2018

BIG DATA - Primo Progetto

Gruppo: AdR&SS

Angelo del Re 441476
Stefano Silvi 461993

Codice del progetto disponibile su GitHub:
https://github.com/mrsambo93/BigData2018.git

Job 1

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.

Soluzione

MapReduce

Nel Mapper viene fatto il parsing del CSV, dal quale ricaviamo i campi Time e Summary. Il Time viene formattato, in modo tale da ottenerne l'anno. Dal Summary rimuoviamo tutti caratteri speciali, dopo viene separato il campo in parole singole. Il Mapper, alla fine, restituirà delle coppie anno-parola. il Reducer ha in input l'anno e tutte le parole inerenti a quell'anno. Da questa lista di parole viene creata una mappa con chiave-parola e valore-frequenza (della parola). Su questa mappa il Reducer memorizza quante volte ogni parola è presente. Per facilitare le operazioni di ordinamento, abbiamo creato un oggetto con due attributi: parola e frequenza. Questi oggetti sono comparabili in base alla frequenza. Dalla mappa precedente otteniamo una lista di quest'ultimi, la ordiniamo in maniera discendente e la tagliamo ai primi dieci elementi. Il Reducer, alla fine, scriverà l'anno seguito dalla lista di coppie parola-frequenza.

Hive

Dopo aver creato la tabella e caricato i dati del CSV, utilizziamo due liste come supporto alla query finale. La prima ci consente di convertire il Time in anno e a ripulire il campo Summary. Nella seconda separiamo il campo Summary in parole singole. Nella query selezioniamo l'anno, la parola e quante volte ricorre quest'ultima. Per contare la frequenza delle parole e selezionare le prime dieci, utilizziamo una funzione "row_number()", che conta il numero di righe in cui compare una parola in base all'anno, ordinando dalla più alla meno frequente. Dopo imponiamo che sia minore-uguale a dieci.

Spark

All'inizio abbiamo diviso il job in due fasi: nella prima otteniamo un "JavaPairRDD", che sarà formato da coppie anno-lista di parole di quell'anno. Per fare ciò, innanzitutto carichiamo il CSV, facciamo il parsing di ogni linea e ricaviamo il Time, convertito in anno, e la lista di parole, "splittando" il campo Summary. Dopo aver ripulito tutte le parole, raggruppiamo, per ogni anno, le parole in un'unica lista. Nella seconda fase, per ottenere il risultato finale, prendiamo, da ogni anno, l'inerente lista di parole e, analogamente come fatto nel MapReduce, creiamo una mappa con coppie parola-frequenza. Per mantenere la scrittura tramite funzioni lambda, abbiamo calcolato la frequenza di ogni parola utilizzando gli "stream" introdotti nella versione 8 di Java. Ordiniamo la mappa in base ai valori, inseriamo le coppie chiave-valore in una lista e ne

prendiamo i primi dieci elementi. Alla fine avremo un “JavaPairRDD” contenente coppie anno e lista di coppie parola-frequenza.

Output

(1999,[(fairy,3), (day,3), (modern,3), (tale,3), (a,3), (is,2), (child,1), (entertainingl,1), (funny,1), (every,1)])

(2000,[(a,11), (bettlejuice,9), (master,6), (version,5), (great,4), (fabulous,3), (fresh,3), (original,3), (directed,3), (funny,3)])

(2001,[(beetlejuice,7), (the,6), (dvd,6), (is,4), (a,4), (greatmovie,3), (great,3), (on,3), (terrible,3), (they,3)])

(2002,[(a,20), (the,15), (great,14), (beetlejuice,12), (of,9), (this,9), (is,9), (movie,9), (for,8), (it,8)])

(2003,[(the,23), (of,12), (not,11), (great,10), (and,9), (best,8), (in,8), (for,8), (a,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), (this,70), (it,70)])

...

Pseudocodice

MapReduce

```
1.  map(key, value, context):
2.      tokens = "[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"']";
3.      //regular expression for useless characters
4.
5.      if key == 0:
6.          return; //skip the header of csv file
7.
8.      //fields is an array of strings containing the values of the csv
9.      fields[] = parseCSVLine(value);
10.     //converting unix time to year in the yyyy format
11.     year = convertUnixTime2Year(fields[Constants.TIME_INDEX]); //TIME_INDEX == 7
12.     //normalizing each word in summary field
13.     cleanSummary = fields[Constants.SUMMARY_INDEX].toLowerCase().replaceAll(tokens," ");
14.     //SUMMARY_INDEX == 8;
15.
16.     for word in cleanSummary:
17.         context.write(year, word);
18.
19. reduce(key, values, context):
20. //summary2frequency is a dictionary with a word as key and the frequency of that word as
     value
21.     for word in values:
22.         frequency = 1;
23.         if(summary2frequency.containsKey(word)):
24.             frequency = summary2frequency.get(word) + 1;
25.         summary2frequency.put(word, frequency);
26.
27.     //SingleReview is an object with two attributes: word and frequency
28.     //reviews is a list of SingleReview objects
29.     for word in summary2frequency.keySet():
30.         singleReview = new SingleReview(word, summary2frequency.get(word));
31.         reviews.add(singleReview);
32.
```

```
33.     //SingleReview objects are comparable by frequency value
34.     reviews.descendantSort();
35.
36.     len = reviews.size();
37.     if len > 10:
38.         len = 10;
39.
40.     //cut the list to first 10 or less elements
41.     reviews.cut(len);
42.
43.     context.write(key, buildStringFromList(reviews));
```

Hive

```
1.  CREATE TEMPORARY FUNCTION unix_date AS 'u2d.u2d.Unix2Date';
2.
3.  CREATE OR REPLACE VIEW cleaned AS
4.  SELECT unix_date(time) AS year, regexp_replace(regexp_replace(lower(summary), "[^a-zA-
    Z0-9]", " "), "[ ]+","#") AS clean
5.  FROM amazonfoodreviews;
6.
7.  CREATE OR REPLACE VIEW w AS
8.  SELECT year, exp.word
9.  FROM cleaned
10. LATERAL VIEW explode(split(clean, '#')) exp AS word
11. WHERE exp.word REGEXP "[^ ]+";
12.
13. SELECT year, word, word_count
14. FROM (
15.     SELECT year, word, count(1) as word_count,
16.            row_number() over (partition by year order by count(1) desc) AS row_num
17.     FROM w
18.     GROUP BY year, word
19. ) T
20. WHERE row_num <= 10;
```

Spark

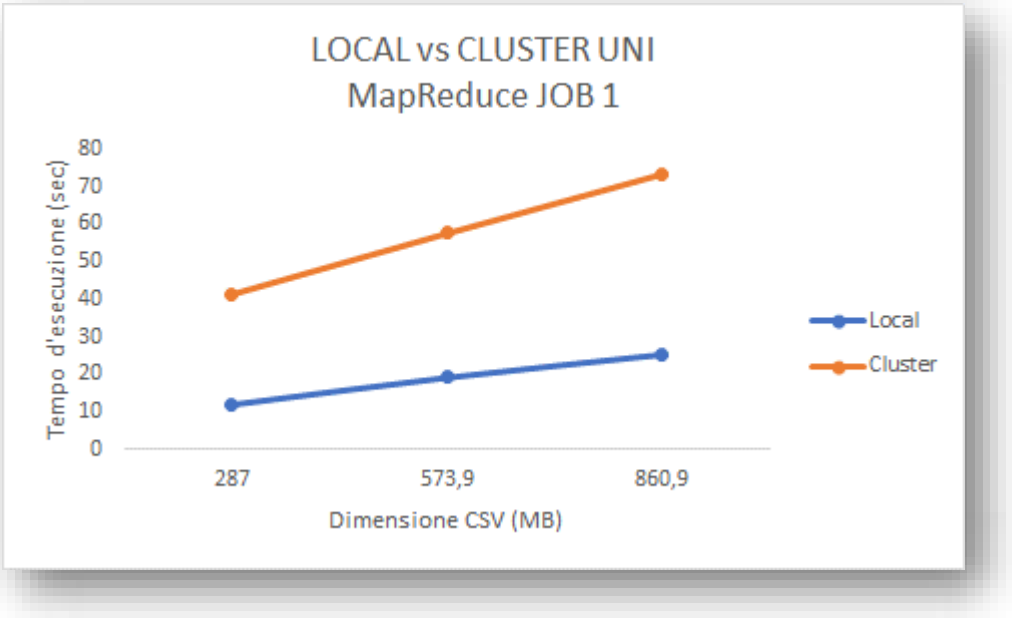
```
1.  loadData(sparkContext):
2.      tokens = "[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"']";
3.      //regular expression for useless characters
4.      //load the csv file
5.      words = sparkContext.textFile(inputPath);
6.      //couples is an RDD of pairs of the form (year, list of words of the year)
7.      couples = words.mapPartitionsWithIndex((index, iterator) ->
8.          if index == 0: //skip the header of csv file
9.              iterator.next();
10.         return iterator;
11.     ).mapToPair(line ->
12.         //fields is an array of strings containing the values of the csv
13.         fields[] = parseCSVLine(line);
14.         summary = fields[Constants.SUMMARY_INDEX].toLowerCase().replaceAll(tokens, " ");
    //SUMMARY_INDEX == 8;
15.         //converting unix time to year in the yyyy format
16.         year = convertUnixTime2Year(fields[Constants.TIME_INDEX]); //TIME_INDEX == 7
17.         //return a pair in the form of (year, list of words)
18.         return (year, list(summary.split(" ")));
19.     ).mapValues(words ->
20.         //trim each word and remove empty ones
```

```
21.         for word in words:
22.             word.trim();
23.             if(word.isEmpty()):
24.                 words.remove(word);
25.         return words;
26.     ).groupByKey().flatMapValues(x -> x);
27. //regroup all list of words of each single year in a list of lists and then flatten it
28. 
29.     return couples;
30. 
31. run(sparkContext):
32.     couples = loadData(sparkContext);
33. 
34.     //result is an RDD of pairs of the form (year, list of pairs (word, frequency))
35.     result = couples.mapToPair(couple ->
36.    //summary2frequency is a dictionary with a word as key and the frequency of that word
     as value
37.         for word in couple._2:
38.             frequency = 1;
39.             if(summary2frequency.containsKey(word)):
40.                 frequency = summary2frequency.get(word) + 1;
41.             summary2frequency.put(word, frequency);
42. 
43.         summary2frequency.sortByValues();
44. 
45.         //reviews is a list of pairs
46.         for word in summary2frequency.keySet():
47.             reviews.add((word, summary2frequency.get(word)));
48. 
49.         len = reviews.size();
50.         if len > 10:
51.             len = 10;
52. 
53.         //cut the list to first 10 or less elements
54.         reviews.cut(len);
55. 
56.         return (couple._1, reviews); //(year, list of (word, frequency))
57.     ).sortByKey();
58. 
59.     return result;
```

Tempi d'esecuzione

MapReduce

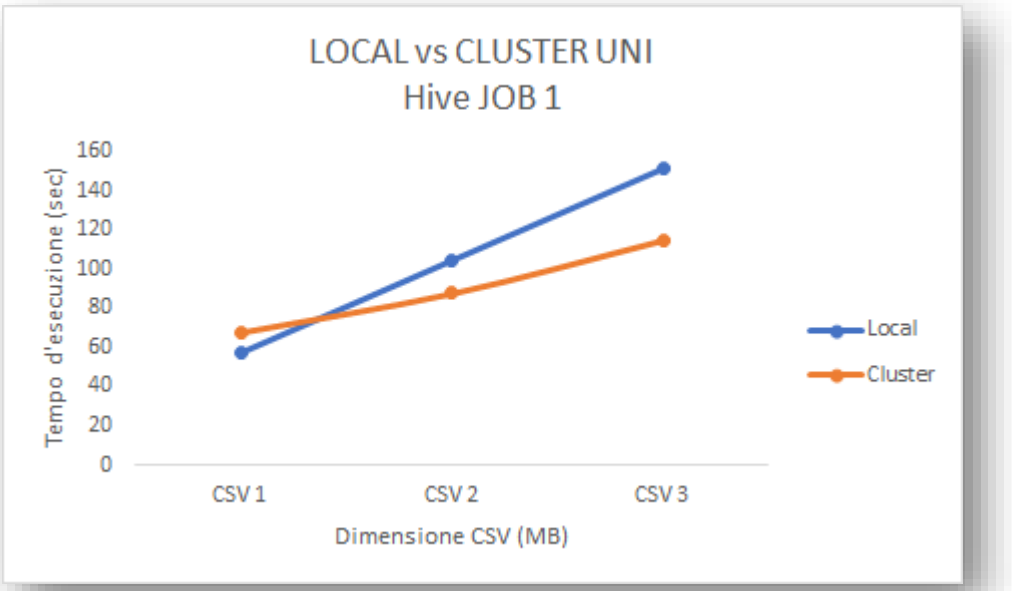
	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	12,065 sec	19,092 sec	25,043 sec
Cluster	41,404 sec	57,644 sec	73,371 sec



Hive

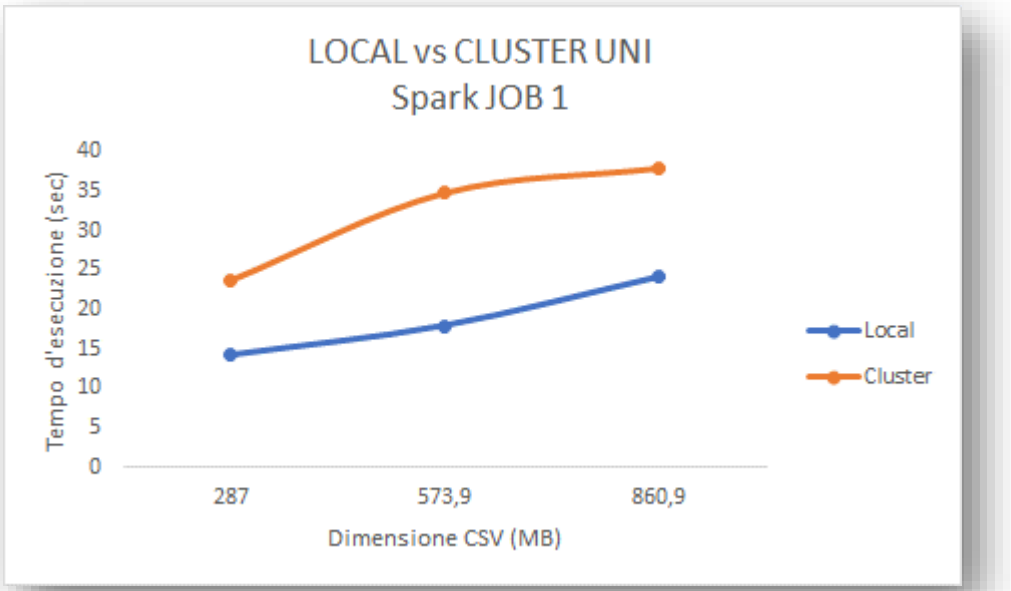
Tempi d'esecuzione calcolati sulle query (esclusi creazione e riempimento della tabella)

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	56,886 sec	104,158 sec	151,075 sec
Cluster	67,584 sec	87,419 sec	114,383 sec



Spark

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	14,145 sec	17,938 sec	24,213 sec
Cluster	23,713 sec	34,85 sec	38 sec



Job 2

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.

Soluzione

MapReduce

Nel Mapper viene fatto il parsing del CSV, dal quale ricaviamo i campi Time, ProductId e Score. Dopo aver effettuato la conversione da Unix Time in anno, verifichiamo se è compreso tra il 2003 e il 2012 (inclusi). Se l'anno è presente in questo arco di tempo, creiamo un oggetto con due attributi: anno e punteggio. Alla fine il Mapper scriverà coppie idprodotto-oggetto creato. Il Reducer avrà a disposizione tutti i prodotti e la lista di oggetti creati con anno e punteggio. Cicliamo per ogni anno dal 2003 al 2012 e contiamo la media del punteggio del prodotto inerente a quell'anno. Il Reducer restituirà l'id dei prodotti seguiti dalla lista composta da coppie anno-punteggio medio.

Hive

Dopo aver creato la tabella e caricato i dati del CSV, abbiamo usato una lista per convertire il Time in anno e per selezionare i campi interessati (ProductId e Score). Nella query principale calcoliamo la media del punteggio, con la condizione che l'anno deve essere compreso tra il 2003 e il 2012.

Spark

All'inizio abbiamo diviso il job in due fasi: nella prima otteniamo un "JavaPairRDD", che sarà formato da idprodotto e coppia anno-punteggio. Per fare ciò, carichiamo il CSV, facciamo il parsing di ogni linea, ricaviamo il Time, convertito in anno, e prendiamo i campi ProductId e Score. Da questi costruiamo le coppie necessarie per lo svolgimento del job. Nella seconda fase raggruppiamo in base all'idprodotto, tutte le coppie anno-punteggio medio in una lista. Utilizzando gli "stream" di Java filtriamo le coppie, in modo da selezionare solo quelle dell'arco di tempo richiesto e calcoliamo la media dei punteggi per ogni anno. Ci costruiamo una lista con le coppie risultanti, alla fine ritornerà il "JavaPairRDD" con il risultato del job.

Output

(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)])

...

Pseudocodice

MapReduce

```
map(key, value, context):
    if key == 0:
        return; //skip the header of csv file

    //fields is an array of strings containing the values of the csv
    fields[] = parseCSVLine(value);
    //converting unix time to year in the yyyy format
    year = convertUnixTime2Year(fields[Constants.TIME_INDEX]); //TIME_INDEX == 7
    if year >= 2003 and year <= 2012:
        productID = fields[Constants.PRODUCT_ID_INDEX]; //PRODUCT_ID_INDEX == 1
        //Year2Score is an object with 2 attributes: year and score
        year2score = new Year2Score(year, fields[Constants.SCORE_INDEX]);
        //SCORE_INDEX == 6
        context.write(productID, year2score)

reduce(key, values, context):
    //results is a list of Year2Score objects
    for(i = 2003; i < 2013; i++):
        cont = 0;
        score = 0;
        for year2score in values:
            if year2score.year == i:
                score += year2score.score;
                cont++;
        mean = 0.0;
        if cont != 0:
            mean = score/cont;
            newElem = new Year2Score(i, mean);
            results.add(newElem);

    //if results is empty, don't do anything
    if results.isEmpty():
        return;
    context.write(key, buildStringFromList(results));
```

Hive

```
CREATE TEMPORARY FUNCTION unix_date AS 'u2d.u2d.Unix2Date';

CREATE OR REPLACE VIEW product_year AS
SELECT productID, unix_date(time) AS year, score
FROM amazonfoodreviews;

SELECT productID, year, avg(score) AS scr
FROM product_year
WHERE year >= 2003 AND year <= 2012
GROUP BY productID, year;
```

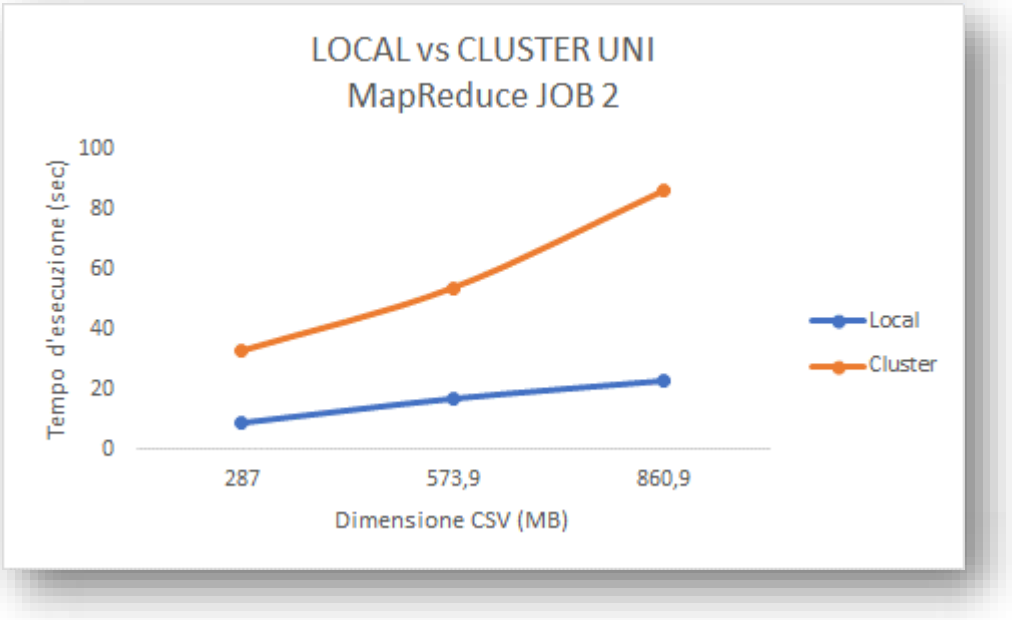
Spark

```
loadData(sparkContext):
    //load the csv file
    words = sparkContext.textFile(inputPath);
    //couples is an RDD of pairs of the form (product, (year, score))
    couples = words.mapPartitionsWithIndex((index, iterator) ->
        if index == 0: //skip the header of csv file
            iterator.next();
        return iterator;
    ).mapToPair(line ->
        //fields is an array of strings containing the values of the csv
        fields[] = parseCSVLine(line);
        productID = fields[Constants.PRODUCT_ID_INDEX]; //PRODUCT_ID_INDEX == 1
        score = fields[Constants.SCORE_INDEX]; //SCORE_INDEX == 6;
        //converting unix time to year in the yyyy format
        year = convertUnixTime2Year(fields[Constants.TIME_INDEX]); //TIME_INDEX == 7
        //return a pair in the form of (product, (year, score))
        return (productID, (year, score));
    );

    return couples;

run(sparkContext):
    couples = loadData(sparkContext);
  //result is an RDD of pairs of the form (productID, list of pairs (year, avgScore))
    result = couples.groupByKey() //regroup all the pairs of each product in a list
    .mapToPair(couple ->
        //pairs is a list of pairs (year, avgScore)
        for(i = 2003; i < 2013; i++):
            cont = 0;
            score = 0;
            for (year, sc) in couple._2:
                if year == i:
                    score += year2score.score;
                    cont++;
            mean = 0.0;
            if cont != 0:
                mean = score/cont;
                pairs.add((i, mean));

        return (couple._1, pairs); //(product, list of (year, avgScore))
    ).filter(el -> !el._2.isEmpty()) //filter products having scores
    .sortByKey();
  return result;
```

Tempi d'esecuzione

MapReduce

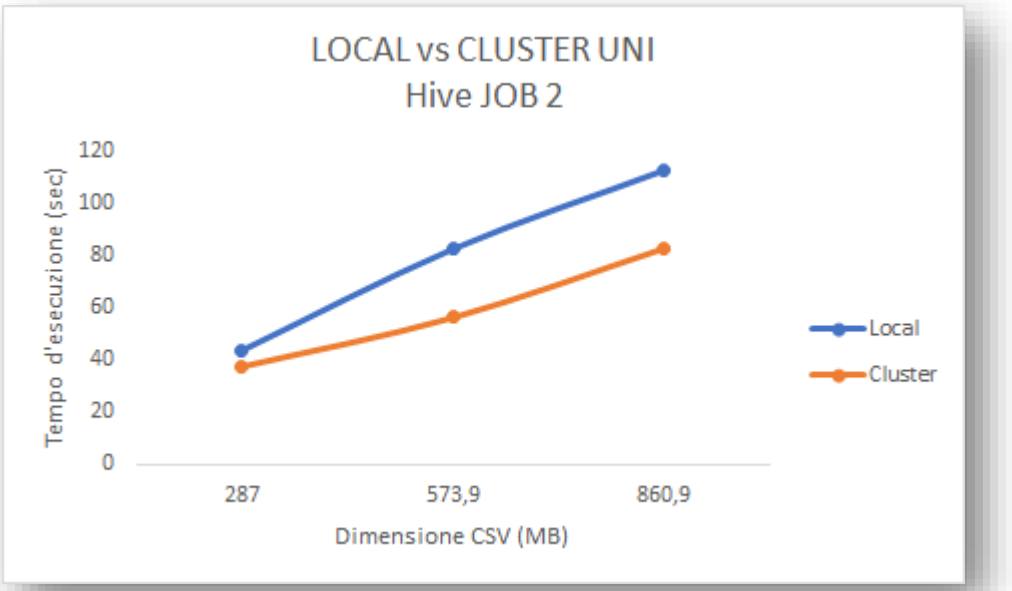
	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	8.996 sec	17,119 sec	23,079 sec
Cluster	32,632 sec	53,758 sec	86,398 sec



Hive

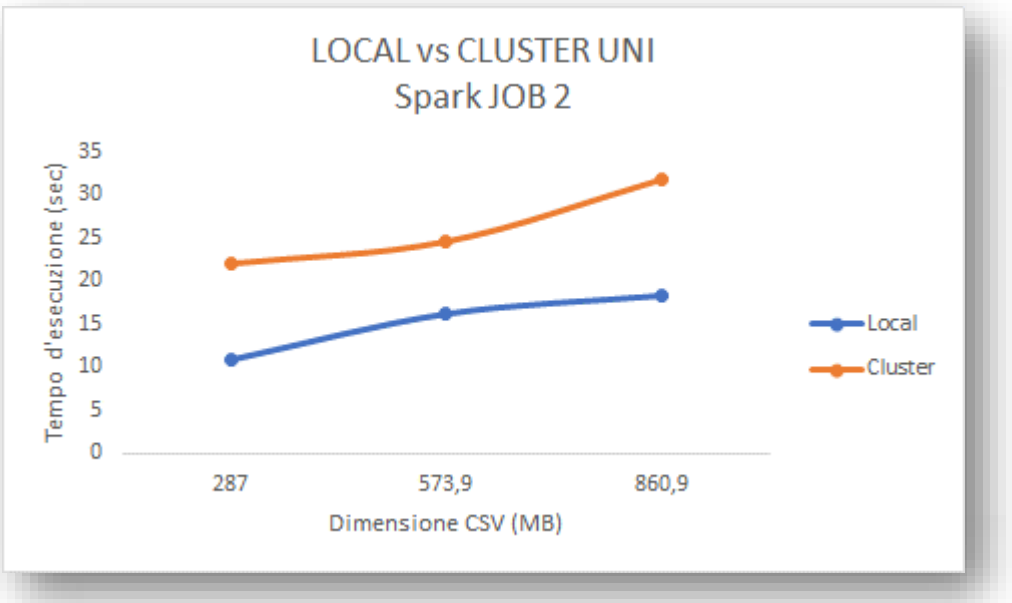
Tempi d'esecuzione calcolati sulle query (esclusi creazione e riempimento della tabella)

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	43,579 sec	82,27 sec	112,234 sec
Cluster	37,626 sec	56,134 sec	82,309



Spark

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	10,842 sec	16,239 sec	18,375 sec
Cluster	22,059 sec	24,594 sec	31,701 sec



Job 3

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.

Soluzione

MapReduce

Abbiamo usato due cicli MapReduce: nel Mapper del primo ciclo viene fatto il parsing del CSV e ricaviamo coppie UserId-ProductId. Nel Reducer avremo tutti i prodotti di un utente, eliminiamo tutti i doppioni, li scorriamo e costruiamo tutte le possibili coppie in base al loro id. Il Reducer ritornerà i prodotti e l'utente. Nel secondo ciclo il mapper prende le coppie di prodotti e l'utente, restituendoli. Il Reducer avrà per ogni coppia di prodotti, la lista di utenti che l'hanno acquistati, elimina gli eventuali doppioni e ne conta il numero.

Hive

Dopo aver creato la tabella e caricato i dati del CSV, facciamo il join della tabella con se stessa sull'UserId per ottenere tutte le coppie di prodotti per utente. Dal risultato del join, filtriamo le righe con uguali prodotti e contiamo il numero di righe.

Spark

All'inizio abbiamo diviso il job in due fasi: nella prima otteniamo un "JavaPairRDD", che sarà formato da coppie UserId-ProductId. Per fare ciò, innanzitutto carichiamo il CSV, facciamo il parsing di ogni linea e ricaviamo i campi necessari. Nella seconda fase prendiamo il "JavaPairRDD" precedente ed effettuiamo il join su se stesso. In questo modo avremo utente-coppie di prodotti, filtriamo solo quelle validi, costruiamo delle nuove coppie in cui al primo elemento abbiamo i due prodotti e al secondo l'utente. Raggruppiamo tutti gli utenti riferiti ad una coppia di prodotti in un'unica lista e contiamo quanti sono questi utenti.

Output

(0006641040, B0005XN9HI,1)

(0006641040, B00061EPKE,1)

(0006641040, B000EM00YU,1)

(0006641040, B000FDQV46,1)

(0006641040, B000FV8LPU,1)

...

(7310172001, 7310172101,167)

(7310172001, B00004S1C6,1)

(7310172001, B000084EZ4,1)

(7310172001, B00008CQVA,1)

(7310172001, B00008JOL0,1)

…

(7310172001, B0002DGRPC,167)

(7310172001, B0002DGRQ6,167)

(7310172001, B0002DGRRA,167)

(7310172001, B0002DGRSY,167)

(7310172001, B0002DGRZC,167)

…

Pseudocodice

MapReduce

```
1.  map1(key, value, context):
2.      if key == 0:
3.          return; //skip the header of csv file
4.
5.      //fields is an array of strings containing the values of the csv
6.      fields[] = parseCSVLine(value);
7.
8.      userID = fields[Constants.USER_ID_INDEX]; //USER_ID_INDEX == 2
9.      productID = fields[Constants.PRODUCT_ID_INDEX]; //PRODUCT_ID_INDEX == 1
10.
11.     context.write(userID, productID);
12.
13. reduce1(key, values, context):
14.     //productSet is a set of productIDs
15.     for productID in values:
16.         productSet.add(productID);
17.
18.     //products is a list of productsIDs
19.     products.addAll(productSet);
20.
21.     for(i = 0; i < products.size() - 1; i++):
22.         for(j = i + 1; j < products.size(); j++):
23.             product1 = products.get(i);
24.             product2 = products.get(j);
25.             //making a single string from two products
26.             couple = product1 + "," + product2;
27.             context.write(couple, key);
28.
29. map2(key, value, context):
30.     //value is a string like "<products couple>\t<userID>"
31.     fields = value.split("\t");
32.
33.     productCouple = fields[0];
```

```
    userID == fields[1];

    context.write(productCouple, userID);

reduce2(key, values, context):
    //userIDs is a set of users
    userIDs.addAll(values);

    context.write(key, userIDs.size());
```

Hive

```
SELECT
    t1.productId AS product1,
    t2.productId AS product2,
    COUNT(1) AS cnt
FROM
    (
    SELECT DISTINCT userId, productId
    FROM amazonfoodreviews
    ) t1
JOIN
    (
    SELECT DISTINCT userId, productId
    FROM amazonfoodreviews
    ) t2
ON (t1.userId = t2.userId)
GROUP BY t1.productId, t2.productId
HAVING t1.productId < t2.productId
ORDER BY t1.productId;
```

Spark

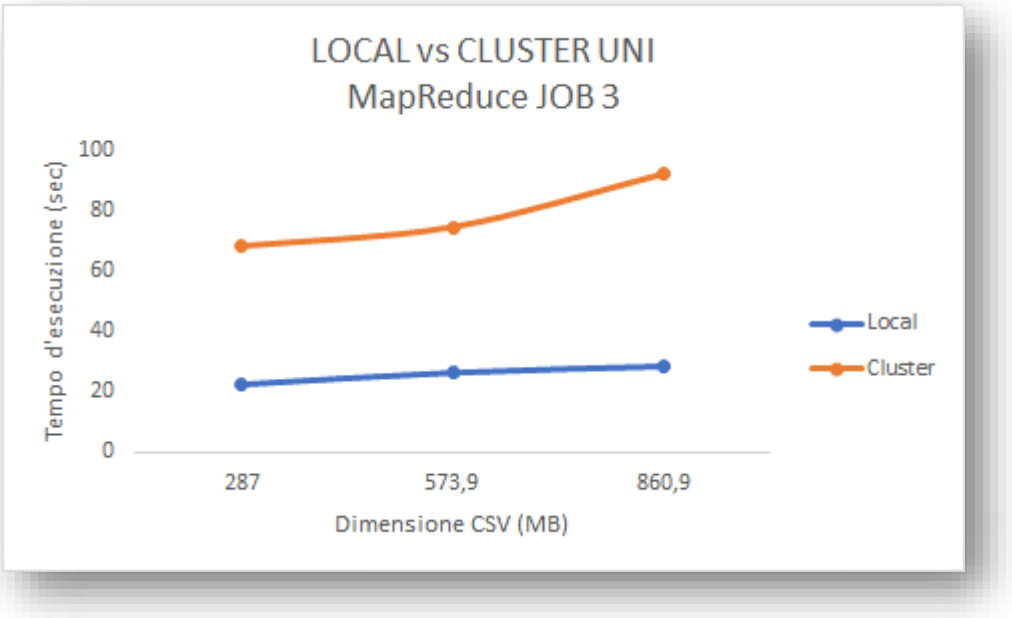
```
loadData(sparkContext):
    //load the csv file
    words = sparkContext.textFile(inputPath);
    //couples is an RDD of pairs of the form (user, product)
    couples = words.mapPartitionsWithIndex((index, iterator) ->
        if index == 0: //skip the header of csv file
            iterator.next();
        return iterator;
    ).mapToPair(line ->
        //fields is an array of strings containing the values of the csv
        fields[] = parseCSVLine(line);
        productID = fields[Constants.PRODUCT_ID_INDEX]; //PRODUCT_ID_INDEX == 1
        userID = fields[Constants.USER_ID_INDEX]; //USER_ID_INDEX == 2;
        //return a pair in the form of (user, product)
        return (userID, productID);
    );

    return couples;

run(sparkContext):
    couples = loadData(sparkContext);

    //result is an RDD of pairs of the form ("productID1, productID2", number of users)

    result = couples.join(couples)
```

```
25.     //join on userIDs produce pairs in the from of (userID, (product1, prdouct2))
26.     //remove all the pairs where product1 and product2 are equals or pairs already visit
    ed
27.     .filter(user2couple -> user2couple._2._1.compareTo(user2couple._2._2) < 0)
28.     .mapToPair(elem -> (elem._2._1 + ", "+ elem._2._2, elem._1))
29.     //insert products in a single string ("productID1, productID2", user)
30.     .groupbyKey() //group all the users of a products couple
31.     .sortByKey()
32.     //counting users
33.     .mapToPair(couple2users ->
34.         //users is a set of users
35.         user.addAll(coupe2users._2);
36.         return (couple2users._1, users.size());
37.     //("productID1, productID2", number of users)
38.     ).sortByKey();
39.
40.     return result;
```

Tempi d'esecuzione

MapReduce

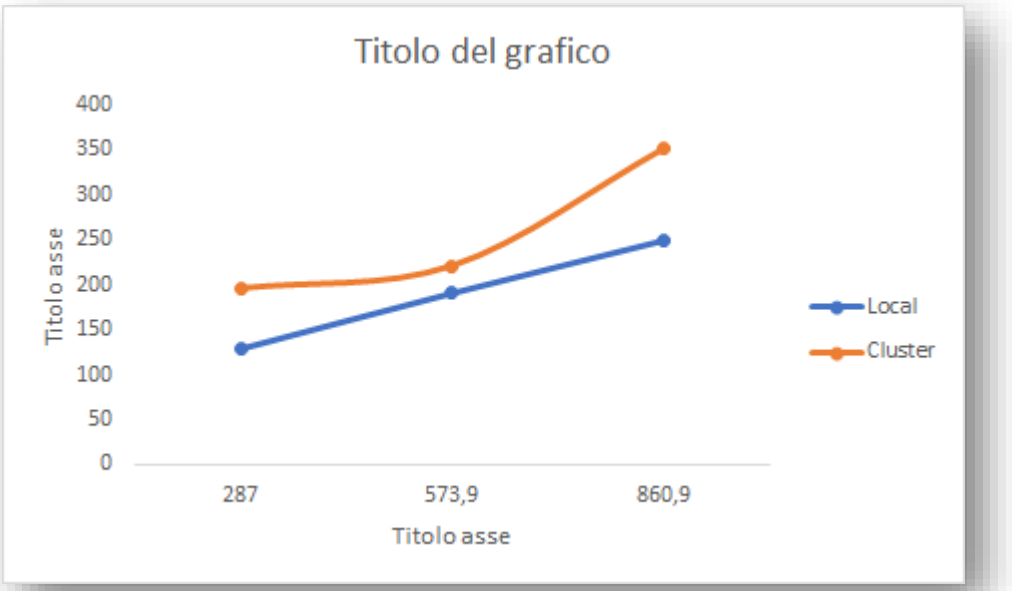
	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	22,202 sec	26,174 sec	28,238 sec
Cluster	68,238 sec	74,578 sec	92,197 sec



Hive

Tempi d'esecuzione calcolati sulle query (esclusi creazione e riempimento della tabella)

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	128,262 sec	191,101 sec	250,293 sec
Cluster	196,371 sec	221,569 sec	352,262 sec



Spark

	CSV 1 (287 MB)	CSV 2 (573,9 MB)	CSV 3 (860,9 MB)
Local	25,476 sec	64,818 sec	120,394 sec
Cluster	51,691 sec	64,187 sec	85,813 sec