Big Data 2018

Primo Progetto

Analisi delle performance MapReduce, Hive, Spark

Nome Gruppo: Nòesis

Componenti: Antonio Matinata (ant.matinata@stud.uniroma3.it , a.matinata@gmail.com)

Immagine generata con wordart : rappresenta le 1000 parole più ricorrenti nel campo summary delle reviews.

Sommario

Introduzione

Scopo del progetto

Lo scopo del progetto è quello di andare a progettare e realizzare in MapReduce, Hive e Spark 3 job attraverso i quali confrontare questi sistemi. Il confronto è basato sull'analisi dei tempi di esecuzione dei job in locale e su cluster sulla base di dimensioni variabili dell'input.

Struttura del documento

Nella prima parte verranno presentati i dettagli del dataset a disposizione e dei job da realizzare, nel caso questa parte sia già nota è possibile passare direttamente alla sezione successiva.

Nel secondo capitolo sono mostrate in dettaglio le implementazioni dei 3 job con le diverse tecnologie MapReduce, Hive e Spark. È inoltre mostrato un piccolo esempio degli output generati per ciascuno dei tre job realizzati

Nella terza parte sono illustrate le metodologie adottate per effettuare i vari confronti tra le tecnologie. Inoltre vengono illustrati i test eseguiti ed i risultati ottenuti.

Specifiche del progetto

Dataset

Il dataset da utilizzare, disponibile all'indirizzo del corso, è quello di AWS Amazon Fine Food. Nel dataset sono presenti esattamente 568454 reviews di prodotti su items da parte degli utenti, raccolte negli anni 1999-2012.

Struttura del dataset

Il dataset è organizzato come segue:

NOME CAMPO	TIPO	DESCRIZIONE
Id	Intero	Id (progressivo) della review
ProductId	Text	Id univoco del prodotto recensito
UserId	Text	Id univoco dell'utente autore della recensione
ProfileName	Text	Nome del profilo utente
HelpfulnessNumerator	Intero	Numero utenti che hanno trovato utile la review
HelpfulnessDenominator	Intero	Numero di utenti che hanno valutato la review
Score	Intero	Punteggio
Timestamp	UnixTimestamp	Data in cui è stata lasciata la recensione
Summary	Text	Riassunto della recensione
Text	Text	La recensione

Analisi del dataset

Dal punto di vista quantitativo, il dataset originale contiene le seguenti informazioni:

Numero di Reviews	568454
Numero di Utenti	256059
Numero di Prodotti	74258

Pulizia del dataset

È stato realizzato un modulo in python che permette di effettuare una pulizia sul dataset e ottenere il file in modo che tutti i tempi vengano correttamente convertiti in un formato più pratico per l'elaborazione.
Nel particolare dalle reviews sono stati alterati i campi con i seguenti tipi:

- UnixTimestamp: Si è passato dal formato originale "UnixTimestamp" al formato "YYYY-MM-DD" in modo da riuscire a lavorare facilmente con le date.
- Text: Dal campo originale è stata eliminata la punteggiatura, in modo da non avere problemi in fase di lettura e splitting delle colonne per il CSV.

Lo script è visionabile al percorso 'utils/data_cleaning/CleanReviews.py'. Nel particolare sono state utilizzate le librerie 'Pandas' (per lavorare facilmente sui CSV) e 'Datetime' (per lavorare facilmente sulle date).

La versione di python utilizzata è la 3.5.6.

Data Augmentation

Poichè l'obiettivo del progetto era quello di testare le performance dei vari sistemi in funzione di grandezze diverse del dataset è stato generato un uno Script in python che permettesse di ottenere files di diverse dimensioni da utilizzare come datasets per le varie prove.

Lo script non fa altro che effettuare un certo numero di append (indicato come parametro durante l'invocazione) al file originale.

Sono stati creati dunque i seguenti dataset:

NOME DATASET	PESO (GB)	DESCRIZIONE
Reviews_clean.csv	0.3	Il file originale dopo la fase di cleaning.
Reviews_clean_3.csv	1.0	Il file originale è stato concatenato 3 volte.
Reviews_clean_17.csv	5.0	Il file originale è stato concatenato 17 volte.

Job

In questa sezione vengono descritti nel dettaglio i tre job che sono stati realizzati con esempi di un possibile output per ognuno di essi.

Job 1

Il primo job ha l'obiettivo di generare, per ciascun anno, le dieci parole che sono state più usate nelle recensioni (campo 'Summary') in ordine di frequenza, indicando, per ogni parola, la sua frequenza, ovvero il numero di occorrenze della parola nelle recensioni di quell'anno.

Esempio (Le entry rappresentano le varie recensioni, le altre colonne sono state omesse):

Id	Summary	Time
1	w1 w2 w3 w4 w5 w9 w10 w11	2000-12-01
2	w1 w2 w4 w5 w6 w7 w8	2000-02-10
4	w1 w2	2005-10-10
5	w1 w2	2005-11-30

1. 2000 w1 2
2. 2000 w2 2
3. 2000 w3 1
4. 2000 w4 2
5. 2000 w5 2
6. 2000 w6 1
7. 2000 W7 1
8. 2000 w8 1
9. 2000 w9 1
10. 2000 w10 1
11. 2005 w1 2
12. 2005 w2 2

Job 2

Il secondo job ha l'obiettivo di generare, per ciascun prodotto, lo score medio ottenuto in ciascuno degli anni compresi tra il 2003 e il 2012. Il formato in output è quello in cui è presente il ProductId seguito dallo score medio ottenuto in quell'anno, così per ognuno degli anni dell'intervallo.

Esempio:

Id	ProductId	Time	Score
1	P1	2000-12-01	5
2	P1	2000-02-10	3
4	P1	2001-10-10	2
5	P1	2001-11-30	4
6	P2	2000-10-12	3

Product	Year	Average
P1	2000	4.0
P1	2001	3.0
P2	2000	3.0

Job 3

Il terzo job consiste nel generare coppie di prodotti che hanno almeno un utente in comune. (Un utente è comune a 2 prodotti se questi ultimi sono stati recensiti da uno stesso utente).

Il job indica, per ciascuna coppia di prodotti, il numero di utenti in comune.

Il risultato è ordinato in base al 'ProductId' del primo elemento della coppia e, non devono sono presenti duplicati.

Id	ProductId	UserId
1	P1	U1
2	P1	U2
3	P2	U1
4	P2	U2
5	P2	U3
6	P2	U4
7	P3	U2
8	P4	U7

Product 1	Product 2	Common Users
P1	P2	2
P1	P3	1
P2	P3	1

Progettazione e Sviluppo

Map Reduce

I Job MapReduce sono sviluppati in Java, di seguito verranno mostrati gli approcci seguiti per la realizzazione.

Tutti i job sono realizzati in modo da prendere in input almeno 2 parametri che sono: il file di input e la cartella di output.

Job 1

L'obiettivo del primo job era quello di effettuare una classifica delle parole più utilizzate nel campo summary per ogni anno.

Il job è stato reso parametrico pertanto è possibile impostare la lunghezza della lista da ottenere. Di default è impostata a 10.

Il mapper, quindi, andrà ad emettere delle coppie chiave valore fatte in questo modo:

- Chiave: è semplicemente una stringa formata dalla concatenazione dell'anno e della parola.
- Valore: sarà semplicemente 1.

```
def map(key, value):
      line = value.split(FIELD_SEPARATOR)
      year = Date(line["Time"]).getYear()
      words = line["Summary"]
      for(w in words:
            emit(year-word, 1)
```

Ogni reducer, a questo punto, si ritroverà, ad ogni esecuzione, per ogni chiave (ovvero la coppia "anno-parola"), tutti gli 1 relativi ad una parola in un anno e quello che farà è sommare i valori emessi dal mapper per una certa parola in un certo anno in modo da ottenerne la frequenza. Il tutto viene memorizzato in una mappa condivisa che ha per chiave l'anno e per valore una coppia "parola-frequenza", (camuffata da una classe

appositamente costruita per facilitare le operazioni di ordinamento prima della scrittura dei risultati).

```
common_values = map()

def reduce(key, values):
      sum = 0
      year, word = key.split(FIELD_SEPARATOR)
      for v in values:
            sum = sum + v
      common_values[year] = (word, sum)
```

Dopo aver popolato la struttura che consente di mantenere i risultati dai vari reducer viene effettuata la chiamata al metodo cleanup che permette, per ogni anno, di ottenere le dieci parole più frequenti e scrive effettivamente tali risultati su file.

```
def cleanup(context):
      for year in common_values.keys():
            word_freq_list = common_values[year]
            word_freq_list.sort("DESC")
            count = 0
            for w, f in word_freq_list:
                if count < TOP_N:
                    contex.emit(year, w + "\t" + f)
                else:
                    break
```

Job 2

L'obiettivo del secondo job era quello di generare lo score medio dei prodotti ottenuti durante gli anni in un intervallo. È possibile lanciare lo script andando indicando solo i 2 parametri di input e di output, in questo caso l'intervallo considerato sarà di default quello degli anni tra il 2003 e il 2012. Opzionalmente è possibile specificare anche altri due parametri al job e indicare gli anni per i quali si vuole effettuare il calcolo del punteggio medio.

La fase di map del job prevede di andare ad emettere una coppia chiave valore così composta:

- Chiave: concatenazione dell'Id del prodotto e dell'anno in cui è stata fatta la review.

- Valore: lo score assegnato al prodotto in quella review.

```
def map(key, value):
      line = value.split(FIELD_SEPARATOR)
      year = Date(line["Time"]).getYear()
      score = line["Score"]
      product = line["ProductId"]
      emit(product +"-"+ year, score)
```

Ogni reducer quindi, andrà a processare, per ogni coppia “prodotto-anno” gli score ottenuti nelle recensioni di quell’anno.
Questi dati verranno inseriti in una struttura che ha come chiave il prodotto e come valore la coppia anno, punteggio medio.

```
common_values = dict()

def reduce(key, values):
      sum = 0
      product, year = key.split("-")
      for v in values:
            sum = sum + v
      common_values[product].add((year, sum/len(values)))
```

La fase successiva è quella di cleanup in cui vengono effettivamente prodotti i risultati in output. In questa fase, quello che viene fatto, è ordinare la lista degli score medi per ogni anno in base all’anno.

```
def cleanup(context):
      for product in common_values.keys():
            year_scores = common_values["product"]
            year_scores.sort("ASC")
            for year, score in year_scores:
               contex.emit(product, year + "\t" + score)
```

Job 3

Nel terzo job l'obiettivo era quello di calcolare una matrice che consentisse di identificare, per ogni coppia di prodotti, il numero di utenti comune che lo hanno recensito.

La fase di map è stata organizzando andando a produrre, per ogni linea processata, un coppia chiave valore fatta in questo modo:

- Chiave: UserId dell'utente che ha lasciato la recensione
- Valore: ProductId del prodotto che è stato recensito.

```
def map(key, value):
      line = value.split(FIELD_SEPARATOR)
      userId = line["UserId"]
      productId = line["ProductId"]
      emit(userId, productId)
```

Nella fase di reduce, dunque, arriveranno ad ogni reducer tutti i prodotti di un determinato utente. I reducer a questo punto andranno a popolare una struttura in cui saranno inseriti come chiave le coppie di prodotti e come valore il numero di volte che la coppia è stata incontrata.

```
common_val = dict()
# key = userId
def reduce(key, values):
      sum = 0
      already_added = set()
      for product_1 in values:
            for product_2 in values:
                if product_1 != product_2:
                    prod_couple = product_1+"-"+product_2
                    if product_1 > product_2:
                        prod_couple = product_2+"-"+product_1
                    if prod_couple not in already_added:
                        common_val[prod_couple] = 1
                        already_added.add(prod_couple)
                    else:
                        Common_val[prod_couple] = common_val[prod_couple]+1
```

Dopo la fase di reduce viene effettuato il cleanup in modo da produrre effettivamente l'output nel formato desiderato. Avendo avuto cura, in fase di reduce, di organizzare la struttura con una mappa ed inserire le coppie di prodotti in maniera già ordinata (viene effettuato un confronto tra le stringhe dei productId in modo da inserire chiavi in cui il primo productId sia minore rispetto al secondo), non è necessario andare ad effettuare un successivo ordinamento ma semplicemente scrivere l'output.

```
def cleanup(context):
      for products in common_val.keys():
            product1, product2 = products.split("-")
            contex.emit(product1 +"\t"+ product2, common_val[products])
```

Hive

Per quanto riguarda hive, il primo passo è stato quello di creare il database e la relativa tabella sulla quale effettuare le query.

```
CREATE DATABASE amazon_reviews WITH DBPROPERTIES ("creator" = "Antonio Matinata");
CREATE TABLE IF NOT EXISTS amazon_reviews.reviews (
      id INT COMMENT "Review unique id",
      productId STRING COMMENT "Unique identifier for the product",
      userId STRING COMMENT "Unique identifier for the user",
      profileName STRING COMMENT "Account name of the user",
      helpfulnessNumerator INT COMMENT "# of users who found the review helpful",
      helpfulnessDenominator INT COMMENT"# of users who graded the review",
      score INT COMMENT "Rating between 1 and 5",
      time DATE COMMENT "Date of the review expressed in the format YYYY-MM-DD",
      summary STRING COMMENT "Summary of the review",
      text STRING COMMENT "Text of the review"
      )
      ROW FORMAT DELIMITED
      FIELDS TERMINATED BY ','
      LINES TERMINATED BY '\n'
      STORED AS TEXTFILE;
```

Successivamente è stata popolata la tabella con i dati del file utilizzando il seguente comando.

```
LOAD DATA INPATH '/user/root/reviews.csv' OVERWRITE INTO TABLE amazon_reviews.reviews;
```

Job 1

Per quanto riguarda il primo job la soluzione utilizzata è stata costruita in modo da :

- Ottenere una sottotabella che, per ogni anno contasse le occorrenze di ogni parola.
- Creasse una sorta di risultato ordinato in base alle occorrenze della parola nell'anno.
- Estraesse solamente le 10 più frequenti.

La query risultato è dunque la seguente:

```
INSERT OVERWRITE DIRECTORY '/user/root/output_hive/1'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT ranked.year, ranked.word, ranked.frequency
FROM (
   SELECT temp.year, temp.word, temp.frequency,
          rank() over (PARTITION BY temp.year ORDER BY temp.frequency DESC) as rank
   FROM (
         SELECT YEAR(Time) as year, t.t1 as word, count(*) as frequency
         FROM reviews LATERAL VIEW EXPLODE(SPLIT(Summary," ")) t as t1
          GROUP BY t.t1, YEAR(Time)
          ORDER BY year ASC, frequency DESC, t.t1) temp
       ) ranked
WHERE ranked.rank < 10;
```

È stata utilizzata la funzione rank() per poter ordinare le parole di uno specifico anno. Quello che permette di fare questa funzione è di andare a costruire una classifica sulla base di un certo valore (nel nostro caso la frequenza del termine) per ogni partizione dei dati (nel nostro caso l'anno di riferimento).

In questo modo è stato possibile generare una vera e propria classifica delle parole più ricorrenti per ogni anno di interesse.

Job 2

Per quanto riguarda il job2, invece, la query risultante è la seguente:

```
INSERT OVERWRITE DIRECTORY '/user/root/output_hive/2'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT ProductId as product, YEAR(Time) as year, AVG(Score) as mean_score
FROM amazon_reviews.reviews
WHERE YEAR(Time) >= 2003 AND YEAR(Time) <= 2012
GROUP BY ProductId, YEAR(Time)
ORDER BY ProductId ASC, year ASC;
```

L'interrogazione è un'interrogazione classica lato SQL che prevede la selezione del prodotto e dell'anno sulla quale si va ad effettuare il raggruppamento per ottenere il punteggio medio degli score negli anni di interesse.

Job 3

La query risultante per il Job 3 è la seguente:

```
INSERT OVERWRITE DIRECTORY '/user/root/output_hive/3'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT r1.ProductId as p1, r2.ProductId as p2, SUM(1) as common_reviewers
FROM amazon_reviews.reviews r1 JOIN amazon_reviews.reviews r2
WHERE r1.UserId = r2.UserId AND r1.ProductId < r2.ProductId
GROUP BY r1.ProductId, r2.ProductId
ORDER BY r1.ProductId ASC, r2.ProductId ASC;
```

In sostanza, quello che si fa, è andare ad ottenere tutti i prodotti recensiti da un certo utente andando a fare un self-join della tabella in base all 'UserId della recensione. La condizione in AND al join è quella che va a effettivamente ad eliminare i duplicati in quanto vengono mantenute solamente le coppie in cui il primo productId è lessicograficamente minore al secondo.

Fatto il Join è possibile andare ad effettuare la somma delle volte che la coppia di product id compare raggruppando per quest'ultima.

Spark

Per quanto riguarda l'implementazione in spark è stato deciso di sviluppare il tutto in Python.

Sono state realizzate diverse versioni dei vari job che consentissero di effettuare analisi sotto diversi punti di vista. Il primo punto di visto riguarda semplicemente l'ordine di alcune operazioni pesanti quali 'reduceByKey()' o 'groupByKey()'. Il secondo, invece, riguarda l' implementazione in sè: è stata realizzata, per il job 2 e il job 3, una versione che sfrutta gli RDD e una che sfrutta i DataFrame ed effettua le interrogazioni eseguendo delle query.

Per snellire la trattazione verrà presentata l'implementazione della soluzione realizzata con gli RDD in quanto, dal punto di vista concettuale, le query sviluppate sono le stesse di quelle illustrate nella precedente sezione, a differenza di qualche piccolo dettaglio tecnologico.

Job 1

L'implementazione del primo job segue, grosso modo, quella già illustrata nella sezione dei job Map Reduce.

Per ogni linea, dunque vengono prodotti degli RDD che hanno come chiave la concatenazione dell'anno e della parola estratta dal campo "Summary" della review.

Successivamente viene effettuato un raggruppamento per chiave in cui due RDD con la stessa chiave vengono ridotti ad un unico RDD che ha la stessa chiave e come valore la somma dei valori degli RDD. In sostanza viene calcolato, in questo punto, la frequenza di una certa parola in un dato anno.

Viene prodotto, successivamente, un altro RDD che ha come chiave l'anno e come valore delle coppie parola-frequenza. Il tutto viene poi ordinato per chiave e vengono filtrati solamente gli N record più frequenti.

Anche in questo caso, come nel caso del job MapReduce, è possibile specificare la lunghezza della lista, il valore di default è 10.

```
year_word = lines.flatMap(get_year_word_frequency)
year_word_sum = year_word.reduceByKey(lambda v1, v2: v1+v2)
year_word_sum = year_word_sum.map(lambda kv: (kv[0].split("-")[0],
(kv[0].split("-")[1], kv[1])))

year_words = year_word_sum.groupByKey()
output = year_words.flatMap(get_top_n).sortByKey()
```

Job 2

Per quanto riguarda il job 2, invece, la soluzione identificata è la seguente:

```
lines_filtered = lines.filter(filter_review_by_date)
year_product_vote = lines_filtered.map(get_key_value_tuple)
year_product_votes = year_product_vote.groupByKey()
year_product_mean = year_product_votes.mapValues(get_key_value_avg).sortByKey()
output = year_product_mean.map(get_output_from_key_value)
```

La prima operazione è quella di andare a filtrare le review che non fanno parte del range specificato: è possibile indicare un range da riga di comando, il default è quello degli anni 2003-2012.

La seconda operazione permette di ottenere degli RDD la cui chiave è la concatenazione del productId e dell'anno, mentre il valore è lo score ottenuto nella singola review. Viene poi effettuato un raggruppamento per chiave che permette di lavorare sulle review di un particolare prodotto in un anno e ottenerne gli score medi. Il risultato viene poi ordinato in base alla chiave e viene portato in un formato consono alla stampa.
La soluzione alternativa a questa versione è quella che utilizza al posto del raggruppamento in base alla chiave, l'operazione di reduce. In questo caso però è necessario portarsi dietro, una lista di valori, il cui primo è il numero di valori sommati e il secondo è invece la media, in modo da poter calcolare facilmente la media ad ogni raggruppamento di due diverse chiavi.

Job 3

L'implementazione in pseudocodice per il terzo Job è la seguente:

```
user_products = loaded_file.map(get_user_products_tuple).groupByKey()
product_couples = user_products.flatMap(get_product_couples)
product_couples_totals = product_couples.groupByKey()
output = product_couples_totals.map(get_output_string).sortBy(sortby_prod1)
```

La prima operazione è quella che consente di ottenere, dalle varie review, la lista dei prodotti che sono stati recensiti da un utente. Il tutto è ottenuto andando a produrre delle coppie chiave valore, per ogni linea, in cui la chiave è rappresentata dall' userId dell'utente che ha effettuato la recensione mentre il valore è il productId del prodotto cui la recensione si riferisce. Effettuando poi un raggruppamento per chiave vengono ottenuti i prodotti recensiti da ogni utente.

Successivamente vengono calcolate le coppie di prodotti recensite da un utente in comune. Vengono utilizzate le stesse accortezze illustrate nel job MapReduce: per ogni coppia univoca nella lista dei prodotti recensiti da un cliente viene emesso un RDD che ha per chiave la concatenazione dei productId della coppia di prodotti e per valore 1. Anche in questo caso viene utilizzato un confronto tra stringhe per fare in modo che la coppia sia sempre la stessa anche nelle liste di altri utenti.

Il tutto viene poi passato ad una funzione di map che ha il compito di formattare gli RDD in un formato consono alla stampa e infine viene effettuato l' ordinamento.

Esempi di Output

In questa soluzione verranno presentati degli esempi di output per i tre job.

Job 1

```
1999  a 3
1999  day 3
1999  fairy 3
1999  modern 3
1999  tale 3
1999  is 2
1999  book 1
1999  child 1
1999  educational 1
1999  entertainingl 1
2000  a 11
2000  master 6
2000  version 5
2000  great 4
2000  afterlife 3
2000  bettlejuicebettlejuicebettlejuice 3
2000  burton 3
2000  by 3
2000  clamshell 3
2000  comedic 3
2001  beetlejuice 7
...
```

Job 2

```
0006641040  2003  5.0
0006641040  2004  4.33
0006641040  2005  3.25
0006641040  2007  4.5
0006641040  2008  4.0
0006641040  2009  5.0
0006641040  2010  5.0
0006641040  2011  4.17
0006641040  2012  4.0
141278509X  2012  5.0
2734888454  2007  3.5
2841233731  2012  5.0
7310172001  2005  3.5
7310172001  2006  5.0
7310172001  2007  4.9
...
```

Job 3

```
0006641040  B0005XN9HI  1
0006641040  B00061EPKE  1
0006641040  B000EM00YU  1
0006641040  B000FDQV46  1
0006641040  B000FV8LPU  1
0006641040  B000MGOZEO  1
0006641040  B000MLHU3M  1
0006641040  B000UVW59S  1
0006641040  B000UVZRES  1
0006641040  B000UW1Q8I  1
0006641040  B000UXH9X8  1
0006641040  B000UXW95G  1
0006641040  B0013P3KC6  1
0006641040  B0014UFXGG  1
0006641040  B0014WYY1E  1
0006641040  B0015UW23M  1
...
```

Test Eseguiti

In questo capitolo verranno illustrate le metriche utilizzate per confrontare i vari sistemi. Verranno illustrate le specifiche dei sistemi utilizzati per le esecuzioni in locale e su cluster e i risultati dei test effettuati.

Le varie prove verranno analizzate con una suddivisione che permetta di valutare le prestazioni di un sistema sui vari ambienti ma anche di valutare come, su uno stesso ambiente, si comportano i vari sistemi.

Metriche di confronto

Le metriche utilizzate per effettuare il confronto tra i sistemi sono quelle relative a :

- Tempo di esecuzione
- Tempo medio di esecuzione
- Ambiente di esecuzione (Locale/Cluster)
- Dimensione del dataset (secondo quanto illustrato nel primo capitolo)

Specifiche degli Ambienti

Gli ambienti sui quali sono state effettuate le varie prove sono 2: uno locale e uno in cloud.

Ambiente locale

Per l'architettura locale si è deciso di utilizzare una macchina virtuale cloudera-quickstart. La macchina in cui è stato importato l'ambiente disponeva di:

- 4GB di RAM
- Processore con 4 Core

Ambiente Cluster

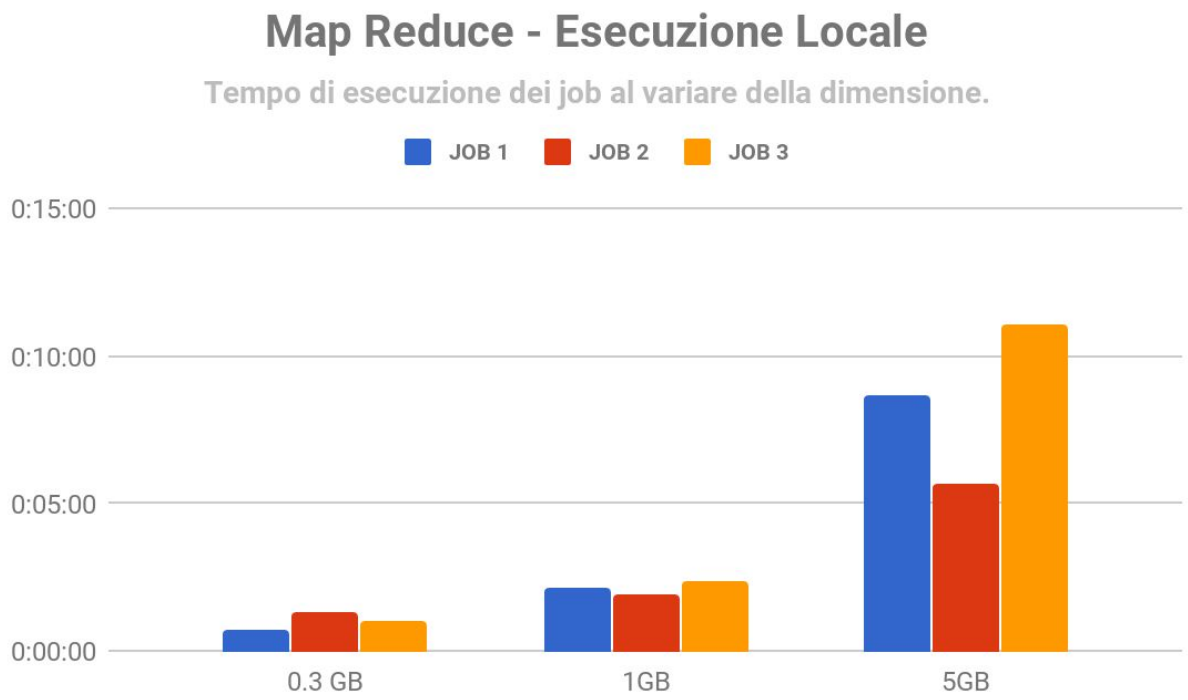
Per l'ambiente Cluster si è deciso di utilizzare l' EMR di Amazon Web Services configurato con 3 nodi totali: un master node e due da data node.

Tutte le macchine sono istanze m4.large, dunque dotate di: 2 processori (4 virtuali), 8 GB di memoria RAM e 32 GB di disco.

MapReduce Jobs

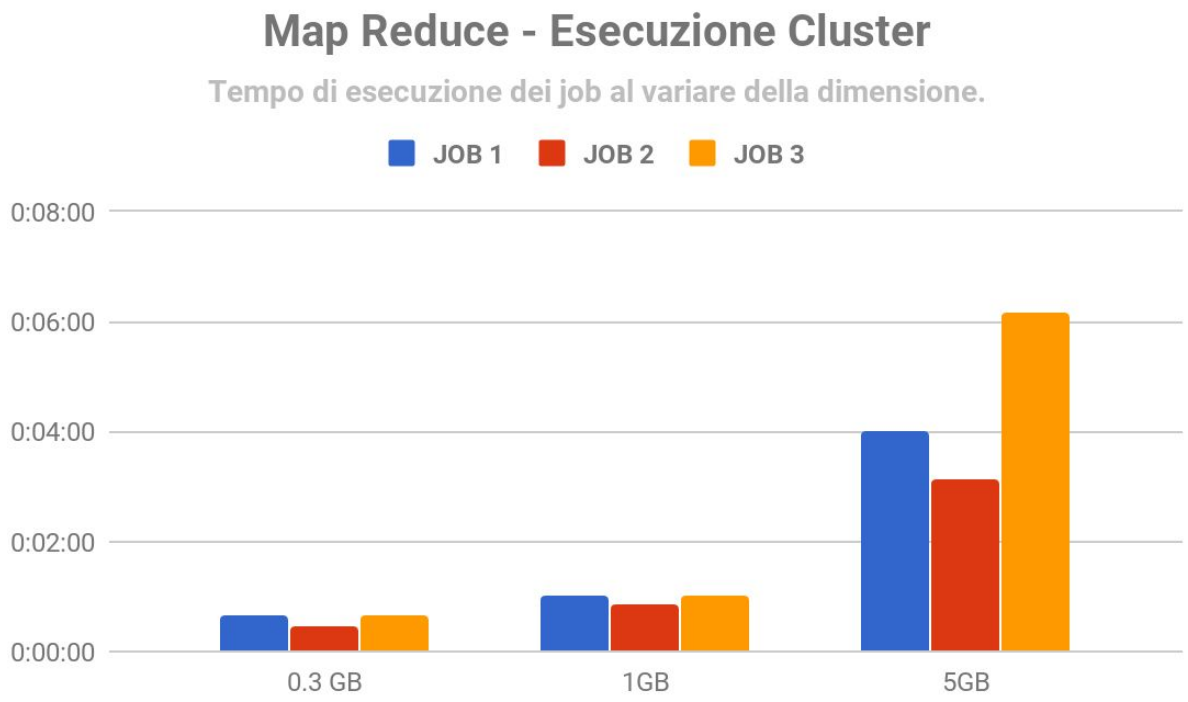
Verranno mostrati ora i confronti relativi ai job MapReduce.

Nel grafico seguente è possibile apprezzare i tempi di esecuzione (formato hh:mm:ss) dei vari Job MapReduce a differenti grandezze dell'input.

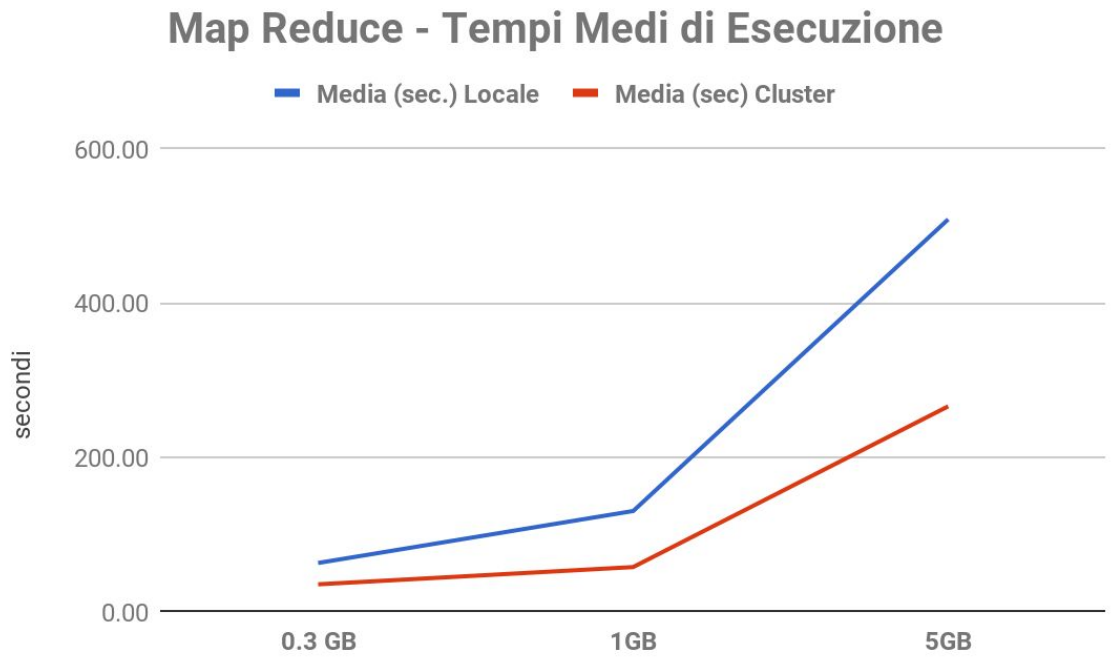


Per quanto riguarda l'esecuzione in locale si nota come su files di dimensioni ridotte i vari tempi tra i job non si discostano di molto. Per i file di dimensioni maggiori, invece, il numero di operazioni impatta pesantemente sui tempi di esecuzione. Per quanto riguarda il primo job, il riordinamento della lista e il ciclo necessario a far si che vengano estratti solamente le 10 parole più frequenti appesantiscono la fase di reduce. La stessa cosa vale per il 3 job in cui la parte pesante è quella della creazione delle varie coppie di prodotti.

Le stesse osservazioni fatte per l'ambiente locale continuano ad essere valide anche per quello cluster anche se i tempi di esecuzione si riducono di molto.

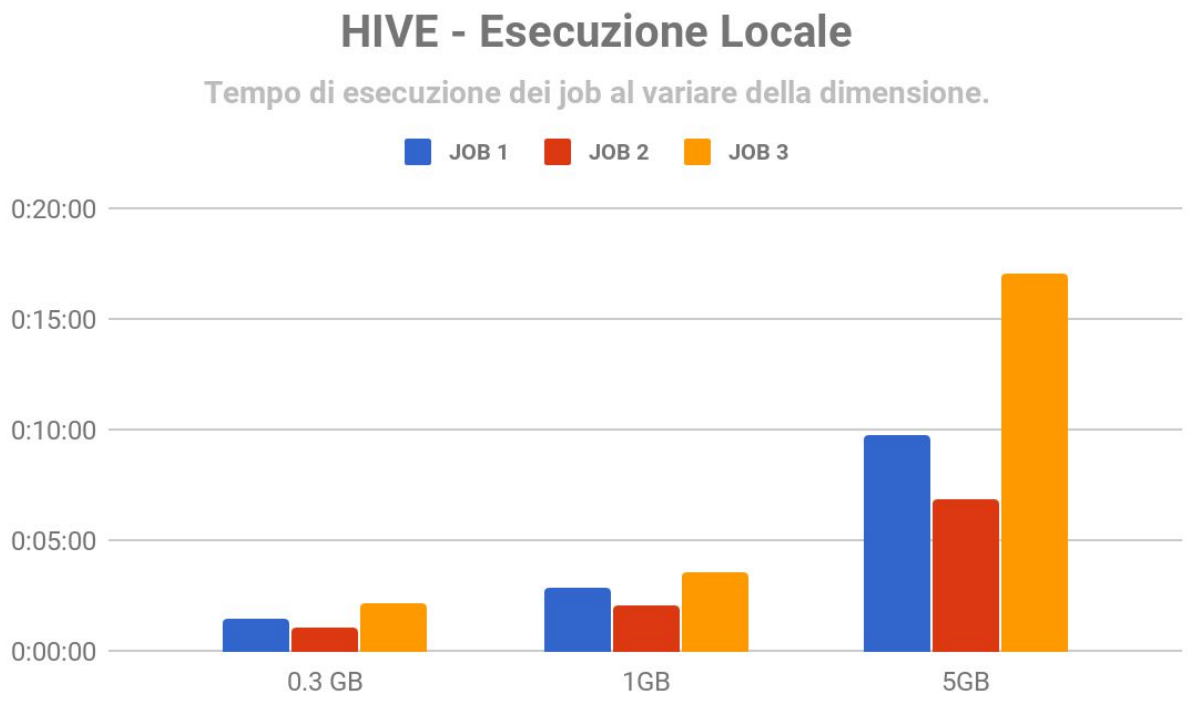


Confrontando i tempi medi di esecuzione (in secondi) dei job raggruppati per grandezza dell'input, si nota come al crescere dell'input cresca il tempo di esecuzione ma è possibile apprezzare anche come il sistema riesca ad essere molto più performante su cluster.



HIVE Jobs

Analizziamo, allo stesso modo di Map Reduce, quello che accade per Hive.

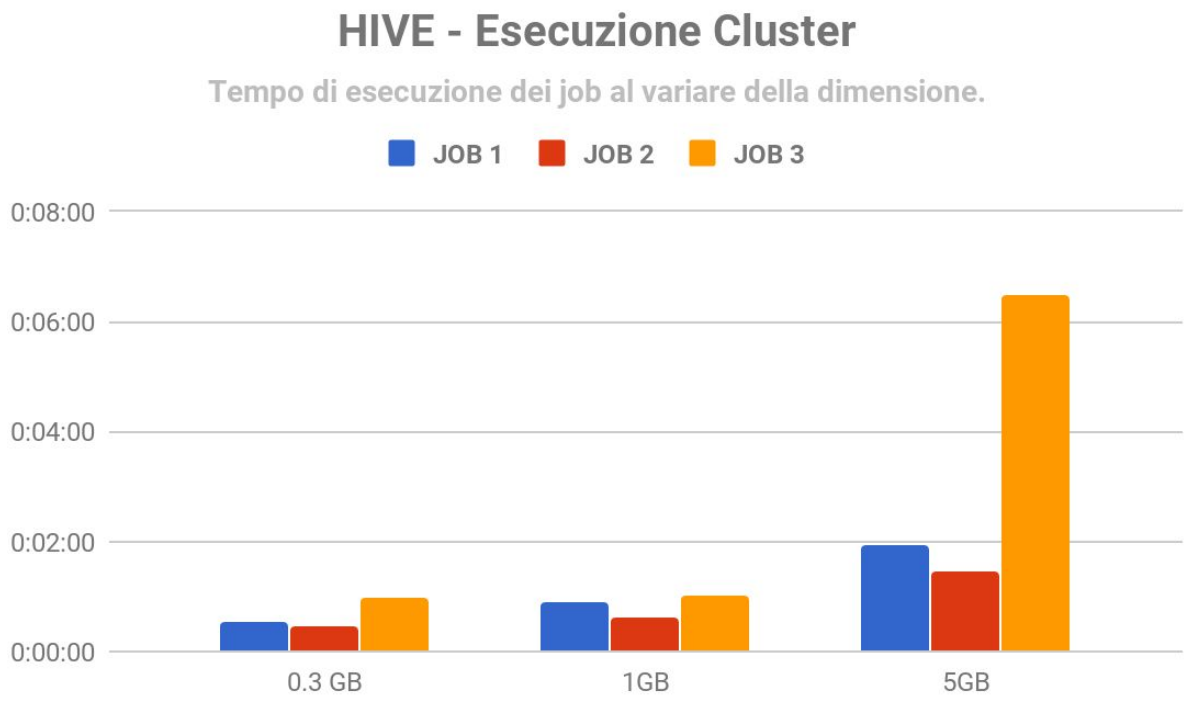


Anche in questo caso sui files di grosse dimensioni l'applicazione di alcune operazioni pesanti impatta di molto l'esecuzione del job.

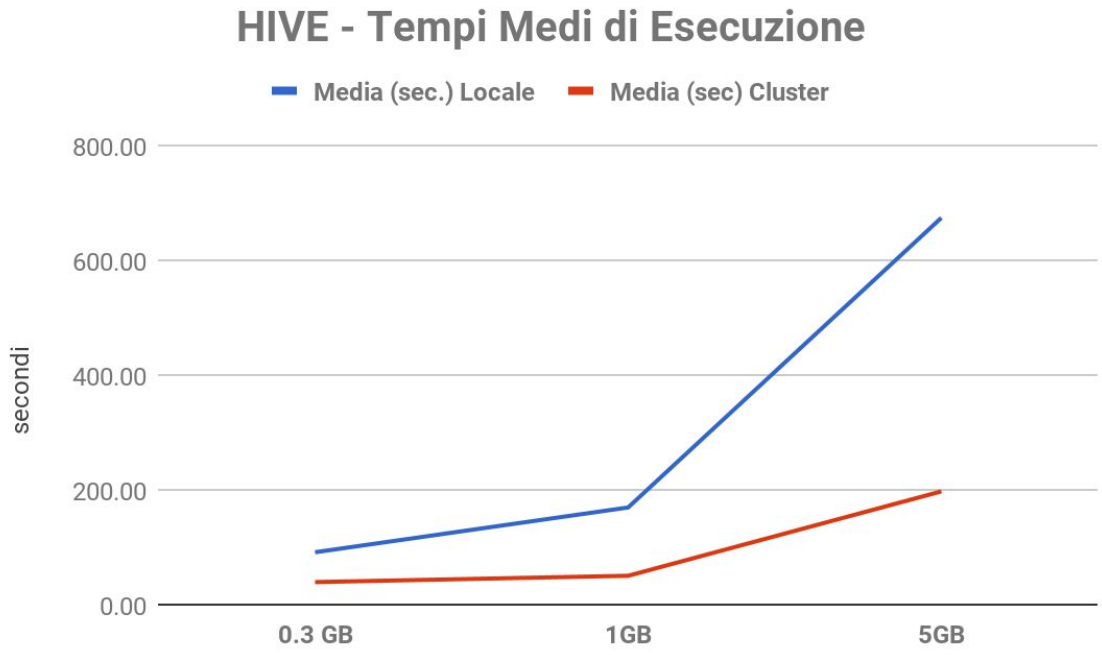
il Job 1 prevede che venga estratta una row per ogni parola del campo "summary", venga effettuato un raggruppamento per anno e in seguito la funzione rank() si occupa di effettuare la classifica;

Per il job 3, invece, essendo effettuato un join della relazione, al crescere delle dimensioni del file il numero di righe da processare cresce di gran lunga andando ad impattare di molto sul tempo totale di esecuzione. Queste osservazioni di carattere generale valgono anche quando ci spostiamo in ambiente cluster anche se, i tempi di esecuzione in se, sono di gran lunga minori.

Guardando ai tempi medi di esecuzione dei vari job in locale e su cluster si scopre come, anche in questo caso, l'esecuzione in locale sia di gran lunga più lenta.



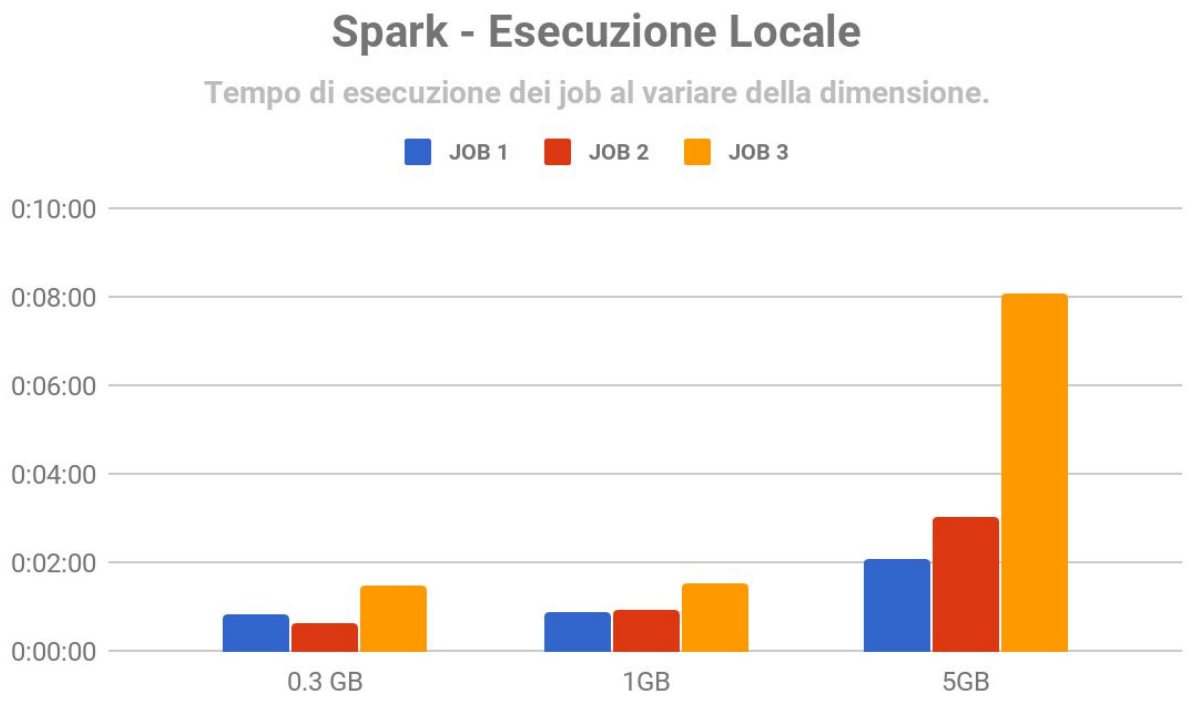
Guardando ai tempi medi di esecuzione scopriamo che l'esecuzione su cluster è più veloce. Inoltre, altra osservazione importante, è il fatto che la pendenza della retta è molto più bassa: questo significa che al crescere dei files il tempo non cresce linearmente come invece accade in locale.



Spark Jobs

L'analisi effettuata su Spark entra un pochino più nel dettaglio in particolare si è cercato di capire come alcune scelte implementative impattassero sulle prestazioni.

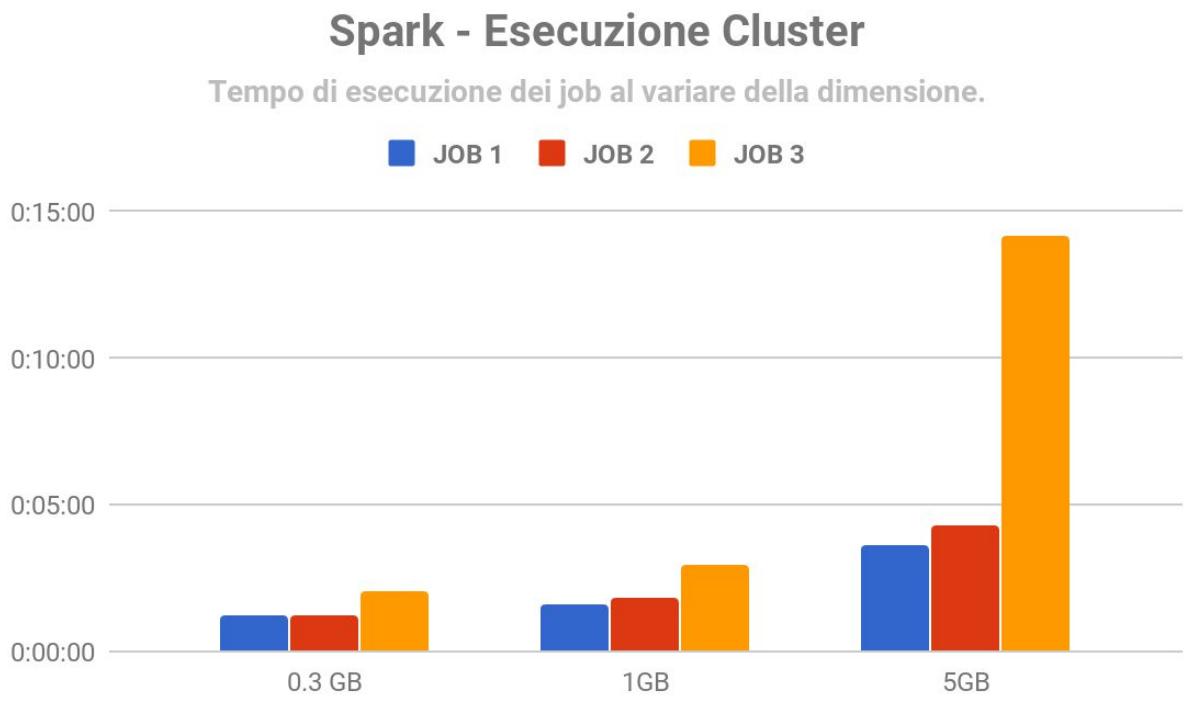
In locale il tutto è stato lanciato dando alla macchina tutti i core che aveva a disposizione. La situazione che si presenta nel grafico vede dunque, ancora una volta, come più pesante il Job 3 che, soprattutto su file grandi ha tempi di esecuzione molto più lunghi rispetto agli altri due (Circa il 60% in più). La cosa interessante da notare consiste nel fatto che almeno per il Job 1 e per il Job 3 spark si comporti molto meglio sul file da 1GB che su quello da 0.3.



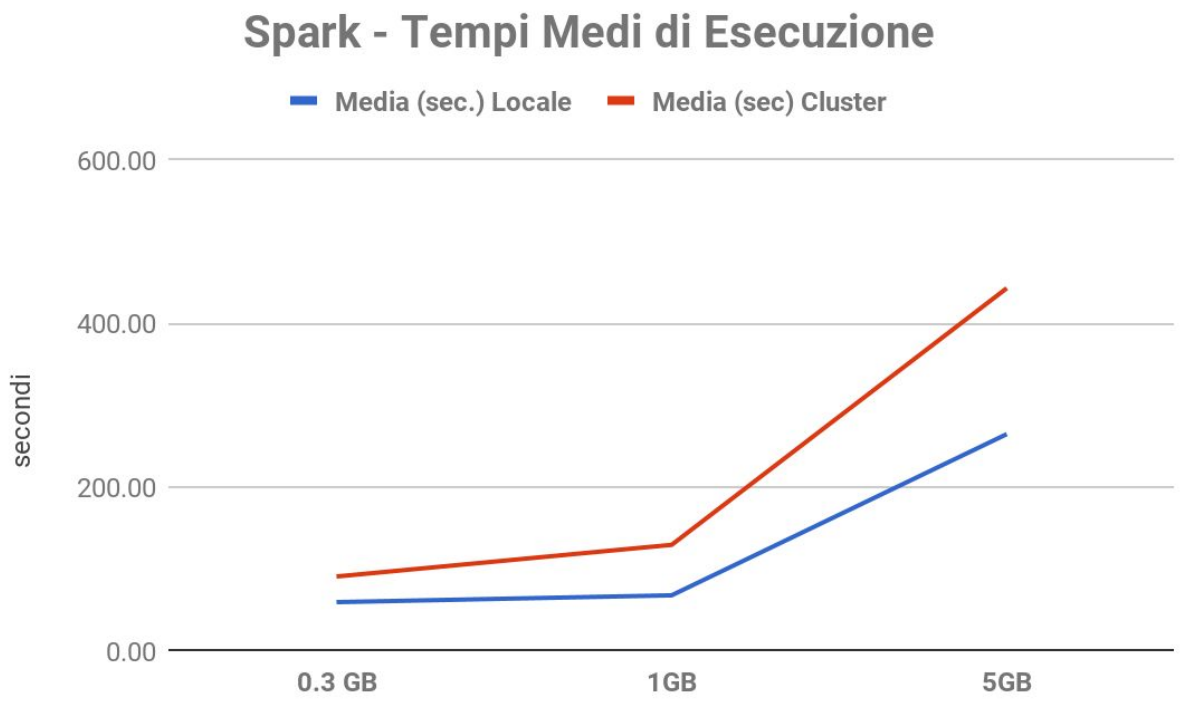
Per quanto riguarda l'esecuzione su cluster, i job sono stati lanciati con il seguente comando:

```
spark-submit --deploy-mode cluster --master yarn --num-executors 2 --executor-cores 4
```

In questo modo è stato possibile indicare "Yarn" come master e come modalità di esecuzione quella su cluster nonchè specificare anche gli altri parametri relativi al numero numero di core e al numero di executors.



Guardando al tempo di esecuzione del 3 job si nota che, essendo previste ben 2 operazioni di Group By Key, l' overhead di comunicazione necessario ai nodi per scambiarsi i dati si traduce in un calo delle performance notevole rispetto al sistema locale.

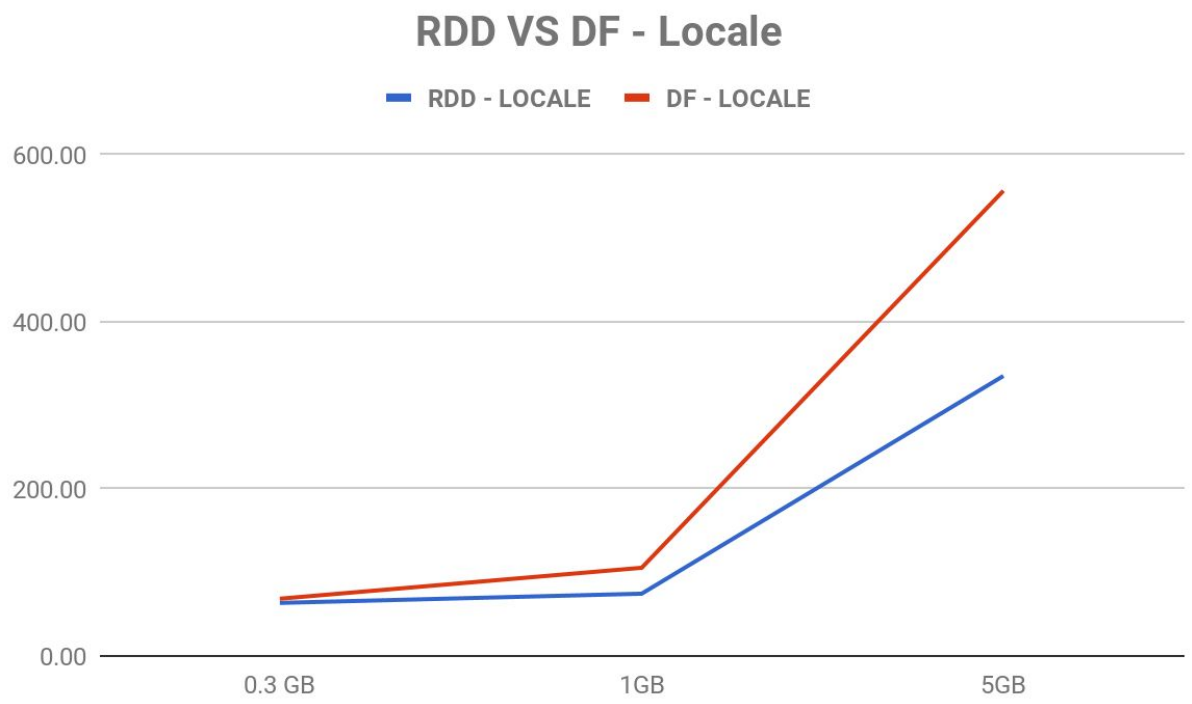


Andando a confrontare i tempi medi di esecuzione si nota come per files piccoli spark si comporti grossomodo allo stesso modo su cluster, contrariamente a quanto veniva fatto in locale.

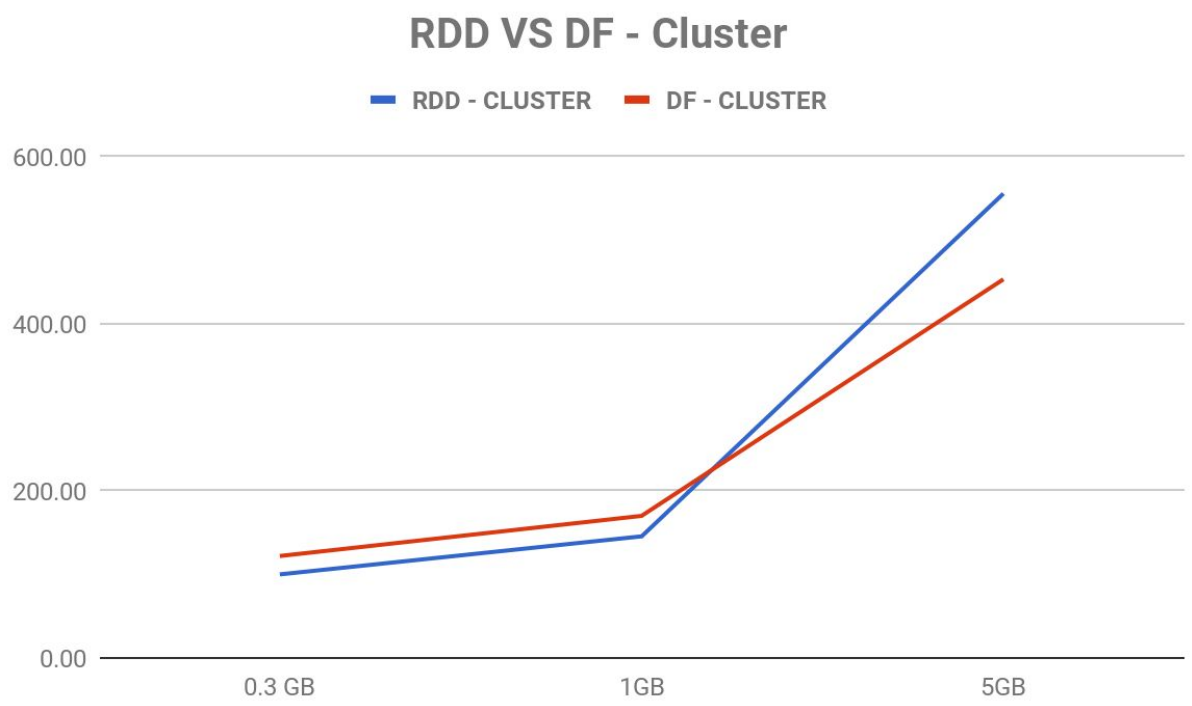
Confronto RDD - SPARK SQL

È stato effettuato, per Spark, anche un confronto che permettesse di stimare le performance del sistema, in locale e su cluster, nel momento in cui piuttosto che utilizzare i classici RDD, i dati vengano caricati in Data Frame e su questi ultimi venga poi eseguita una query SQL.[1]

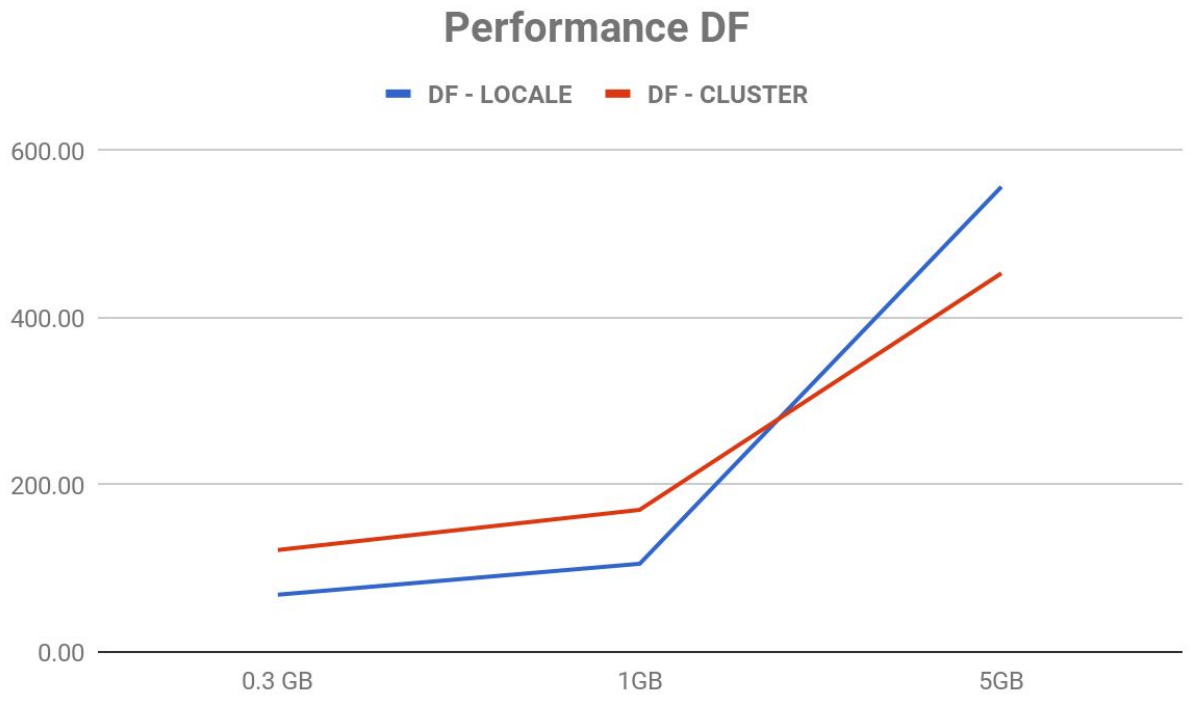
Per quanto riguarda l'ambiente locale, nel grafico successivo, è possibile notare come per i files più piccoli, quello da 0.3gb e quello da 1.0gb effettivamente le classiche operazioni RDD si comportino meglio di poco. La stessa cosa non avviene invece quando ci si confronta con il file da 5GB in cui lo scarto di tempo è davvero notevole (circa 200 sec).



[1] I dettagli implementativi sono apprezzabili nelle classi "spark/bd_ex3ai/Esercizio32.py e spark/bd_ex2ai/Esercizio23.py



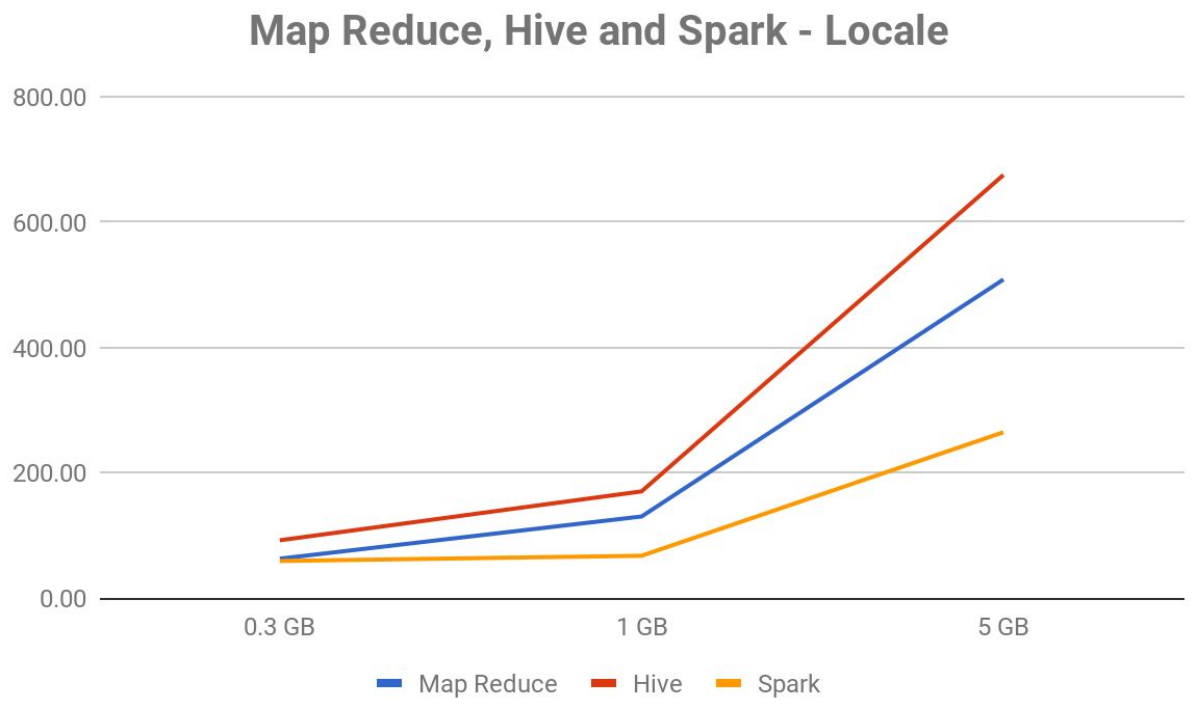
Su ambiente cluster l'utilizzo dei data frame, aumenta di molto le performance su files grandi mentre, sui files piccoli si registrano tempi migliori, anche se di poco, per gli RDD. La stessa esecuzione su cluster è molto più veloce su files grandi quando si i Data Frames.



Confronto Tecnologico

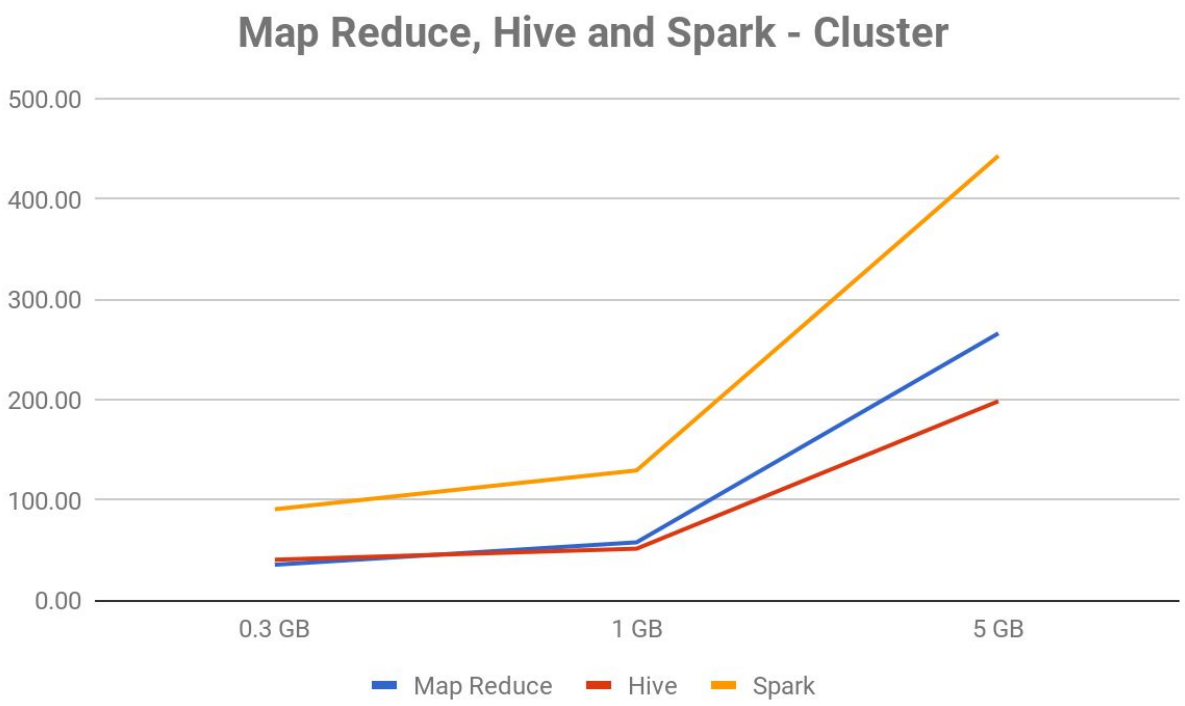
In questa sezione verrà effettuato un confronto fra le varie tecnologie in ambiente cluster e locale andando a comprendere come si comportano i vari sistemi in relazione ai tempi medi registrati nelle varie esecuzioni dei job.

Per quanto riguarda l'ambiente locale abbiamo che la tecnologia più lenta è Hive mentre quella che da sempre risultati migliori è Spark.



La cosa interessante da notare è la pendenza delle rette: per quanto riguarda spark, l'esecuzione su i files piccoli (0.3 GB - 1 GB) non aumenta di molto i tempi di esecuzione, mentre per gli altri sistemi c'è comunque una crescita sostanziale. Allo stesso modo, spostandoci verso files di dimensioni maggiori, abbiamo che MapReduce ed Hive aumentano quasi in maniera lineare i tempi di esecuzione.

Per quanto riguarda l'ambiente cluster, la situazione che si delinea è mostrata nel grafico sottostante.



Quello che accade è che MapReduce ed Hive per files di piccole dimensioni si comportano sostanzialmente allo stesso modo. Mano a mano che i files aumentano di dimensione però Hive riesce a garantire performance migliori rispetto a MapReduce. Spark, invece, risulta essere comunque il più lento, tanto sui files piccoli che su quelli grandi rispetto agli altri due sistemi.