KEY - ↓ key: *(city, year)*, value: *(severityAvg)*

MAP - key: *(city)*, value: *(severityAvg)*

GROUP_BY_KEY - key: *(city)*, value: *([severityAvg2019, severityAvg2018])*

In questo modo è stato possibile filtrare le chiavi che avessero esattamente 2 valori, cioè la gravità media associata agli ultimi due anni.

FILTER - *value.length = 2*

Per tali record è stata calcolata la differenza del valore medio di gravità dell'ultimo anno con quello dell'anno precedente mediante l'utilizzo di una "map", computando tale differenza con un "foldLeft" sui valori delle liste associate alle chiavi.

MAP - key: *(city)*, value: *(diffSeverityAvg)*

In seguito è stato applicato un ordinamento decrescente sui valori (le differenze di gravità media) e un ordinamento crescente sulle chiavi (le città).

SORT_BY - ↑ key: *(city, year)*, ↓ value: *(severityAvg)*

Per ottenere le prime 100 città, e quindi il risultato finale dell'analisi, è stato utilizzato un "zipWithIndex" seguito da una "filter" e una "map".

ZIP_WITH_INDEX - key: *(city, diffSeverityAvg)*, value: *(index)*

FILTER - *index <100*

MAP - key: *(city)*, value: *(diffSeverityAvg)*

Infine, l'RDD è stato ripartizionato in un'unica partizione e salvato su HDFS. Di seguito un estratto dell'output del job:

```
(Waveland,2)
(Waverly Hall,2)
(West Yellowstone,2)
(Woodhull,2)
(Woodstown,2)
(Wylliesburg,2)
(Dillsboro,1.8)
(Guyton,1.8)
(Dyer,1.75)
(Kenna,1.75)
(Keystone,1.7143)
(Attapulgus,1.6667)
(Idalia,1.6667)
(Long Pond,1.6667)
(Millsap,1.6667)
(Orbisonia,1.6667)
(Eastman,1.6429)
(Broad Brook,1.6364)
(Charlotte Court House,1.6)
(Claxton,1.6)
(Maynardville,1.6)
(Pfafftown,1.6)
(Hazen,1.5714)
(Brillion,1.5)
```

Figure 5: Output del job con Spark del tipo: *(city, diffSeverityAvg)*

2.3.2 Considerazioni sulle performance

Per ottimizzare il job, i dati sono stati inizialmente partizionati su 10 partizioni. Per capire le motivazioni faccio delle considerazioni sulla tipologia del cluster e sulle best-practice per la configurazione dei parametri.

Siccome il cluster è composto da 10 nodi, ognuno avente 4 core, è necessario lasciare libero 1 core per nodo, dal momento che ci sono diversi processi che stanno eseguendo in background, come il NameNode, il Secondary NameNode, il JobTracker e il TaskTracker, insieme al sistema operativo sottostante. É inoltre opportuno lasciare anche un nodo libero che sarà utilizzato come driver.

Il client HDFS ha problemi con troppi thread concorrenti. Troppi task per ogni executor causa molto overhead. Inoltre, è stato osservato che HDFS raggiunge il miglior throughput in scrittura con circa 5 task per executor.

Ritornando al tuning degli executor, prendiamo in considerazione 3 configurazioni diverse.

- Consideriamo inizialmente un executor per core:

  num-executors = total-cores-in-cluster = num-cores-per-node * total-nodes-in-cluster = 4 * 10 = 40.

  executor-cores = 1.

  Con solo 1 executor per core non si sfrutta il vantaggio di eseguire multipli task sulla stessa JVM. Inoltre, non si lascia abbastanza memoria per i processi di Hadoop/Yarn, per l'ApplicationManager e per il sistema operativo.

- Consideriamo un executor per nodo:

  num-executors = total-nodes-in-cluster = 10.

  executor-cores = 4, questo perché 1 executor per nodo significa che tutti i core del nodo sono assegnati all'executor.

  Con tutti e 4 i core per executor, non se ne ha libero almeno uno per l'Application Manager e i processi daemon. Inoltre, il throughput HDFS non sarà dei migliori.

- Consideriamo una configurazione ideale, basata sulle raccomandazioni menzionate in precedenza, cioè assegnare (4-1 = 3) core per executor, lasciando per ogni nodo un core libero per gestire i processi in background e il sistema operativo e (10-1 = 9) nodi del cluster, lasciandone libero uno per il driver.

  num-executors = (total-cores / num-cores-per-executors) = ((4-1) * 9) / 3 = 9 - 1 = 8, così lasciamo libero 1 executor per l'ApplicationManager.

  executor-cores = 3

Si è osservato che Spark imposta come default 2 executor per eseguire il job. Con le considerazioni fatte, però, si è deciso di impostare a 3 il numero di core per ogni executor, definibile da riga di comando con: *–executor-cores 3*.

Si è deciso di allocare 5 task per executor, per un throughput in scrittura ottimale di HDFS, per cui 5 * 2 = 10 partizioni, considerando i 2 executor allocati da Spark. Questo motiva la scelta iniziale della partizione dei dati.

Analizzando il seguente DAG è possibile visualizzare la sequenza di computazioni eseguite sui dati. I nodi sono gli RDD e gli archi sono le operazioni sugli RDD. Il DAG è suddiviso in stage sulla base delle trasformazioni.

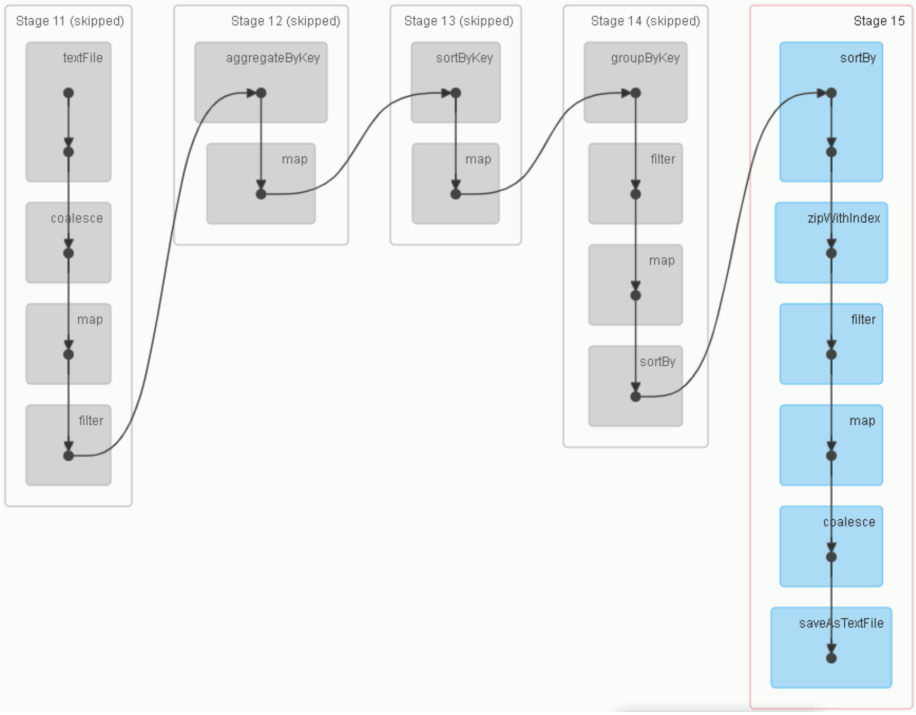

Figure 6: DAG.

Inoltre, è possibile analizzare le statistiche del job al link: `http://isi-vclust0.csr.unibo.it:18089/history/application_1583679666662_4056/jobs/`.