

Amazon Fine Food Reviews Big Data

Primo Progetto

Niccolò Regini
462287

Introduzione

Per il primo progetto di Big Data sono state progettate e realizzate con tre tecnologie (MapReduce, Hive e Spark) tre Job. I Job sono stati eseguiti in locale e su Amazon EMR. In locale sono stati eseguiti su una macchina con processore Intel Core i7 3 GHz, 16 GB di memoria Ram e disco SSD. Su Amazon EMR è stato configurato un cluster di tre istanze di cui uno è il master. La release EMR scelta è la emr-5.13.0. L'istanza è di tipo m3.xlarge che prevede: 8 vCore, 15 GiB memory, 80 SSD GB storage.

Per ogni Job e per ogni tecnologia utilizzata sono stati confrontati i tempi di esecuzione sia in locale che su cluster EMR al variare delle dimensioni dell'input.

L'input utilizzato è il dataset Amazon Fine Food Reviews in formato CSV. Per verificare i tempi di esecuzione al variare delle dimensioni il dataset iniziale è stato duplicato e sono stati creati cinque input CSV come nella tabella che segue:

Nome File	CSV 1	CSV 2	CSV 3	CSV 4	CSV 5
Dimensioni	300,9 MB	601,8 MB	902,7 MB	1,2 GB	1,5 GB

1. Primo Job

Un job che sia in grado 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.

1.1. Map Reduce

- Nella fase di Map viene eseguito il parsing della riga corrente del file CSV e viene verificato che ogni riga contiene esattamente dieci campi. Successivamente per ogni parola del campo "Summary" è stata generata la coppia anno-parola. Dove l'anno è la chiave della Map.
- Dopo lo shuffle and sort nella fase di Reduce si ha per ogni anno una lista di parole. La lista contiene tutte le parole di tutte le recensioni (campo summary) relative ad un certo anno. Si itera quindi sulla lista inserendo le parole in una mappa (chiave = parola, valore = numero di occorrenze di quella parola). Se la parola è già contenuta nella mappa il suo valore viene incrementato. Dopo aver creato la mappa delle parole di un certo anno con il relativo numero di occorrenze viene creata una lista di oggetti TermFreq che rappresentano il termine con la sua frequenza. La collezione viene ordinata in base al numero di occorrenze di ogni parola. Successivamente se è più lunga di dieci elementi viene tagliata. Al termine della fase di Reduce viene ritornata come chiave l'anno passato dalla fase di Map e come valore la lista di parole ordinate in base alla loro frequenza.

PSEUDOCODICE FASE DI MAP:

```
String line = value.toString();
RFC4180Parser parser = new RFC4180Parser();
String[] fields = parser.parseLine(line);
if (fields.length == 10) {
    IntWritable year = time2Year(Time);
    StringTokenizer tokenizer =
        new StringTokenizer(Summary.toLowerCase());

    Text word = new Text();
    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken())
        context.write(year, word)
    }
}
```

PSEUDOCODICE FASE DI REDUCE:

```
for (Text value : values) {
    if(map.containsKey(value)) {
        map.put(value, freq + 1);
    } else
        map.put(value, 1);
}

List<TermFreq> l = new ArrayList<TermFreq>();
for(String s : map.keySet()) {
    l.add(new TermFreq(s, map.get(s)));
}

Collections.sort(l);

if(l.size()>10)
    context.write(key, l.subList(0, 10));
else
    context.write(key, l);
```

PSEUDOCODICE CLASSE TermFreq :

```
class TermFreq
    private Text term;
    private IntWritable frequency;
```

1.2. Hive

- Viene creata la tabella *amazonfoodreviews*
- Per la conversione dell'ora da Unix Time ad anno viene utilizzata la funzione *unix_year* che è importata tramite JAR.
- Viene creata una view *words* che contiene anno e parola, su cui è stato eseguito lo *split*, del campo *summary*.
- Successivamente viene eseguita la *SELECT* su anno, parola e contatore (numero di occorrenze di quella parola nell'anno selezionato). Per ottenere per ogni anno una lista di parole con numero di occorrenze viene utilizzata *collect_set* di Hive.
- Per selezionare solo le prime dieci parole più frequenti si fa utilizzo di una query annidata che ha lo scopo di contare le occorrenze di ogni parola per un certo anno in ordine decrescente e di selezionare solo i primi dieci risultati per ogni anno. Per quest'ultima operazione viene utilizzato *rank() over (partition by year order by COUNT(1) desc) as rank* dove viene assegnato ad ogni partizione (anno) un numero progressivo. Questo ha lo scopo di numerare per ogni anno le parole con la loro frequenza. Successivamente con l'operazione *WHERE rank < 11* vengono selezionate solo le prime dieci.

```

CREATE EXTERNAL TABLE amazonfoodreviews
(id INT, productId STRING, userId STRING, profileName STRING,
helpfulnessNumerator INT,
helpfulnessDenominator INT, score INT, time BIGINT, summary STRING,
text STRING) PARTITIONED BY (dt = ${INPUTDUE})
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES
    ("separatorChar" = ",",
    "quoteChar"      = "\"",
    "escapeChar"     = "\\0")

TBLPROPERTIES("skip.header.line.count"="1");

LOAD DATA LOCAL INPATH 'CSV PATH'

OVERWRITE INTO TABLE amazonfoodreviews;

add jar JAR PATH;

CREATE TEMPORARY FUNCTION unix_year AS 'unix2year.Unix2Year';

CREATE OR REPLACE VIEW words AS
SELECT unix_year(time) as year, exp.word
FROM amazonfoodreviews
LATERAL VIEW explode(split(lower(summary), ' ')) exp AS word;

INSERT OVERWRITE DIRECTORY '/user/hive/warehouse/job1Hive'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
SELECT year,
    collect_set(CONCAT(CONCAT("(",
        CONCAT_WS(" ", word, CAST(cnt AS STRING))), ")"))
FROM
    (SELECT year, word,
        COUNT(1) AS cnt,
        rank()
            over ( partition by year order by COUNT(1) desc) as rank
    FROM words
    GROUP BY year, word
    ORDER BY year, cnt DESC) tmp
WHERE rank < 11
GROUP BY year;

```

1.3. Spark

- Il metodo `loadData()` ha lo scopo di leggere il file CSV e di eseguire il parsing del file. Viene eseguito lo `split` per il campo `Summary`. È poi generato un `JavaPairRDD<Integer, List<String>>` che contiene l'anno e la lista di parole del campo `Summary` di una certa recensione (riga del file CSV).
- Successivamente viene eseguita una `groupByKey()` sul `JavaPairRDD` ritornato dal metodo `loadData()`. Si ottiene così per ogni anno una lista di liste di parole.
- Si itera sulla lista di liste di parole e si inserisce ogni parola in una mappa dove il valore è la parola e la chiave è il numero di occorrenze della parola.
- La mappa viene ordinata in base al valore decrescente delle sue chiavi.
- Se la mappa ha più di dieci elementi viene tagliata.
- Viene generata una `Tupla2` dove il primo campo è l'anno ed il secondo campo è la mappa dove ho come chiave la parola e come valore il numero di occorrenze della parola nel dato anno.
- Al termine viene eseguito una `sortByKey` per avere l'ordine sulla chiave anno.

PSEUDOCODICE SPARK:

loadData()

```
JavaRDD words = textFile(inputFilePath);

JavaPairRDD couple = words.mapToPair(row -> {

    ArrayList list = new ArrayList<>();
    StringTokenizer tokenizer = new StringTokenizer(Summary);

    while (tokenizer.hasMoreTokens()) {
        String word = tokenizer.nextToken();
        list.add(word);
    }

    return new Tuple2<>(Time, list);
});

return couple;
```

run()

```
JavaPairRDD couple = loadData(sc).groupByKey();

JavaPairRDD result = couple.mapToPair( x -> {
    LinkedHashMap map = new LinkedHashMap<>();
    Iterable<List<String>> it = x._2;
    for(List<String> l : it) {
        for(String s : l) {
            if (map.containsKey(s))
                map.put(s, freq+1);
            else
                map.put(s, 1);
        }
    }

    LinkedHashMap sortedMap = sortByValues(map);
    if(sortedMap.size()>10)
        LinkedHashMap sortedMapSeized = sortedMap.limit(10);
        return new Tuple2 (x._1(), sortedMapSeized);
    else
        return new Tuple2(x._1(), sortedMap);
});

result.sortByKey().saveAsTextFile(outputFolderPath);
```


1.4. Porzione di output. Job1 Map Reduce

1999	[a 3, tale 3, modern 3, day 3, fairy 3, is 2, educational 1, funny! 1, book 1, this 1]
2000	[a 14, master 6, is 5, version 5, great 5, - 4, tv 3, your 3, fanasy 3, research 3]
2001	[a 18, is 9, great 8, dvd 6, the 6, master 6, version 5, your 4, beetlejuice! 4, - 4]
2002	[a 38, the 21, great 21, is 18, beetlejuice! 13, this 10, it 9, of 9, - 8, to 7]
2003	[a 46, the 44, great 31, is 23, of 21, this 15, for 14, - 13, it 13, beetlejuice! 13]
2004	[the 160, a 95, best 76, great 66, is 61, for 57, good 47, of 47, and 41, i 40]
2005	[the 373, a 220, best 180, for 163, great 160, good 126, and 118, of 117, is 116, this 96]
2006	[the 1244, a 893, great 852, best 667, for 657, and 588, good 556, tea 423, is 386, not 375]

2. Secondo Job

Un job che sia in grado di generare, per ciascun prodotto, lo score medio ottenuto in ciascuno degli anni compresi tra il 2003 e il 2012, indicando ProductId seguito da tutti gli score medi ottenuti negli anni dell'intervallo. Il risultato deve essere ordinato in base al ProductId.

2.1. Map Reduce

- Nella fase di Map del Job2 per ogni riga del file CSV viene creata una coppia con ProductId come chiave e l'oggetto ScoreByYear come valore.
- L'oggetto ScoreByYear contiene lo score e l'anno.
- Dopo lo shuffle and sort si ha per ogni productId una lista di oggetti ScoreByYear.
- Nella fase di Reduce per ogni ProductId viene calcolata la media degli score per gli anni compresi tra il 2003 e il 2012.
- Il risultato è ordinato in base al ProductId perché è stato scelto come chiave.

PSEUDOCODICE FASE DI MAP:

```
String line = value.toString();
RFC4180Parser parser = new RFC4180Parser();
String[] fields = parser.parseLine(line);

if (fields.length == 10) {
    IntWritable year = Time;
    IntWritable score = Score;

    ScoreByYear s = new ScoreByYear(score, year);

    context.write(ProductId, s);
}
```

PSEUDOCODICE FASE DI REDUCE:

```
HashMap map = new HashMap<Integer, Double>();
ArrayList score = new ArrayList<ScoreByYear>();

for (ScoreByYear s : values) {
    ScoreByYear tmp = new ScoreByYear(s.getScore(), s.getYear());
    score.add(tmp);
}

for(int i = StartYear; i<=FinalYear; i++) {

    for (ScoreByYear s : score) {
        if (s.getYear()==i) {
            somma = somma + s.getScore();
            num++;
        }
    }

    if(num!=0) {
        media = MathsUtils.round(somma/num, 2);
    }

    map.put(i, media);
}

context.write(key, map);
```

2.2. Hive

- Nella SELECT viene selezionato il productId, l'anno e la media degli score.
- Nella WHERE viene indicato il periodo per cui si vuole calcolare la media per ogni anno.
- Nella GROUP BY viene indicato il productId e l'anno. Questo serve alla funzione che calcola la media avg(score) di calcolare la media degli score dei prodotti per ogni anno.
- Al termine della query viene eseguito l'ordinamento in base al productId.

```
CREATE TABLE IF NOT EXISTS amazonfoodreviews
(id INT, productId STRING, userId STRING, profileName STRING,
helpfulnessNumerator INT,
helpfulnessDenominator INT, score INT, time BIGINT, summary STRING, text
STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "\"",
"escapeChar" = "\\0")
TBLPROPERTIES("skip.header.line.count"="1");
```

```
LOAD DATA LOCAL INPATH 'CSV PATH'
OVERWRITE INTO TABLE amazonfoodreviews;
```

```
add jar JAR PATH;
```

```
CREATE TEMPORARY FUNCTION unix_year AS 'unix2year.Unix2Year';
```

```
INSERT OVERWRITE DIRECTORY '/user/hive/warehouse/job2Hive'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
SELECT tmp.productId, collect_list(tmp.year2score) FROM
  (SELECT productId, CONCAT(CONCAT("(", CONCAT_WS(", ",
unix_year(time), cast(avg(score) as STRING))), ")") AS year2score
  FROM amazonfoodreviews
  WHERE unix_year(time)>2002 AND unix_year(time)<2013
  GROUP BY productId, unix_year(time)
  ORDER BY productId, year2score) tmp
GROUP BY productId
ORDER BY productId;
```

2.3. Spark

- Nel metodo loadData() viene caricato il file CSV.
- Viene creato un JavaPairRDD di tipo String, Tuple2<Integer, Integer> dove è memorizzato il ProductId e lo score composto dai campi Time (anno) e Score (valutazione prodotto).
- Viene eseguito sul JavaPairRDD una groupByKey() così da ottenere una lista di Tuple2<Integer, Integer> per ogni ProductId.
- Viene poi creato un JavaPairRDD<String, LinkedHashMap<Integer, Double>> dove è memorizzato per ogni ProductId una mappa che contiene anno e media delle recensioni per l'anno corrispondente.
- Il risultato viene poi ordinato in base alla chiave ProductId.

loadData()

```
JavaRDD<String> words = textFile(inputFilePath);
JavaPairRDD<String, Tuple2<Integer, Integer>> data =
    words.mapToPair(row -> {
        RFC4180Parser parser = new RFC4180Parser();
        String[] values = parser.parseLine(row);

        return new Tuple2(ProductId,
                           new Tuple2(Time, Score));
    });

return data;
```

```

run()
JavaPairRDD data = loadData().groupByKey();
JavaPairRDD result = data.mapToPair( x -> {
    LinkedHashMap<Integer, Double> map = new LinkedHashMap();
    Iterable<Tuple2<Integer, Integer>> iterable = x._2();
    for(int i = StartYear; i<=FinalYear; i++) {
        for (Tuple2<Integer, Integer> t : iterable) {
            if (t._1()==i) {
                somma = somma + t._2();
                num++;
            }
        }

        if(num!=0)
            media = MathsUtils.round(somma/num, 2);

        map.put(new Integer(i), new Double(media));

    }

    return (x._1(), map);
});

result.sortByKey().saveAsTextFile(outputFolderPath);

```

2.4. Porzione di output. Job2 Map Reduce

0006641040	{2003=5.0, 2004=4.33, 2005=3.25, 2006=0.0, 2007=4.5, 2008=4.0, 2009=5.0, 2010=5.0, 2011=4.17,
2012=4.0}	
141278509X	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=0.0, 2011=0.0, 2012=5.0}
2734888454	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=3.5, 2008=0.0, 2009=0.0, 2010=0.0, 2011=0.0, 2012=0.0}
2841233731	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=0.0, 2011=0.0, 2012=5.0}
7310172001	{2003=0.0, 2004=0.0, 2005=3.5, 2006=5.0, 2007=4.91, 2008=4.55, 2009=4.73, 2010=4.77, 2011=4.78,
2012=4.79}	
7310172101	{2003=0.0, 2004=0.0, 2005=3.5, 2006=5.0, 2007=4.91, 2008=4.55, 2009=4.73, 2010=4.77, 2011=4.78,
2012=4.79}	
7800648702	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=0.0, 2011=0.0, 2012=4.0}
9376674501	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=0.0, 2011=5.0, 2012=0.0}
B00002N8SM	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=2.0, 2008=1.0, 2009=2.5, 2010=1.25, 2011=2.25,
2012=1.56}	
B00002NCJC	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=4.5, 2011=0.0, 2012=0.0}
B00002Z754	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=0.0, 2008=0.0, 2009=0.0, 2010=0.0, 2011=0.0, 2012=0.0}
B00004CI84	{2003=3.71, 2004=4.69, 2005=4.2, 2006=4.25, 2007=4.62, 2008=4.12, 2009=4.77, 2010=4.86, 2011=4.91,
2012=4.5}	
B00004CXX9	{2003=3.71, 2004=4.67, 2005=4.0, 2006=4.25, 2007=4.62, 2008=4.11, 2009=4.67, 2010=4.6, 2011=4.77,
2012=4.38}	
B00004RAMS	{2003=0.0, 2004=0.0, 2005=0.0, 2006=1.0, 2007=1.5, 2008=4.0, 2009=5.0, 2010=3.75, 2011=4.17,
2012=3.67}	
B00004RAMV	{2003=0.0, 2004=0.0, 2005=0.0, 2006=0.0, 2007=4.0, 2008=0.0, 2009=2.5, 2010=0.0, 2011=2.0, 2012=1.0}

3. Terzo Job

Un job in grado di generare coppie di prodotti che hanno almeno un utente in comune, ovvero che sono stati recensiti da uno stesso utente, indicando, per ciascuna coppia, il numero di utenti in comune. Il risultato deve essere ordinato in base allo ProductId del primo elemento della coppia e, possibilmente, non deve presentare duplicati.

3.1. Map Reduce

Il terzo Job è stato realizzato con un Map Reduce a due fasi.

- Nella prima fase della Map viene selezionato da ogni riga del file CSV l'utente (chiave) e il prodotto recensito (valore).
- Dopo lo shuffle and sort si ottiene per ogni utente la lista dei prodotti che ha recensito.
- Nella prima fase di Reduce per ogni coppia di prodotti della lista viene generata una coppia: [productId 1, productId 2] chiave e userId valore
- Alla fine della fase di Reduce si ottiene una lista di coppie di prodotti recensite da uno stesso utente con associato l'utente che le ha recensite.
- Nella seconda fase del MapReduce è necessario aggregare le coppie di prodotti e contare gli utenti univoci che le hanno recensite.
- Nella seconda fase di Map viene solamente riportato il risultato della fase precedente.
- Dopo lo shuffle and sort della seconda fase si ha per ogni coppia di prodotti una lista di userId che hanno recensito entrambi i prodotti.
- Nella seconda fase di Reduce la lista di utenti per una coppia di prodotti viene inserita in un HashSet così da poter eliminare i duplicati.
- Viene calcolata la dimensione del set e viene associata alla coppia di prodotti.

- Al termine della seconda fase di reduce si ha per ogni coppia di prodotti il numero di utenti che li ha recensiti entrambi. L'ordinamento è in base al productId del primo elemento perché la coppia di prodotti è la chiave.

PSEUDOCODICE FASE DI MAP 1:

```
String line = value.toString();
RFC4180Parser parser = new RFC4180Parser();
String[] fields = parser.parseLine(line);

if (fields.length == 10) {
    String user = UserId;
    String productId = ProductId;

    context.write(new Text(user), new Text(productId));
}
```

PSEUDOCODICE FASE DI REDUCE 1:

```
reduce(Text key, ... .. ){

    HashSet<String> set = new HashSet<String>();

    for(Text t : values) {
        set.add(t.toString());
    }

    ArrayList<String> list = new ArrayList<String>(set);

    for(int i=0; i<list.size()-1; i++) {
        for(int j=i+1; j<list.size(); j++) {
            context.write(list.get(i) " " list.get(j)), key);
        }
    }
}
```

PSEUDOCODICE FASE DI MAP 2:

```
map(LongWritable key, Text value, ... .. ){  
  
    String[] parts = value.toString().split("\t");  
    String couple = parts[0];  
    String userId = parts[1];  
  
    context.write(new Text(couple), new Text(userId));  
}
```

PSEUDOCODICE FASE DI REDUCE 2:

```
reduce(Text key, Iterable<Text> values, ... .. ) {  
    HashSet<Text> set = new HashSet<Text>();  
    for(Text t : values) {  
        set.add(t);  
    }  
  
    context.write(key, new IntWritable(set.size()));  
}
```

3.2. Hive

In Hive viene utilizzato l'operatore di Join per calcolare le coppie di prodotti recensite dallo stesso utente e per calcolare il numero di utenti che hanno recensito entrambi i prodotti.

- Nella SELECT viene selezionata la coppia di prodotti (item1 e item2) e viene calcolato il numero di utenti che hanno recensito la coppia.
- Nella FROM viene fatto il JOIN tra t1 e t2 in base allo stesso userId.
- Viene eseguita la GROUP BY su i due prodotti.
- Per evitare dopo l'operazione di JOIN coppie di prodotti uguali viene inserito HAVING t1.productId != t2.productId.
- Infine viene ordinato il risultato in base al primo productId.


```

CREATE TABLE IF NOT EXISTS amazonfoodreviews
(id INT, productId STRING, userId STRING, profileName STRING,
helpfulnessNumerator INT,
helpfulnessDenominator INT, score INT, time BIGINT, summary STRING, text
STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES ("separatorChar" = ",", "quoteChar" = "\"",
"escapeChar" = "\\")
TBLPROPERTIES("skip.header.line.count"="1");

LOAD DATA LOCAL INPATH "CSV PATH"
OVERWRITE INTO TABLE amazonfoodreviews;

INSERT OVERWRITE DIRECTORY '/user/hive/warehouse/job3Hive'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
SELECT
t1.productId AS item1, t2.productId AS item2, COUNT(1) AS cnt
FROM
(SELECT DISTINCT productId, userId FROM amazonfoodreviews) t1
JOIN
(SELECT DISTINCT productId, userId FROM amazonfoodreviews) t2
ON (t1.userId = t2.userId)
GROUP BY t1.productId, t2.productId
HAVING t1.productId != t2.productId
ORDER BY item1 ASC;

```

3.3. Spark

In Spark il Job 3 è stato realizzato in modo simile alla versione in Hive.

- La funzione `loadData()` carica il CSV e crea le coppie `userId` (chiave), `productId` (valore).
- Successivamente sul `JavaPairRDD` ritornato da `loadData()` viene eseguito il Join con se stesso: `data.join(data)` e si ottiene un `JavaPairRDD<String, Tuple2<String, String>>` dove il primo valore è la `userId` e la coppia di valori di `Tuple2` corrisponde alla coppia di `productId` recensiti dall'utente.

- Viene creato un nuovo JavaPairRDD <Tuple2<String, String>, String> con i valori precedenti ma invertiti. (Prima la coppia di prodotti e poi l'utente)
- Su quest'ultimo RDD viene eseguita l'operazione groupByKey() così da ottenere per ogni coppia di prodotti una lista di userId.
- La lista di utenti viene inserita in un HashSet per evitare duplicati.
- Al termine viene creato un JavaPairRDD<Tuple2<String, String>, Integer> dove la Tuple2 rappresenta la coppia di prodotti e il secondo valore dell'RDD rappresenta le dimensioni del set ovvero il numero di utenti che hanno recensito la coppia di prodotti.
- Il risultato è ordinato in base al primo prodotto. Ciò avviene attraverso sortByKey(new TupleComparator(), true). TupleComparator() è un comparatore di Tuple2 in base al primo valore della tupla.

loadData()

```

JavaRDD<String> words = textFile(inputFilePath);

JavaPairRDD data = words
    .mapToPair(row -> {
        RFC4180Parser parser = new RFC4180Parser();
        String[] values = parser.parseLine(row);

        return new Tuple2(UserId, ProductId);
    });
return data;
```

```

run()
JavaPairRDD data = loadData();
JavaPairRDD joinedData = data.join(data);
JavaPairRDD<Tuple2<String, String>, String> prod2user =
    joinedData.mapToPair(x -> {
        return new Tuple2(x._2, x._1);
    });
JavaPairRDD list = prod2user.groupByKey();
JavaPairRDD result = list.mapToPair(x -> {
    HashSet<String> users = new HashSet<>();
    x._2.forEach(y -> { users.add(y);});
    Integer i = new Integer(users.size());
    return new Tuple2<>(x._1, i);
});

result.sortByKey(new TupleComparator(),true)
    .saveAsTextFile(outputFolderPath);

```

3.4. Output Parziale. Job3 Map Reduce (1)

[B005ZBZLT4 , B007Y59HVM]	#oc-R115TNMSPFT9I7
[B005HG9ERW , B005HG9ESG]	#oc-R11D9D7SHXIJB9
[B005HG9ERW , B005HG9ET0]	#oc-R11D9D7SHXIJB9
[B005HG9ESG , B005HG9ET0]	#oc-R11D9D7SHXIJB9
[B005ZBZLT4 , B007Y59HVM]	#oc-R11DNU2NBKQ23Z
[B005HG9ERW , B005HG9ESG]	#oc-R11O5J5ZVQE25C
[B005HG9ERW , B005HG9ET0]	#oc-R11O5J5ZVQE25C
[B005HG9ESG , B005HG9ET0]	#oc-R11O5J5ZVQE25C
[B007OSBE1U , B007OSBEV0]	#oc-R12KPBODL2B5ZD

Output Parziale. Job3 Map Reduce (2)

[0006641040 , B0005XN9HI]	1
[0006641040 , B00061EPKE]	1
[0006641040 , B000EM00YU]	1
[0006641040 , B000FDQV46]	1
[0006641040 , B000FV8LPU]	1
[0006641040 , B000MGOZEO]	1
[0006641040 , B000MLHU3M]	1
[0006641040 , B000UVZRES]	1
[0006641040 , B000UW1Q8I]	1
[0006641040 , B000UXH9X8]	1
[0006641040 , B000UXW95G]	1
[0006641040 , B0013P3KC6]	1

4. Esecuzione Job su Amazon EMR

Il codice è stato eseguito in locale e su Amazon EMR.

Su Amazon EMR è stato eseguito attraverso gli step che consentono di eseguire codice con un certo input. Sono stati quindi eseguiti gli step per ogni Job, per ogni tecnologia e per i cinque file CSV.


La creazione di uno step prevede di caricare su S3 il file JAR con le sue dipendenze ed i dati di input. L'output viene salvato su S3.

Step per la creazione del Job3 in Map Reduce

Add step ✕


Step type Custom JAR

Name*

JAR location*  JAR location maybe a path into S3 or a fully qualified java class in the classpath.

Arguments

job3.Job3 s3://progettobigdatanr/input/uno.csv
s3://progettobigdatanr/output/tmpJob3Csv1
s3://progettobigdatanr/output/resultJob3Csv1

 These are passed to the main function in the JAR. If the JAR does not specify a main class in its manifest file you can specify another class name as the first argument.

Action on failure What to do if the step fails.

Cancel Save

Step per la creazione del Job3 in Spark

Add step ✕


Step type Spark application Run Spark application using spark-submit. [Learn more](#)


Name

Deploy mode Run your driver on a slave node (cluster mode) or on the master node as an external client (client mode).

Spark-submit options


--class job1.Job1Spark

 Specify other options for spark-submit.

Application location*  Path to a JAR with your application and dependencies (client deploy mode only supports a local path).

Arguments

s3://progettobigdatanr/input/uno.csv
s3://progettobigdatanr/output/Job1Csv1Spark

 Specify optional arguments for your application.

Action on failure What to do if the step fails.

Cancel Save

Add steps (optional) ⓘ

Name	Action on failure	JAR location	Arguments
Job1 Csv1	Continue	s3://progettobigdatanr/jar/ProgettobigData.jar	job1.Job1 s3://progettobigdatanr/input/un o.csv s3://progettobigdatanr/output/re sultJob1Csv1
Job1 Csv2	Continue	s3://progettobigdatanr/jar/ProgettobigData.jar	job1.Job1 s3://progettobigdatanr/input/du e.csv s3://progettobigdatanr/output/re sultJob1Csv2
Job1 Csv3	Continue	s3://progettobigdatanr/jar/ProgettobigData.jar	job1.Job1 s3://progettobigdatanr/input/tre. csv s3://progettobigdatanr/output/re sultJob1Csv3

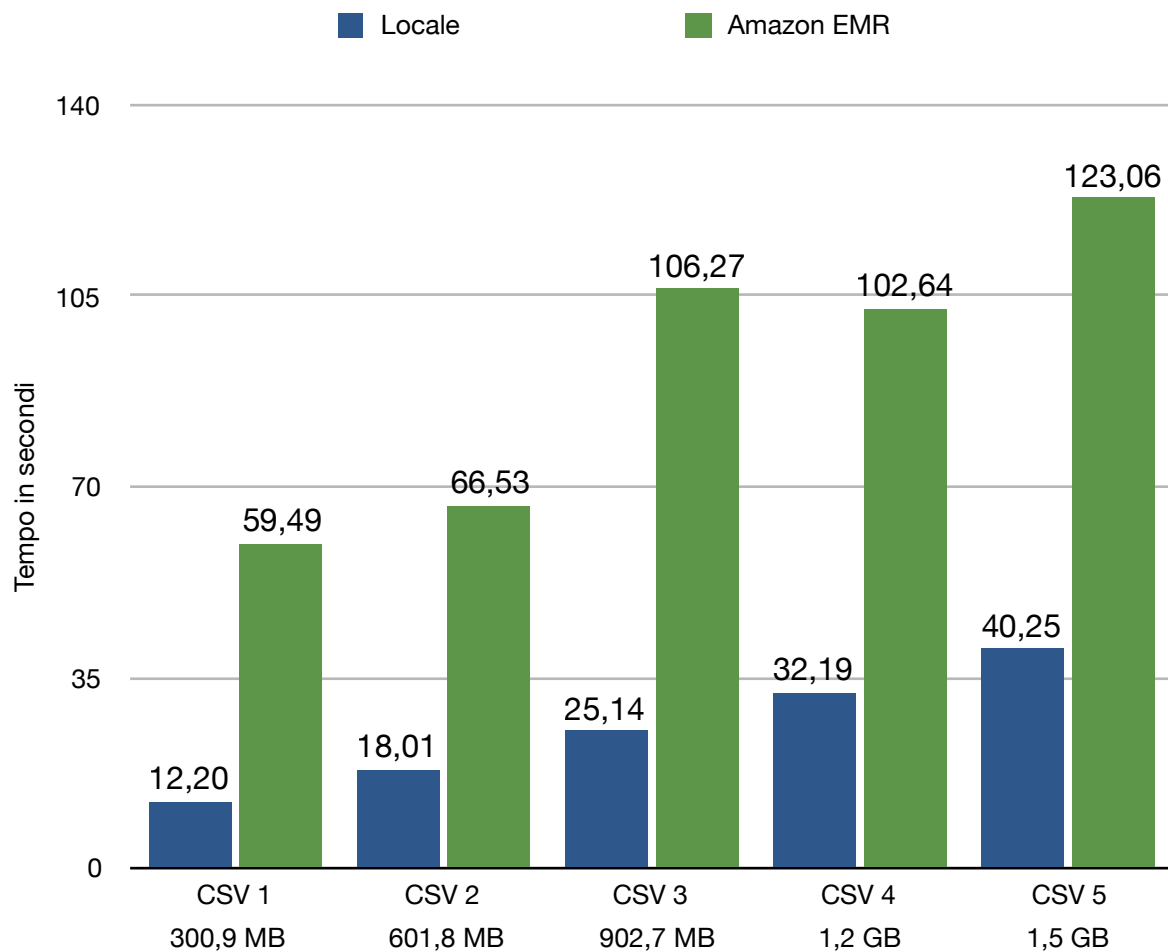
Al termine dell'esecuzione di un dato step è possibile attraverso i log scaricare il risultato dell'esecuzione. Il tempo di esecuzione che viene mostrato in stdout.

		s-1BAFNAAYQ38H9	Job3 Csv3	Pending			View logs
		s-GSEO3LF203K7	Job3 Csv2	Pending			View logs
		s-RHYH0FEG309K	Job3 Csv1	Pending			View logs
		s-W50K7TSBSV7P	Job2 Csv5	Pending			View logs
		s-MZ9YBI8LTXR	Job2 Csv4	Pending			View logs
		s-1LF84AK6MX3MN	Job2 Csv3	Pending			View logs
		s-2GA3X3LF40YW9	Job2 Csv2	Pending			View logs
		s-2XRMG8ZBCQ04I	Job2 Csv1	Pending			View logs
		s-2YBHRHVYYI94	Job1 Csv5	Pending			View logs
		s-2M66RX0F3HUPV	Job1 Csv4	Running	2018-05-24 13:39 (UTC+2)	1 minute	View logs
		s-ANOVK58VO7SX	Job1 Csv3	Completed	2018-05-24 13:37 (UTC+2)	1 minute	View logs
		s-OJHYNEKM5YQ	Job1 Csv2	Completed	2018-05-24 13:35 (UTC+2)	1 minute	View logs
		s-3SQZK3H1POV2X	Job1 Csv1	Completed	2018-05-24 13:34 (UTC+2)	1 minute	View logs
		s-3C9TA1INGWQ6C	Setup hadoop debugging	Completed	2018-05-24 13:34 (UTC+2)	2 seconds	View logs

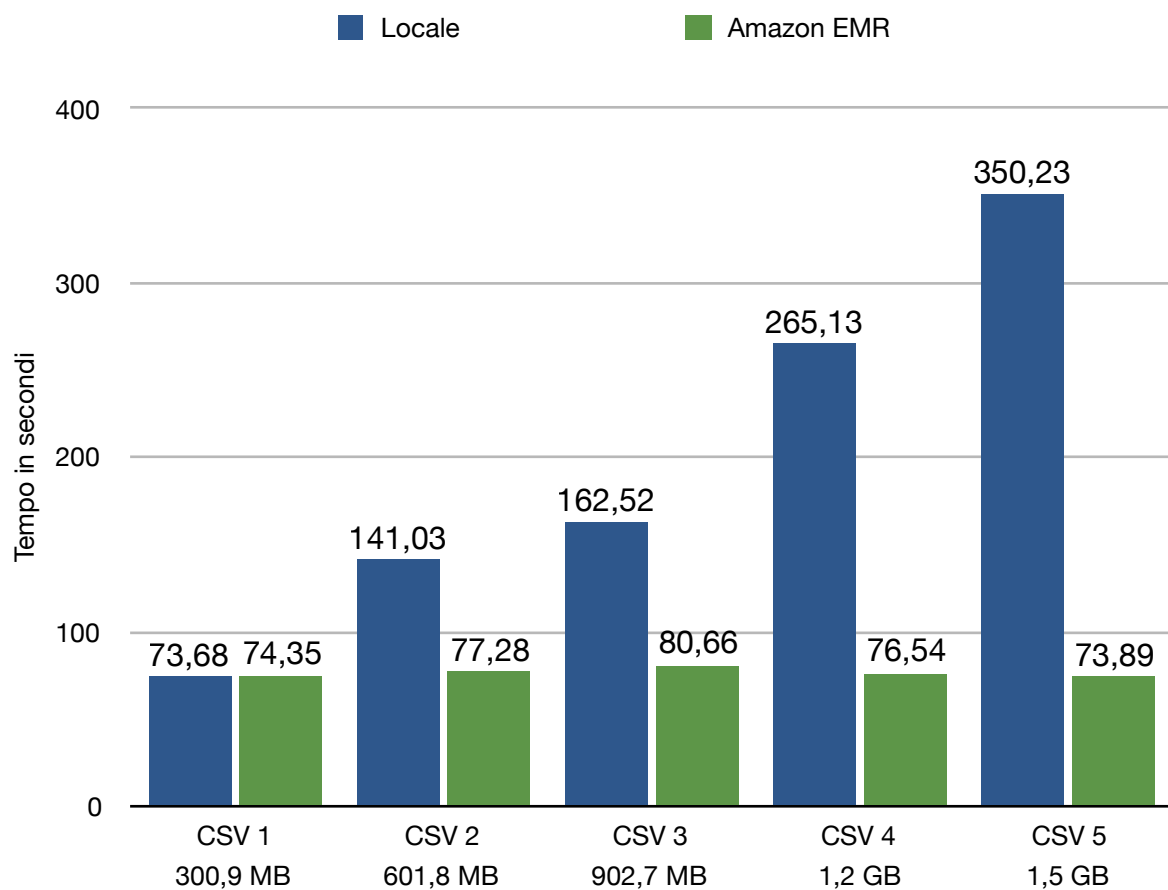
5. Tempi di esecuzione

Vengono ora riportati per ogni Job e per ogni tecnologia i tempi di esecuzione al variare delle dimensioni del file CSV confrontando l'esecuzione locale con quella su Amazon EMR.

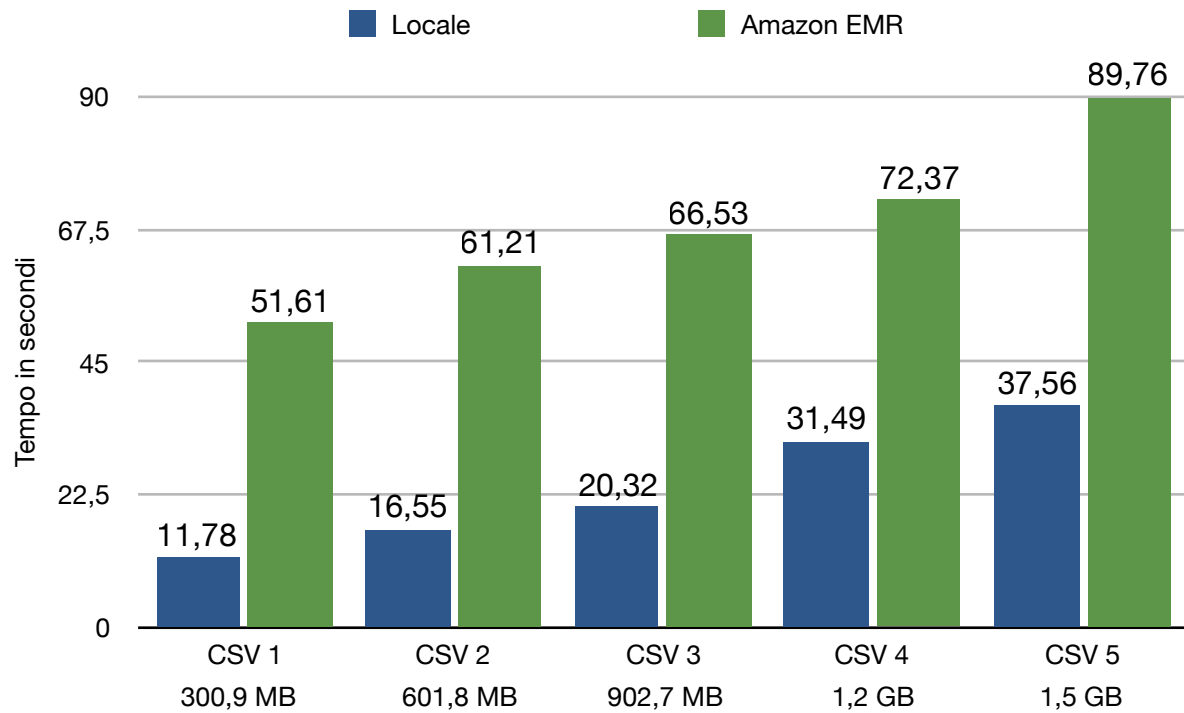
Tempi di Esecuzione MapReduce JOB 1 - Locale vs Amazon EMR



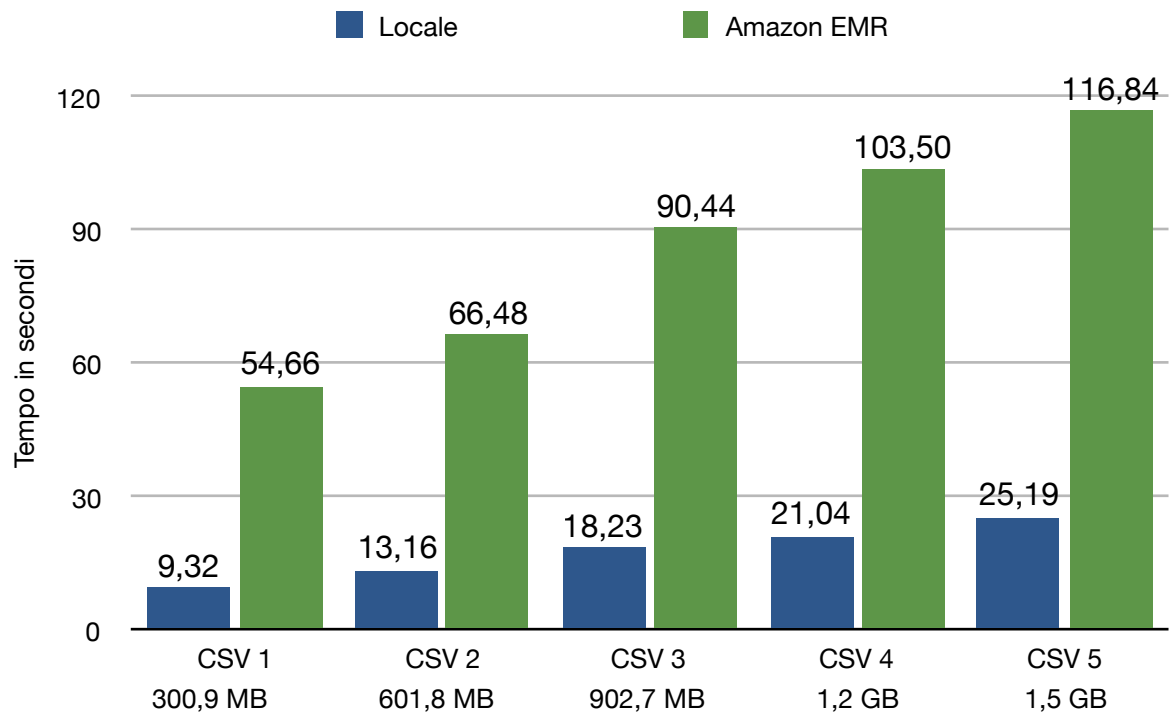
Tempi di Esecuzione Hive JOB 1 - Locale vs Amazon EMR



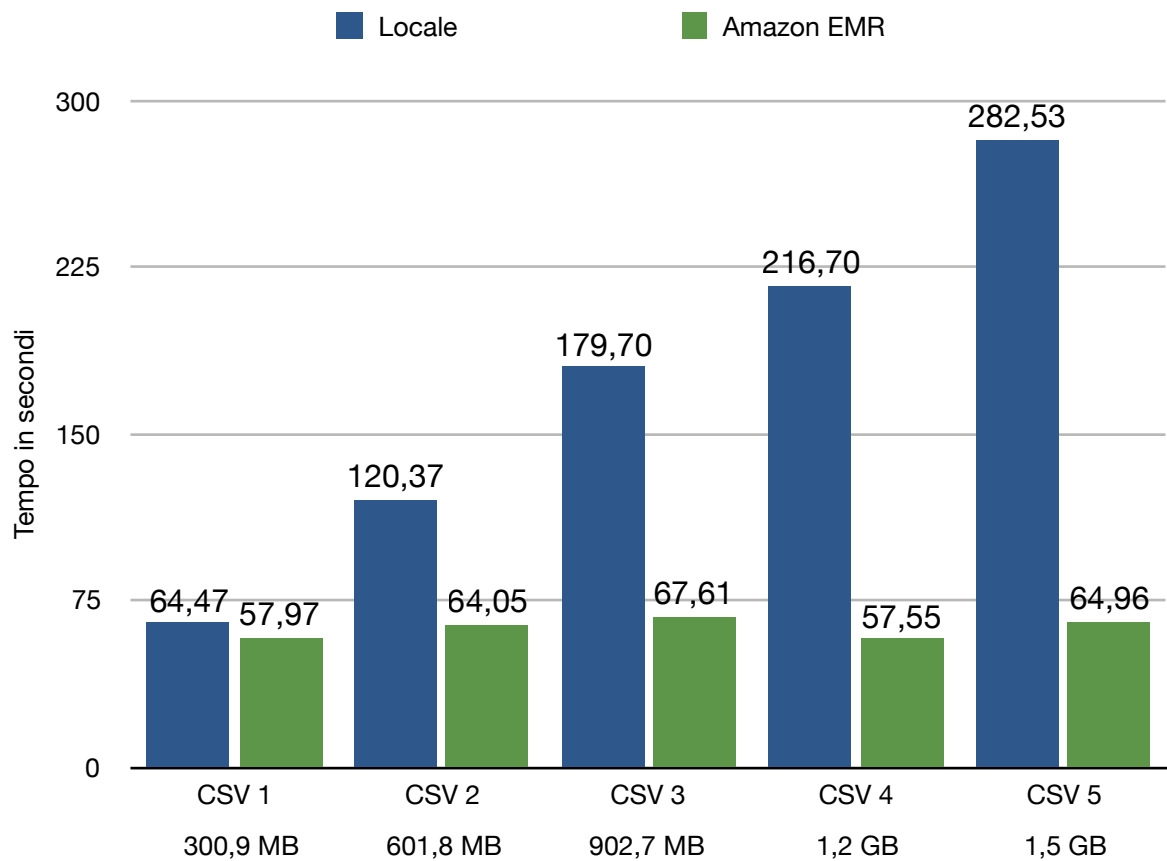
Tempi di Esecuzione Spark JOB 1 - Locale vs Amazon EMR



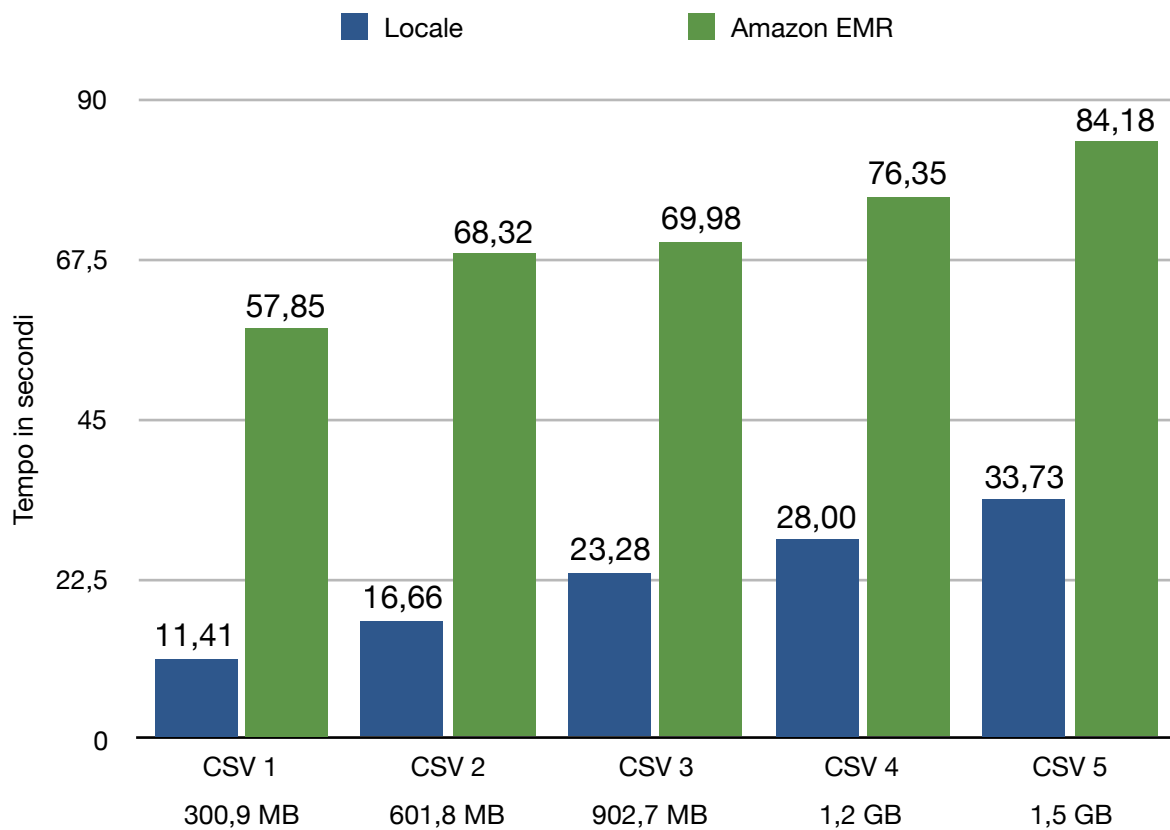
Tempi di Esecuzione MapReduce JOB 2 - Locale vs Amazon EMR



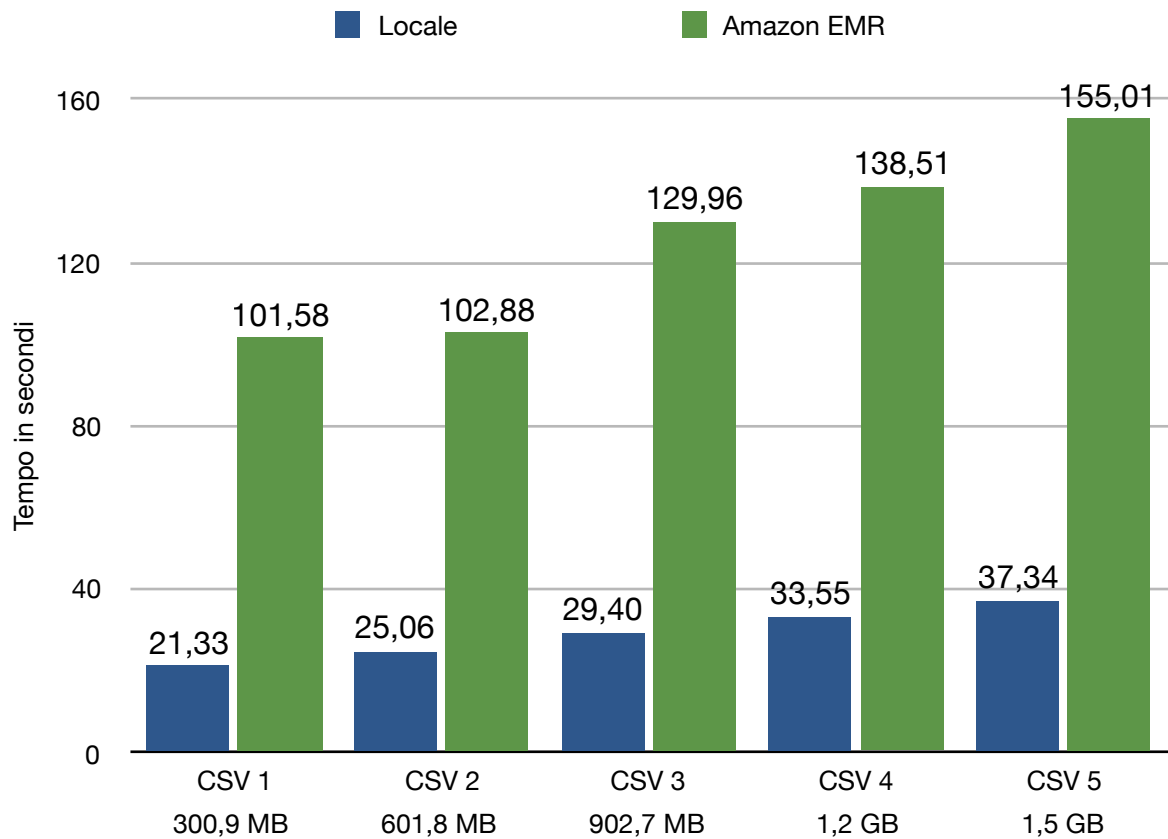
Tempi di Esecuzione Hive JOB 2 - Locale vs Amazon EMR



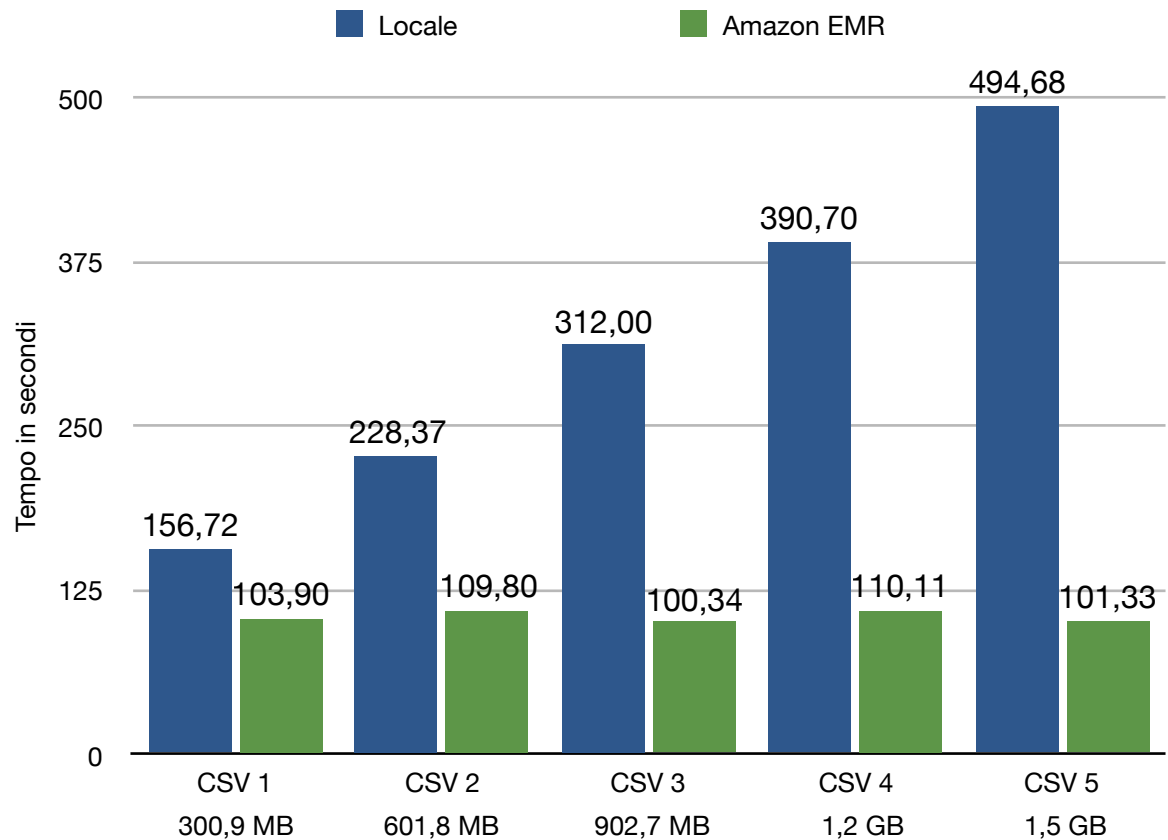
Tempi di Esecuzione Spark JOB 2 - Locale vs Amazon EMR



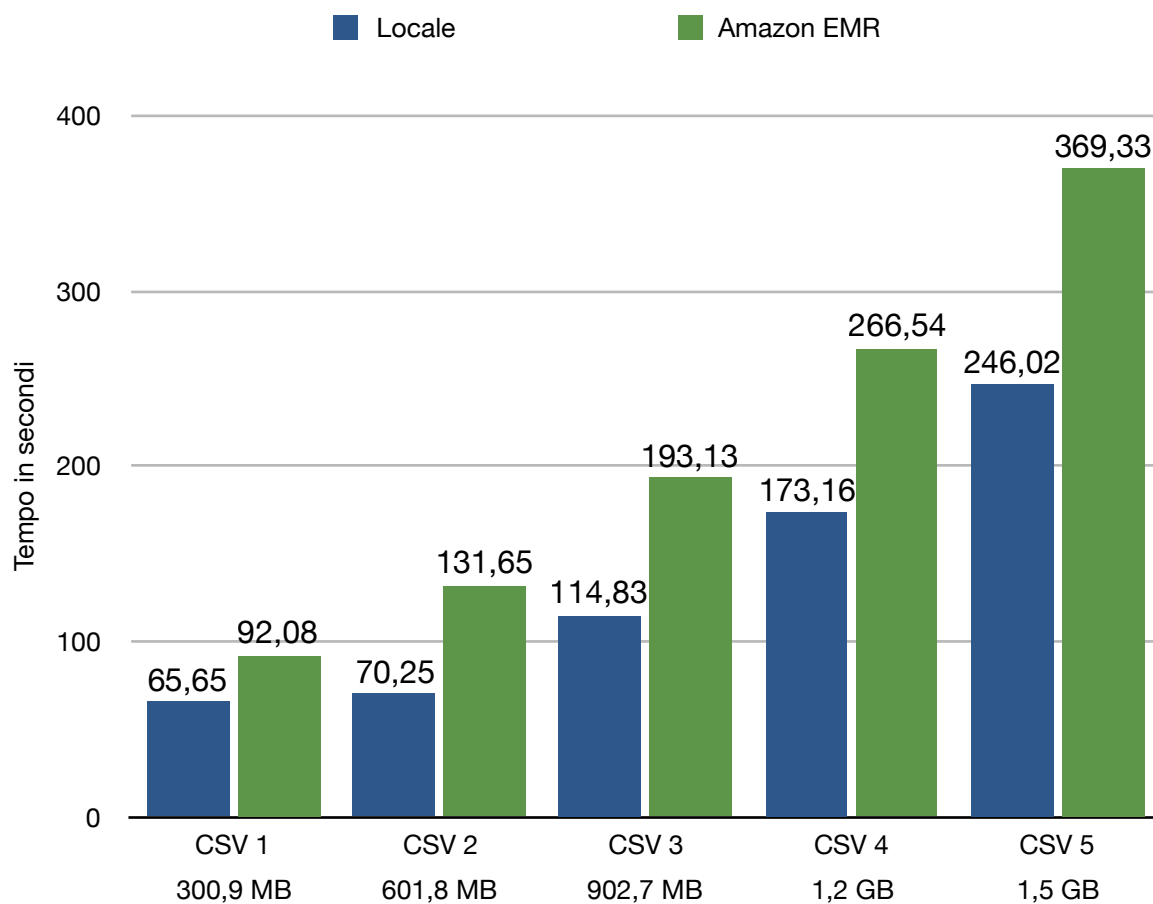
Tempi di Esecuzione MapReduce JOB 3 - Locale vs Amazon EMR



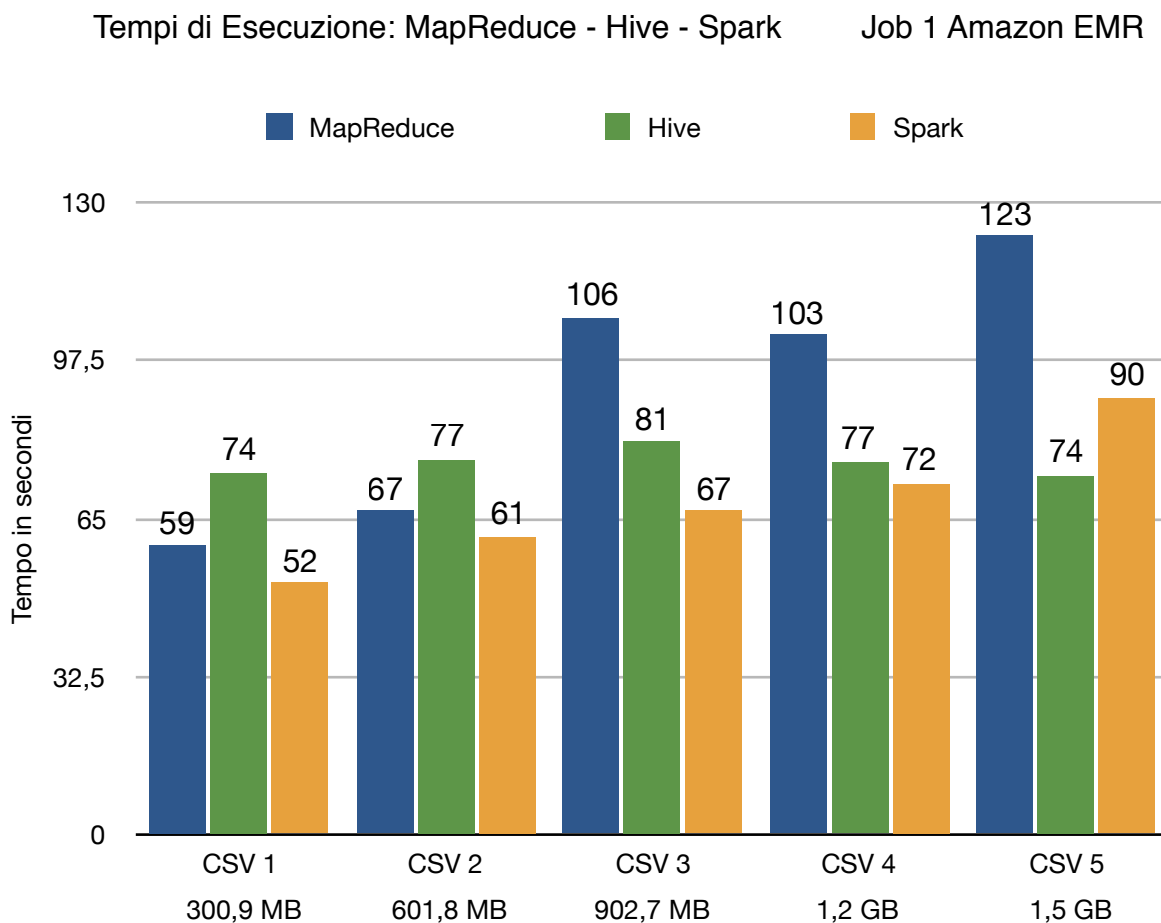
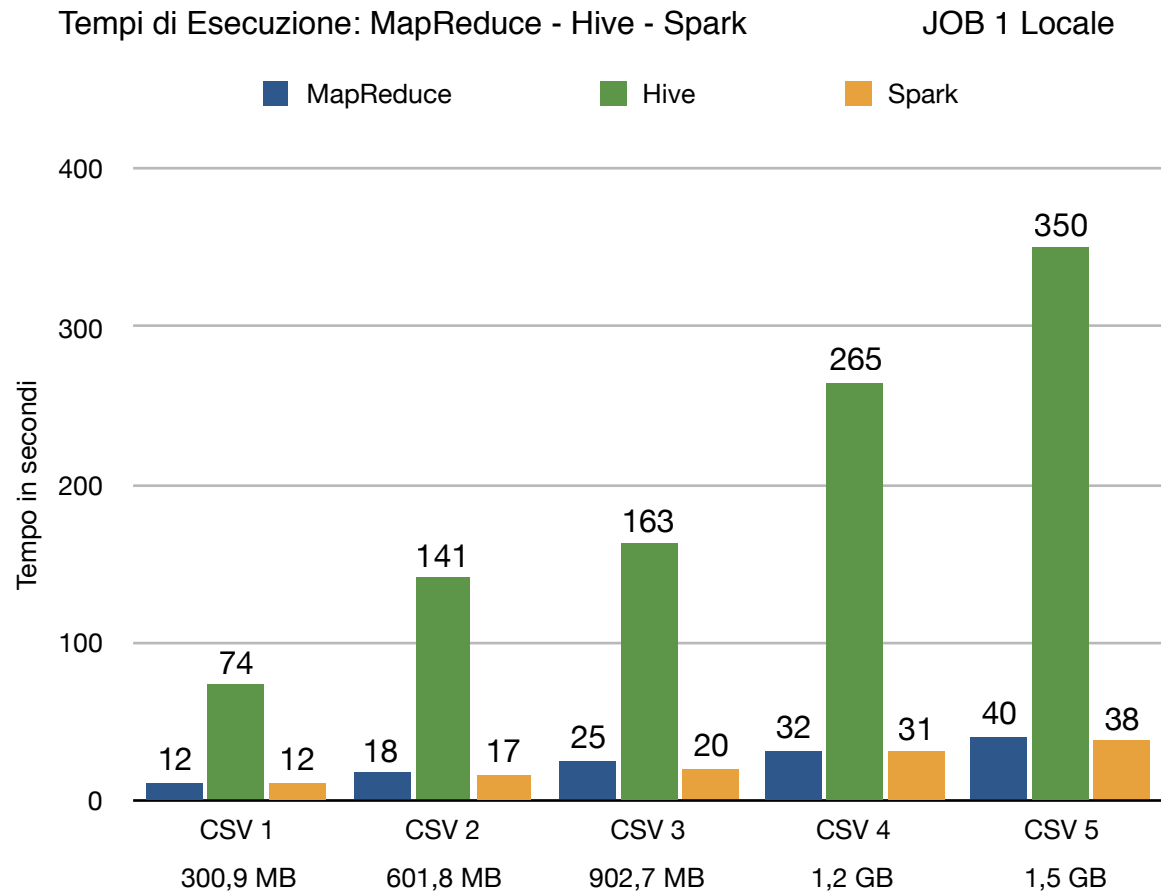
Tempi di Esecuzione Hive JOB 3 - Locale vs Amazon EMR

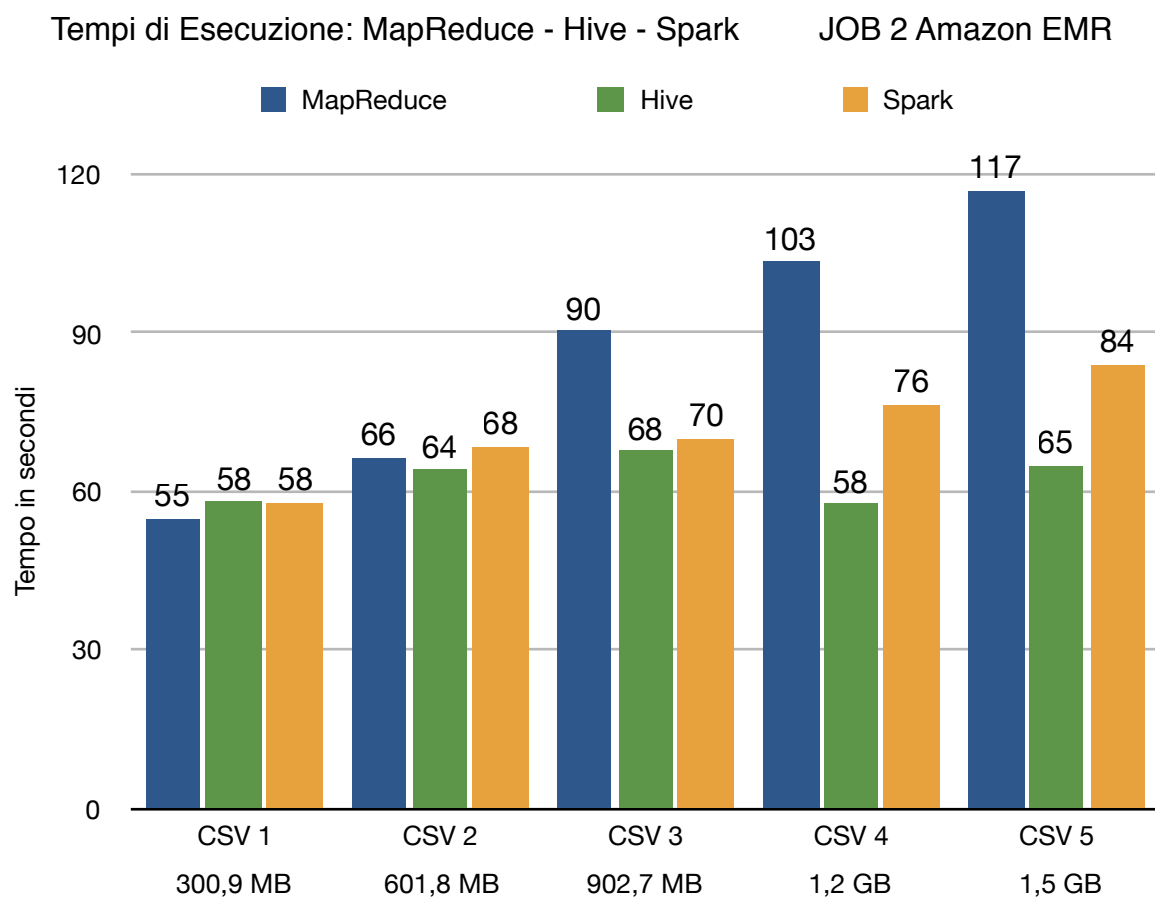
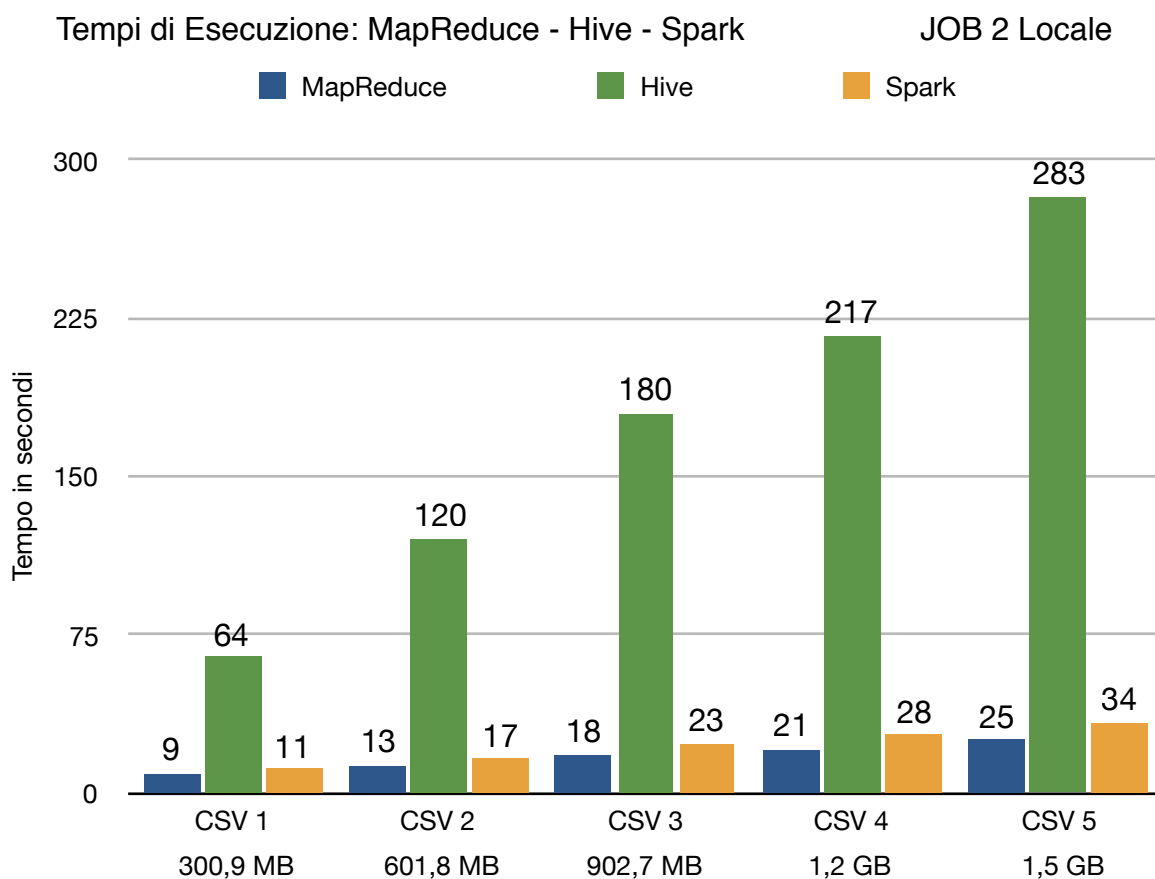


Tempi di Esecuzione Spark JOB 3 - Locale vs Amazon EMR



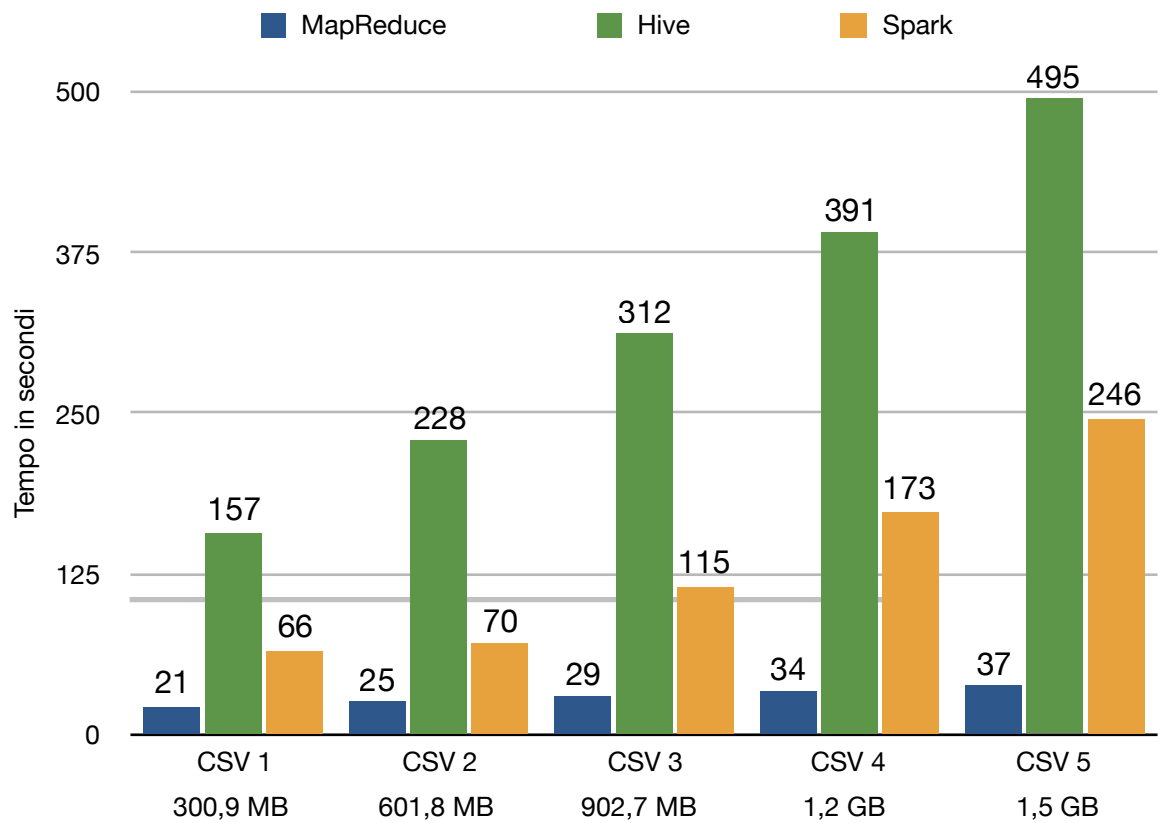
Vengono ora confrontate le tecnologie per ogni Job sia in locale che su Amazon EMR.





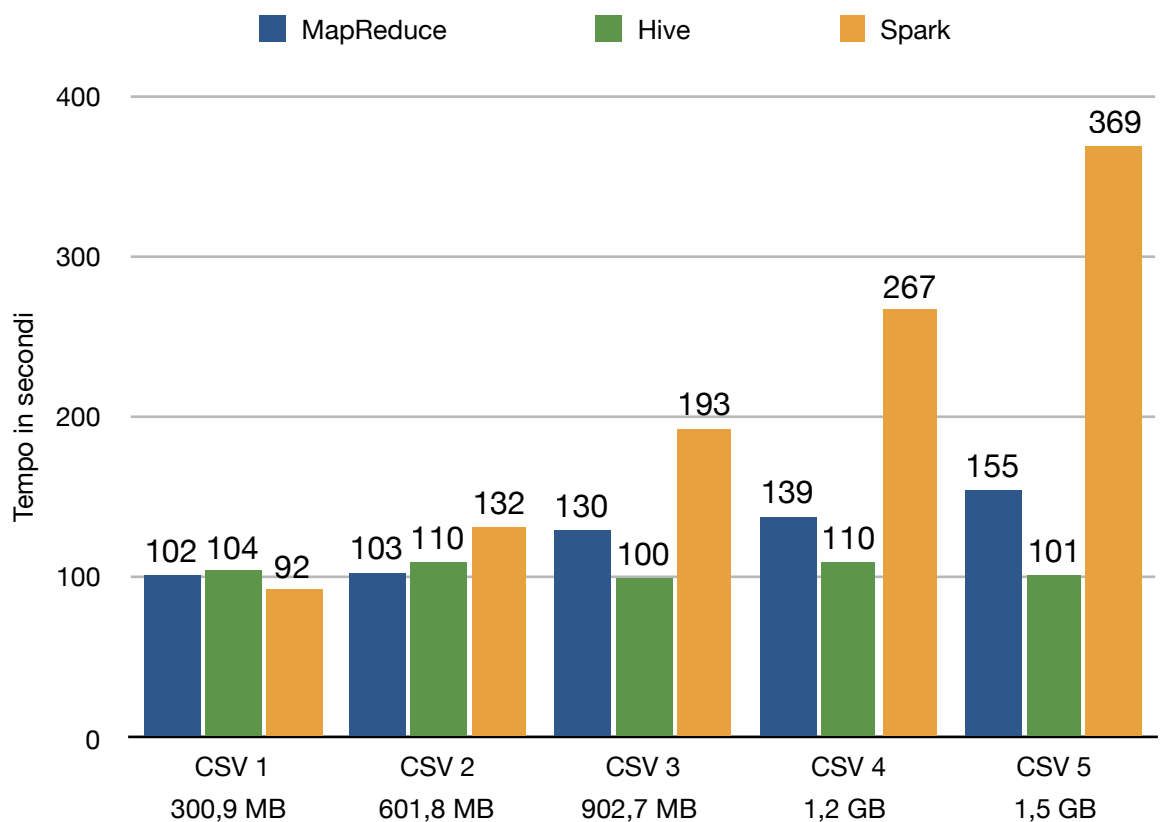
Tempi di Esecuzione: MapReduce - Hive - Spark

JOB 3 Locale



Tempi di Esecuzione: MapReduce - Hive - Spark

JOB 3 Amazon EMR



6. Conclusioni

Dopo aver eseguito i test sia in locale che su Amazon EMR ho notato che in locale la tecnologia con prestazioni più basse è Hive. Sia MapReduce che Spark hanno risultati simili sia nel Job1 che nel Job2. Nel Job3 invece la tecnologia con prestazioni migliori è MapReduce. Spark nel Job3 è decisamente più lento rispetto a MapReduce probabilmente a causa della sua implementazione con il Join. Su cluster Amazon EMR la tecnologia Hive è decisamente più performante ed ho notato che al variare dell'input il tempo di computazioni rimane pressoché invariato. Il Job 1 in Hive su Amazon EMR ha un tempo tra i 74 e gli 81 secondi al variare dell'input (file 300 MB - 1,5 GB). In locale MapReduce e Spark sono stati più performanti rispetto ad Amazon EMR ciò probabilmente è dovuto alle dimensioni del dataset che si ferma ad 1,5 GB. Con file più grandi le performance del cluster Amazon dovrebbero essere più apprezzabili.