

# Primo Progetto Big Data

MapReduce, Hive, Spark

*Castagnacci Giulia, 581749*  
*Giordano Elisabetta, 536265*

**Repo GitHub:** [https://github.com/Elisabetta99/BigData\\_Progetto1.git](https://github.com/Elisabetta99/BigData_Progetto1.git)

## Introduzione

In questa relazione illustriamo le diverse tecnologie utilizzate per l'analisi e l'elaborazione del dataset specifico *Amazon Fine Food Reviews*, che contiene circa 500.000 recensioni di prodotti gastronomici. Ogni riga del dataset comprende una serie di campi informativi come:

- ID,
- ProductId: identificatore univoco del prodotto,
- UserId: identificatore univoco dell'utente,
- ProfileName,
- HelpfulnessNumerator: numero di utenti che hanno trovato la recensione utile, HelpfulnessDenominator: numero di utenti che hanno valutato la recensione,
- Score: valutazione compresa tra 1 e 5,
- Time: timestamp della recensione espresso in Unix time,
- Summary: riassunto della recensione,
- Text: testo della recensione.

Nel contesto di questo progetto, sono stati progettati e realizzati tre diversi job per ciascuna delle seguenti tecnologie: *Map Reduce*, *Hive*, *Spark SQL* e *Spark Core*.

- **Job 1:** generare, per ciascun anno, i 10 prodotti che hanno ricevuto il maggior numero di recensioni e, per ciascuno di essi, le 5 parole con almeno 4 caratteri più frequentemente usate nelle recensioni (campo text), indicando, per ogni parola, il numero di occorrenze della parola.
- **Job 2 :** generare una lista di utenti ordinata sulla base del loro apprezzamento, dove l'apprezzamento di ogni utente è ottenuto dalla media dell'utilità (rapporto tra HelpfulnessNumerator e HelpfulnessDenominator) delle recensioni che hanno scritto, indicando per ogni utente il loro apprezzamento.
- **Job 3:** generare gruppi di utenti con gusti affini, dove gli utenti hanno gusti affini se hanno recensito con score superiore o uguale a 4 almeno 3 prodotti in comune, indicando, per ciascun gruppo, i prodotti condivisi. Il risultato deve essere ordinato in base allo UserId del primo elemento del gruppo e non devono essere presenti duplicati.

Prima della realizzazione dei Job, è stata effettuata una pulizia del dataset originale. In particolare sono state eliminate le colonne superflue, e tutti i caratteri come virgole, spazi bianchi, punti esclamativi, punti interrogativi, caratteri HTML dai singoli campi del dataset. In modo tale da non incorrere in errori soprattutto nel caso del Job1 per la ricerca delle parole nel campo text.

Gli script relativi a ciascun job sono disponibili nel repository GitHub dedicato al progetto. I dataset utilizzati per i test non sono stati inclusi nel repository per motivi di spazio.

I Job sono stati testati sia in ambiente **locale** che su **cluster AWS**.

## Indice

<b>Introduzione.....</b>	<b>1</b>
<b>Indice.....</b>	<b>2</b>
<b>Job1.....</b>	<b>3</b>
• Map Reduce.....	3
• Hive.....	5
• Spark Core.....	6
• Spark SQL.....	7
• Analisi dei tempi.....	9
<b>Job 2.....</b>	<b>11</b>
• Map Reduce.....	11
• Hive.....	13
• Spark SQL.....	14
• Spark Core.....	16
• Analisi dei tempi.....	17
<b>Job 3.....</b>	<b>19</b>
• Map Reduce.....	19
• Hive.....	21
• Spark Core.....	24
• Analisi dei tempi.....	25

## Job1

L'obiettivo del seguente Job è quello di restituire per ciascuno anno i dieci prodotti che hanno ricevuto il maggior numero delle recensioni, e per ciascuno di essi, le cinque parole con almeno quattro caratteri più frequentemente usate nelle recensioni, e per ogni parola il numero delle occorrenze della parola. Pertanto le uniche colonne del dataset necessarie in questo job sono Time, ProductId, Text.

- **Map Reduce**

L'implementazione MapReduce è stata svolta creando due script, uno script mapper ed uno reducer. Il mapper legge il file csv una riga per volta ed esegue uno split della riga in tre parti: time, productId, text. Ciascuna riga contiene il valore time nel formato Unix time, che viene convertito per ottenere l'anno e il testo che viene ripulito da eventuali segni di punteggiatura, spazi in eccesso ecc.

Il mapper per ciascun time, productId e text ( recensione) stampa una stringa contenente l'anno, il prodotto, il testo e l'intero 1. L'intero 1 viene utilizzato nel file reducer per contare il numero di recensioni per la coppia (anno, prodotto).

---

**Algorithm 1** Mapper Job 1

---

```
1: for line in stdin do  
2:   line ← line.strip()  
3:   productId, time, text ← line.split()  
4:   Output (productId, time, text, 1)  
5: end for
```

---

Infatti, lo script reducer, prende in input le stringhe stampate dal mapper ed utilizza dizionari opportunamente creati in modo tale che per ogni coppia anno, prodotto, si conti il numero delle recensioni e si concateni il testo di queste recensioni. Una volta fatto ciò con il ciclo for si selezionano per ogni anno, i dieci prodotti con numero più grande di recensioni. Questo risultato, viene passato in input ad una funzione aggiuntiva creata, words\_count, che per ogni (anno, prodotto), effettua lo split delle parole nelle recensioni, seleziona solo quelle con lunghezza  $\geq 4$  e restituisce un dizionario contenente parola, conteggio parola. Vengono selezionate successivamente solo le cinque parole con conteggio parola migliore e l'output finale sarà dato da anno, prodotto, numero recensioni, e le top 5 parole con il relativo conteggio.

---

**Algorithm 1** Reducer Job 1

---

```
1: Initialize reviews_year dictionary
2: for line in stdin do
3:   line  $\leftarrow$  line.strip()
4:   productId, time, text  $\leftarrow$  line.split()
5:   if time not in reviews_year then
6:     Initialize reviews_year[time] dictionary
7:   end if
8:   if productId not in reviews_year[time] then
9:     Initialize reviews_year[time][productId] dictionary
10:  end if
11:  reviews_year[time][productId][valore] += numeroRecensioni
12:  reviews_year[time][productId][testo] += text
13:
14: end for
15: for time in sorted(reviews_year) do
16:   for productId in sorted(reviews_year[time]) do
17:     seleziono i migliori 10 productId
18:     words.count(time,productId)  $\leftarrow$  dictionaryofwords,count
19:     select top five words
20:   end for
21:   Output (time, productId, numeroRecensioni topFivewords, count-
    Words)
22: end for
23: words.count(time,productId) crea funzione di supporto che per ogni
    words in text, seleziona le words con lunghezza almeno pari a 4 ed effet-
    tua il conteggio.
```

---

I due script utilizzati sono mapper.py e reducer.py e sono presenti all'interno del Repo GitHub.

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:

part-00000 ~/ProgettoBigData/output/outputProgettoFinal		
2012	B007JFMH8M	911
	cookie 338	
	cookies 324	
	soft 287	
	this 268	
	these 246	
	B005ZBZLT4	480
	coffee 245	
	this 166	
	have 96	
	that 87	
	like 84	

Sono stati, inoltre, calcolati i tempi di esecuzione di map reduce in locale e su cluster AWS:

- ❖ Tempo locale: 10.05 secondi
- ❖ Tempo cluster: 29 secondi

- **Hive**

Codice *hive.hql*:

```
DROP TABLE documents;
DROP TABLE info;
DROP TABLE output_job1;
DROP TABLE topTen;
DROP TABLE topFive;
DROP TABLE mergedTable;

CREATE TABLE documents(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 ',';

LOAD DATA LOCAL INPATH '/home/fabio/Scaricati/dataset/Reviews.csv' OVERWRITE INTO TABLE documents;

CREATE TABLE info AS
SELECT ProductId, YEAR(from_unixtime(CAST(time AS BIGINT))) AS time, text
FROM documents;

CREATE TABLE output_job1 AS
SELECT time, ProductId, val, parola, COUNT(*) AS conteggio_parola
FROM (
    SELECT time, ProductId, val, split(testo_concatenato, ' ') AS parole
    FROM (
        SELECT time, ProductId, COUNT(ProductId) AS val, concat_ws(' ', collect_list(text)) AS testo_concatenato
        FROM info
        GROUP BY time, ProductId
    ) t
) t2
LATERAL VIEW explode(parole) exploded_table AS parola
WHERE parola != '' AND LENGTH(parola)>=4
GROUP BY time, ProductId, val, parola;

CREATE TABLE topTen AS
SELECT time, productId, testoConcatenato, valoretesto_concatenato
FROM(
    SELECT *, row_number()
        over ( PARTITION BY time ORDER BY time, val DESC) as dimensione
    FROM (
        SELECT time, productId, val,concat_ws(' ', collect_list(parola)) as testoConcatenato, concat_ws(' ', collect_list(CAST(
        conteggio_parola AS STRING))) as valoretesto_concatenato
        FROM output_job1
        GROUP BY time, productId, val
    ) p
    ORDER BY time DESC, dimensione) dimensione_table
WHERE dimensione_table.dimensione < 11;

CREATE TABLE topFive AS
SELECT time, ProductId, parola, conteggio_parola
FROM (
    SELECT time, ProductId, parola, conteggio_parola,
        ROW NUMBER() OVER (PARTITION BY time, ProductId ORDER BY conteggio_parola DESC) AS rn
    FROM output_job1
) t
WHERE rn <= 5;

CREATE TABLE mergedTable AS
SELECT tf.time, tf.productId, tf.parola, tf.conteggio_parola
FROM topFive tf
JOIN (
    SELECT time, productId
    FROM topTen
    GROUP BY time, productId
) tt ON tt.time = tf.time AND tt.productId = tf.productId;

SELECT * FROM mergedTable;

DROP TABLE documents;
DROP TABLE info;
DROP TABLE output_job1;
DROP TABLE topTen;
DROP TABLE topFive;
DROP TABLE mergedTable;
```

In Hive, viene popolata una tabella info, contenente solamente le informazioni necessarie per il job, productId, time convertito dal formato unix time, text. La tabella output\_job1 permette per ogni anno, prodotto, di raggruppare le recensioni e di calcolarne il numero. La tabella topTen restituisce i dieci prodotti migliori per ogni anno in base al numero di recensioni. Nella terza query, quella che crea la tabella topFive vengono selezionate per ogni anno e prodotto, le cinque parole con lunghezza  $\geq 4$  con più occorrenze. Infine viene effettuato un merge tra le tabelle topTen e topFive in base ad anno e prodotto, ottenendo così i dieci prodotti migliori per ogni anno, con le cinque parole più occorrenti. Infine vengono eliminate tutte le tabelle create.

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:

part-00000 ~/ProgettoBigData/outputHive			
2012	B007JFMH8M	cookies	361
2012	B007JFMH8M	cookie	354
2012	B007JFMH8M	soft	326
2012	B007JFMH8M	this	275
2012	B007JFMH8M	these	253
2012	B005ZBZLT4	coffee	265
2012	B005ZBZLT4	this	171
2012	B005ZBZLT4	have	96
2012	B005ZBZLT4	that	88
2012	B005ZBZLT4	like	84

Sono stati, inoltre, calcolati i tempi di esecuzione di hive in locale e su cluster AWS:

- ❖ Tempo locale: 143 secondi
- ❖ Tempo cluster: 102 secondi

## ● *Spark Core*

Una volta caricato il file csv nell'RDD di input sono state eseguite, come nelle precedenti implementazioni, operazioni di selezione delle relative righe necessarie, time, productId, text e operazioni per la conversione dell'anno da unix time. Da questo RDD è stato creato un nuovo RDD contenente come chiave l'anno e il prodotto e come valore l'intero 1. L'intero 1 è stato utilizzato nel reduceByKey per poter contare per ogni anno le occorrenze di quel prodotto.

Successivamente sono state eseguite operazioni di ordinamento e raggruppamento per ottenere un RDD di output contenente per ogni anno i dieci prodotti con più recensioni. Si è creato inoltre un altro RDD da quello di partenza per contare le occorrenze delle parole nelle recensioni per ogni coppia (anno, prodotto). Si sono selezionate solo le parole con lunghezza  $\geq 4$ , ottenendo così tramite operazioni di conteggio, ordinamento, raggruppamento le cinque migliori parole per ogni (anno, prodotto). si è infine effettuato il merge tra i migliori dieci prodotti per ogni anno e le

migliori cinque parole per quella stessa coppia (anno, prodotto), ottenendo così il risultato finale desiderato.

---


**Algorithm 1** Spark core Job 1


---

```
1: lines_RDD  $\leftarrow$  inputfile
2: year_cleaned_text_RDD  $\leftarrow$  time, productId, text
3: year_product_rec  $\leftarrow$  ((time, productId), recensione)
4: year_product_RDD  $\leftarrow$  (time, productId), 1
5: year_product_sum_RDD  $\leftarrow$  reduceByKey(year_product_RDD) and sort
6: output_RDD  $\leftarrow$  (time, (productId, countRec))
7: output_year_top10_product_RDD  $\leftarrow$  groupByKey(year_product_sum_RDD)
   and get top 10 product
8: list_word  $\leftarrow$  da year_product_rec ottengo ((time, productId), words with
   len maggiore o uguale di 4
9: word_count  $\leftarrow$  (time, (productId, word, countWord))
10: final_out  $\leftarrow$  groupByKey(word_count) and sort
11: get top 5 words for (time, productId)
12: join top 10 product by time with top 5 words for (time, productId) and
   return top ten product with top 5 words.
```

---

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```
Apri  part-00000
~/ProgettoBigData/outputSpark
(('2012', 'B007JFMH8M'),
 [('cookie', 536),
 ('cookies', 505),
 ('these', 418),
 ('this', 417),
 ('soft', 415)])
(('2012', 'B005ZBZLT4'),
 [('coffee', 669),
 ('this', 466),
 ('that', 348),
 ('have', 287),
 ('like', 265)])
```

Sono stati, inoltre, calcolati i tempi di esecuzione di spark core in locale e su cluster AWS:

- ❖ Tempo locale: 103.94577050209045 secondi
- ❖ Tempo cluster: 66.65 secondi

- **Spark SQL**

Codice *sparkSQL.py*:

```
#!/usr/bin/env python3
"""job1sparksql.py"""

import argparse
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
import time
from datetime import datetime
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, split, explode, count, from_unixtime, year, desc, length, lower, regexp_replace, col

parser = argparse.ArgumentParser()
parser.add_argument("--input_path", type=str, help="Input file path")
parser.add_argument("--output_path", help="Output file path")
# parse arguments
args = parser.parse_args()
input_filepath, output_filepath = args.input_path, args.output_path

# initialize SparkSession with the proper configuration
spark = SparkSession \
    .builder \
    .appName("Spark SQL JOB 1") \
    .getOrCreate()

# Define the schema structure
schema = StructType([
    StructField("Id", IntegerType(), True),
    StructField("productId", StringType(), True),
    StructField("userId", StringType(), True),
    StructField("profileName", StringType(), True),
    StructField("helpfulnessNumerator", IntegerType(), True),
    StructField("helpfulnessDenominator", IntegerType(), True),
    StructField("score", StringType(), True),
    StructField("time", IntegerType(), True),
    StructField("summary", StringType(), True),
    StructField("text", StringType(), True)
])

start_time = time.time()

# read from csv file
input_df = spark.read.option("quote", "\"") \
    .csv(input_filepath, header=True, schema=schema) \
    .cache()

# convert to time
input_df = input_df.withColumn("time", input_df["time"].cast("timestamp"))
input_df = input_df.withColumn("year", year(from_unixtime(input_df["time"].cast("bigint")))).cache()

# TOP10 PER OGNI ANNO
count_reviews = input_df.groupBy("year", "productId").agg(count("*").alias("conteggio_recensioni")).cache()

window = Window.partitionBy("year").orderBy(desc("conteggio_recensioni"))

top10prod = count_reviews.withColumn("row_number", row_number().over(window))
top10prod = top10prod.filter(top10prod["row_number"] <= 10).drop("row_number").cache()

# top 5 words per ogni prodotto di ogni anno. faccio lo split di ogni parola nelle recensioni
searchWords = input_df.withColumn("words", explode(split(input_df["text"], " "))).cache()

searchWords = searchWords.filter(length(searchWords["words"]) > 3).groupBy("year", "productId", "words").agg(count("*").alias("conteggio_parole"))

window = Window.partitionBy("year", "productId").orderBy(desc("conteggio_parole"))

topWords = searchWords.withColumn("row_number", row_number().over(window))
topWords = topWords.filter(topWords["row_number"] <= 5).drop("row_number").cache()

# unisco le info
outputJob = top10prod.join(topWords, ["year", "productId"]).cache()

end_time = time.time()

outputJob.write.csv(output_filepath, mode="overwrite")
# visualizzo il tempo impiegato
print("Time : ", end_time - start_time)
```

Una volta creato lo schema e convertito il tempo, viene effettuato il conteggio delle recensioni per ogni anno e per ogni prodotto utilizzando `groupBy()`, e i risultati vengono memorizzati nella variabile `count_reviews`. Viene definita una finestra di partizionamento per raggruppare i dati per anno e ordinare per conteggio delle recensioni in ordine decrescente. Si calcola poi la top 10 dei prodotti per ogni anno utilizzando `row_number()` e la finestra di partizionamento, filtrando solo le righe con numero di riga inferiore o uguale a 10, e i risultati vengono memorizzati nella



variabile top10prod . Viene effettuata una suddivisione delle parole in ciascuna recensione utilizzando split() e explode(), creando nuove righe per ogni parola. Le parole devono avere una lunghezza superiore a 3 caratteri. Si effettua il conteggio delle parole per ogni anno, prodotto e parola utilizzando groupBy(), e i risultati vengono memorizzati nella variabile SearchWords. Si definisce una nuova finestra di partizionamento per raggruppare i dati per anno, prodotto e ordinare per conteggio delle parole in ordine decrescente. Si selezionano le cinque parole con più occorrenze per ogni anno, prodotto. Infine si uniscono le informazioni della top10 prodotti e top 5 parole

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```

Apri  part-00000.csv
~/ProgettoBigData/outputsparksq

2012,B007JFMH8M,911,cookie,502
2012,B007JFMH8M,911,cookies,481
2012,B007JFMH8M,911,soft,408
2012,B007JFMH8M,911,this,399
2012,B007JFMH8M,911,these,395
2012,B005ZBZLT4,480,coffee,599
2012,B005ZBZLT4,480,this,404
2012,B005ZBZLT4,480,that,299
2012,B005ZBZLT4,480,have,251
2012,B005ZBZLT4,480,like,225

```

Sono stati, inoltre, calcolati i tempi di esecuzione di spark sql in locale e su cluster AWS:

- ❖ Tempo locale: 4.505791187286377 secondi
- ❖ Tempo cluster: 1.08 secondi

### ● *Analisi dei tempi*

Di seguito vengono riportate le tabelle e grafici di confronto dei tempi di esecuzione in locale e su cluster di Map reduce, Hive, SparkCore e SparkSQL, con dimensioni crescenti dell'input

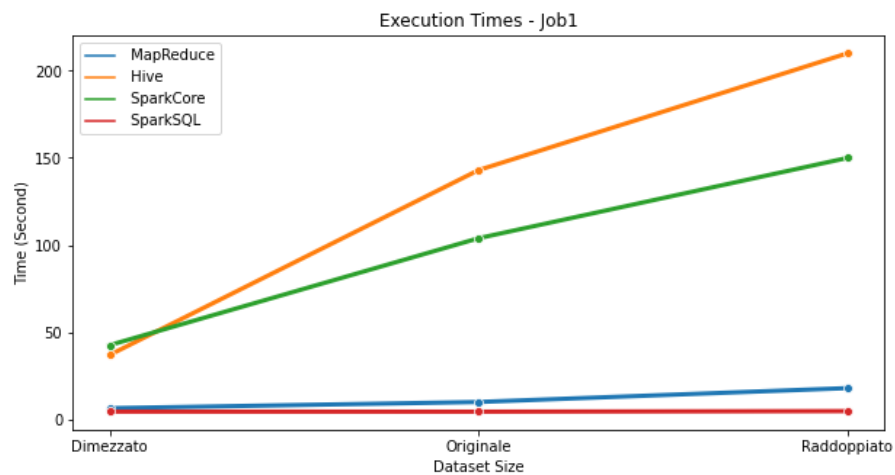
In particolare:

- ❖ Nel file dimezzato sono presenti 284227 righe
- ❖ Nel file originale sono presenti 568455 righe
- ❖ Nel file raddoppiato sono presenti 1136909 righe

Tempi ottenuti in **locale**:

	MapReduce	Hive	SparkCore	SparkSQL
Dimezzato	6.50	37	42.63	4.53
Originale	10.05	143	103.94	4.50
Raddoppiato	18.00	210	150.00	4.82

Dal grafico si può osservare che l'implementazione Spark SQL è la più veloce e Hive e Spark Core sono le più lente. Il tempo di esecuzione in tutte le implementazioni cresce al crescere della dimensione del file preso in input, tranne in Spark SQL che rimane più o meno costante.



Tempi ottenuti sul **cluster AWS**:

	MapReduce	Hive	SparkCore	SparkSQL
Dimezzato	27	80	43.31	0.96
Originale	29	102	66.65	1.08
Raddoppiato	30	109	88.75	1.15

Anche in questo caso possiamo notare che l'implementazione in SparkSQL è la più veloce, mentre l'implementazione in Hive è la più lenta. Il tempo di esecuzione in cluster non varia di molto in base alla dimensione dell'input in MapReduce e SparkSQL mentre si discosta di più in Hive e Spark Core.



## Job 2

L'obiettivo di questo Job è creare una lista di utenti ordinata in base al loro grado di apprezzamento. Il grado di apprezzamento viene calcolato prendendo in considerazione la media delle valutazioni di utilità delle recensioni scritte da ciascun utente. Il risultato finale fornirà, per ogni utente, il grado di apprezzamento.

- **Map Reduce**

L'implementazione di MapReduce è stata svolta, come per il Job1, creando due script, uno per il mapper e uno per il reducer.

La classe "*mapper.py*" è responsabile di elaborare le righe di input e produrre coppie chiave-valore. Successivamente, estrae l'*ID utente*, *HelpfulnessNumerator* e *HelpfulnessDenominator* dalle rispettive colonne. Utilizzando queste informazioni, viene calcolato il valore di utilità come il rapporto tra il numeratore e il denominatore. Infine, l'output viene emesso con l'*ID utente* come chiave e il valore di utilità come valore associato.

Pseudocodice *mapper.py*:

---

### Algorithm 1 Mapper Job 2

---

```

1: for line in stdin do
2:   line ← line.strip()
3:   userId, help_num, help_den ← line.split()
4:   if help_den ≠ 0 then
5:     usefulness ← help_num / help_den
6:     Output(userId, usefulness)
7:   end if
8: end for

```

---

La classe "*reducer.py*" si occupa di aggregare i dati prodotti dal Mapper, eseguire il calcolo dell'apprezzamento medio e ordinare gli utenti in base all'apprezzamento. In particolare, attraverso l'utilizzo del dizionario *userAppreciation* come struttura dati, viene gestita l'aggregazione dei dati per ogni ID utente. Se l'ID utente è già presente nel dizionario, vengono aggiornati la somma delle valutazioni di utilità e il conteggio delle recensioni. Se l'ID utente è nuovo, viene creato un nuovo record con i valori iniziali.

Viene quindi calcolato l'apprezzamento medio per ogni utente dividendo la somma delle valutazioni di utilità per il numero delle recensioni scritte dall'utente stesso. I valori di apprezzamento medio vengono quindi assegnati al dizionario.

Infine, gli utenti vengono ordinati in base all'apprezzamento medio in ordine decrescente utilizzando una funzione di ordinamento *sorted*. L'output finale viene emesso con l'ID utente e l'apprezzamento medio corrispondente.

Pseudocodice *reducer.py*:

---

**Algorithm 1** Reducer Job2

---

```

1: Initialize userAppreciation dictionary
2: for line in stdin do
3:   line ← line.strip()
4:   userId, usefulness ← line.split()
5:   if userId not in userAppreciation then
6:     userAppreciation[userId] dictionary
7:   else
8:     userAppreciation[userId]['usefulness_sum'] += usefulness
9:     userAppreciation[userId]['review_count'] += 1
10:  end if
11: end for
12: for user in userAppreciation do
13:   appreciation =  $\left( \frac{\text{usefulness\_sum}}{\text{review\_count}} \right)$ 
14: end for
15: sort(userAppreciation)
16: for user, appreciation in sorted(userAppreciation) do
17:   Output(user, appreciation)
18: end for

```

---

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```

part-00000
~/PycharmProjects/BigData_Progetto1/Job2/MapReduce/output_mr2
1 AZZY649VVAHQ5 1.0
2 AZZUQYE2C1LNI 1.0
3 AZZU5BA2CHYVF 1.0
4 AZZU4D6TZ2L6J 1.0
5 AZZU1VE08KUXH 1.0
6 AZZRFMU060L7J 1.0
7 AZZR020VST1WG 1.0
8 AZZOMF6HZYFL7 1.0
9 AZZMDW27MUJR6 1.0
10 AZZFLKZ198VZV 1.0

```

In generale, l'apprezzamento degli utenti va da un valore minimo 0 ad un valore massimo 1.

Sono stati, inoltre, calcolati i tempi di esecuzione di map reduce in locale e su cluster AWS:

- ❖ Tempo locale: 10 secondi
- ❖ Tempo cluster: 26.23 secondi

- *Hive*

```
CREATE TABLE IF NOT EXISTS user_reviews (  
  Id int,  
  ProductId string,  
  UserId string,  
  ProfileName string,  
  HelpfulnessNumerator int,  
  HelpfulnessDenominator int,  
  Score int,  
  Time int,  
  Summary string,  
  Text string  
)  
COMMENT 'User Reviews Table'  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ',';  
  
LOAD DATA LOCAL INPATH '/home/elisabetta/BigDataCluster/dataset/Reviews.csv' overwrite INTO TABLE user_reviews;  
  
CREATE TABLE user_appreciation AS  
SELECT  
  UserId,  
  SUM(CASE WHEN HelpfulnessDenominator != 0 THEN HelpfulnessNumerator / HelpfulnessDenominator ELSE 0 END) / COUNT(*) AS appreciation  
FROM user_reviews  
GROUP BY UserId;
```

Inizialmente, viene definita una tabella chiamata "*user\_reviews*" che corrisponde ai dati delle recensioni degli utenti e viene creata una tabella chiamata "*user\_appreciation*" che contiene l'ID utente e il suo apprezzamento.

L'apprezzamento è stato calcolato come segue: nella query SELECT, la funzione di aggregazione SUM viene utilizzata insieme a una clausola CASE per calcolare la somma dei valori di utilità (HelpfulnessNumerator / HelpfulnessDenominator), viene quindi calcolata la media dividendo la somma per il conteggio totale delle recensioni scritte dall'utente. La clausola GROUP BY viene utilizzata per raggruppare i risultati per ID utente.

```

CREATE TABLE sorted_users AS
SELECT UserId, appreciation
FROM user_appreciation
ORDER BY appreciation DESC, UserId DESC;

INSERT OVERWRITE DIRECTORY 'file:///home/elisabetta/Scrivania/BigData/output_hive2'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT *
FROM sorted_users;

```

Una volta ottenuti gli apprezzamenti per ciascun utente creiamo una tabella "sorted\_users" che conterrà gli utenti ordinati in base all'apprezzamento medio. L'ordinamento viene eseguito in ordine decrescente utilizzando la clausola ORDER BY sull'apprezzamento.

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```

1 AZZY649VYAHQS,1.0
2 AZZUQYE2C1LNI,1.0
3 AZZU5BA2CHYVF,1.0
4 AZZU4D6TZ2L6J,1.0
5 AZZU1VE08KUXH,1.0
6 AZZRFRMU060L7J,1.0
7 AZZR020VST1WG,1.0
8 AZZOMF6HZYFL7,1.0
9 AZZMDW27MUJR6,1.0
10 AZZFLKZ198VZV,1.0

```

Sono stati, inoltre, calcolati i tempi di esecuzione di hive in locale e su cluster AWS:

- ❖ Tempo locale: 14 secondi
- ❖ Tempo cluster: 0.31 secondi

- **Spark SQL**

Inizialmente, viene creato un parser per leggere gli argomenti di input e output e viene inizializzata la sessione Spark.

Per calcolare l'apprezzamento per ogni utente viene effettuato un controllo sulla colonna *HelpfulnessDenominator* per verificare se è diversa da zero. Se la condizione è soddisfatta, viene calcolato il rapporto tra *HelpfulnessNumerator* e *HelpfulnessDenominator*, ottenendo in questo modo l'utilità della recensione; in caso contrario, viene assegnato il valore zero. Il risultato viene aggregato raggruppando per l'ID utente utilizzando il metodo *groupBy* e calcolando la media dell'apprezzamento utilizzando le funzioni di aggregazione *sum* e *count*. Il DataFrame risultante viene denominato *user\_appreciation\_df*.

```

# Creazione parser
parser = argparse.ArgumentParser()
parser.add_argument("--input_path", type=str, help="Input file path")
parser.add_argument("--output_path", type=str, help="Output file path")

# Argomenti parser
args = parser.parse_args()
input_filepath, output_filepath = args.input_path, args.output_path

# Inizializzazione della sessione Spark
spark = SparkSession.builder.appName("UserAppreciation").getOrCreate()

# Avvia il timer
start_time = time.time()

# Lettura del file CSV del dataset Amazon Fine Food
reviews_df = spark.read.csv(input_filepath, header=True)

# Calcolo dell'apprezzamento per ogni utente
utility = when(col("HelpfulnessDenominator") != 0, col("HelpfulnessNumerator") / col("HelpfulnessDenominator")).otherwise(0)
user_appreciation_df = reviews_df.groupBy("UserId") \
    .agg((sum(utility) / count("*")).alias("Appreciation"))

```

Successivamente, viene ordinata la lista di utenti in base all'apprezzamento utilizzando il metodo `orderBy` su `user_appreciation_df`. L'ordinamento viene effettuato in ordine decrescente in base all'apprezzamento.

```

# Ordinamento della lista di utenti in base all'apprezzamento
sorted_users = user_appreciation_df.orderBy("Appreciation", ascending=False)

# Visualizzazione dei risultati
sorted_users.show()

# Salvataggio dei risultati in un file di output
sorted_users.write.csv(output_filepath)

# Calcola il tempo di esecuzione
execution_time = time.time() - start_time
print("Tempo di esecuzione: %.2f secondi" % execution_time)

```

Il codice utilizza le funzioni di manipolazione dei DataFrame offerte da Spark SQL, l'utilizzo di funzioni di aggregazione e ordinamento. Inoltre, viene misurato il tempo di esecuzione dell'applicazione utilizzando il modulo `time` di Python.

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```
1 AZZY649VYAHQS,1.0
2 AZZUQYE2C1LNI,1.0
3 AZZU5BA2CHYVF,1.0
4 AZZU4D6TZ2L6J,1.0
5 AZZU1VE08KUXH,1.0
6 AZZRFMU060L7J,1.0
7 AZZR020VST1WG,1.0
8 AZZOMF6HZYFL7,1.0
9 AZZMDW27MUJR6,1.0
10 AZZFLKZ198VZV,1.0
```

Sono stati, inoltre, calcolati i tempi di esecuzione di spark SQL in locale e su cluster AWS:

- ❖ Tempo locale: 11.09 secondi
- ❖ Tempo cluster: 0.16 secondi

- **Spark Core**

Una volta caricato il file .csv nell'RDD di input, è stata eseguita una mappatura dell'RDD per separare i campi e creare una tupla con l'*ID utente* come chiave e una tupla contenente i valori di *HelpfulnessNumerator* e *HelpfulnessDenominator*. Successivamente, viene applicata una trasformazione *mapValues* per calcolare l'utilità e il numero di recensioni per ogni utente.

Viene eseguita una riduzione per chiave *reduceByKey* per aggregare le informazioni sull'utilità e il numero di recensioni per ogni utente.

Viene applicata una trasformazione *mapValues* per calcolare l'apprezzamento medio dividendo l'utilità totale per il numero totale di recensioni.

L'RDD risultante viene ordinato in base all'apprezzamento in ordine decrescente utilizzando il metodo *sortBy*.

L'implementazione di Spark Core è stata quindi effettuata tramite operazioni di pre-elaborazione, filtraggio e raggruppamento.



Pseudocodice *SparkRDD.py*:

---

**Algorithm 1** Sistema di elaborazione

---

```
1: input_RDD ← input file
2: user_helpfulness_RDD ← (userId, (HelpfulnessNumerator, Helpfulness-
   Denominator))
3: user_utility_reviews_RDD ← (userId, (utility, num_reviews))
4: user_total_utility_reviews_RDD ← (userId, (total_utility, total_reviews))
5: user_appreciation_RDD ← (userId, appreciation)
6: output_RDD ← (userId, appreciation), sort userId on appreciation
```

---

Di seguito vengono mostrate le prime 10 righe dell'output ottenuto:



```
1 ('AZZY649VYAHQS', 1.0)
2 ('AZZUQYE2C1LNI', 1.0)
3 ('AZZU5BAZCHYVF', 1.0)
4 ('AZZU4D6TZ2L6J', 1.0)
5 ('AZZU1VE08KUXH', 1.0)
6 ('AZZRFMU060L7J', 1.0)
7 ('AZZR02OVST1WG', 1.0)
8 ('AZZOMF6HZYFL7', 1.0)
9 ('AZZMDW27MUJR6', 1.0)
10 ('AZZFLKZ198VZV', 1.0)
```

Sono stati, inoltre, calcolati i tempi di esecuzione di spark Core in locale e su cluster AWS:

- ❖ Tempo locale: 2.46 secondi
- ❖ Tempo cluster: 14.69 secondi

- **Analisi dei tempi**

Di seguito vengono riportate le tabelle e grafici di confronto dei tempi di esecuzione in locale e su cluster di Map reduce, Hive, SparkCore e SparkSQL, con dimensioni crescenti dell'input.

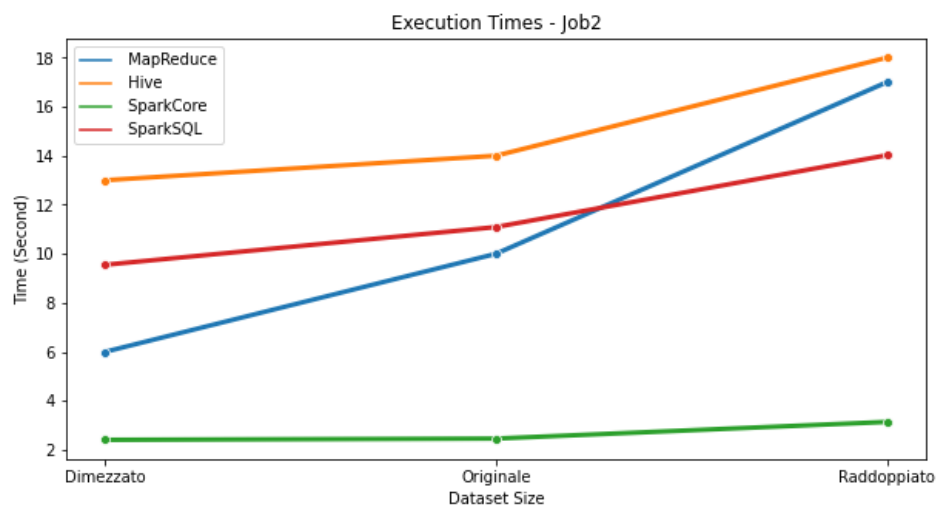
In particolare:

- ❖ Nel file dimezzato sono presenti 284227 righe
- ❖ Nel file originale sono presenti 568455 righe
- ❖ Nel file raddoppiato sono presenti 1136909 righe

Tempi ottenuti in **locale**:

	MapReduce	Hive	SparkCore	SparkSQL
Dimezzato	6	13	2.41	9.55
Originale	10	14	2.46	11.09
Raddoppiato	17	18	3.14	14.02

Dal grafico possiamo notare che l'implementazione in SparkCore è la più veloce e Hive è la più lenta. Il tempo di esecuzione in tutte le implementazioni cresce al crescere della dimensione del file preso in input.



Tempi ottenuti sul **cluster AWS**:

	MapReduce	Hive	SparkCore	SparkSQL
Dimezzato	25.00	0.29	13.44	0.11
Originale	26.23	0.31	14.69	0.16
Raddoppiato	28.65	0.39	15.53	0.24

In questo caso possiamo notare che le implementazioni in SparkSQL e Hive sono le più veloci (sovrapposte), mentre l'implementazione in Map Reduce è la più lenta. Il

tempo di esecuzione in cluster di tutte le implementazioni non cresce di molto all'aumentare della dimensione del file di input.



### Job 3

Lo scopo principale di questo Job consiste nella creazione di cluster di utenti che condividono interessi simili. Gli utenti sono considerati affini se hanno recensito almeno tre prodotti in comune con un punteggio di 4 o superiore. Per ogni cluster generato, sarà indicata la lista dei prodotti condivisi. L'output ottenuto verrà ordinato in base all'ID dell'utente presente come primo elemento del gruppo, e si garantirà l'assenza di duplicati.

- **Map Reduce**

Nella fase del mapper, vengono estratti i valori dell'*UserId*, del *ProductId* e dello *Score*. Se il valore dello *Score* è maggiore o uguale a 4, il mapper emette una coppia chiave-valore, nel formato (*userId*, *productId*), in modo da poter poi raggruppare le recensioni per utente.

Pseudocodice *mapper.py*:

---

**Algorithm 1** Mapper Job 3

---

```

1: for line in stdin do
2:   line ← line.strip()
3:   userId, productId, score ← line.split()
4:   if score ≥ 4 then
5:     Output (userId, productId)
6:   end if
7: end for

```

---

Nella fase del reducer, creiamo un dizionario *userProduct* che memorizza i prodotti recensiti da ogni utente. La funzione *split()* viene nuovamente utilizzata per dividere la riga di input nelle due chiavi *userId* e *productId*. La chiave *userId* viene utilizzata

per accedere al dizionario e viene aggiunto il productId all'insieme dei prodotti recensiti dall'utente corrispondente. Dopo aver letto tutti i dati, si procede a filtrare gli utenti che hanno recensito almeno tre prodotti.

Successivamente, vengono individuati gli utenti con almeno tre prodotti in comune tra di loro. Per far questo utilizziamo un dizionario *affinityGroup* che si occupa di memorizzare i gruppi di affinità, dove ogni gruppo è rappresentato da una coppia contenente una lista di userId e l'insieme di prodotti comuni. Vengono utilizzate le funzioni *enumerate()* e *intersection()* per confrontare i prodotti recensiti da ogni utente e verificare se soddisfano i criteri di affinità.

Utilizziamo le funzioni *sorted()* e *join()* per ordinare e formattare correttamente i risultati.

In fine, viene verificato se gli utenti nel gruppo sono già stati processati. Se il gruppo non contiene utenti che sono già presenti nell'insieme processedUsers, il gruppo viene aggiunto alla lista uniqueGroups insieme ai prodotti comuni. Questo garantisce che solo i gruppi unici vengano conservati, evitando duplicati basati sugli utenti.

Pseudocodice *reducer.py*:

---

**Algorithm 1** Reducer Job 3

---

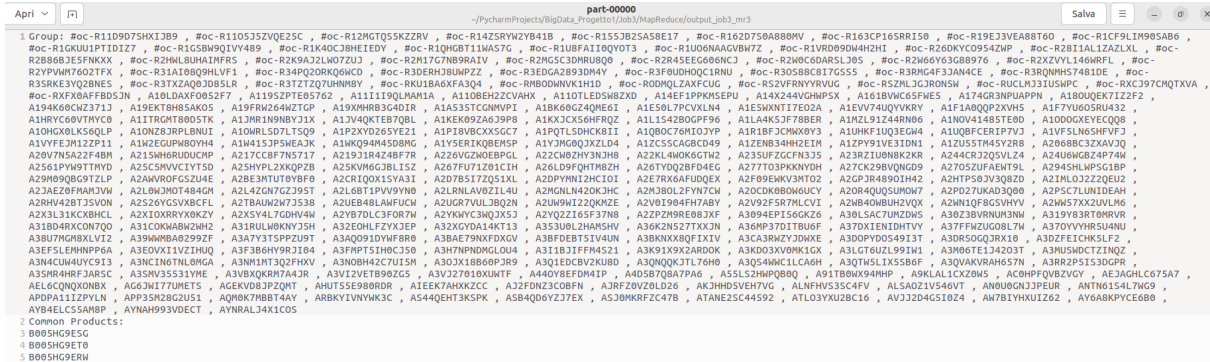
```

Initialize userProducts dictionary
2: Initialize affinityGroups dictionary
   for line in stdin do
4:   userId, productId ← line.split()
     Initialize userProducts[userId].add(productId)
6: end for
Initialize filteredUsers list
8: for userId, products in userProducts.items() do
   if len(products) ≥ 3 then
10:   filteredUsers.append(userId)
   end if
12: end for
   for i, user1 in enumerate(filteredUsers) do
14:   group ← {user1}
     commonProducts ← userProducts[user1]
16:   for i, user2 in enumerate(filteredUsers[i + 1:]) do
     if len(commonProducts.intersection(userProducts[user2])) ≥ 3
   then
18:     group.add(user2)
       commonProducts.intersection(userProducts[user2])
20:   end if
   end for
22:   affinityGroups.append((sorted(list(group)), commonProducts))
   end for
24: sortedGroups ← sorted affinityGroups on first UserId
   processedUsers ← set()
26: Initialize uniqueGroups list
   for group, commonProducts in sortedGroups do
28:   if not any(user in processedUsers for user in group) then
     uniqueGroups.append((group, commonProducts))
30:     processedUsers.update(group)
   end if
32: end for
   for group, commonProducts in uniqueGroups do
34:   Output(group, commonProducts)
   end for

```

---

Di seguito viene mostrato il primo gruppo con i prodotti in comune:



Sono stati, inoltre, calcolati i tempi di esecuzione di map reduce in locale e su cluster AWS:

- ❖ Tempo locale: 9.5 secondi
- ❖ Tempo cluster: 39 secondi

## ● *Hive*

```
CREATE TABLE IF NOT EXISTS reviews_job3 (  
    Id int,  
    ProductId string,  
    UserId string,  
    ProfileName string,  
    HelpfulnessNumerator int,  
    HelpfulnessDenominator int,  
    Score int,  
    Time int,  
    Summary string,  
    Text string  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ',';  
  
LOAD DATA LOCAL INPATH '/home/elisabetta/Scrivania/Reviews.csv' OVERWRITE INTO TABLE reviews_job3;  
  
-- Creazione della tabella filtered_reviews (filtro prodotti con score >= 4)  
CREATE TABLE IF NOT EXISTS filtered_reviews AS  
SELECT UserId, ProductId, Score  
FROM reviews_job3  
WHERE Score >= 4;
```

Per prima cosa viene creata una tabella *reviews\_job3* con le colonne corrispondenti ai campi del file di input *Reviews.csv* e vengono caricati dalla directory locale. Successivamente, viene creata una tabella *filtered\_reviews* che contiene gli *UserId*, *ProductId* e *Score* dei record con *Score* maggiore o uguale a 4.

```
-- Creazione della tabella user_groups per gli utenti con 3 prodotti in comune
CREATE TABLE IF NOT EXISTS user_groups AS
SELECT a.UserId AS user1, collect_set(b.UserId) AS users, collect_set(b.ProductId) AS common_products
FROM (
    SELECT DISTINCT UserId, ProductId
    FROM filtered_reviews
) a
JOIN (
    SELECT DISTINCT UserId, ProductId
    FROM filtered_reviews
) b ON a.ProductId = b.ProductId AND a.UserId < b.UserId
GROUP BY a.UserId
HAVING size(collect_set(b.ProductId)) >= 3;
```

A questo punto creiamo una tabella *user\_groups* selezionando gli utenti e i prodotti unici dalla tabella *filtered\_reviews*. Successivamente, vengono effettuate due *join* sulla stessa tabella, in modo che i record siano combinati solo se hanno lo stesso *ProductId* ma *UserId* diversi e l'*UserId* del primo record è inferiore a quello del secondo; ciò garantisce che ogni coppia di utenti condivida almeno un prodotto e che il gruppo venga rappresentato solo una volta. La funzione *collect\_set* viene utilizzata per raggruppare gli *UserId* e i *ProductId* comuni all'interno di un insieme. Infine, i gruppi vengono raggruppati per *UserId* e solo quelli con almeno 3 prodotti comuni vengono mantenuti. La tabella risultante contiene quindi il primo utente (*user1*) di ogni gruppo, gli utenti (*users*) appartenenti al gruppo e i prodotti (*common\_products*) che condividono.

```
-- Creazione della tabella affinity_groups con un unico gruppo contenente tutti gli utenti con gli stessi prodotti in comune
CREATE TABLE IF NOT EXISTS affinity_groups AS
SELECT collect_set(user1) AS users, common_products
FROM user_groups
GROUP BY common_products;
```

La tabella *affinity\_groups* viene utilizzata per rappresentare un unico gruppo contenente tutti gli utenti con gli stessi prodotti in comune. Viene utilizzata una query che raggruppa gli utenti in base ai prodotti in comune, utilizzando la funzione *collect\_set* per ottenere un set univoco di *UserId* per ogni gruppo di prodotti.

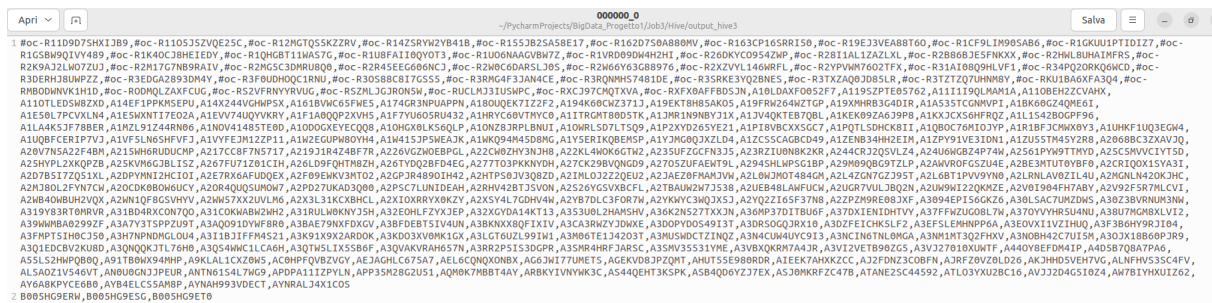
```
-- Ordinamento dei gruppi in base all'UserId del primo utente del gruppo
CREATE TABLE IF NOT EXISTS sorted_groups AS
SELECT users, common_products
FROM affinity_groups
WHERE size(users) > 1
ORDER BY users[0];

-- Scrittura del risultato nella cartella di output
INSERT OVERWRITE DIRECTORY 'file:///home/elisabetta/Scrivania/output_hive3'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT *
FROM sorted_groups;
```

Infine, viene creata la tabella *sorted\_groups* che contiene i gruppi ordinati in base all'UserId del primo utente del gruppo, per far questo utilizziamo la clausola ORDER BY sulla colonna users[0] per ordinare i gruppi.

I risultati vengono scritti nella directory di output specificata, all'interno di un file.

Di seguito viene mostrato il primo gruppo con i prodotti in comune:



```
1 #oc-R11D9075H1J39,#oc-R11053J5VQE25C,#oc-R12MGTQ5SKZ2RV,#oc-R14ZSRVY2YB418,#oc-R1553B25A5BE17,#oc-R162D750AB80MV,#oc-R163CP165RR150,#oc-R19E33VEA88T60,#oc-R1CF9LI9M95A86,#oc-R1GKU1PTID1Z7,#oc-
R1CSB9Q1V489,#oc-R1K40CJ8HEEDY,#oc-R1QHCBT1WASTG,#oc-R1UBFA11QOYOT3,#oc-R1U06NAGVBM7Z,#oc-R1V0R9904HHDH1,#oc-R26DKVC095A2WP,#oc-R2B1IAL1ZAZLXL,#oc-R2B6B0JESFNKX,#oc-R2HML8UHA1MFRS,#oc-
R2K9A3J2LM0ZUJ,#oc-R2M17G7NB9RAIV,#oc-R2MCS3DMRU8Q,#oc-R2R4SEEG66NCJ,#oc-R2M6C6DARSL3B5,#oc-R2M6V63G8B976,#oc-R2XZVYL146MRFL,#oc-R2YPMW7602TFX,#oc-R31A10BQ9HLVF1,#oc-R34PQ20RQ6WCD,#oc-
R3DERH3BUMPZJ,#oc-R3EDGA2893DM4Y,#oc-R3F8UDHQC1RNU,#oc-R30588CB17C555,#oc-R3RMG4F3JAN4CE,#oc-R3RQNMHS7481DE,#oc-R3SKE3YQ2BNE5,#oc-R3TXAQ63D85LR,#oc-R3TZT2Q7UHNMBY,#oc-RKU1BA6XFA3Q4,#oc-
RMB0DNNVK1H1D,#oc-RB0DQLZAXFCUG,#oc-R5ZVFRNYYRVUG,#oc-R5ZHLJGRON5W,#oc-RUCLMJ31IUSWP,#oc-RXC397CMTXVA,#oc-RXF8BAFFBDS3N,A18LDAXF0852F7,A1195ZPT85762,A1111I9QLNAMI1,A110BEHZCZVAHX,
A110TLED5W8ZXD,A14EF1PPKXSEPU,A14X244GVHPSX,A161BVWC65FHE5,A174GR3NPUAPPN,A180UQK712ZF2,A194K6G8CZ371J,A19EKT8HBSAKOS,A19FHW264NZTOP,A19XWMB3GADIR,A1AS35TCONVPP1,A1BK69CZ4QNE61,
A1E59L7PCVJLM4,A1ESKXNT17ED2A,A1EVV74UQVVKRY,A1F1A8Q0P2XVH5,A1F7YU605RU43Z,A1HRYCV68VTRYC0,A11TRGHT88D5TK,A1JMR1N9NBV21X,A1JV4QKTEB7QBL,A1KEK69ZAG39PB,A1KXJCS6HFRQZ,A1L1542B0CP96,
A1LA4K53F78BER,A1M2L91Z44RNB6,A1NOV41485TEB0,A1ODGCXEYCQ08,A1OHGX8LKS6QLP,A1ONZ83RPLBNUI,A1OMRL5D7LTSQ9,A1P2XYD265YE21,A1P18VBCXSGC7,A1PQTL5DHCK811,A1QB0C76MIO2YP,A1R1BF3CMX0Y3,A1UHKF1UQ3EGW4,
A1UQBFCERIP7VJ,A1VF5LH6SHFVFJ,A1VYFE3M12ZP11,A1W2EUPW80YH4,A1W4153PSMEAJK,A1WQ94M45D8HG,A1Y5ERIKQ8EMSP,A1YJMG0QJXZLD4,A1ZC55CAGBCD49,A1ZENB34HHZEIM,A1ZPY91VE3IDN1,A1ZU55TH45Y2R8,A20688C3ZAVJQ,
A20V7N5A22F4B8,A215HHHRUDJCHP,A217C8B7N5717,A21931R4Z4BF7R,A226VQZWBEPGL,A22CMB02HY3MJB,A22KLAWK6GTW2,A23JUFZCCFN3J5,A23R210NBK2KR,A244CR32QSVLZ4,A24U6HGBZ4P74W,A2561PM99T7HYD,A25C5NVCVIVT5D,
A25HVP12XKQP2B,A25KVM6G3BL15Z,A267FUT1Z0IC1H,A26L9FQHTNBZH,A26YDQ2BF4HEG,A277T03PKKNYH,A27CKZ98YQND9,A2705ZUFAENT9L,A2945HLNPSG1BP,A29M9Q8Q9TZLP,A29MVR0FGCSU4E,A29E3KHTU8YBFB,A29CRIQX15VA31,
A2D7B517ZQ51XL,A2DPYMN12HC101,A2E7RX6AFUQEX,A2F09EKKV3MT02,A2GP3R4890IH42,A2HTP503V3Q8ZD,A2I1ML0322ZQEJ2,A2JAEZ8FMAH3VM,A2L6W3MOT484GM,A2L4ZQNTGZ395T,A2L6BT1PVV9YB0,A2LRNLAV0ZIL4U,A2MGNL420KJHC,
A2HJ80L2FYN7CH,A2ODCK080M6UCY,A2OR4QUQSUN0W7,A2PD27UKAD3Q00,A2P5C7LUNIDEAH,A2RHV42BT3SVON,A2S26VGSVXBCFL,A2TBAUW2H73538,A2UEB48LAWFUCH,A2UGR7VUL3BQ2N,A2UW9H122QKHEZ,A2V01904FH7ABY,A2V92F5R7MLCVI,
A2W40MBH2VQX,A2WN1QF8C5SVHV,A2WHS7XZUVLH6,A2X3L3IKXBHCL,A2X10XRYK8ZY,A2XSV4L7QDHPVH,A2VB7DL3CF07M,A2YKWC3WQJX5J,A2YQZ215F37M8,A2Z2M9RE08JX,A2B94EP156GZ6,A3BL3ACVJ2K045,A3B23BVNMUK3N8,
A319VB38T8MVRU,A31B048XCON79T,A31C0K04MBQ2W42,A31RULMKNVJ5H,A32E0HLFZYXJEP,A32XCVDA14KT13,A33UBU2N4HSHV,A36K2N527XXJW,A36P37D1TBUEF,A37DXIENIDHTV,A37FFWZUGOBL7M,A37OVVYHRSU4MJ,A38U7MGM8XLV1Z,
A39MMBA0299ZF,A3A7Y3T5PPZ9T,A3AQ091DVMF8B0,A3BAE79NFXDGV,A3BFDEBTS1V4UN,A3BKXKX8QF1XIV,A3CB3H2VJDNKE,A3D0PYD054913T,A3DR50GQ3RX10,A3DZFE1CHKSLF2,A3EFSLEHNNP6A,A3EDVX11VZ1HUQ,A3F3B6HY9R3J04,
A3FMPT51H0CJ50,A3H7NPNMDGLOU4,A3I1B3J1FFM4521,A3K91X9X2ARDOK,A3K0D3XVBMK1GX,A3LGT6U2L99IWI,A3M06TE1J4203T,A3MUSHDCTZINQZ,A3N4CU4UYC913,A3NCIN6TLM0GA,A3NM1HT3Q2FHVV,A3N0BH42CTUI5H,A3OJX18B60P3R9,
A3Q1EDCBK2X0B0,A3Q3NQKQ3T176H0,A3Q54HWC1LCAH,A3Q7H5L1K55BF6,A3QVAKYRAH627N,A3R3P2513SDGPR,A3SR4HRF3AKR,A3SVN3531YME,A3VBKXKR7A43R,A3V1ZVETB90ZG5,A3V3Z781BXNITF,A440YBEF0M41P,A405B70A7P46,
A55L52HWPQ8Q0,A5IT8B0K44H8P,A5KLAL1CXZ0W5,A5CHP1QV8ZGV,A5JAGHL675A7,A5L6CQKQX0NBX,A5G3M177UMET5,A5GKV0B3PZQMT,A5HT55E980B0R,A5EEK7AHXZCC,A5JFDNZ3C087N,A5JF2BVZBLD26,A5JHDSVEH7VC,A5NFHV535CAFV,
A5L5A0Z1V5460T,A5NBU6GN3JPEUR,A5NT6154L7WC9,APDPA11ZPYLN,APP35N28C2U51,AQ0M6K7MBT4AY,ARBKY1VNYK3C,AS44QHT3K5PK,ASB4QD6Y237EX,AS30MKRFZC47B,ATANE25C4459Z,ATL03YXU2BC16,AVJ32D4G510Z4,AH781YHUIZ62,
AY6ABKPYCE60B,AYB4ELC5SAMBP,AYNAH993VDECT,AYNRA1J4X1C05
2 B06SHC9ERH,B06SHG9E5G,B06SHG9ET0
```

Sono stati, inoltre, calcolati i tempi di esecuzione di hive in locale e su cluster AWS:

- ❖ Tempo locale: 150 secondi
- ❖ Tempo cluster: 142.10 secondi

- **Spark Core**

L'implementazione Spark è stata effettuata, come nelle precedenti implementazioni di Spark, partendo da un RDD di input nel quale viene caricato il file csv. L'RDD è stato filtrato per ottenere solo productId e userId con score  $\geq 4$  e dopo l'operazione di filtraggio l'RDD ha la seguente forma: (k: productId, v: userId). Questo RDD è stato raggruppato per chiave e la lista di utenti che si genera è stata ordinata. Da questa lista, in un nuovo RDD, sono state create le coppie di utenti.

Successivamente le coppie di utenti sono diventate la chiave per un nuovo RDD, quindi il seguente RDD ha la seguente forma: (k: couple user, v: productId). Con questo RDD è stato eseguito un raggruppamento per chiave così da ottenere per ogni coppia di utenti la lista dei prodotti recensiti in comune. Quindi si sono create coppie di utenti che possiedono almeno 3 prodotti in comune. Viene poi creato un RDD chiamato coupleRDD che rappresenta tutte le possibili combinazioni di prodotti tra le coppie di utenti. Viene utilizzata la funzione `itertools.combinations` per ottenere tutte le combinazioni di prodotti a gruppi di 3. Successivamente, vengono applicate una serie di trasformazioni agli RDD per ottenere gli "affinity groups" tra le coppie di utenti. Viene utilizzata la funzione `reduceByKey` per combinare gli insiemi di utenti che hanno prodotti in comune e la funzione `sortBy` per ordinare gli "affinity groups" in base al primo elemento dell'insieme. Ottenendo così come risultato finale gruppi che condividono gli stessi prodotti, almeno 3 in comune con uno score  $\geq 4$ .

---

**Algorithm 1** Spark core Job 3

---

```

1: linesRDD  $\leftarrow$  input file
2: userProductFilteredScoreRDD  $\leftarrow$  productId, userId, filtered score
   maggiore o uguale a 4
3: productUserRDD  $\leftarrow$  productId, userId
4: productUsersRDD  $\leftarrow$  groupByKey() and sort userId list
5: productCoupleUsersRDD  $\leftarrow$  productId, couple users list
6: usersProductRDD  $\leftarrow$  coupleuser, productId
7: usersProductsRDD  $\leftarrow$  groupByKey() and filtered length productId list
   maggiore o uguale a 3
8: coupleRDD  $\leftarrow$  generated all combination of products
9: groupAffinityRDD  $\leftarrow$  reduceByKey and sort
10: get the final output

```

---

Per motivi di spazio occupato si è riuscito a memorizzare solamente il file di output ottenuto dal test del dataset dimezzato. Per verificare che il risultato fosse coerente e corretto si è provato il dataset dimezzato anche con le tecnologie Hive e Map Reduce ottenendo gli stessi risultati. Riportiamo di seguito le prime righe del dataset dimezzato:



```

part-00000
~/ProgettoBioData/outputSparkJob3
Salva
((('B002D4DY8', 'B0041NVV8E', 'B004BKLH05'), (('AUAX1QWUCYKSX', 'AYNAH993VDECT'))))
((('B007TJGY46', 'B006N3IG4K', 'B003VXHGP'), (('ATOEUHIO4IU5D', 'AYHH3IL637HZJ'), ('A2KUUMP4ZZMOBP', 'ATOEUHIO4IU5D'), ('A2KUUMP4ZZMOBP', 'AYHH3IL637HZJ'))))
((('B007TJGY46', 'B003VXF44', 'B003VXHGP'), (('ATOEUHIO4IU5D', 'AYHH3IL637HZJ'), ('A2KUUMP4ZZMOBP', 'ATOEUHIO4IU5D'), ('A2KUUMP4ZZMOBP', 'AYHH3IL637HZJ'))))
((('B007TJGY46', 'B003VXF44', 'B006N3IG4K'), (('ATOEUHIO4IU5D', 'AYHH3IL637HZJ'), ('A2KUUMP4ZZMOBP', 'ATOEUHIO4IU5D'), ('A2KUUMP4ZZMOBP', 'AYHH3IL637HZJ'))))
((('B004ZIER34', 'B007RTR8AM', 'B004BKLH05'), (('ATL03YXU2BC16', 'AZA595ZPIG240'))))
((('B001LGGH40', 'B003ZT61E2', 'B004JRMG98'), (('ATL03YXU2BC16', 'AZ9WQM7SLUX7E'))))
((('B001LGGH40', 'B003ZT61E2', 'B004R8J8E0'), (('ATL03YXU2BC16', 'AZ9WQM7SLUX7E'))))
((('B003ZT61E2', 'B004R8J8E0', 'B004JRMG98'), (('ATL03YXU2BC16', 'AZ9WQM7SLUX7E'))))
((('B002TMV3E4', 'B0041NVV8E', 'B004BKLH05'), (('ATL03YXU2BC16', 'AYNAH993VDECT'))))
((('B002TMV3E4', 'B001LGGH40', 'B0041NVV8E'), (('ATL03YXU2BC16', 'AYNAH993VDECT'))))
((('B002TMV3E4', 'B001LGGH40', 'B004BKLH05'), (('ATL03YXU2BC16', 'AYNAH993VDECT'))))
((('B002IEZJMA', 'B001LGGH40', 'B004R8J8E0'), (('A1LJR5IS0B6ADX', 'ATL03YXU2BC16'), ('ATL03YXU2BC16', 'AXVKMYWNIHK7W'), ('A1LJR5IS0B6ADX', 'AXVKMYWNIHK7W'), ('A1LJR5IS0B6ADX', 'A3H8PA7AG48K33'), ('A3H8PA7AG48K33', 'ATL03YXU2BC16'))))
((('B006BXUZV0', 'B004YV800E', 'B004BKLH05'), (('ATL03YXU2BC16', 'AWKZAU0D8DYL'))))
((('B006BXUZV0', 'B002LANN56', 'B004BKLH05'), (('ATL03YXU2BC16', 'AWKZAU0D8DYL'))))
((('B006BXUZV0', 'B001LGGH40', 'B004BKLH05'), (('ATL03YXU2BC16', 'AWKZAU0D8DYL'))))
((('B006BXUZV0', 'B001LGGH40', 'B002LANN56'), (('A1LHAXBM5GBJS2', 'ATL03YXU2BC16'), ('A1LHAXBM5GBJS2', 'AWKZAU0D8DYL'), ('ATL03YXU2BC16', 'AWKZAU0D8DYL'))))
((('B006BXUZV0', 'B002IEZJMA', 'B004BKLH05'), (('ATL03YXU2BC16', 'AWKZAU0D8DYL'))))
((('B0041NVV8E', 'B005IW4WFY', 'B004BKLH05'), (('ATL03YXU2BC16', 'AUAX1QWUCYKSX'), ('A2R80172BBNSTA', 'A37WVR9M1STQDU'), ('A2R80172BBNSTA', 'ATL03YXU2BC16'), ('A2R80172BBNSTA', 'AUAX1QWUCYKSX'), ('A37WVR9M1STQDU', 'AUAX1QWUCYKSX'))))
((('B008RWKXK', 'B0041NVV8E', 'B004BKLH05'), (('ATL03YXU2BC16', 'AUAX1QWUCYKSX'), ('A2R80172BBNSTA', 'A37WVR9M1STQDU'), ('A2R80172BBNSTA', 'ATL03YXU2BC16'), ('A37WVR9M1STQDU', 'ATL03YXU2BC16'), ('A2R80172BBNSTA', 'AUAX1QWUCYKSX'), ('A37WVR9M1STQDU', 'AUAX1QWUCYKSX'))))

```

Sono stati, inoltre, calcolati i tempi di esecuzione di spark core in locale e su cluster AWS:

- ❖ Tempo locale: 1.556300401687622 secondi
- ❖ Tempo cluster: 4.19 secondi

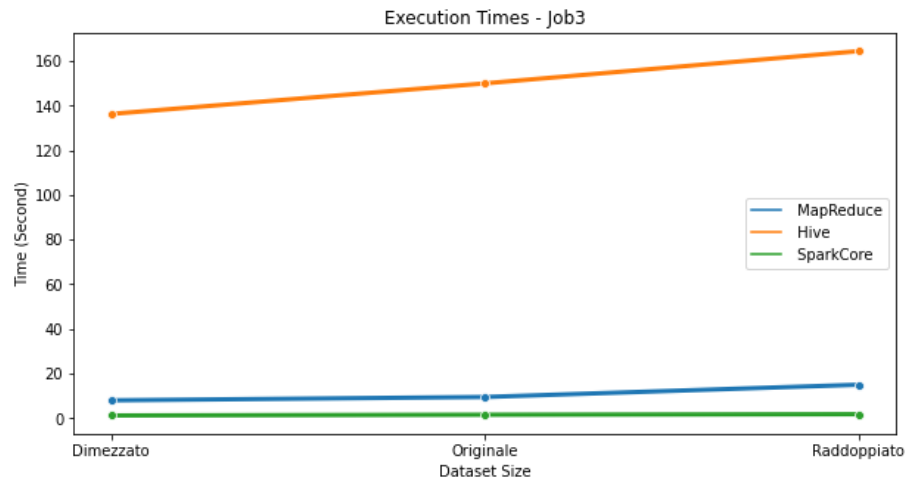
● *Analisi dei tempi*

Di seguito vengono riportate le tabelle e grafici di confronto dei tempi di esecuzione in locale e su cluster di Map Reduce, Hive e SparkCore, con dimensioni crescenti dell’input.

Tempi ottenuti in **locale**:

	MapReduce	Hive	SparkCore
Dimezzato	8.0	136.31	1.23
Originale	9.5	150.00	1.56
Raddoppiato	15.0	164.46	1.80

Dal grafico si può osservare che l’implementazione Spark core è la più veloce e Hive è la più lenta. Il tempo di esecuzione in MapReduce e SparkCore non cresce di troppo rispetto alla dimensione dell’input, mentre in Hive si nota più distacco.



Tempi ottenuti sul **cluster AWS**:

	MapReduce	Hive	SparkCore
Dimezzato	30.89	114.68	3.52
Originale	39.00	142.10	4.19
Raddoppiato	42.00	158.32	4.83

Anche sul cluster Spark Core rimane il più veloce, anche se in questo caso risulta più veloce l'esecuzione in locale anziché su cluster. Il tempo di esecuzione cresce all'aumentare del file in input.

