Big Data

Relazione Primo Progetto

Job 1:

MapReduce:

Per il primo job ho utilizzato un mapper ed un reducer, ed ho utilizzato Python come linguaggio.
Il job viene quindi eseguito tramite hadoop-streaming nel seguente modo:

- il mapper so occupa di ricevere le linee in input, e di restituire in output coppia (Year, Summary). Ad esempio verrà restituito “1999 pippo”.
- il reducer riceve l’output del mapper e crea un dizionario che ha come chiave l’anno e come valore il contenuto di ogni campo summary dello stesso anno. Creando ad esempio : “{ 1999: pippo pluto database ecc}”. Successivamente viene scansionato il dizionario in base alle chiavi, e per ogni valore vengono estratte le 10 parole più frequenti con il relativo numero di occorrenze.

Nell’immagine seguente viene mostrato lo pseudocodice di entrambi

```
"""mapper.py"""


columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator",
"HelpfulnessDenominator","Score","Time","Summary","Text"]


# input comes from STDIN (standard input)
for words in line_in_input:


    summary = delete_punctuation(words['Summary']).lower()
    year = getYear(words['Time']))

    print('%s\t%s' % (year, summary))
```

```
"""reducer.py"""


# maps words to their counts
text2year = {}


# input comes from STDIN
for line in input_line:

    # parse the input we got from mapper.py
    try:
        year,text = line.split('\t')
    except:
        continue

    try:
        text2year[year] = text2year[year] + " " + text
    except:
        text2year[year] = text


# write the tuples to stdout
for year in text2year.keys():
    words_freq = mostFrequent(text2year[year])

    print ('%s\t%s' % (year, words_freq) )

```

Hive:

In figura il codice del primo job Hive

```
CREATE TABLE Reviews ( Id STRING, ProductId STRING, UserId STRING, ProfileName STRING, HelpfulnessNumerator STRING, HelpfulnessDenominator STRING,
        Score STRING, Time STRING, Summary STRING, Text STRING )

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

LOAD DATA LOCAL INPATH '/home/davide/Documenti/BIG_DATA/codice/Reviews_nofirstline.csv' INTO TABLE Reviews;

add jar /home/davide/Documenti/BIG_DATA/codice/unix_date.jar;

CREATE TEMPORARY FUNCTION get_year AS 'unix_date.Unix2Date';

SELECT Id, get_year(Time) FROM Reviews LIMIT 10;

CREATE TABLE mapper_out AS
  ( SELECT get_year(r.Time) as year, regexp_replace(lower(r.Summary), "\!|\"|\,|\;|\-|\~|\:", "") as summary
       FROM Reviews as r);

CREATE TABLE year_word AS
SELECT mp.year, exp.splitted as word
FROM mapper_out mp
LATERAL VIEW explode(split(summary, ' ')) exp AS splitted;

CREATE TABLE year_word_count AS
SELECT year, word, count(1) as word_count
FROM year_word yw
WHERE year >= 1999
GROUP BY year, word;

CREATE TABLE result_1 AS
SELECT year, word, word_count
FROM
(SELECT year, word, word_count,
       rank() over (partition by year order by word_count desc) as rank
FROM year_word_count) t
WHERE rank <11;

SELECT * FROM result_1;
```

Nel codice viene creata la tabella relativa al file in input, oltre che una funzione temporanea scritta in Java che permette di effettuare il parsing da timestamp ad anno.

La tabella mapper_out viene creata prendendo le colonne relative all'anno (utilizzando la funzione temporanea) e il campo summary.
Sulla base della tabella mapper_out viene creata la tabella year_word, eseguendo lo split (in base allo spazio) sul valore del campo summary di mapper_out e quindi associando ad ogni anno una singola parola.

Year_word_count contiene le colonne : Year, Word, Word_count. Tramite il raggruppamento, viene associato ad ogni parola comparsa in un anno, il numero di occorrenze della parola stessa nell'anno preso in esame.

Nell'ultimo passaggio viene creata la tabella con i risultati, ordinando in ordine decrescente le parole (per ogni anno) e limitandosi a prendere le prime 10.

Spark:

Per eseguire il Job1 ho utilizzato pyspark.
Data la rappresentazione dei dati in input, ho scelto di utilizzare i dataframe.
Di seguito lo pseudocodice.

```
columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator","HelpfulnessDenominator",
           "Score","Time","Summary","Text"]

dataframe2 = spark.read.format('csv').option('header','true').option('mode','DROPMALFORMED')\
.load("hdfs:///input/Reviews_4.csv")

year = dataframe2.withColumn('Time', from_unixtime('Time')).cache()
year = year.select(year.Time.substr(1,4).alias('Time'), 'Summary')

year = year.withColumn('Summary', lower(dataframe2.Summary)).cache()
year = year.withColumn('Summary', regexp_replace(year.Summary,'\p{Punct}', ''))

year = year.select('Time', explode(split(col("Summary"), "\s+")).alias("Word"))
year = year.groupBy("Time", "Word").agg(count('Word').alias('Word_count'))


year = year.orderBy('Word_count', ascending=False)


window = Window.partitionBy(year['Time']).orderBy(year['Word_count'].desc())

year = year.select('*', rank().over(window).alias('rank')).filter(col('rank') <= 10).select('Time', 'Word', 'Word_count').cache()

year.write.save('hdfs:///output/job1_spark.csv', format='csv', mode='overwrite')


```

Viene caricato il file, spark si occupa di eliminare le righe malformate. Successivamente vengono selezionate solo le colonne di interesse, ossia Time e Summary. Dopo averle parsate, estrapolando solo l'anno e rimuovendo la punteggiatura (es '... ! ' ecc), la colonna Summary viene splittata in base allo spazio, creando quindi nuove righe che associano una singola parola per ogni anno.
In seguito viene effettuato il group by in base all'anno, aggiungendo una colonna Word_count, che contiene le occorrenze della parola nella colonna Word nell'anno della colonna Time.
Infine vi è un passaggio di ordinamento e di filtraggio, per estrapolare solo le prime 10 parole con più occorrenze per ogni anno.

I risultati del Job 1 utilizzando le varie tecnologie:
Map reduce:

```
1999  [('tale', 3), ('fairy', 3), ('day', 3), ('modern', 3), ('a', 3), ('is', 2), ('funny', 1), ('entertainingl', 1), ('child', 1), ('your', 1)]
2000  [('a', 11), ('master', 6), ('version', 5), ('', 4), ('great', 4), ('success', 3), ('afterlife', 3), ('bettlejuicebettlejuicebettlejuice', 3), ('research', 3), ('tv', 3)]
2001  [('beetlejuice', 7), ('dvd', 6), ('the', 6), ('a', 4), ('is', 4), ('terrible', 3), ('movie', 3), ('great', 3), ('on', 3), ('you', 3)]
2002  [('a', 20), ('the', 15), ('great', 14), ('beetlejuice', 12), ('of', 9), ('is', 9), ('this', 9), ('movie', 9), ('it', 8), ('for', 8)]
2003  [('the', 23), ('of', 12), ('not', 11), ('great', 10), ('and', 9), ('a', 8), ('for', 8), ('in', 8), ('best', 8), ('excellent', 7)]
2004  [('the', 116), ('best', 73), ('a', 50), ('for', 43), ('good', 41), ('is', 39), ('great', 36), ('i', 34), ('', 33), ('and', 31)]
2005  [('the', 213), ('a', 129), ('best', 121), ('', 111), ('for', 106), ('great', 102), ('good', 90), ('and', 77), ('of', 71), ('this', 68)]
2006  [('the', 873), ('great', 760), ('a', 675), ('best', 567), ('good', 552), ('for', 499), ('and', 479), ('tea', 441), ('', 411), ('not', 315)]
2007  [('great', 3095), ('the', 2341), ('good', 2055), ('best', 1693), ('a', 1476), ('for', 1448), ('tea', 1360), ('and', 1357), ('', 1205), ('not', 948)]
2008  [('great', 4487), ('the', 3770), ('good', 3222), ('a', 2610), ('for', 2481), ('best', 2373), ('and', 2244), ('', 1816), ('not', 1735), ('tea', 1720)]
2009  [('great', 7383), ('the', 5751), ('good', 4925), ('a', 4149), ('best', 3887), ('for', 3767), ('and', 3431), ('', 2998), ('not', 2937), ('my', 2695)]
2010  [('great', 11193), ('the', 8366), ('good', 7786), ('for', 6454), ('a', 5954), ('best', 5733), ('and', 5244), ('not', 4781), ('my', 4581), ('', 4359)]
2011  [('great', 20849), ('the', 15773), ('good', 14515), ('for', 11999), ('a', 11400), ('not', 10179), ('and', 9811), ('best', 9075), ('my', 9058), ('love', 8430)]
2012  [('great', 24675), ('good', 18300), ('the', 17979), ('a', 13957), ('for', 13600), ('and', 12239), ('not', 12235), ('it', 10315), ('love', 10219), ('coffee', 10209)]
```

Hive:

```
2011◁best◁26124
2011◁my◁25839
2011◁not◁24945
2011◁and◁24864
2011◁love◁23982
2012◁great◁68895
2012◁good◁49374
2012◁the◁49056
2012◁for◁37086
2012◁a◁37026
2012◁and◁31284
2012◁not◁30033
2012◁love◁28869
2012◁my◁28767
2012◁best◁28143
```

Spark:

```
+----+--------------------+----------+
|Time|                Word|Word_count|
+----+--------------------+----------+
|2012|               great|     22631|
|2012|                good|     16209|
|2012|                 the|     15089|
|2012|                 for|     11529|
|2012|                   a|     11364|
|2012|                and|     10342|
|2012|                not|     10011|
|2012|               love|      9274|
|2012|                  it|      8903|
|2012|              coffee|      8902|
|2005|                 the|       171|
|2005|                   a|       106|
|2005|                best|       101|
|2005|               great|        94|
|2005|                 for|        91|
|2005|                good|        77|
|2005|                 and|        65|
```

I risultati sono pressocchè uguale, al netto di piccole differenze sulle occorrenze, dovuto alle differenti librerie utilizzate per leggere il file csv.

Infine, di seguito sono riportati i grafici e le tabelle relative al tempo di esecuzione delle tre tecnologie proposte, sulla base di dimensione in input variabile.
In particolare, in input è stato dato un quarto dell'input originale, la metà, il file originale, il doppio e il triplo. I file "sintetici" sono stati creati aggiungendo in modo casuale le righe del file stesso, cambiando alcuni valori.

Per l'esecuzione in locale, la macchina utilizzata ha le seguenti caratteristiche:

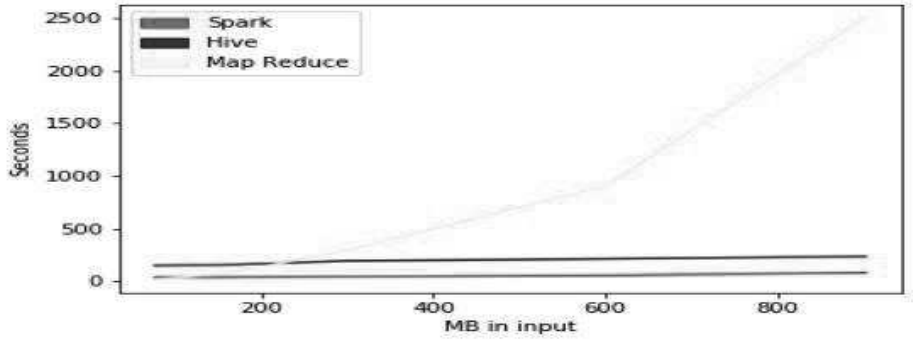
- 4 cores i7, 12 GiB Ram, 1 TB storage

Per quanto riguarda l'esecuzione su cluster, ho creato un cluster utilizzando Amazon Web Services, con 3 nodi (1 master, 2 slaves) con le seguenti caratteristiche:

- 8 vCore, 15 GiB memory, 80 SSD GB storage

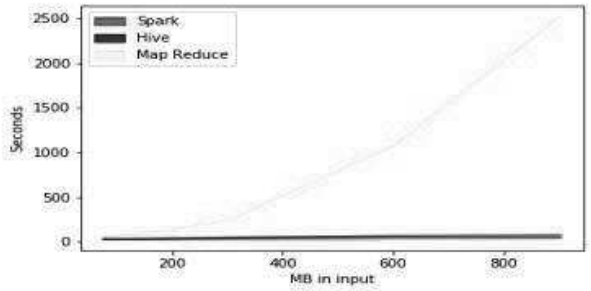
JOB 1 (locale)

Tecnologia\dim. input (MB)	76	152	300	600	900
Map Reduce	16s	67s	295s	909s	2505s
Hive	150s	155s	195s	212s	236s
Spark	37s	37s	43s	56s	84s



Nell'immagine seguente viene presentato un confronto fra l'esecuzione su cluster ed in locale (tabella) e l'andamento delle tre tecnologie al variare della dimensione dell'input in relazione al tempo (espresso in secondi).
Si può notare come la differenza in termini di prestazioni risulti sempre più netta all'aumentare della dimensione dell'input.

Tecnologia\d:	76		152		300		600		900	
	locale	cluster	locale	cluster	locale	cluster	locale	cluster	locale	cluster
Map Reduce	16s	55s	67s	91s	295s	240s	909s	1080s	2505s	2520s
Hive	150s	34s	155s	36s	195s	49s	212s	66s	236s	74s
Spark	37s	30s	37s	31s	43s	36s	56s	41s	84s	47s



Job 2

Map reduce:

Per il secondo Job ho utilizzato un mapper ed un reducer, anche in questo caso implementati come script Python.

Il mapper, per ogni riga del file che riceve, se essa fa riferimento ad un anno compreso tra 2003 e 2012, estrapola codice prodotto e score associato e stampa:

- CodiceProdotto Anno:Score

Di seguito lo pseudocodice

```
"""mapper.py"""

columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator","HelpfulnessDenominator",
           "Score","Time","Summary","Text"]

# input comes from STDIN (standard input)
for words in line_input:

    year = datetime.words['Time']

    if(year >= 2003 and year <= 2012):

        product = words['ProductId']
        score = words['Score']

        print('%s\t%s:%s' % (product, year, score))
```

Il reducer prende in input l'output del mapper e si occupa come prima cosa di estrapolare year, prodotto e score (split su "\t" e su ":"), inserendo le informazioni in un dizionario che ha come chiave il codice prodotto e come valore un altro dizionario che ha come chiave l'anno e come valore gli score di quell'anno (per un determinato prodotto).
A questo punto viene ordinato il dizionario in base alla chiave (prodotto) e successivamente viene scansionato sulla base delle chiavi. Per ogni chiave viene calcolata la media degli score per ogni anno.

Di seguito lo pseudocodice:

```
"""reducer.py"""

# maps year to their scores
product2year_score = {}

# input comes from STDIN
for line in input_line:

    # parse the input we got from mapper.py
    product, year_score = line.split('\t')
    year, score = year_score.split(':')

    if product in product2year_score:
        if year in product2year_score[product]:
            product2year_score[product][year].append(score)
        else:
            product2year_score[product][year] = [score]
    else:
        product2year_score[product] = {year:[score]}


#sort dictionary by key (product)
prod2 = (sorted(product2year_score.items()))


#get dict from sorted list of tuples (product, dict)
prod2_dict = dict(prod2)

for k in prod2_dict.keys():
    avg = average_year(prod2_dict[k])

    print ('%s\t%s' % (k, avg) )
```

Hive:

Codice:

```
CREATE TABLE Reviews ( Id STRING, ProductId STRING, UserId STRING, ProfileName STRING, HelpfulnessNumerator STRING, HelpfulnessDenominator STRING,
        Score STRING, Time STRING, Summary STRING, Text STRING )

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

LOAD DATA LOCAL INPATH 'Reviews_nofirstline.csv' INTO TABLE Reviews;

add jar unix_date.jar;

CREATE TEMPORARY FUNCTION get_year AS 'unix_date.Unix2Date';

CREATE TABLE mapper_out_tmp AS
SELECT ProductId as product, get_year(Time) as year, Score as score FROM Reviews;

CREATE TABLE mapper_out AS
SELECT * FROM mapper_out_tmp mp WHERE mp.year >= 2003 and mp.year <= 2012 ORDER BY product;

CREATE TABLE result_2 AS
SELECT product, year, avg(score) FROM mapper_out GROUP BY product, year;
```

Dopo aver creato la tabella relativa al file in input, e una funzione temporanea per estrapolare l'anno dal timestamp, viene creata una tabella mapper_out_tmp selezionando le colonne ProductID, Year e Score.
Da questa tabella viene creata una tabella mapper_out estrapolando le righe che hanno Year compreso tra 2003 e 2012.
La tabella result viene ottenuta facendo il group by sy prodotto e anno, e la media fra i valori della colonna score.

Spark:

Pseudocodice:

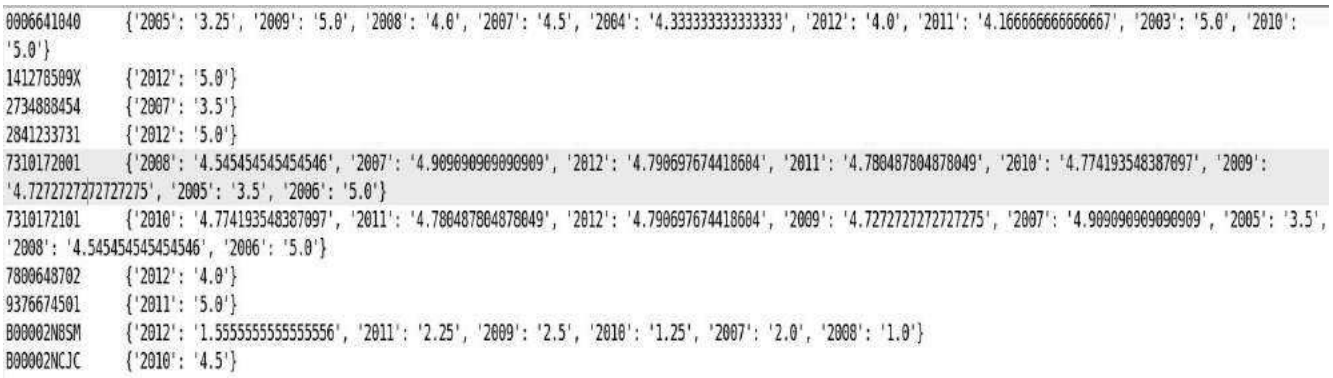
```
columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator","HelpfulnessDenominator",
           "Score","Time","Summary","Text"]

dataframe2 = spark.read.format('csv').option('header','true').option('mode','DROPMALFORMED')\
.load("hdfs:///input/Reviews.csv")

product = dataframe2.withColumn('Time', from_unixtime('Time')).cache()
product = product.select('ProductId', product.Time.substr(1,4).alias('Time'), 'Score')
product = product.orderBy('ProductId')

product = product.filter(product.Time >= '2003').filter(product.Time <= '2012')

product = product.groupBy('ProductId', 'Time').agg(avg('Score').alias('avg_score'))

product = product.orderBy('ProductId','Time').cache()
product.write.save('hdfs:///output/job2_spark.csv', format='csv', mode='overwrite')
```

Anche in questo caso è stato utilizzato un dataframe.
Dopo aver caricato il file dato, sono state caricate le colonne ‘productId’, ‘Time’ (opportunamente parsata), e ‘Score’, ed è stato effettuato un ordinamento in base al prodotto.

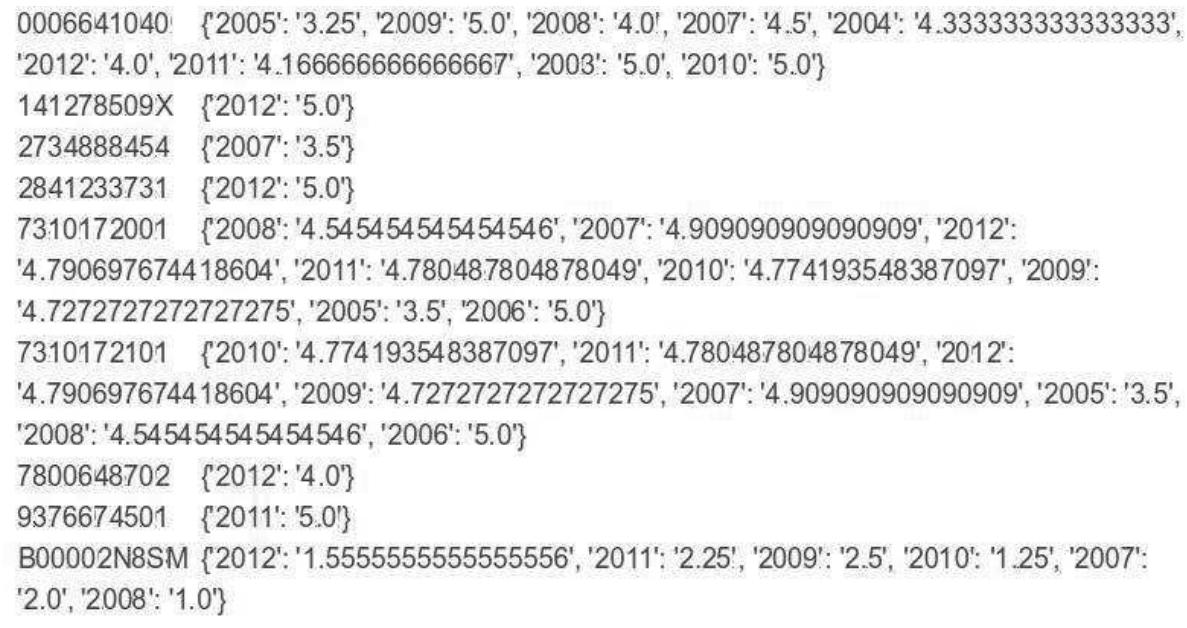
In seguito vengono filtrate le righe che hanno il campo Time compreso fra 2003 e 2012, e poi viene calcolata la media degli score sulla base del productId e dell’anno di riferimento.

I risultati (le prime righe) del Job 2 ottenuti tramite le tre tecnologie sono presentati nelle immagini a seguire.

Map reduce:



Hive:



Spark:

```
0006641040  {'2005': '3.25', '2009': '5.0', '2008': '4.0', '2007': '4.5', '2004': '4.333333333333333', '2012': '4.0', '2011':
'4.166666666666667', '2003': '5.0', '2010': '5.0'}
141278509X  {'2012': '5.0'}
2734888454  {'2007': '3.5'}
2841233731  {'2012': '5.0'}
7310172001  {'2008': '4.545454545454546', '2007': '4.909090909090909', '2012': '4.790697674418604', '2011':
'4.780487804878049', '2010': '4.774193548387097', '2009': '4.7272727272727275', '2005': '3.5', '2006': '5.0'}
7310172101  {'2010': '4.774193548387097', '2011': '4.780487804878049', '2012': '4.790697674418604', '2009':
'4.7272727272727275', '2007': '4.909090909090909', '2005': '3.5', '2008': '4.545454545454546', '2006': '5.0'}
7800648702  {'2012': '4.0'}
9376674501  {'2011': '5.0'}
B00002N8SM {'2012': '1.5555555555555556', '2011': '2.25', '2009': '2.5', '2010': '1.25', '2007': '2.0', '2008': '1.0'}
B00002NCJC  {'2010': '4.5'}
```

Infine vengono presentati i risultati sperimentali dell'esecuzione del Job2 in locale e su cluster.
Per l'esecuzione in locale, la macchina utilizzata ha le seguenti caratteristiche:

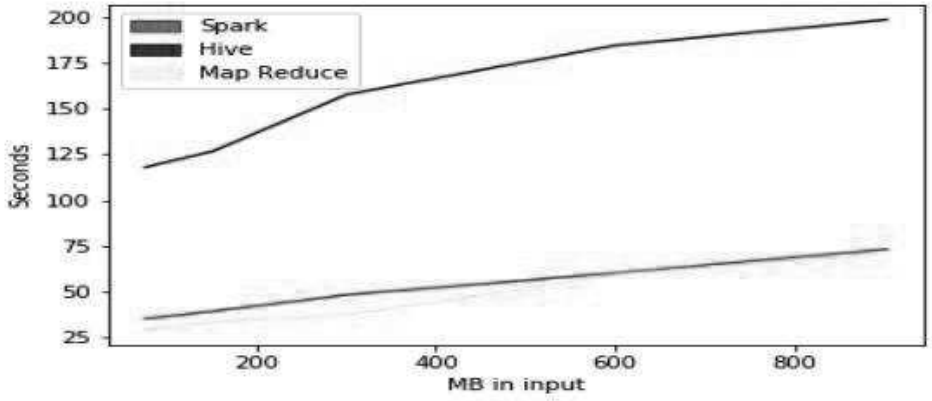
- 4 cores i7, 12 GiB Ram, 1 TB storage

Per quanto riguarda l'esecuzione su cluster, ho creato un cluster utilizzando Amazon Web Services, con 3 nodi (1 master, 2 slaves) con le seguenti caratteristiche:

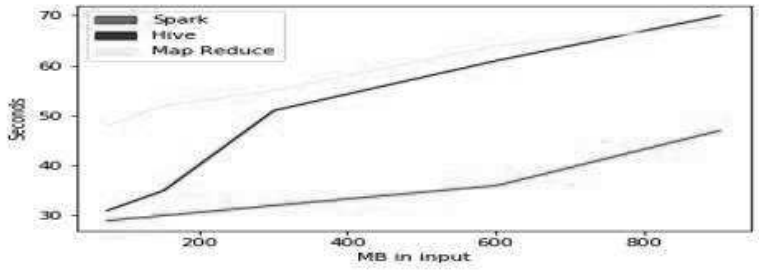
- 8 vCore, 15 GiB memory, 80 SSD GB storage

JOB 2 (locale)

Tecnologia\dim. input (MB)	76	152	300	600	900
Map Reduce	29s	33s	37s	59s	71s
Hive	118s	127s	158s	185s	199s
Spark	35s	39s	48s	60s	73s



Tecnologia\d	76		152		300		600		900	
	locale	cluster	locale	cluster	locale	cluster	locale	cluster	locale	cluster
Map Reduce	29s	48s	33s	52s	37s	55s	59s	64s	71s	68s
Hive	118s	31s	127s	35s	158s	51s	185s	61s	199s	70s
Spark	35s	29s	39s	30s	48s	32s	60s	36s	73s	47s



Job 3

Map reduce:

Per il Job 3 sono stati utilizzati 2 mapper e due reducer, al fine di rendere l'esecuzione più efficiente.

```
columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator","HelpfulnessDenominator",
           "Score","Time","Summary","Text"]


for words in input_line:

    userID = words['UserId']
    product= words['ProductId']
    print('%s\t%s' % (userID, product))
```

Il primo mapper estrapola e manda in output "userID ProductId".

il primo reducer (figura sotto) prende input l'output del primo mapper, e crea una lista di prodotti associati ad un user. In seguito emette tutte le permutazioni della lista di prodotti associati all'utente come coppia, ed il numero 1.
Es: '(prodotto1,prodotto2) 1

```
current_key = None
key = None
curr_prods = []

def emit_result():
        curr_prods_unique = sorted(curr_prods)
        for pair_prod in itertools.combinations(curr_prods_unique, 2):
                print('{}\t{}'.format(pair_prod, 1))


# user1 [prod1, prod2]
for line in sys.stdin:

        key, prod = line.split('\t')

        if current_key == key:
                curr_prods.append(prod)
        else:
                if current_key == None:
                        current_key = key
                        curr_prods.append(prod)
                else:
                        emit_result();

                        current_key = key
                        curr_prods = []
                        curr_prods.append(prod)

emit_result();
```

Il secondo mapper esegue lo split sul carattere “\t” e fornisce al secondo reducer la coppia ed il numero 1, ed il reducer2 si occupera di contare le occorrenze di ogni coppia, che rappresentano il numero di utenti che hanno recensito entrambi i prodotti.
Di seguito il secondo mapper ed il secondo reducer.

```
"""mapper2.py"""

for line in input_line:

        key, value = line.split('\t')
        key = key.split()

        print('{}\t{}'.format(key, value))
```

```
"""reducer2.py"""

current_key = None
key = None
current_count = []

def emit_result():
        print('{}\t{}'.format(current_key, len(current_count)))

for line in input_line:
        key, count = line.split('\t')

        try:
                count = int(count)
        except ValueError:
                continue

        if current_key == key:
                current_count.append(count)
        else:
                if current_key == None:
                        current_key = key
                        current_count = [count]
                else:
                        emit_result()
                        current_key = key
                        current_count = [count]
emit_result()
```

Hive:

Il codice del Job3 in Hive è il seguente:

```
CREATE TABLE Reviews ( Id STRING, ProductId STRING, UserId STRING, ProfileName STRING, HelpfulnessNumerator STRING, HelpfulnessDenominator STRING,
        Score STRING, Time STRING, Summary STRING, Text STRING )

ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

LOAD DATA LOCAL INPATH '/home/davide/Documenti/BIG_DATA/codice/Reviews_nofirstline.csv' INTO TABLE Reviews;

add jar /home/davide/Documenti/BIG_DATA/codice/unix_date.jar;

CREATE TEMPORARY FUNCTION get_year AS 'unix_date.Unix2Date';

CREATE TABLE mapper_out AS
SELECT ProductId as product, UserId as user_ FROM Reviews;

CREATE TABLE result_3 AS
SELECT
 t1.product AS item1,
 t2.product AS item2,
 COUNT(1) AS cnt
FROM
(
 SELECT DISTINCT product, user_
 FROM mapper_out
) t1
JOIN
(
 SELECT DISTINCT product, user_
 FROM mapper_out
) t2
ON (t1.user_ = t2.user_)
GROUP BY t1.product, t2.product
HAVING t1.product != t2.product
ORDER BY t1.product;
```

Dopo aver riempito la tabella di input, vengono selezionate le colonne di interesse, creando la tabella mapper_out.

Il passaggio successivo è rappresentato da un join sugli utenti della tabella mapper_out con se stessa, eseguendo il groupy by sui prodotti e eliminando le righe che rappresentano la coppia con lo stesso prodotto. Infine viene ordinato il risultato sulla base del primo prodotto della coppia.

Spark:

```
columns = ["Id","ProductId","UserId","ProfileName","HelpfulnessNumerator","HelpfulnessDenominator",
           "Score","Time","Summary","Text"]


dataframe2 = spark.read.format('csv').option('header','true').option('mode','DROPMALFORMED')\
.load('hdfs:///input/Reviews.csv')

product1 = dataframe2.select(dataframe2.ProductId.alias('prod1'), dataframe2.UserId.alias('user1'))
product1 = product1.orderBy('prod1')

product2 = dataframe2.select(dataframe2.ProductId.alias('prod2'), dataframe2.UserId.alias('user2'))
product2 = product2.orderBy('prod2')

df = product1.join(product2, product1.user1 == product2.user2).filter(product1.prod1 != product2.prod2)\
    .orderBy(product1.prod1)

df = df.select(df.prod1.alias('Product1'), df.prod2.alias('Product2'), df.user1.alias('User'))


df = df.groupBy('Product1', 'Product2').count()
```

Il codice di Spark è molto simile al codice di Hive, avendo utilizzato anche in questo caso i dataframe, e quindi avendo fatto le operazioni sulle colonne.

Dopo aver caricato il dataset, vengono creati due dataframes uguali, con il codice prodotto e il codice utente.

Successivamente viene fatto il join sulla base degli utenti fra i due dataframe, filtrando le righe che hanno lo stesso codice prodotto.
Infine, dopo aver ordinato in base al primo prodotto, vengono contate le occorrenze facendo il group by su entrambi i prodotti.

Le prime righe dei risultati sono visibile nelle immagini seguenti:

Map reduce:

["('000664104O',", "'B0005XN9HI')"]	1
["('000664104O',", "'B00061EPKE')"]	1
["('000664104O',", "'B000EM00YU')"]	1
["('000664104O',", "'B000FDQV46')"]	1
["('000664104O',", "'B000FV8LPU')"]	1
["('000664104O',", "'B000MGOZEO')"]	1
["('000664104O',", "'B000MLHU3M')"]	1
["('000664104O',", "'B000UVW59S')"]	1
["('000664104O',", "'B000UVZRES')"]	1
["('000664104O',", "'B000UW1Q8I')"]	1
["('000664104O',", "'B000UXH9X8')"]	1
["('000664104O',", "'B000UXW95G')"]	1
["('000664104O',", "'B0013P3KC6')"]	1
["('000664104O',", "'B0014UFXGG')"]	1
["('000664104O',", "'B0014WYY1E')"]	1
["('000664104O',", "'B0015UW23M')"]	1
["('000664104O',", "'B00178U95K')"]	1

Hive:

```
0006641040◁B0013P3KC6◁1
0006641040◁B0014WYY1E◁1
0006641040◁B0015UW23M◁1
0006641040◁B00178U95K◁1
0006641040◁B001DR0LU8◁1
0006641040◁B001ELL4ZY◁1
0006641040◁B001EQ55RW◁1
0006641040◁B001LG940E◁1
0006641040◁B000FDQV46◁1
0006641040◁B003WFUW40◁1
0006641040◁B004OV5QRS◁1
0006641040◁B00503DOWS◁1
0006641040◁B006N3I79Y◁1
0006641040◁B000UW1Q8I◁1
0006641040◁B000MLHIU3M◁1
0006641040◁B00061EPKE◁1
0006641040◁B000UVW59S◁1
```

Spark:

```
+----------+----------+-----+
|  Product1|  Product2|count|
+----------+----------+-----+
|7310172001|B0050TDMD8|    1|
|7310172101|B000UG3FD8|    1|
|7310172101|B000LKTE2E|    1|
|B00004RAMY|B0090X8IPM|    1|
|B00004RBDZ|B003GTR8IO|    1|
|B00004RBDZ|B005HG9ET0|    1|
|B00004RYGX|B005VOONLW|    1|
|B00004S1C6|B004V3IO0U|    1|
|B000052Y74|B001E5308O|    1|
|B000052Y74|B001GZ2ZB0|    1|
|B00005344V|B000WB1YSE|    1|
|B0000691JF|B000084EKY|    1|
|B00006IDK9|B0014WYXYW|    1|
|B00006LL38|B000E5AO8O|    1|
|B000084346|B00008CQVA|    1|
|B000084E66|B000FDDET6|    1|
|B000084E6V|B000C0E2NM|    1|
```

La differenza fra i risultati di spark e quelli di map reduce e hive potrebbe essere dovuta alle righe malformate che spark elimina automaticamente.

Infine vengono presentati i risultati sperimentali dell'esecuzione del Job3 in locale e su cluster.
Per l'esecuzione in locale, la macchina utilizzata ha le seguenti caratteristiche:

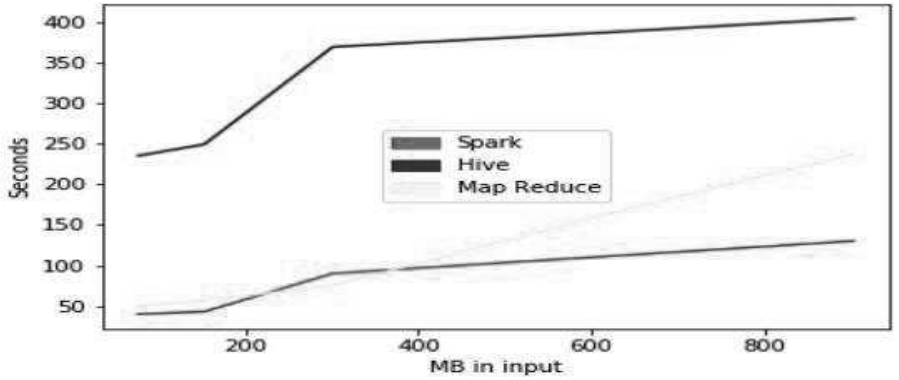
- 4 cores i7, 12 GiB Ram, 1 TB storage

Per quanto riguarda l'esecuzione su cluster, ho creato un cluster utilizzando Amazon Web Services, con 3 nodi (1 master, 2 slaves) con le seguenti caratteristiche:

- 8 vCore, 15 GiB memory, 80 SSD GB storage

JOB 3 (locale)

Tecnologia\dim. input (MB)	76	152	300	600	900
Map Reduce	50s	57s	75s	159s	237s
Hive	235s	249s	369s	386s	404s
Spark	40s	43s	90s	110s	130s



Tecnologia\di	76		152		300		600		900	
	locale	cluster	locale	cluster	locale	cluster	locale	cluster	locale	cluster
Map Reduce	16s	55s	67s	91s	295s	240s	909s	1080s	2505s	2520s
Hive	235s	35s	249s	45s	369s	70s	386s	81s	404s	86s
Spark	40s	54s	43s	54s	90s	59s	110s	68s	130s	110s