1° Progetto

Gruppo: Los Datos Hermanos

Jhonattan Christian Loza – 452252
Nicholas Tucci – 461669

Codice del progetto disponibile anche presso l'indirizzo GitHub:
https://github.com/Vzzarr/BigData---FineFoodReviews

TASK1: generare, per ciascun mese, i cinque prodotti che hanno ricevuto lo score medio più alto, indicando prodotto e score medio e ordinando il risultato temporalmente

SOLUZIONE

- MapReduce: abbiamo usato come chiave per il Mapper la coppia Data|ProductId, così da poter raggruppare per ogni prodotto all'interno di uno stesso mese tutti gli score che ha ricevuto; in modo tale da poter poi calcolare una semplice media degli score nel Reducer relativa al singolo prodotto nello specifico mese. A tal punto si può poi gestire una lista ordinata (mantenendone la sua dimensione inferiore o uguale a 5) che alla fine del processo restituirà l'output desiderato.
- Hive: con una prima view è stata calcolata la media degli score relativi al prodotto nel mese specifico, raggruppandoli tramite un'opportuna GROUP BY. Questo risultato verrà poi sfruttato per poter ordinare ogni raggruppamento in ordine decrescente in base allo score medio, per poi poterne selezionare le prime 5 righe di ogni raggruppamento.
- Spark: nel primo flusso si selezionano solo i campi di interesse dal file di input, mettendo come "chiave" della Tupla sempre la coppia Data|ProductId e come valore lo score (come fatto in precedenza); nel successivo si calcola una media tramite un'operazione di Reduce, per poi in seguito spostare il ProductId nel valore della Tupla in un ulteriore flusso. Infine con una GroupByKey si selezionano per ogni mese i primi 5 prodotti con score medio più alto.

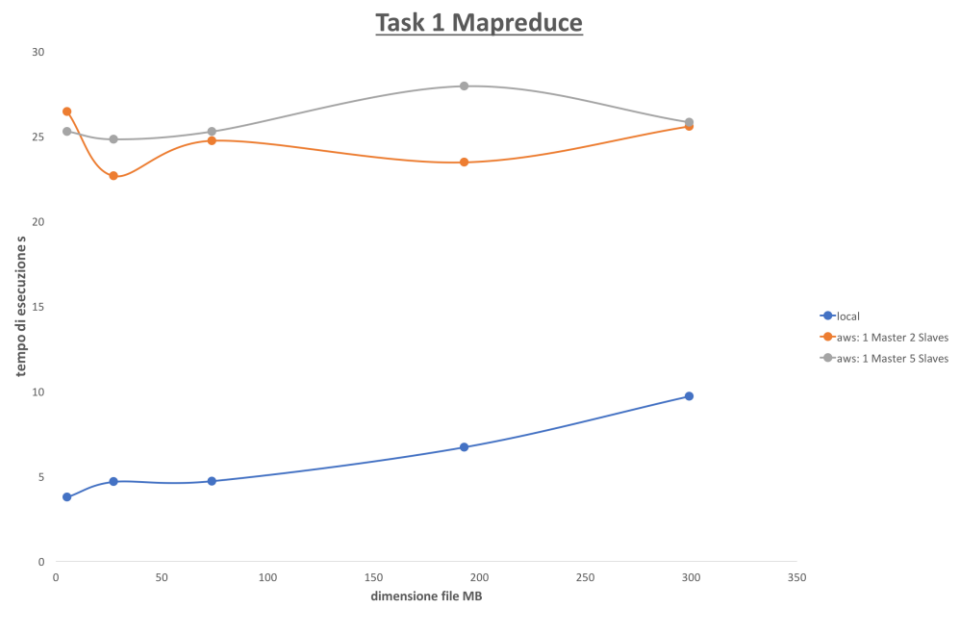
PRIME RIGHE OUTPUT

(1999-10, 0006641040|5.0)
(1999-12, B00004CI84|5.0 B00004CXX9|5.0 B00004RYGX|5.0)
(2000-01, B00004RYGX|3.0 B00004CI84|3.0 B00004CXX9|3.6666666666666665 B00002N8SM|5.0)
(2000-02, B00004RYGX|4.0 B00004CXX9|4.0 B00004CI84|4.0)
(2000-06, B00004CXX9|5.0 B00002Z754|5.0 B00004CI84|5.0 B00004RYGX|5.0)
...

TEMPI DI ESECUZIONE

MapReduce

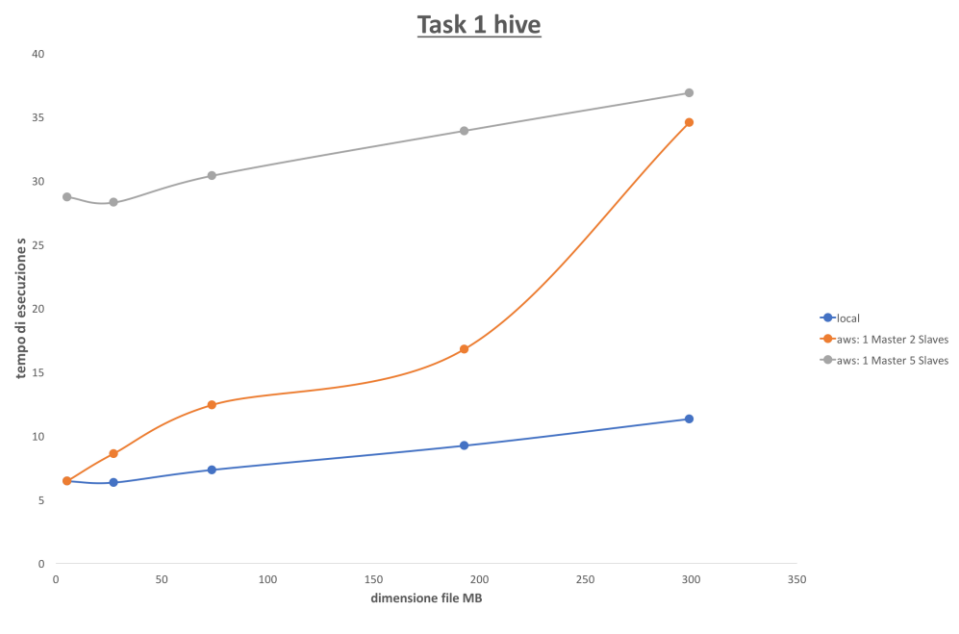
#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	3,774	4,672	4,712	6,706	9,696
aws: 1 Master 2 Slaves	26,433	22,662	24,726	23,461	25,569
aws: 1 Master 5 Slaves	25,271	24,807	25,261	27,925	25,806



Hive

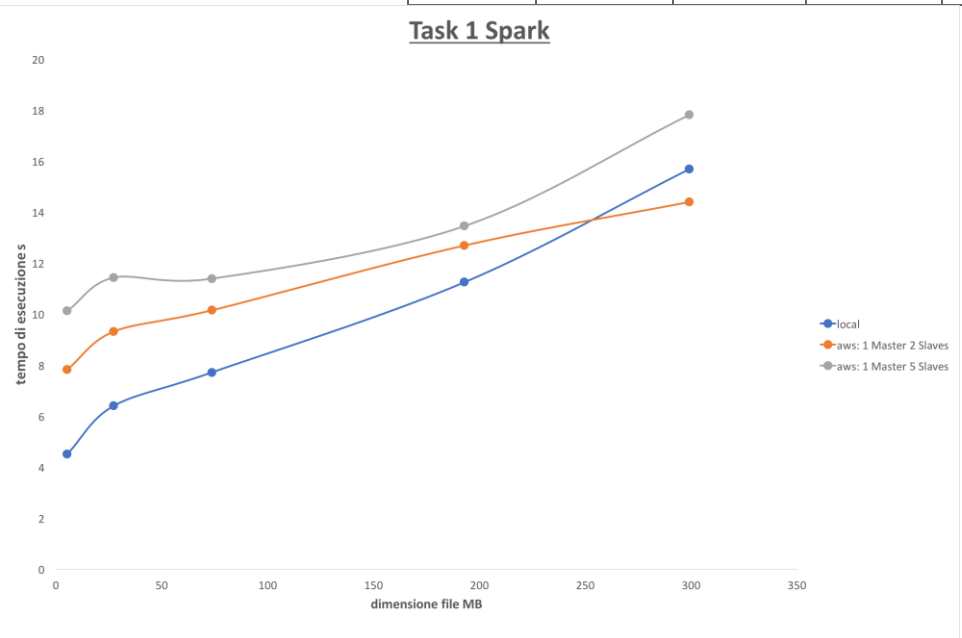
#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	6,441	6,34	7,325	9,22	11,298
aws: 1 Master 2 Slaves	6,44	8,6	12,4	16,78	34,54

aws: 1 Master 5 Slaves	28,713	28,287	30,377	33,897	36,86



Spark

#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	4,508	6,41	7,717	11,252	15,687
aws: 1 Master 2 Slaves	7,822	9,318	10,149	12,686	14,391
aws: 1 Master 5 Slaves	10,125	11,435	11,391	13,453	17,813



PSEUDOCODICE

MapReduce

Map

```
1.  //method used to map input parameters
2.  map(key, value, context) {
3.          //line is an array string
4.          line[] = value.split("\t")
5.          unixTime = parseStringToLong(line[7]);
6.          productId = line[1];
7.          score = parseStringToDouble(line[6]);
```

```
8.
9.          //convertUnixTime(): transforms a long expressed in unixtime to the
    date format YYYY:MM
10.              //key<date|input> -> value<score>
11.         context.write(convertUnixTime(unixTime) + "|" + productId), score);
12. }
```

Reduce

```
1.      best5products //string list
2.      date2idProduct_value //map<String, list(String)> whose purpose is to maintain a
    collection of [product – avg score] relative to a certain YYYY:MM
3.
4.      reduce(key, values, context) {
5.          key[] = key.toString().split("\\|")
6.          avg = count = 0; //double
7.          for each value in values
8.              avg+=value
9.              count++;
10.         end for
11.         //support method that adds an element to date2idProduct_value
12.         add_product2scores(key[0], key[1] + "|" + (avg/count));
13.     }
14.
15.     cleanup(context) {
16.         for each date in date2idProduct_value.keySet()
17.             best5products //initialize string list
18.             for each idProduct_avg in date2idProduct_value.get(date))
19.                 //p is a product object
20.                 idProduct_avg[] = idProduct_avg.split("\\|")
21.                 p = new Product(idProduct_avg[0])
22.                 p.setAverage(parseStringToDouble(idProduct_avg.[1]));
23.                 //inserts a product in order in best5products (according to his average)
    maintaining 5 as size
24.                 addValue(p);
25.             end for
26.             //getIdProducsAvg() returns a string of the list best5products
27.             context.write(date, getIdProducsAvg(best5products))
28.         end for
29.     }
```

Hive

```
1.  CREATE TABLE IF NOT EXISTS foodreviews
2.  (id INT, productId STRING, userId STRING, profileName STRING, helpfulnessNumerator INT,
3.  helpfulnessDenominator INT, score INT, time BIGINT, summary STRING, text STRING)
4.  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
5.
6.  LOAD DATA LOCAL INPATH 'path your file'
7.  OVERWRITE INTO TABLE foodreviews;
8.
9.  CREATE VIEW average_score AS
10. SELECT dg.ym, dg.productId, avg(dg.score) AS average
11. FROM(
12. SELECT from_unixtime(time, 'Y/MM') AS ym, productId, score
13. FROM foodreviews) dg
14. GROUP BY dg.productId, dg.ym
15. ORDER BY dg.ym, dg.productId;
16.
17. SELECT t.ym, t.productId, t.average
18. FROM(
19. SELECT ym, productId, average, row_number() over(PARTITION BY ym ORDER BY average DESC) AS
     rank
```

```
20. FROM average_score) t   WHERE t.rank <= 5;
```

Spark

```
1.  JavaRDD<String> data = sc.textFile(inputFilePath);
2.  //key = date_idProduct  values = score
3.  data.mapToPair(row -> {
4.      String[] tsvValues = row.split("\t");
5.      return new Tuple2<>(convertUnixTime(Long.parseLong(tsvValues[UNIXTIME])) + "|" + tsvVa
    lues[ID_PRODUCT], new Tuple2<>(Double.parseDouble(tsvValues[SCORE]), new Double(1.0)));

6.      // avg score for each product in each month
7.  })
8.  .reduceByKey((a,b) -> new Tuple2(a._1 + b._1, a._2 + b._2)).mapValues(tuple -
    > tuple._1 / tuple._2)
9.      //change map -> key = date   values = idProduct_score
10.     .mapToPair(row -> {
11.            String[] rowsValues = row._1.split("\\|");
12.            return new Tuple2<>(rowsValues[0], rowsValues[1] + "|" + row._2); })
13.         //select top k elements with avg score with method addValue() and print into file

14.     .groupByKey().mapToPair(partitionDate -> {
15.         Iterator<String> it = partitionDate._2.iterator();
16.         List<Product> best5products = new LinkedList<>();
17.         while(it.hasNext()){
18.             String[] id_avg = it.next().split("\\|");
19.             Product product = new Product(id_avg[0]);
20.             product.setAverage(Double.parseDouble(id_avg[1]));
21.             best5products = addValue(best5products, product);
22.         }
23.         String result = " ";
24.         for (Product product : best5products)
25.             result += (product.getIdProduct() + "|" + product.getAverage() + " ");
26.         return new Tuple2<>(partitionDate._1, result);
27.         })
28.        .sortByKey()
29.        .saveAsTextFile(outputFolderPath);
```

TASK2: generare, per ciascun utente, i 10 prodotti preferiti (ovvero quelli che ha recensito con il punteggio più alto), indicando prodotto e score e ordinando il risultato in base allo User ID

SOLUZIONE

- MapReduce: abbiamo usato lo UserId come chiave nel Mapper e la coppia ProductId|score come valore, in modo da raggruppare per ogni utente tutti gli score che ha assegnato ai rispettivi prodotti. Analogamente al task1, abbiamo poi inserito il prodotto in modo ordinato (decrescente) in base al suo score, mantenendo la sua dimensione pari o inferiore a 10, ottenendo così l'output desiderato.
- Hive: vengono selezionati per ogni utente i suoi 10 preferiti prodotti in ordine di score, tramite l'apposita funzione di Hive row_number()
- Spark: sono state selezionate soltanto i campi di interesse dal file di input, ossia lo UserId come chiave e la coppia ProductId|score come valore. Tramite una GroupByKey abbiamo quindi raggruppato tutti gli score assegnati da un utente ai rispettivi ProductId; questi sono stati poi inseriti in modo ordinato (decrescente) all'interno di una collezione che mantenga una dimensione non superiore a 10, riuscendo così a calcolare l'output.

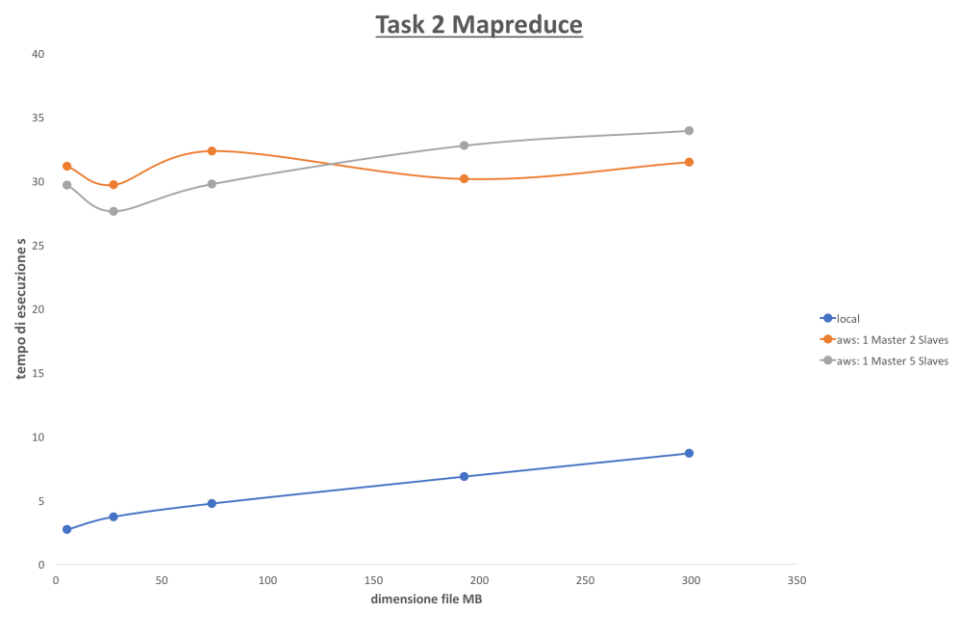
PRIME RIGHE OUTPUT

(A100CY9WRC18I2, B000CQG84Y|1.0)
(A101CCC619GN4S, B00017L1UK|5.0)
(A101VS17YZ5ZEJ, B0004LW990|5.0)
(A103OZ75AVET1Y, B000CBOR60|5.0)
(A1048CYU0OV4O8, B00004CXX9|5.0 B00004CI84|5.0 B00004RYGX|5.0)
...

TEMPI DI ESECUZIONE

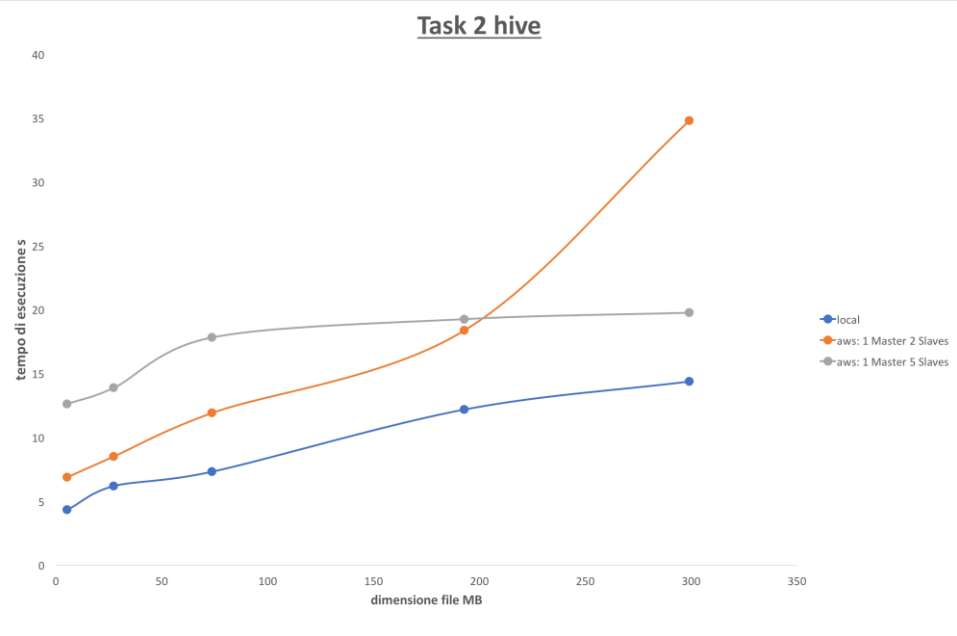
MapReduce

#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	2,705	3,707	4,741	6,856	8,681
aws: 1 Master 2 Slaves	31,131	29,683	32,336	30,143	31,46
aws: 1 Master 5 Slaves	29,659	27,615	29,737	32,747	33,908



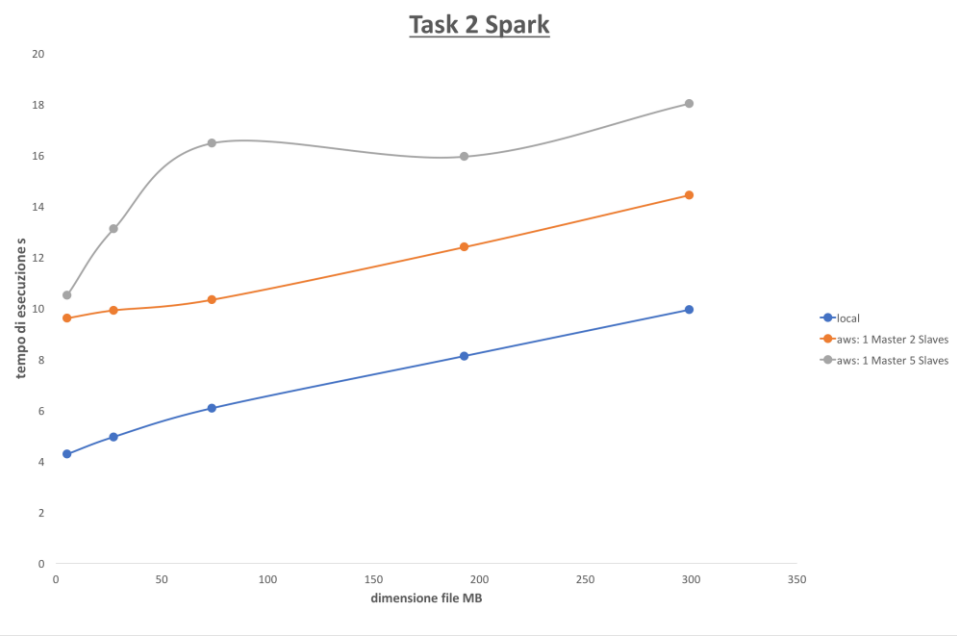
Hive

#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	4,332	6,192	7,312	12,176	14,369
aws: 1 Master 2 Slaves	6,88	8,51	11,91	18,38	34,79
aws: 1 Master 5 Slaves	12,621	13,891	17,819	19,253	19,755



Spark

#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
local	4,273	4,951	6,074	8,122	9,938
aws: 1 Master 2 Slaves	9,607	9,916	10,328	12,395	14,431
aws: 1 Master 5 Slaves	10,507	13,115	16,467	15,942	18,021



PSEUDOCODICE

MapReduce

Map

```
1.  //method used to map input parameters
2.  map(key, value, context) {
3.          line = value.split("\t") //array String
4.          userId = line[2], productId = line[1]
5.          score = parseStringToDouble(line[6]);
6.          //key<userId> -> value<productId|score>
7.          context.write(userId, productId + "|" + score);
8.  }
```

Reduce

```
best10products //list of ten best products
reduce(key, values, context) {
        best10products = new List

        for each value in values
            productId = value.split("\\|")[0];
            score = parseStringToDouble(value.split("\\|")[1]);
            product = new Product(productId);
            product.setScore(score)
            //inserts a product in order in best10products (according to his score) maintaining 10 as size
            addValue(product)
        end for

        result = ""

        for each  prodotto of best10products
            //Create output
            result += " " + prodotto.getIdProduct() + "|" + prodotto.getScore()
        end for
        context.write(key, result)
}
```

Hive

```
CREATE TABLE IF NOT EXISTS foodreviews
(id INT, productId STRING, userId STRING, profileName STRING, helpfulnessNumerator INT,
helpfulnessDenominator INT, score INT, time BIGINT, summary STRING, text STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH ''
OVERWRITE INTO TABLE foodreviews;

SELECT t.userId, t.productId, t.score
FROM(
SELECT userid, productId, score, row_number() over(PARTITION BY userid ORDER BY score DESC) as rank
FROM foodreviews) t
WHERE t.rank <= 10;
```

Spark

```
JavaRDD<String> data = sc.textFile(inputFilePath);

        //key = idUser  values = idProduct_score
        data.mapToPair(row -> {
            String[] tsvValues = row.split("\t");
            return new Tuple2<>(tsvValues[ID_USER],
                    new Tuple2<>(tsvValues[ID_PRODUCT], Double.parseDouble(tsvValues[SCORE])));
            // select best 10 products for each user
    }).groupByKey().mapToPair(partitionUser -> {
            Iterator<Tuple2<String, Double>> it = partitionUser._2.iterator();
            List<Product> best10products = new LinkedList<>();
            while(it.hasNext()){
                Tuple2<String, Double> productId_score = it.next();
                Product product = new Product(productId_score._1);
                product.setScore(productId_score._2);
                best10products = addValue(best10products, product);
            }
            String result = " ";
            for (Product product : best10products)
```

```
20.                 result += (product.getIdProduct() + "|" + product.getScore() + " ");
21.             return new Tuple2<>(partitionUser._1, result);
22.     }).sortByKey().saveAsTextFile(outputFolderPath);
```

TASK3: generare coppie di utenti con gusti affini, dove due utenti hanno gusti affini se hanno recensito con score superiore o uguale a 4 almeno tre prodotti in comune, indicando le coppie di utenti e i prodotti recensiti da entrambi e ordinando il risultato in base allo User ID del primo elemento della coppia

SOLUZIONE

- MapReduce: in un primo Mapper, filtrando le recensioni con score maggiore di 3, abbiamo usato il ProductId come chiave e lo UserId come valore, raggruppando quindi per ogni prodotto tutti gli utenti che abbiano recensito lo stesso prodotto; nel successivo Reducer è stata fatta quindi una "sorta di JOIN" tra i diversi utenti (evitando duplicati), ottenendo quindi per ogni prodotto tutte le possibili coppie di utenti che lo abbiano recensito. Questo risultato è stato usato per un ulteriore Mapper che stavolta ha la coppia di utenti precedentemente generata come chiave ed il prodotto come valore, raggruppando così per ogni coppia di utenti tutti i prodotti che sono stati recensiti da entrambi gli utenti; a questo punto nel Reducer basta controllare che il numero di prodotti da entrambi recensiti sia almeno 3.
- Hive: attraverso una prima view viene fatto un JOIN in base allo UserId, evitando di avere coppie di utenti duplicati, contando quindi il numero di prodotti per coppia di utenti che ha assegnato degli score maggiori o uguali a 4. Questo risultato viene usato in seguito per poter selezionare solo le coppie di utenti che hanno recensito almeno 3 prodotti.
- Spark: tramite un filtro iniziale si selezionano soltanto le recensioni con score maggiore di 3, per poi selezionare dal file di input soltanto i campi di interesse, ossia il ProductId come chiave della Tupla e lo UserId come valore. A questo punto si effettua un'operazione di JOIN in base alla chiave ProductId (precedentemente creata), per poi filtrare questo risultato eliminando le coppie di utenti ripetute. A questo punto si passa alla coppia di utenti come chiave e il ProductId recensito da entrambi come valore, per poi poter fare una GroupByKey, ottenendo così per ogni coppia di utenti la collezione di prodotti recensiti da entrambi; a questo punto basta filtrare le collezioni la cui dimensione sia maggiore o uguale a 3.

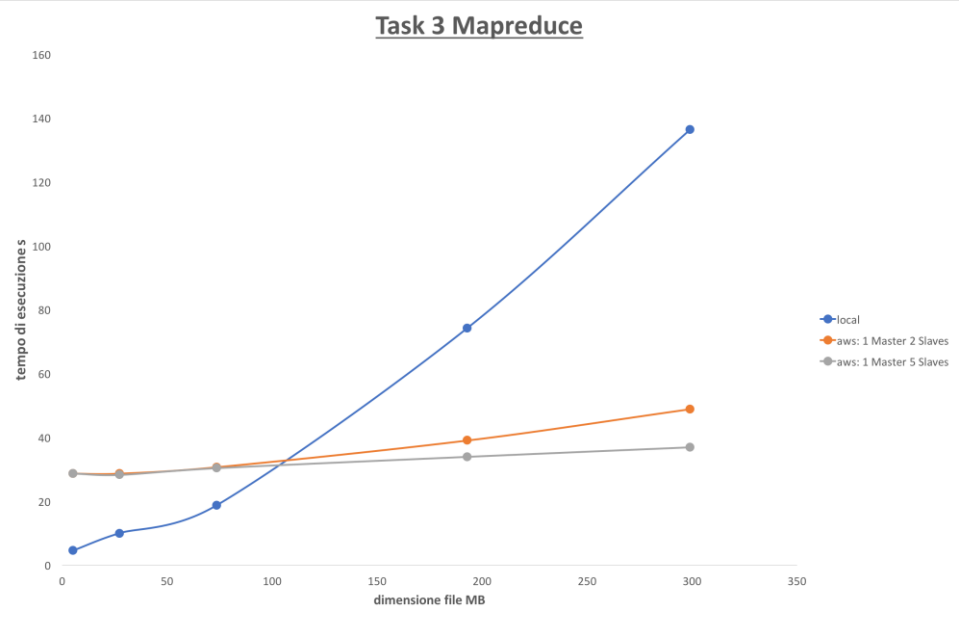
PRIME RIGHE OUTPUT

(A1048CYU0OV4O8|A157XTSMJH9XA4,[B00004CXX9, B00004CI84, B00004RYGX])
(A1048CYU0OV4O8|A19JYLHD94K94D,[B00004CXX9, B00004CI84, B00004RYGX])
(A1048CYU0OV4O8|A1BZEGSNBB7DVS,[B00004CXX9, B00004CI84, B00004RYGX])
(A1048CYU0OV4O8|A1CAA94EOP0J2S,[B00004CXX9, B00004CI84, B00004RYGX])
(A1048CYU0OV4O8|A1CZICCYP2M5PX,[B00004CXX9, B00004CI84, B00004RYGX])
...

TEMPI DI ESECUZIONE

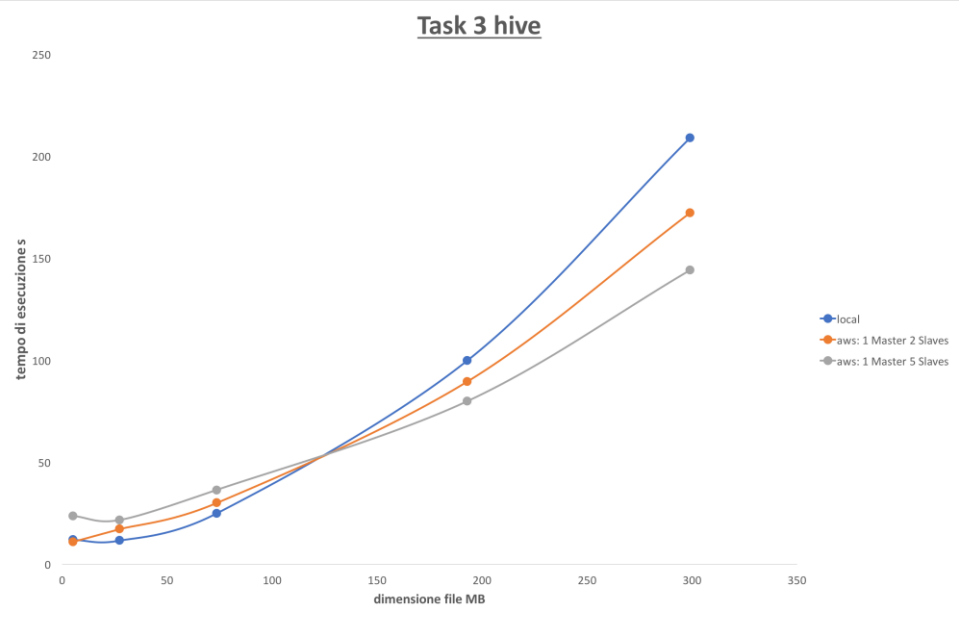
MapReduce

#nodes / file dimension MB	**5,2**	**27,3**	**73,6**	**192,9**	**299,1**
Local	4,545	9,98	18,715	74,149	136,337
aws: 1 Master 2 Slaves	28,705	28,689	30,698	39,058	48,869
aws: 1 Master 5 Slaves	28,713	28,287	30,377	33,897	36,86



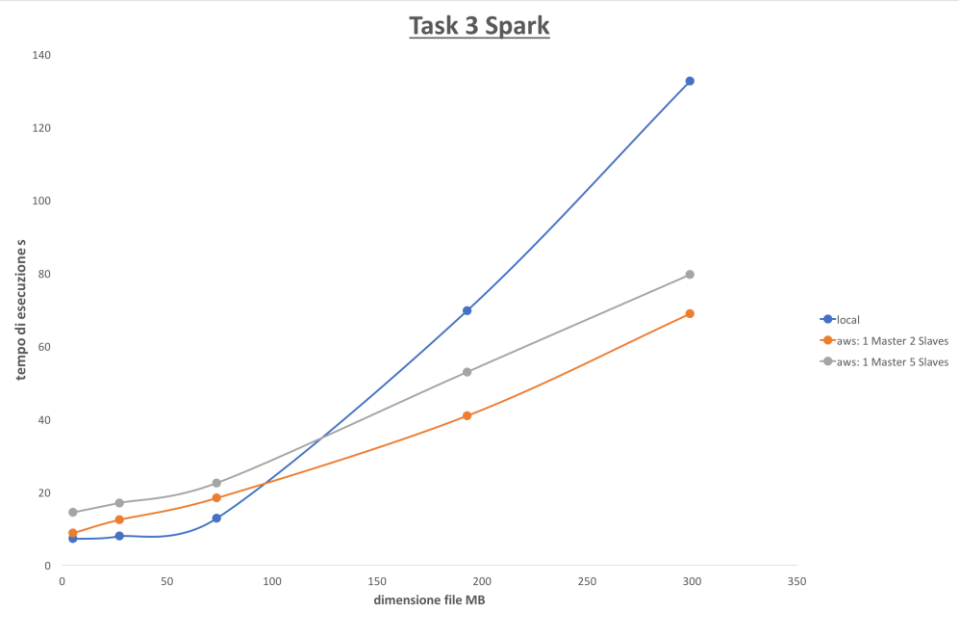
Hive

#nodes / file dimension MB	5,2	27,3	73,6	192,9	299,1
Local	12,005	11,62	24,95	99,729	208,847
aws: 1 Master 2 Slaves	11,04	17,24	30,08	89,48	172,17
aws: 1 Master 5 Slaves	23,618	21,613	36,379	79,829	144,122



Spark

#nodes / file dimension MB	5,2	27,3	73,6	192,9	299,1
local	7,219	7,944	12,855	69,72	132,543
aws: 1 Master 2 Slaves	8,817	12,462	18,406	40,927	68,856
aws: 1 Master 5 Slaves	14,42	17,021	22,479	52,905	79,612



PSEUDOCODICE

MapReduce

Map1

```
map(key, value, context) {
        String[] line = value.split("\t")

        String productId = line[1], userId = line[2];
        int score = line[6];
        if(score > 3)
            //key<productId> -> value<userId>
            context.write(productId, userId)
}
```

Reduce1

```
reduce(key, values, context){
    setUserKey_id //is a set of strings
    foreach value of values
        setUserKey_id.add(value);
    end foreach

    listUsers //is a list of string maked by values of setUserKey_id
    for i = 0 to listUsers.length
        for j = i+1 to listUsers.length
            coupleUsers = listUsers.get(i)+"|"+listUsers.get(j);
            //is a map <userID|userID, productId>
            context.write(coupleUsers, key)
        end for
    end for
}
```

Map2

```
map(key, value, context) {
    String[] line = value.split("\t")

    String coupleUser = line[0], productId = line[1];
    //crate a map <userID|userId, productId>
    context.write(coupleUser, productId);
```

```
}
```

Reduce2

```
reduce(key, values, context) {
    products //is a ordinate set of strings
    foreach text in values
        products.add(text)
    end foreach
    if products.length>2 then
        productsIds = "";
        foreach product in products
            productsIds += product + " ";
        end foreach
        //this is a output userId|userID : productID, productID, productID...
        context.write(key, productsIds);
    end if
}
```

Hive

```
CREATE TABLE IF NOT EXISTS foodreviews
(id INT, productId STRING, userId STRING, profileName STRING, helpfulnessNumerator INT,
helpfulnessDenominator INT, score INT, time BIGINT, summary STRING, text STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH 'path your file'
OVERWRITE INTO TABLE foodreviews;

CREATE VIEW similaruser as
SELECT a.userid as first, b.userid as second, COUNT(DISTINCT a.productid) as common_interests,
 collect_set(a.productId) as elencoProdottiComuni
FROM foodreviews a
JOIN foodreviews b on a.productId=b.productId
WHERE a.score > 3 and b.score > 3 and a.userid > b.userid
GROUP BY a.userid,b.userid;

SELECT sq.first, sq.second, sq.common_interests,sq.elencoProdottiComuni
FROM similaruser sq
WHERE sq.common_interests > 2
ORDER BY sq.first,sq.second;
```

Spark

```
JavaRDD<String> userRelateds = sc.textFile(inputFilePath);

JavaPairRDD<String, String> idProduct2user =  userRelateds
        .filter(filter -> Integer.parseInt(filter.split("\t")[SCORE]) > 3)
        .mapToPair(row -
> new Tuple2<>(row.split("\t")[ID_PRODUCT], row.split("\t")[ID_USER]));
        //key v1, key v2 -> key(v1, v2)
        idProduct2user.join(idProduct2user)
        .filter(product2coupleUser -
> product2coupleUser._2._1.compareTo(product2coupleUser._2._2) < 0)
        .mapToPair(row -
> new Tuple2<>(row._2._1 + "|" + row._2._2, row._1)).groupByKey()
        .mapToPair(coupleUser_products -> {
            Set<String> products = new HashSet<>();
            for (String iterable_element : coupleUser_products._2)
                products.add(iterable_element);
            return new Tuple2<>(coupleUser_products._1, products);
        })
```

```
16.         .filter(filter -> filter._2.size() > 2)
17.         .sortByKey()
18.         .saveAsTextFile(outputFolderPath);
```