## **JOB1**:

## MapReduce

La soluzione a questo problema si compone di un Mapper, un Combiner e un Reducer. Il compito del Mapper è quello di generare, per ogni anno, le coppie (anno, (parola, 1)). Il Combiner somma le occorrenze di ogni parola in un certo anno producendo le coppie (anno, (parola, N)). Il Reducer seleziona per ogni anno le 10 parole con il numero maggiore di occorrenze. Per far ciò si conta inizialmente il numero di occorrenze di ogni parola. Scorrendo tutte le parole (per un certo anno) queste vengono inserite in un TreeMap insieme al relativo numero di occorrenze. La TreeMap è ordinata in base alla frequenza delle parole precedentemente calcolata e, appena la sua dimensione supera 10, l'ultimo elemento viene eliminato. L'uso di una TreeMap permette di mantenere una complessità pari a O(n). Invece, ordinare le parole in base alla loro frequenza e prendere le prime 10 richiederebbe una complessità pari a O(n log n).

**Map:**

```
map (key, value, context) {
        values = parse(value);   // parsing del file CSV
        controlli sulla consistenza della porzione del file in esame
        year = getYear(values[TIME_POSITION]); // unix time -> anno
        summary = values[SUMMARY_POSITION];
        words = summary.split(" +") *
        verifica anno nel range
        for (w : words)
                context.write( year, (w, 1)*);
}
```

1* Prima di effettuare lo split, tutti i segni di interpunzione sono rimpiazzati da spazi

2* La coppia (w, 1) è un oggetto del tipo WordOccurrences, un tipo da noi definito con i campi word e occurrences che rappresentano rispettivamente una parola e il numero delle sue occorrenze in un anno.

**Combiner:**

```
combiner (key = year, values = [(word1, 1), …(wordN, 1)], context) {
      Map<String, Int> word2occurrences = raggruppa values per word e somma i
                                          valori del campo occurrences
      for ((word, occurrences) : word2occurrences)
        context.write(year, (word, occurrences))
}
```

**Reducer:**

```
combiner (key = year, values = [(word1, N), …(wordM, K)], context) {
        Map<String, Int> word2occurrences = raggruppa values per word e somma i
                                            valori del campo occurrences
        TreeMap<String, Int> ordered = TreeMap in cui gli elementi vengono ordinati con il
                                       seguente comparatore
                                       (a, b) -> word2occurrences[a] – word2occurrences[b]
```

```
            for ((word, occurrences) : word2occurrences)
                ordered.put(word, occurrences)
                if (ordered.size() > 10) rimuovi ultimo elemento di ordered

            context.write(key, valori in ordered)
        }
```

## Hive

Le esecuzioni su hive sfruttano una tabella *reviews* in cui i dati vengono caricati una volta per tutte. Questo fa sì che in alcuni casi i tempi impiegati per le esecuzioni di hive possono essere notevolmente minori rispetto alle altre tecnologie (La query che carica questi dati e presente alla fine di questo documento).

```
-- crea una tabella <anno> <parola> rimpiazzando i segni di interpunzione con spazi
-- ed eseguendo uno split su " +"
create table es1_year2word as (
    select year, exp.splitted as word
    from reviews
    lateral view explode(split(lower(trim(regexp_replace(summary,
    '[\'_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"]', " "))), " +")) exp as splitted
);
-- Crea una tabella <anno> <parola> <occorrenze di quella parola in quell'anno>
create table es1_ywc as (
    select year, word, count(1) as freq
    from es1_year2word
    group by year, word
);
-- ad ogni parola, per ogni anno, assegna una posizione in base alla frequenza
drop table if exists es1_rank;
create table es1_rank as (
    select year, word, freq, row_number() over (partition by year order by freq desc) as rnk
    from es1_ywc
);
-- seleziona solo le parole
drop table if exists es1_final;
create table es1_final as (
    select year, word, freq from es1_rank where rnk <= 10 order by year, freq desc);
```

## Spark

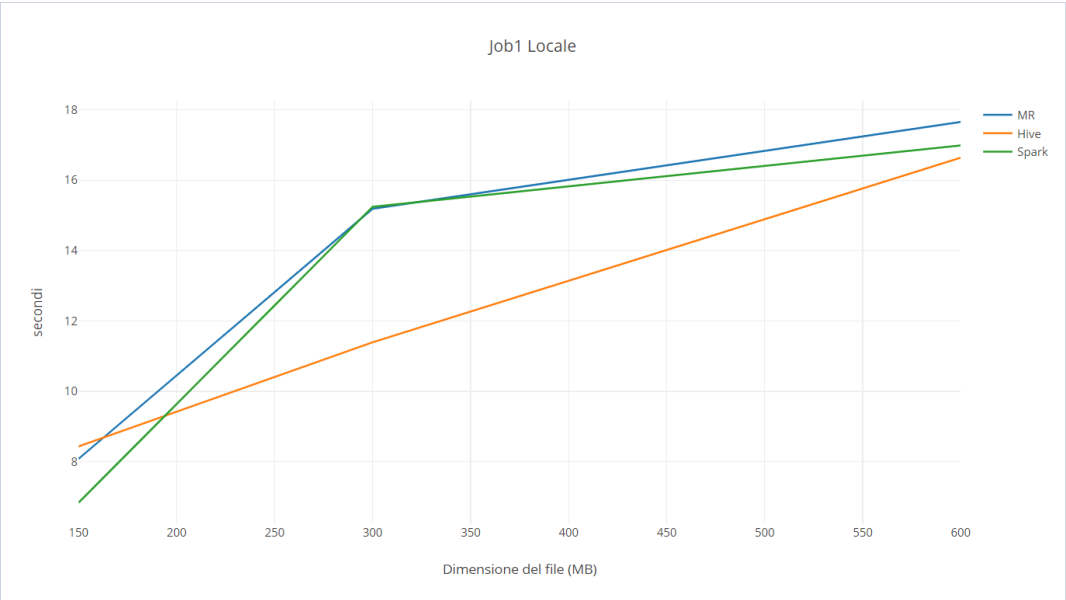
Durante l'implementazione dei tre job in Spark è stata usata una classe Record per poter avere in un unico oggetto tutti i valori di una riga del file csv.

La soluzione proposta per questo job prevede di produrre delle tuple ((anno, parola), 1), con un reduceByKey è quindi possibile contare le occorrenze di una certa parola per anno. A questo punto (con una map) si generano le coppie (occorrenze, (anno, parola)) e si ordina per occorrenze. Mantenendo questo ordinamento si generano le coppie (anno, (occorrenze, parola)). Si esegue poi una groupbyKey che restituisce, per ogni anno, le tuple (anno, [(occorrenze1, parola1), [(occorrenze2, parola2), …, [(occorrenzeN, parolaN)]). Grazie al precedente ordinamento, le parole sono ancora ordinate per numero di occorrenze ed è quindi sufficiente prelevare le prime 10.

```
RDD<Record> records = loadData(…); // caricamento dati dal csv
RDD<(int, String), int> records = Per ogni parola nel summary genera una coppia ((anno, parola), 1)
    records.reduceByKey((a, b) -> a + b) // conta le occorrenze di una parola per anno
    .map( ((anno, parola), occorrenze) -> (occorrenze, (anno, parola)) )
    .sortByKey() // in ordine decrescente
    .map((occorrenze, (anno, parola)) -> (anno, (occorrenze, parola)))
    .groupByKey() // si ottengono le parole ordinate grazie all'ordinamento precedente
    Prendi i primi 10 per ogni anno
```

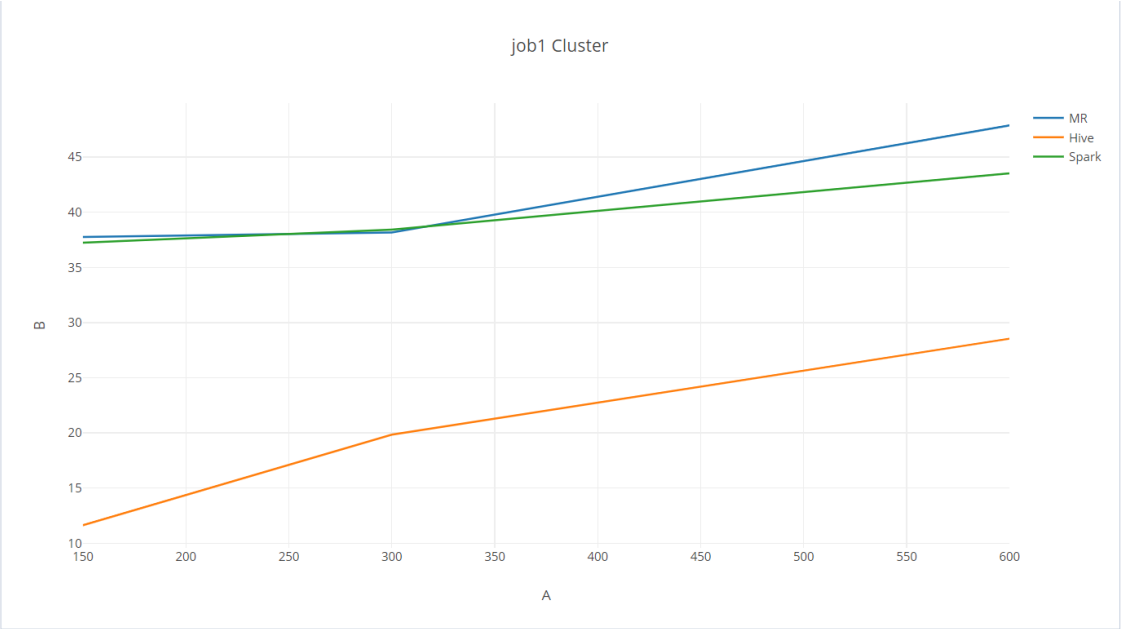
## Tempi in locale

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 8.082 | 15.188 | 17.656 |
| Hive | 8.436 | 11.394 | 16.638 |
| Spark | 6.837 | 15.244 | 16.987 |



Tempi su cluster (3 nodi m4.large)

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 37.755 | 38.161 | 47.877 |
| Hive | 11,63 | 19,84 | 28,54 |
| Spark | 37,233 | 38.431 | 43,521 |




## Risultati

### MapReduce

1999 a 3 day 3 fairy 3 modern 3 tale 3 is 2 book 1 child 1 educational 1 entertainingl 1
2000 a 11 bettlejuice 9 master 6 version 5 great 4 afterlife 3 burton 3 by 3 clamshell 3 comedic 3
2001 beetlejuice 7 dvd 6 the 6 a 4 is 4 but 3 cheated 3 great 3 greatmovie 3 movie 3
2002 a 20 the 15 great 14 beetlejuice 12 is 9 movie 9 of 9 this 9 for 8 it 8
2003 the 23 of 12 not 11 great 10 and 9 a 8 best 8 for 8 in 8 but 7
2004 the 116 best 73 a 53 for 43 good 41 i 40 is 39 great 36 and 31 coffee 31
2005 the 213 a 128 best 121 for 108 great 102 good 91 and 82 of 72 it 70 this 70
2006 the 880 great 766 a 681 best 569 good 553 for 499 and 497 tea 442 it 377 not 316
2007 great 3116 the 2356 good 2069 best 1702 a 1509 for 1450 tea 1375 and 1364 not 959 i 943
2008 great 4524 the 3791 good 3274 a 2626 for 2492 best 2378 and 2272 not 1745 tea 1734 it 1707
2009 great 7448 the 5774 good 4972 a 4177 best 3911 for 3772 and 3465 not 2982 it 2724 my 2704
2010 great 11264 the 8422 good 7840 for 6468 a 6001 best 5751 and 5325 not 4862 my 4597 it 4596
2011 great 20979 the 15872 good 14626 for 12021 a 11489 not 10320 it 9964 and 9935 best 9128 my 9074

2012 great 24840 good 18420 the 18104 a 14005 for 13634 not 12387 and 12362 it 12054 coffee 10364 love 10287

## Hive

1999 a 3
1999 tale 3
1999 fairy 3
1999 modern 3
1999 day 3
1999 is 2
1999 funny 1
1999 entertainingl 1
1999 great 1
1999 series 1
2000 a 11
2000 bettlejuice 9
2000 master 6
...
2012 great 24840
2012 good 18420
2012 the 18104
2012 a 14006

## Spark

(1999,[(3,fairy), (3,a), (3,tale), (3,day), (3,modern), (2,is), (1,this), (1,your), (1,to), (1,time)])
(2000,[(11,a), (9,bettlejuice), (6,master), (5,version), (4,great), (3,from), (3,funny), (3,research), (3,fanasy), (3,comedic)])
(2001,[(7,beetlejuice), (6,dvd), (6,the), (4,is), (4,a), (3,great), (3,on), (3,cheated), (3,but), (3,greatmovie)])
...
(2011,[(20979,great), (15872,the), (14626,good), (12021,for), (11489,a), (10320,not), (9964,it), (9935,and), (9128,best), (9074,my)])
(2012,[(24840,great), (18420,good), (18104,the), (14005,a), (13634,for), (12387,not), (12362,and), (12054,it), (10364,coffee), (10287,love)])

## **JOB2**:

## MapReduce

In questo job la nostra proposta di soluzione è stata sviluppata con due classi Mapper, due classi Reducer, e un Combiner. I risultati della prima passata di MapReduce sono già corretti ma in un formato non conforme con quanto richiesto nell'esercizio. La seconda passata di MapReduce è quindi usata per ovviare a questo problema senza complicare la logica della prima. Inoltre, sono

state implementate due classi che implementano WritableComparable per il passaggio di opportuni oggetti dai Mapper ai rispettivi Reducer.

**Primo Map:**

```
map (key, value, context) {
        values = parse(value);   // parsing del file CSV
        controlli sulla consistenza della porzione del file in esame
        year = getYear(values[TIME_POSITION]); // unix time -> anno
        product = values[PRODUCT_POSITION];
        stars = values[STARS_POSITION];
        verifica anno nel range
        context.write( (year, product), (stars, 1));
}
```

Con (year, product) si intende un tipo composto, da noi definito, che implementa l'interfaccia WritableComparable. Lo stesso vale per (stars, 1) usato per poter effettuare una media parziale nel combiner e mantenere anche il numero di elementi su cui la media è stata calcolata (usati come peso nel reducer).

**Combiner**:

```
combiner (key = (year, product), values, context) {
        avg = calcola la media su values
        cont = calcola il numero di elementi in values
        // passo alla reduce la chiave = (anno, prodotto) e valore = (media, num. di elementi)
        context.write( key, (avg, cont) );
}
```

**Primo Reduce**:

```
private totalWeigth = 0;
private count (avg = (starsAvg, count)) {
        totalWeigth += avg.getCount(); // getCount() restituisce il peso di questa media
        return avg.getValue() * avg.getCount();  // getValue() restituisce la media parziale
}
reduce (key = (year, product), values, context) {
        // calcolo la media finale dello score per la coppia (anno, prodotto) in esame
        totalAverage = map.(this::count).sum();
        totalAverage = totalAverage/totalWeigth;
        context.write( (year, product) , totalAverage );

}
```

Nota: la prima map produce i risultati nel seguente formato (valori separati da \t):

| <anno> | <prodotto> | <score medio> |
|---|---|---|
| 2011 | BXXFFF01010 | 4.5 |

2012 BXXFFF01010 4.16666667

Ecc…

Per ottenere il formato <prodotto> <anno> <score medio> <anno> <score medio> … è stato necessario eseguire una seconda passata di MapReduce

**Secondo Map:**

```
// in questa seconda map si avrà come input l'output della prima reduce
map (key, value, context) {
        values = value.split("\t");
        product = values[1], year = values[0], avg = values[2];
        context.write( product , (year, avg) );
}
```

**Secondo Reduce:**

```
reduce (key = product, values, context) {
        sorted = sort(values); // ordinamento per anno
        context.write( product , sorted );
}
```

## Hive

```
create table es2 as (
    SELECT product, cast(year as string) as year, cast(AVG(score) as string) as average
        FROM reviews
        where year >= 2003
        group by product, year
        order by product);
```

## Spark

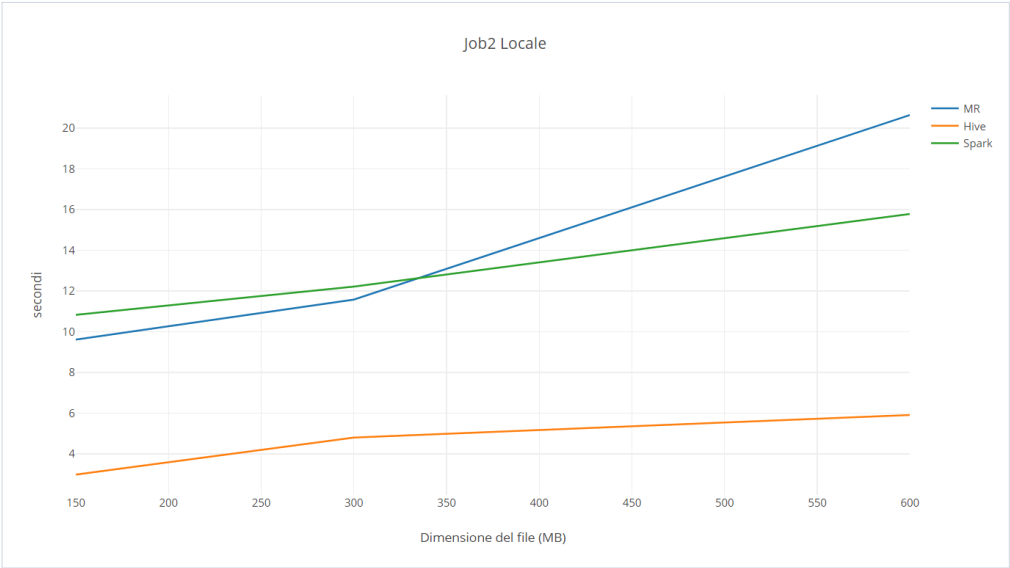
Per il job2 la soluzione da noi proposta prevede di raggruppare i record per prodotto e anno. Successivamente, per ogni prodotto e per ogni anno, viene calcolata la media delle *stars.* Si effettua poi una map per avere come chiave il prodotto e come valore l'anno e la media precedentemente calcolata. Si effettua poi una groupByKey per avere il formato

<prodotto> <anno><media> <anno><media> …

```
RDD<Record> records = loadData(…); // caricamento dati dal csv
RDD<(String, int), [Record]> prodYear2record =  records.groupBy(r -> (r.product, r.year));
prodYear2avg = prodYear2record.map(((product, year), records) -> ((product, year), records.avg()))
    .map(((prod, year), avg) -> (prod, (anno, avg)))
    .gruopByKey()
    .sortByKey();
```

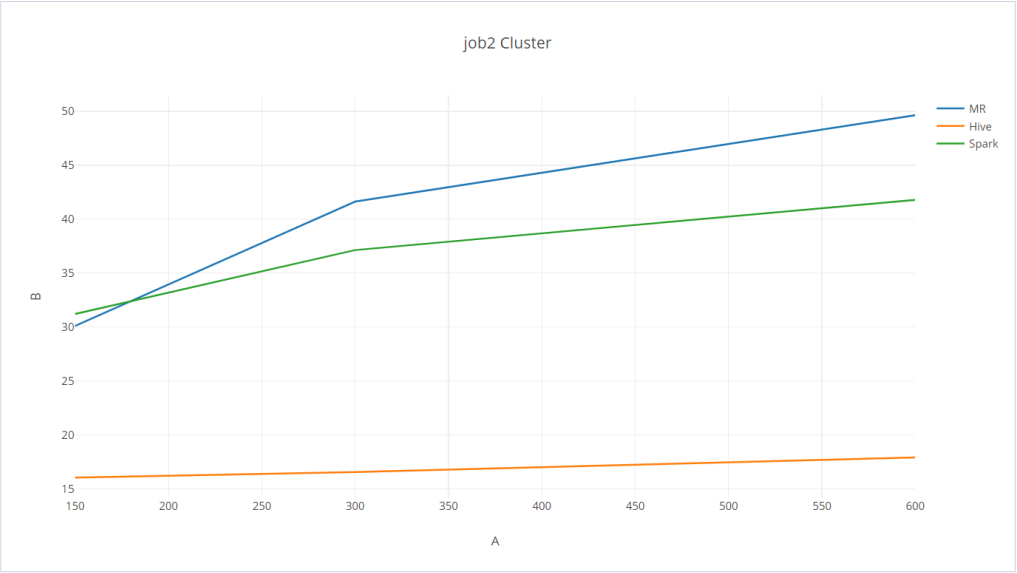
Tempi in locale

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 9.612 | 11.576 | 20.642 |
| Hive | 2.984 | 4.807 | 5.911 |
| Spark | 10.83 | 12.214 | 15.783 |




Tempi su cluster (3 nodi m4.large)

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 30.102 | 41.632 | 49,652 |
| Hive | 16,03 | 16,55 | 17,91 |
| Spark | 31.212 | 37,133 | 41,785 |



## Risultati

### MapReduce

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

### Hive

```
0006641040
        ["2004:4.333333333333333","2003:5.0","2007:4.5","2008:4.0","2009:5.0","2005:3.25","2
        010:5.0","2011:4.166666666666667","2012:4.0"]
141278509X ["2012:5.0"]
2734888454 ["2007:3.5"]
2841233731 ["2012:5.0"]
7310172001
        ["2008:4.545454545454546","2005:3.5","2006:5.0","2007:4.909090909090909",
        "2009: 4.7272727272727275","2010:4.774193548387097",
        "2011:4.780487804878049","2012:4.790697674418604"]
7310172101
        ["2005:3.5","2006:5.0","2007:4.909090909090909","2008:4.545454545454546",
        "2009:4.7272727272727275","2010:4.774193548387097",
        "2011:4.780487804878049","2012:4.790697674418604"]
7800648702 ["2012:4.0"]
9376674501 ["2011:5.0"]
B00002N8SM
        ["2011:2.25","2007:2.0","2008:1.0","2009:2.5","2010:1.25","2012:1.555555555555556"]
B00002NCJC  ["2010:4.5"]
```

## Spark

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

## **JOB3:**

## MapReduce

La soluzione proposta per questo job prevede sempre due Mapper e due Reducer.
Nel primo Mapper si creano le coppie <utente, prodotto> e il primo Reducer compone, per ogni utente, tutte le possibili coppie di prodotti: (p1, p2) in cui p1 < p2, questo è un modo veloce per evitare duplicati.

Nel secondo Mapper, per ogni coppia (p1, p2), si generano le coppie ((p1, p2), 1) e il secondo Reducer conta il numero di coppie con gli stessi prodotti.

**Primo Map:**

```
map (key, value, context) {
        values = parse(value);   // parser del file CSV
        controlli sulla consistenza della porzione del file in esame
        user = values[USER_POSITION];
        product = values[PRODUCT_POSITION];
        context.write( user, product );
}
```

**Primo Reduce:**

```
reduce (key = user, values = [prod1, prod2, …, prodN], context) {
        values = set(values) *
        for (v1 : values) // Generazione di tutte le coppie
            for (v2 : values)
                if (v1 < v2) // Evita duplicati
```

```
                context.write(v1, v2);
        }
```

* La creazione di un set risolve il seguente problema: se un utente avesse comprato due oggetti dello stesso tipo: X e X e un terzo oggetto Y, risulterebbero due coppie: <X, Y>, <X, Y> il che equivale a due utenti distinti che hanno comprato la stessa coppia di oggetti. Creando un set si eliminano invece i duplicati, risolvendo questo problema.

**Secondo Map (ha in input l'output del precedente reducer)**

```
map (key, value = (product1, product2), context) {
        context.write( value , 1 );
}
```

**Secondo Reduce**

```
reduce (key = (product1, product2), values, context) {
        sum = values.sum(); // somma degli utenti in comune
        context.write( key , sum );
}
```

## Hive

```
create table es3_pre as (
        select distinct usr, product from reviews);
create table es3 as (
        select r1.product as p1, r2.product as p2, count(1)
                from es3_pre r1 join es3_pre r2 on r1.usr = r2.usr and r1.product < r2.product
                group by r1.product, r2.product
                order by r1.product, r2.product);
```

* La prima query ha il solo scopo di eliminare i duplicati: stesso utente che ha acquistato più volte lo stesso prodotto (Il problema è stato descritto nel caso di MapReduce). L'operazione distinct è però piuttosto onerosa e non è presa in considerazione nei tempi.

## Spark

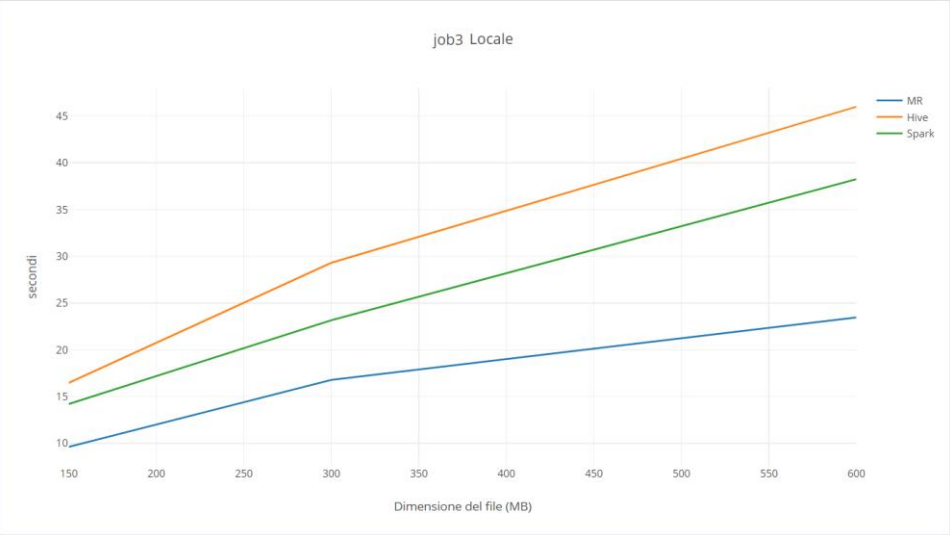
La soluzione prevede di effettuare un join tra un RDD di coppie (utente, prodotto) e sé stesso. Il risultato di questo join è un RDD che per ogni utente contiene tutte le coppie di prodotti da esso acquistati. Di queste coppie si selezionano solo quelle in cui product1 < product2 per evitare duplicati. Per ogni coppia si produce la tupla ((product1, product2), 1) e con una reduceByKey si calcola il numero di utenti che hanno comprato quei due prodotti congiuntamente.

```
RDD<Record> records = loadData(…); // caricamento dati dal csv
user2product = records.map(r -> (r.user, r.product)).distinct(); // eliminazione duplicati, vedi sopra
sameUser = user2product.join(user2product);
counted = sameUser.map((user, product1, product2) -> ((product1, product2), 1))
```

```
.filter(product1 < product2) // evita i duplicati
.reduceByKey((a, b) -> a + b) // somma il numer di coppie
.sortByKey();
```

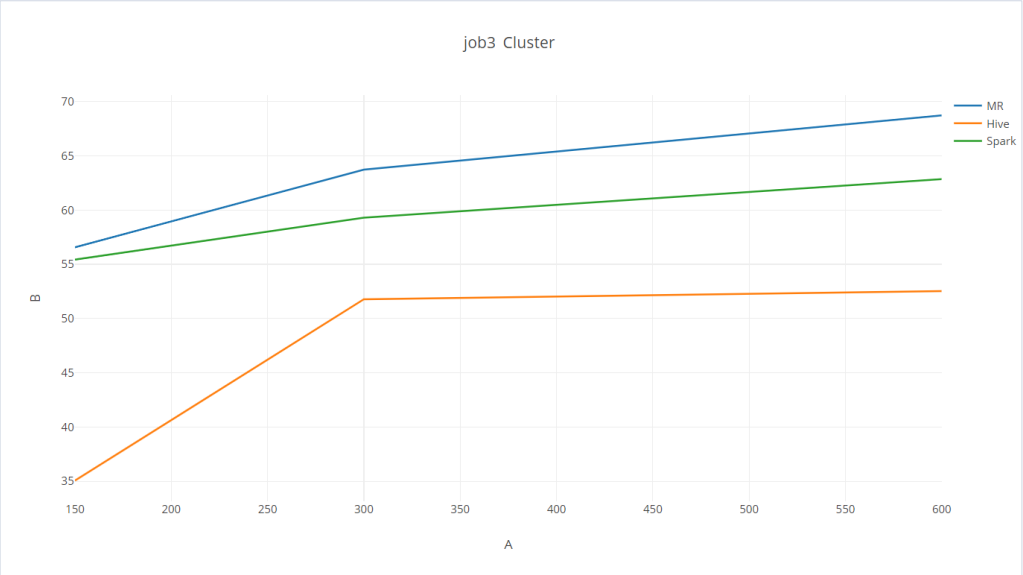
## Tempi in locale

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 9.607 | 16.776 | 23.461 |
| Hive | 16.461 | 29.303 | 45.999 |
| Spark | 14.199 | 23.164 | 38.247 |




## Tempi su cluster (3 nodi m4.large)

| | 150 MB | 300 MB | 600 MB |
|---|---|---|---|
| MR | 56.564 | 63.737 | 68.522 |
| Hive | 35,06 | 51,78 | 52,53 |
| Spark | 55,425 | 59,303 | 62,851 |




## Risultati

### MapReduce

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

### Hive

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

### Spark

((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)

## Hive – Creazione della tabella reviews

```
add jar csv-serde-1.1.3-1.2.1-all.jar;
DROP TABLE IF EXISTS reviews_tmp;
DROP TABLE IF EXISTS reviews;
CREATE TABLE reviews_tmp (id INT, product STRING, usr STRING, pn STRING, hn STRING, hd STRING, score
                                        INT, time BIGINT, summary STRING, t STRING)
row format serde 'com.bizo.hive.serde.csv.CSVSerde';
LOAD DATA LOCAL INPATH 'data/Reviews.csv'
OVERWRITE INTO TABLE reviews_tmp;
create table reviews as ( select id, product, usr, cast(score as int),
     cast(date_format(from_unixtime(cast(time as bigint)),'yyy') as int) as year, summary from reviews_tmp)
```

Simone Ceccarelli 475146

Damiano Massarelli 473486