

PROGETTO 1 – CORSO BIG DATA 2021/2022  
UNIVERSITÀ ROMA TRE

Realizzazione di jobs nell'ecosistema Hadoop utilizzando MapReduce, Hive e Spark

**MARIO CUOMO**



## Sommario

<b>PROGETTO .....</b>	<b>1</b>
<b>AZIONI PRELIMINARI .....</b>	<b>2</b>
<b>STRUTTURA DEL PROGETTO .....</b>	<b>3</b>
<b>JOB 1 .....</b>	<b>4</b>
MAPREDUCE .....	5
HIVE .....	7
SPARK-CORE .....	9
SPARK-SQL .....	10
OUTPUT .....	11
<b>JOB 2 .....</b>	<b>12</b>
MAPREDUCE .....	13
HIVE .....	15
SPARK-CORE .....	16
SPARK-SQL .....	17
OUTPUT .....	18
<b>JOB 3 .....</b>	<b>19</b>
MAPREDUCE .....	20
HIVE .....	24
SPARK-CORE .....	26
SPARK-SQL .....	28
OUTPUT .....	29
<b>RISULTATI SPERIMENTALI – IN LOCALE .....</b>	<b>30</b>

# PROGETTO

## CORSO DI BIG DATA

### Primo Progetto

21 aprile 2022

Si consideri il dataset **Amazon Fine Food Reviews** di Kaggle<sup>1</sup>, che contiene circa 500.000 recensioni di prodotti gastronomici rilasciati su Amazon dal 1999 al 2012. Il dataset è in formato CSV e ogni riga ha i seguenti campi:

- Id,
- ProductId (unique identifier for the product),
- UserId (unique identifier for the user),
- ProfileName,
- HelpfulnessNumerator (number of users who found the review helpful),
- HelpfulnessDenominator (number of users who graded the review),
- Score (rating between 1 and 5),
- Time (timestamp of the review expressed in Unix time),
- Summary (summary of the review),
- Text (text of the review).

Dopo avere eventualmente eliminato dal dataset dati errati o non significativi, progettare e realizzare in: (a) MapReduce, (b) Hive e (c) Spark core (quindi senza usare Spark SQL):

1. Un job che sia in grado di generare, per ciascun anno, le dieci parole che sono state più usate nelle recensioni (campo text) in ordine di frequenza, indicando, per ogni parola, il numero di occorrenze della parola nell'anno.
2. Un job che sia in grado di generare, per ciascun utente, i prodotti preferiti (ovvero quelli che ha recensito con il punteggio più alto) fino a un massimo di 5, indicando ProductId e Score. Il risultato deve essere ordinato in base allo UserId.
3. Un job in grado di 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 che condividono. Il risultato deve essere ordinato in base allo UserId del primo elemento della coppia e non deve presentare duplicati.

Per ciascun job bisogna illustrare e documentare in un rapporto finale:

- Una possibile implementazione MapReduce (pseudocodice), Hive e Spark (pseudocodice).
- Le prime righe dei risultati dei vari job.
- Tabella e grafici che confrontano i tempi di esecuzione in locale dei vari job con dimensioni variabili dell'input<sup>2</sup>.
- Il relativo codice completo MapReduce e Spark (da allegare al documento)
- Un test di uso con logs e file di output (da allegare)
- [Facoltativo] Eseguire i vari job su un cluster a propria scelta (per esempio su dataproc di Google (<https://cloud.google.com/dataproc>) che offre 300\$ di credito gratuito per 90 giorni) e confrontare i tempi con l'esecuzione in locale di uno o più dei job realizzati.

Tutte le specifiche non definite in questo documento possono essere scelte liberamente. Consegnare il rapporto **entro il 20 maggio 2022** in un unico file compresso di formato a piacere sul sito moodle del corso disponibile all'indirizzo: <https://ingegneria.el.uniroma3.it/course/view.php?id=386>.

---

<sup>1</sup> <https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews>

<sup>2</sup> Per aumentare le dimensioni dell'input si suggerisce di generare copie del file dato, eventualmente alterando alcuni dati.

## AZIONI PRELIMINARI

Prima di lavorare con gli strumenti map-Reduce, hive e Spark ho effettuato una pulizia del dataset eliminando i dati non significativi e modificando la struttura di quelli significativi in modo tale da effettuare renderne facilitato l'utilizzo.

Tra i vari campi del dataset quelli di interesse sono sostanzialmente 4.

ProductId – UserId – Score – Time – Text

La prima operazione è stata quindi quella di proiezione dei soli attributi sopra elencati.

A questo punto ho eliminato eventuali repliche in cui lo stesso utente ha recensito lo stesso prodotto più volte: non ho applicato nessun criterio di selezione e per semplicità mantengo la prima replica.

Successivamente utilizzando un Named Entity Recognition system ho pulito il contenuto del campo Text eliminando tutte quelle che sono le stop word. Questa operazione poteva essere raffinata in quanto nel campo Text sono presenti anche tag html rimasto dopo la pulizia.

Dato che del campo Time mi interessa solo ed esclusivamente l'anno, ho convertito il timestamp mantenendo la sola informazione dell'anno.

Per semplicità, ho cambiato il separatore dei campi utilizzando il simbolo di tabulazione.

Infine ho creato diversi file con diversa cardinalità mischiando il contenuto delle ennuple.

Questi file serviranno a testare le varie implementazioni.

ProductId	UserId	Score	Time	Text
B001E4KFG0	A3SGXH7AUHU8GW	5	2011	bought several vitality canned dog food products found good quality product looks like stew processed meat smells better labrador finicky appreciates product better
B00813GRG4	A1D87F6ZCVE5NK	1	2012	product arrived labeled jumbo salted peanuts peanuts actually small sized unsalted sure error vendor intended represent product jumbo
...	...	...	...	...

## STRUTTURA DEL PROGETTO

Il progetto è disponibile all'indirizzo [https://uniroma3-my.sharepoint.com/:f:/g/personal/mario\\_cuomo3\\_uniroma3\\_it/EoL9faVt4V1IqKdNILfHfo4BEDP26yaA0BP\\_DT5qBlxk5w?e=H0f7tc](https://uniroma3-my.sharepoint.com/:f:/g/personal/mario_cuomo3_uniroma3_it/EoL9faVt4V1IqKdNILfHfo4BEDP26yaA0BP_DT5qBlxk5w?e=H0f7tc) e ha la seguente struttura.

```
.
├── ./datasets
├── ./esercizio-1
│   ├── ./esercizio-1/hive
│   ├── ./esercizio-1/map-reduce
│   ├── ./esercizio-1/spark-core
│   └── ./esercizio-1/spark-sql
├── ./esercizio-2
│   ├── ./esercizio-2/hive
│   ├── ./esercizio-2/map-reduce
│   ├── ./esercizio-2/spark-core
│   └── ./esercizio-2/spark-sql
├── ./esercizio-3
│   ├── ./esercizio-3/hive
│   ├── ./esercizio-3/map-reduce
│   ├── ./esercizio-3/spark-core
│   └── ./esercizio-3/spark-sql
├── ./helper
└── ./resources
```

La cartella `datasets` contiene il dataset di partenza scaricato da kaggle e le varie versioni pulite e maggiormente strutturate utilizzate per l'analisi.

La cartella `helper` contiene due script di supporto per la pulizia del dataset e la creazione dei grafici mostrati in questo elaborato (contenuti nella cartella `resources`).

Ogni cartella `hive`, `map-reduce`, `spark-core` e `spark-SQL` contiene l'output del relativo job e rispettivi file di log.

Tutti gli script sono realizzati in python e hql – ad eccezione dell'`esercizio-3/spark-sql` in cui la soluzione è presentata in linguaggio scala.

Il codice sorgente è disponibile qui <https://github.com/mariocuomo/hadoopAtWork/tree/main/progetto1>

## JOB 1

Realizzare un job che sia in grado di generare, per ciascun anno, le dieci parole che sono state più usate nelle recensioni (campo text) in ordine di frequenza, indicando, per ogni parola, il numero di occorrenze della parola nell'anno

## MapReduce

Ho realizzato il job con un singolo stage MapReduce.

Nella fase di map scandisco le varie ennuple del dataset e restituisco in output delle tuple che hanno la forma (anno, parola, occorrenze). L'anno è acquisito dalla entry che sto considerando. La parola è acquisita dal campo text e occorrenze indica quante volte la parola è presente nella recensione in questione.

### ESEMPIO

...	Time	...	Text		OUTPUT
	2008		'ottimo prodotto, ottimo acquisto'	➔	(2008, ottimo, 2) (2008, prodotto, 1) (2008, acquisto, 1)
	2010		'pessimo'		(2010, pessimo, 1)
	2008		'ottimo servizio'		(2008, ottimo, 1) (2008, servizio, 1)

### PSEUDOCODICE

```
for linea in file_di_input
    anno = acquisisciAnno(linea)
    testo_recensione = acquisisciTestoRecensione(linea)

    dizionario_parole_occorrenze={}
    for parola in testo_recensione
        dizionario_parole_occorrenze[parola]+=1

    for parola in dizionario_parole_occorrenze.keys
        stampa(anno, parola, dizionario_parole_occorrenze[parola])
```

Nella fase di reduce ricevo in input l'output della map che ha la forma (anno, parola, occorrenze).

L'idea è di realizzare un dizionario di dizionari.

Il dizionario più esterno ha come chiave l'anno mentre i dizionari interni hanno come chiave la singola parola; questo mi permette di avere una struttura che emula una chiave composta dalla coppia parola-anno.

Scandisco l'input, accedo al dizionario relativo all'anno in questione e incremento il numero di volte che compare parola di una quantità occorrenze.

Al termine, per ogni anno, acquisisco le 10 parole più utilizzate in quell'anno – trasformando i dizionari interni in liste di coppie parola-occorrenze e ordinandole in modo decrescente rispetto al valore di occorrenze.

## ESEMPIO

INPUT	STRUTTURA dict di dict
(2008, ottimo, 2) (2008, prodotto, 1) (2008, acquisto, 1) (2010, pessimo, 1) (2008, ottimo, 1) (2008, servizio, 1)	{ 2008: { ottimo: 3, prodotto: 1, servizio: 1, acquisto: 1 }, 2010: { pessimo: 1 } }

## PSEUDOCODICE

```
dizionario_anno_occorrenze={}

for (anno, parola, occorrenze) in input
    dizionario_anno_occorrenze[anno][parola]+=occorrenze

for anno in dizionario_anno_occorrenze.keys
    parole-occorrenze = dizionario_anno_occorrenze[anno]

// ordina è una funzione che trasforma un dizionario in una lista di
// coppie ordinata in modo decrescente rispetto al value
parole-occorrenze-list = ordina(parole-occorrenze)

stampaTop10(parole-occorrenze-list)
```



## Hive

Ho realizzato il job utilizzando una relazione (cte) di supporto.

Per prima cosa creo una tabella che rispetti la struttura del file di input e la popolo con i dati.

```
CREATE TABLE info(ProductId STRING, UserId STRING, Score STRING, Timee
                    STRING, Textt STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH 'ReviewsPulitoN.txt'
OVERWRITE INTO TABLE info;
```

Successivamente creo una tabella di supporto annoParole caratterizzata dai campi anno e parola. Per ogni ennupla della relazione info si chiama una user defined function che genera n ennuple anno-parola se n sono le parole nel campo Textt.

```
CREATE TABLE annoParole AS
  SELECT TRANSFORM(info.Timee, info.Textt)
  USING 'python3 text_converter.py'
  AS anno, parola
  FROM info;
```

A questo punto per ogni coppia anno-parola si contano quante volte la parola occorre nell'anno. Creo una tabella annoParoleOccorrenze che contiene le informazioni di interesse.

```
CREATE TABLE annoParoleOccorrenze AS
  SELECT anno, parola, Count(*) as occorrenze
  FROM annoParole
  GROUP BY anno, parola;
```

Per ogni anno si selezionano le parole più utilizzate effettuando una partizione sulla base del campo anno – analogo al ragionamento per la risoluzione del job2.

```
CREATE TABLE result AS
  WITH cte AS (
    SELECT anno, parola, occorrenze, ROW_NUMBER() OVER (PARTITION BY anno
    ORDER BY occorrenze DESC) AS rn
    FROM annoParoleOccorrenze)

  SELECT anno, parola, occorrenze
  FROM cte
  WHERE rn <= 10
  ORDER BY occorrenze DESC;
```

Con una user defined function emulo il comportamento della LISTAGG – operazione non disponibile nel RDBMS *Apache Derby* preinstallato con hive. L'idea è quella di raggruppare tutte le ennuple per anno e concatenare in una sola ennupla la lista delle parole con le relative occorrenze.

## Spark-Core

Ho realizzato il job utilizzando sia operazioni narrow che wide.

Per prima cosa si caricano i dati in un RDD che e si filtrano i soli dati di interesse.

Ogni record dell'RDD ha la forma (anno, recensione).

Con l'utilizzo di una flatmap trasformato splitto ogni parola della recensione e produco record nella forma (anno, parola).

Quello che si vuole fare è contare ogni parola quante volte compare nell'anno: utilizzando una map trasformato tutti i record dell'RDD in ((anno, parola), 1) e con una reduceByKey ottengo delle coppie che hanno la forma ((anno, parola), occorrenze) con la semantica che occorrenze indica quante volte parola compare nelle recensioni in quell'anno.

Dato che si vuole sapere per ogni anno quali sono le parole più frequenti effettuo uno swap delle coppie con una map da ((anno, parola), occorrenze) a forma (anno, (parola, occorrenze)).

Con una seconda reduceByKey raccolgo per ogni anno tutte le coppie (parola, occorrenze) avendo una struttura finale dei record (anno, lista\_di\_coppie(parola, occorrenze)).

A questo punto ordino ogni lista\_di\_coppie(parola, occorrenze) di ogni anno applicando l'operatore mapValues e contestualmente acquisisco le 10 parole più frequenti.

### ESEMPIO

anno	Recensione	RDD <sub>i</sub>	RDD <sub>j</sub>	RDD <sub>k</sub>
2001	'ottimo prodotto, ottimo acquisto'	(2001, ottimo) (2001, prodotto) (2001, ottimo) (2001, acquisto) (2010, pessimo) (2010, prodotto) (2001, ottimo) (2001, servizio)	(2001, (ottimo, 3)) (2001, (prodotto, 1)) (2001, (acquisto, 1)) (2001, (servizio, 1)) (2010, (pessimo, 1)) (2010, (prodotto, 1))	(2001, [(ottimo,3), (prodotto, 1), (acquisto, 1), (servizio, 1)]) (2010, [(pessimo, 1), (prodotto, 1)])
2010	'pessimo prodotto'			
2001	'ottimo servizio'			

### PSEUDOCODICE

```
input_RDD = caricaInCache(file_di_input)

// RDD con forma ((anno, parola), 1)
anno_parola_uno_RDD = input_RDD.map()

// RDD con forma ((anno, parola), occorrenze)
anno_parole_occorrenze_RDD = anno_parola_uno_RDD.reduceByKey()

// RDD con forma (anno, (parola, occorrenze))
anno_parole_occorrenze_swap_RDD = anno_parole_occorrenze_RDD.map()

// RDD con forma (anno, lst(parola, occorrenze))
anno_lst_parole_occ_RDD = anno_parole_occorrenze_swap_RDD.reduceByKey()

// stampa l'RDD con forma (anno, lstOrdinata(parola, occorrenze))
stampa(anno_lst_parole_occ_RDD.mapValues())
```

## Spark-SQL

Ho realizzato il job utilizzando sia operazioni narrow che wide.

Per prima cosa si carica il file di input in un dataframe che ne rispecchi la struttura.

Successivamente si mantengono solo le colonne di interesse che sono anno e recensione.

Utilizzando una groupby rispetto al campo anno è possibile concatenare in un unico campo recensione tutte le recensioni effettuate per ogni anno. Il campo è una stringa.

Si applica un map per estrarre le 10 parole usate per ogni anno: la stringa textt è convertita in una lista di parole, la lista di parole è convertita in una lista di tuple in cui il primo elemento indica la parola e il secondo elemento il numero di occorrenze. Si ordina la lista di liste rispetto al numero di occorrenze e si ritornano le 10 parole più usate.

### ESEMPIO

anno	recensione	DF <sub>i</sub>	DF <sub>j</sub>
2001	'ottimo prodotto ottimo acquisto'	(anno, recensione) (2001, 'ottimo prodotto ottimo acquisto ottimo prodotto' )  (2010, 'pessimo prodotto')	(anno, lst_parola_occorrenze) (2001, [ (ottimo, 3), (prodotto, 2), (acquisto, 1)])  (2010, [ (prodotto, 1), (pessimo, 1)])
2010	'pessimo prodotto'		
2001	'ottimo prodotto'		

### PSEUDOCODICE

```
// DF con forma (prodotto, utente, voto, anno, recensione)
input_DF = caricaInCache(file_di_input)

// DF con forma (anno, recensione)
anno_recensione_DF = input_DF.drop()

// DF con forma (anno, recensione)
anno_recensioni_DF = anno_recensione_DF.groupBy("anno")

// DF con forma (anno, 10_parole_più_frequenti_per_anno)
anno_parole_top_DF = anno_recensioni_DF.map()

// stampa il DF
stampa(anno_parole_top_DF)
```

Output

1999		2000	
absurdity	3	film	46
amp	3	quot	40
book	3	beetlejuice	36
burton	3	movie	25
captured	3	find	23
davis	3	burton	21
dull	3	one	18
film	3	p	18
geena	3	comedy	15
keaton	3	dead	15

## JOB 2

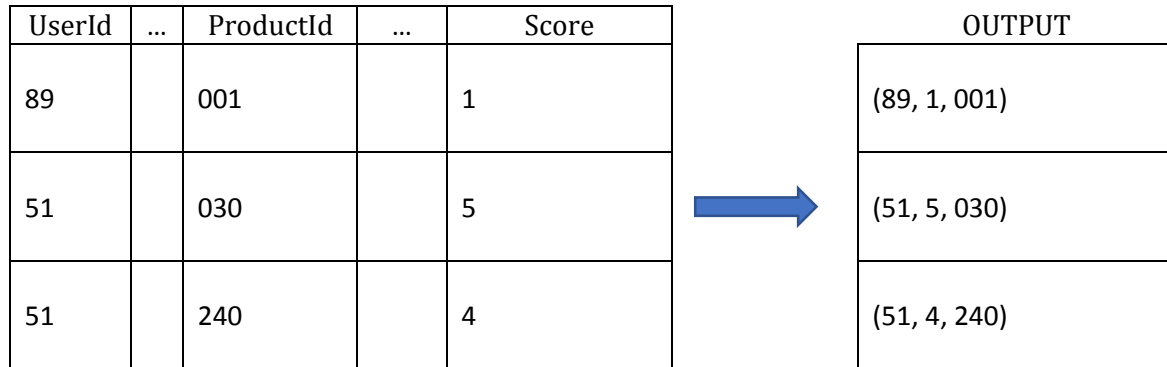
Un job che sia in grado di generare, per ciascun utente, i prodotti preferiti (ovvero quelli che ha recensito con il punteggio più alto) fino a un massimo di 5, indicando ProductId e Score. Il risultato deve essere ordinato in base allo UserId.

## MapReduce

Ho realizzato il job con un singolo stage MapReduce.

Nella fase di map scandisco le varie ennuple del dataset e restituisco in output delle tuple che hanno la forma (idUser, voto, idProdotto). È effettuata sostanzialmente una proiezione degli attributi.

### ESEMPIO



### PSEUDOCODICE

```
for linea in file_di_input
    idUtente = acquisisciId(linea)
    voto = acquisisciVoto(linea)
    idProdotto = acquisisciIdProdotto(linea)

    stampa(anno, parola, dizionario_parole_occorrenze[parola])
```

Nella fase di reduce ricevo in input l'output della map con forma (idUsernte, voto, idProdotto).

L'idea è di realizzare un dizionario di liste di coppie.

Il dizionario ha come chiave l'idUtente e la lista di coppie contiene i prodotti recensiti dall'utente in questione con il relativo voto assegnato.

Scandisco l'input e aggiorno la lista assegnata all'utente idUtente inserendoci la coppia idProdotto e voto.

Al termine, per ogni utente, ordino la lista associata – in ordine decrescente rispetto al voto – e restituisco al più i primi 5 prodotti recensiti.

#### ESEMPIO

INPUT	STRUTTURA dict di tuple list
(89, 1, 001) (51, 5, 030) (51, 4, 240)	{ 51: [ [030,5], [240,4] ], 89: [ [001,1] ] }

#### PSEUDOCODICE

```
dizionario_utente_listaProdotti={}

for (idUsernte, voto, idProdotto) in input
    dizionario_utente_listaProdotti[idUtente].add(idProdotto,voto)

lista-utenti = dizionario_utente_listaProdotti.keys
lista-utenti = ordinalista(lista-utenti)

for utente in lista-utenti
    prodotto-voto-list = dizionario_utente_listaProdotti[utente]

// ordina è una funzione che ordina la lista passata come parametro in
// in modo decrescente rispetto al voto
ordina(prodotto-voto-list)

stampaTop5(prodotto-voto-list)
```



## Hive

Ho realizzato il job utilizzando una relazione (cte) di supporto.

Per prima cosa creo una tabella che rispetti la struttura del file di input e la popolo con i dati.

```
CREATE TABLE info(ProductId STRING, UserId STRING, Score STRING, Timee  
                  STRING, Textt STRING)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t';  
  
LOAD DATA LOCAL INPATH '/home/mariocuomo/Desktop/progetto-big-  
data/ReviewsN.txt'  
  
OVERWRITE INTO TABLE info;
```

Successivamente creo una tabella di supporto temporanea (cte) caratterizzata dai campi UserId, Score, ProductId e rn. rn (row number) è un campo che si ottiene partizionando la tabella di partenza rispetto lo UserId e ordinando rispetto lo Score – così facendo le ennuple che hanno un rn da 1 a 5 rappresentano i 5 prodotti preferiti per ogni utente.

```
CREATE TABLE result AS  
  WITH cte AS (  
    SELECT UserId, Score, ProductId, ROW_NUMBER() OVER(  
      PARTITION BY UserId ORDER BY Score DESC) AS rn  
    FROM info)  
  
  SELECT UserId,Score,ProductId  
  FROM cte  
  WHERE rn <= 5  
  ORDER BY Score DESC;
```

A questo punto non bisogna fare altro che ordinare il risultato rispetto lo UserId.

```
SELECT * FROM  
ORDER BY UserId;
```

Con una user defined function emulo il comportamento della LISTAGG – operazione non disponibile nel RDBMS *Apache Derby* preinstallato con hive. L'idea è quella di raggruppare tutte le ennuple per utente e concatenare in una sola ennupla la lista dei prodotti da lui recensiti con i relativi voti.

## Spark-Core

Ho realizzato il job utilizzando per la maggior parte dei casi operazioni narrow.

L'idea è quella di leggere l'input riga per riga e applicare una prima map che seleziona i campi di interesse (UserId, ProductId, Score) e una per ognuna di esse produce coppie (UserId, [ProductId, Score]).


A questo punto si applica una reduceByKey che produce coppie (UserId, lst) – dove lst è una lista di coppie [ProductId, Score]. Così facendo per ogni utente si ha la lista dei prodotti recensiti con i relativi voti assegnati.

Attraverso l'utilizzo di una map ordino le liste associate a ogni utente e recupero per ognuno di essi i 5 prodotti maggiormente piaciuti.

Con l'utilizzo dell'operazione sortBy è possibile ordinare l'RDD in base all'identificativo dell'utente.

Segue la stampa sul file.

### ESEMPIO

UserId	ProductId	Score		RDD <sub>i</sub>	RDD <sub>j</sub>	RDD <sub>k</sub>
89	001	1		(89,[001,1])	(89, [ [001,1], [170,2] ] )	(51, [ [030,5], [240,4] ] )
				(51,[030,5])	(51, [ [030,5], [240,4] ] )	(89, [ [170,2], [001,1] ] )
				(51,[240,4])		
				(89,[170,2])		
51	030	5				
51	240	4				
89	170	2				

### PSEUDOCODICE

```
input_RDD = caricaInCache(file_di_input)

// RDD con forma (userId, ProductId, Score)
utente_prodotto_voto_RDD = input_RDD.map()

// RDD con forma (userId, [[ProductId, Score], [ProductId, Score] ... ] )
utente_lista_prodotti_voti_RDD = utente_prodotto_voto_.reduceByKey()

// RDD con forma (userId, lst) in cui lst è una lista di coppie [ProductId, Score] di al più 5 elementi e ordinata rispetto lo score
utente_lista_prodotti_topK = utente_prodotto_voto_.map()

// RDD con stessa forma di utente_lista_prodotti_topK ma ordinato anche rispetto lo userId
utente_lista_prodotti_topK_sorted = utente_lista_prodotti_topK.sortBy()

stampa(utente_lista_prodotti_topK_sorted)
```

## Spark-SQL

Ho realizzato il job utilizzando sia operazioni narrow che wide.

Per prima cosa si carica il file di input in un dataframe che ne rispecchi la struttura.

Successivamente si mantengono solo le colonne di interesse che sono utente prodotto e voto.


Utilizzando una groupby rispetto al campo utente e si concatena in un unico campo prodottoVoto i prodotti recensiti dall'utente con i relativi voti. Il campo è una lista di tuple.

Con una operazione di map – per ogni utente – la lista è ridotta ai 5 prodotti che ha recensito con voti alti.

### NOTA

l'ordine rispetto a utente1 è garantito dall'operazione di groupBy

### ESEMPIO

UserId	ProductId	Score		DF <sub>i</sub>	DF <sub>k</sub>
89	001	1		(user, lst_prodotto_voto)	(user, lst_prodotto_voto)
				(89, [ [001,1], [170,2] ] )	(51, [ [030,5], [240,4] ] )
				(51, [ [030,5], [240,4] ] )	(89, [ [170,2], [001,1] ] )
51	030	5			
51	240	4			
89	170	2			

### PSEUDOCODICE

```
// DF con forma (prodotto, utente, voto, anno, recensione)
input_DF = caricaInCache(file_di_input)

// DF con forma (utente, prodotto, voto)
utente_prodotto_voto_DF = input_DF.drop()

// DF con forma (utente, lista prodotto_voto)
utente_prodottiVoti_DF = utente_prodotto_voto_DF.groupBy("utente")

// stampa il DF
stampa(utente_prodottiVoti_sorto_DF)
```

## Output

#oc-R103C0QSV1DF5E	B006Q820X0:5
#oc-R109MU5OBBZ59U	B008I1XPKA:5
#oc-R10LFEMQEW6QGZ	B008I1XPKA:5
#oc-R10LT57ZGIB140	B0026LJ3EA:3
#oc-R10UA029WVWIUI	B006Q820X0:1
#oc-R115TNMSPFT9I7	B007Y59HVM:2; B005ZBZLT4:2
#oc-R119LM8D59ZW8Y	B005DVVB9K:1
#oc-R11D9D7SHXIJB9	B005HG9ESG:5; B005HG9ET0:5; B005HG9ERW:5
#oc-R11D9LKDAN5NQJ	B008I1XPKA:3
#oc-R11DNU2NBKQ23Z	B005ZBZLT4:1; B007Y59HVM:1

## JOB 3

Un job in grado di 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 che condividono. Il risultato deve essere ordinato in base allo UserId del primo elemento della coppia e non deve presentare duplicati.

## MapReduce


Ho realizzato il job con due stage MapReduce.

Il primo stage prende in input il file di partenza e produce in output un file le cui le righe hanno la forma (utente1, utente2, productId) ad indicare che utente1 e utente2 hanno entrambi dato un voto  $\geq 4$  al prodotto identificato da productId.

Nella fase di map scandisco le varie ennuple del dataset e restituisco in output delle tuple che hanno la forma (prodotto, lista\_di\_utenti\_che\_lo\_hanno\_recensito\_positivamente).

### ESEMPIO

ProductId	UserId	Score
010	1	5
010	2	3
022	1	5
002	1	4
002	4	3
022	3	3
010	3	4
010	5	4



OUTPUT	
( 010, [1, 5, 3] )	
( 022, [2] )	
( 002, [1, 4] )	

### PSEUDOCODICE

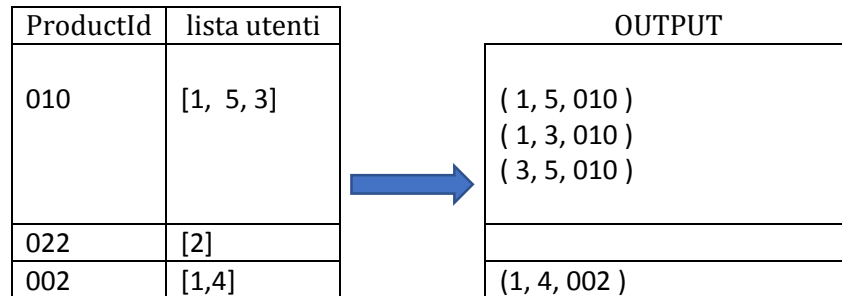
```
product_users={}
for linea in file_di_input
    user, prodotto, voto = acquisisciCampi(linea)

    if voto>3
        product_users[prodotto].add(user)

for prodotto in product_users.keys
    stampa(prodotto, product_users[prodotto])
```

Nella fase di reduce ricevo in input l'output della map che ha la forma (prodotto, lista\_utenti).  
L'idea è quella di produrre in output un file che ha la forma (utente1, utente2, productId); per fare ciò è sufficiente generare tutte le coppie utente1, utente2 a parte da lista\_utenti.

#### ESEMPIO



#### PSEUDOCODICE

```
product_users={}
for linea in file_di_input
    prodotto, lista_utenti = acquisisciCampi(linea)

    product_users[prodotto].add(lista_utenti)

for prodotto in product_users.keys
    lista_utenti = product_users[prodotto]

    // generaCombinazioni è prende in input una lista e produce una lista
    // coppie le cui coppie sono le combinazioni degli elementi in lista
    // es: generaCombinazioni([1,2,3,4]) = [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]

    lista_coppie_utenti = generaCombinazioni(lista_utenti)

    for utente1, utente2 in lista_coppie_utenti:
        stampa(utente1, utente2, prodotto)
```

Nel secondo stage si prende il file di output dello stage 1 e si producono il risultato che ci si aspetta per il job – ovvero un file le cui ennuple hanno la forma utente1, utente2, lista\_prodotti\_in\_comune.

Nella fase di map scandisco le varie ennuple del dataset e restituisco in output delle tuple che hanno la forma (utente1,utente2,lista\_prodotti\_in\_comune).

#### ESEMPIO

utente1	utente2	productId
1	5	010
1	5	030
1	4	005
1	3	010
3	5	010
1	4	002

OUTPUT

( 1, 5, [010, 030] )  
( 1, 4, [002, 005] )  
( 1, 3, [010] )  
( 3, 5, [010] )

#### PSEUDOCODICE

```
user1_user2_products={}
for linea in file_di_input
    user1, user2, prodotto = acquisisciCampi(linea)

    user1_user2_products[(user1, user2)].add(prodotto)

for (user1, user2) in user1_user2_products.keys
    lista_prodotti = product_users[prodotto]


    stampa(utente1, utente2, lista_prodotti)
```



Nella fase di reduce ricevo in input l'output della map che ha la forma (user1,user2,lista\_prodotti) e produce in output le sole ennuple che hanno una lunghezza di lista\_prodotti di almeno 3 elementi. È effettuato sostanzialmente un filtraggio.

#### ESEMPIO

utente1	utente2	lista prodotti
1	5	[010, 030, 002]
1	4	[030]
2	3	[005, 001, 002, 040]
1	3	[010, 098]



**OUTPUT**  
( 1, 5, [010, 030, 002])  
( 2, 3, [005, 001, 002, 040] )

#### PSEUDOCODICE

```
user1_user2_products={}
for linea in file_di_input
    user1, user2, lista_prodotti = acquisisciCampi(linea)

    user1_user2_products [(user1, user2)].add(lista_prodotti)

for (user1, user2) in user1_user2_products.keys
    lista_prodotti = product_users[prodotto]

    if len(lista_prodotti)
        stampa(utente1, utente2, lista_prodotti)
```

## Hive

Ho realizzato il job utilizzando una diverse relazioni di supporto.

Per prima cosa creo una tabella che rispetti la struttura del file di input e la popolo con i dati.

```
CREATE TABLE info(ProductId STRING, UserId STRING, Score STRING, Timee
                    STRING, Textt STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH '/home/mariocuomo/Desktop/progetto-big-
                        data/ReviewsN.txt'
OVERWRITE INTO TABLE info;
```

Creo una tabella di supporto – info\_join\_info – che come suggerisce il nome è l'inner join della tabella info con se stessa sulla base del campo ProductId. Una ennupla di questa relazione rappresenta il fatto che l'utente1 e l'utente2 hanno entrambi recensito lo stesso prodotto identificato dal productId. Al termine del join seleziono le sole ennuple tali per cui entrambi gli utenti hanno dato un voto  $\geq 4$ .

```
CREATE TABLE info_join_info AS
  SELECT i.UserId as Utente1, i1.UserId as Utente2, i.ProductId
  as ProductId
  FROM info i
  INNER JOIN info i1
  ON i.ProductId=i1.ProductId
  AND I.UserId <> i1.UserId
  WHERE i.Score>=4 AND i1.Score>=4
```

A questo punto creo una tabella affini che contiene le sole coppie utente1 e utente2 che hanno recensito almeno 3 prodotti in comune. Non mi devo preoccupare dello Score in quanto ho già scartato le recensioni con Score basso.

Ho scelto di mantenere entrambe le coppie (utente1, utente2) e (utente2, utente1).

```
CREATE TABLE AFFINI AS
  WITH coppie AS (
    SELECT Utente1, Utente2, Count(*) as recensioni_comuni
    FROM info_join_info
    GROUP BY Utente1, Utente2 )

  SELECT Utente1, Utente2
  FROM coppie
  WHERE recensioni_comuni>=3
```

Con la tabella affini posso filtrare la tabella info\_join\_info recuperando le sole ennuple in cui utente1 e utente2 sono affini.

```
CREATE TABLE info_join_info_affini AS
  SELECT *
  FROM info_join_info
  WHERE EXISTS (
    SELECT *
    FROM affini
    WHERE affini.Utente1=info_join_info.Utente1 AND
          affini.Utente2=info_join_info.Utente2
```

Con una user defined function emulo il comportamento della LISTAGG – operazione non disponibile nel RDBMS *Apache Derby* preinstallato con hive. L’idea è quella di raggruppare tutte le ennuple rispetto la coppia utente1 e utente2 e concatenare in una sola ennupla la lista dei prodotti recensiti da entrambi.

## Spark-Core

Ho realizzato il job utilizzando sia operazioni narrow che wide che hanno portato alla creazione di 3 stage.

Per prima cosa si caricano i dati in un RDD e si filtrano i campi di interesse.

Ogni record dell'RDD ha la forma (prodotto, utente, voto).

Filtro i record considerando solo quelli che hanno un voto  $\geq 4$ . Questa operazione mi serve per essere sicuro di considerare i soli prodotti che portano a un eventuale affinità tra gli utenti.

Posso eliminare il campo voto con una map e ottenere record con forma (prodotto, utente).

A questo punto per ogni prodotto mi interessa avere la lista degli utenti che lo hanno recensito. Effettuo una reduceByKey che produce record con la forma (prodotto, lista\_utenti).

Applicando opportunamente prima una mapValues, poi una flatMap e infine una reduceByKey all'RDD creato in precedenza si ottiene un nuovo RDD che ha la forma ((utente1, utente2), lista prodotti).

La mapValues si applica all'RDD con forma (prodotto, lista\_utenti) e trasforma lista\_utenti in una lista di coppie (utente1, utente2) per ogni utente in lista\_utenti.

La flatMap effettua uno scambio nei campi del record (prodotto, lst\_coppie\_utenti) tale per cui per ogni coppia di utenti (utente1, utente2) in lst\_coppie\_utenti si genera il record ((utente1, utente2), prodotto).

La reduceByKey permette di raccogliere tutti i prodotti recensiti dalla coppia e ottenere un nuovo RDD con la forma ((utente1, utente2), lista\_prodotto\_recensiti).

Una filter mi permette di eliminare tutti i record tali per cui utente1, utente2 non condividano almeno 3 prodotti.

Dato che si vuole ottenere in output proprio l'ultimo RDD ma ordinato per utente1 quello che faccio è applicare una map e trasformarlo in (utente1, utente2, lista\_prodotto\_recensiti), salvarlo in una lista classica di python con l'operazione collect, ordinare la lista con il metodo sort nativo di python e ricaricarlo in un nuovo RDD con l'operazione parallelize.

### ESEMPIO

UserId	ProductId	Score		RDD <sub>i</sub>	RDD <sub>j</sub>	RDD <sub>k</sub>
89	001	5		(001, [89,51,35] ) (020, [51,89,40,35] )	(001, [ (89, 51), (89, 35), (51, 35)])  (020, [ (89, 51), (51, 40), (51, 35), (89, 40), (89, 35), (40, 35)])	( (89,51), [001, 020] ) ( (89,35), [001] ) ( (51,35), [001, 020] ) ( (51,40), [020] ) ( (89,40), [020] ) ( (40,35), [020] )
51	001	5				
51	020	4				
89	020	4				
40	020	4				
35	001	4				
35	020	4				

## PSEUDOCODICE

```
input_RDD = caricaInCache(file_di_input)

// RDD con forma (userId, ProductId) filtrato con score>3
prodotto_utente_RDD = input_RDD.map()

// RDD con forma (productId, [userId, userId, ...] )
prodotto_1stutenti_RDD = prodotto_utente_RDD.reduceByKey()

// RDD con forma ((user1,user2), prodotto)
utente1_utente2_prodotto_RDD = utente_ prodotto_1stutenti_RDD_.mapValues().
    .flatMap()

// RDD con forma ((user1,user2), lista prodotti)
utente1_utente2_prodotti_RDD = utente1_utente2_prodotto_RDD.reduceByKey()

// RDD con forma ((user1,user2), lista prodotti)
// dove len(lista_prodotti)>=3
utente1_utente2_prodotti_RDD = utente1_utente2_prodotti_RDD.filter()

// lista che ha la stessa forma dell'RDD utente1_utente2_prodotti_RDD
utente1_utente2_prodotti_list = utente1_utente2_prodotti_RDD.collect()

// ordina la lista rispetto a utente1
ordina(utente1_utente2_prodotti_list)

stampa(sparkContext.parallelize(utente1_utente2_prodotti_list))
```

## Spark-SQL

Ho realizzato il job utilizzando sia operazioni narrow che wide.

Per prima cosa si carica il file di input in un dataframe che ne rispecchi la struttura e filtrandolo mantenendo solo le ennuple che hanno uno score maggiore di 3.

Successivamente si mantengono solo le colonne di interesse che sono utente e prodotto.

A questo punto si effettua un inner join del dataframe con se stesso sulla base del valore prodotto e attraverso delle proiezioni e filtraggi che eliminano duplicati si ottiene un dataframe che ha la seguente forma prodotto, utente1, utente2.

Ogni ennupla del dataframe indica che utente1 e utente2 hanno recensito entrambi il prodotto – e con il filtraggio iniziale si è sicuri che tale recensione sia stata positiva.

È possibile raggruppare in una lista tutti i prodotti che sono recensiti dalla coppia utente1, utente2.


Si ottiene un dataframe che ha la forma utente1, utente2, listaProdottiCondivisi e lo si filtra mantenendo le sole ennuple che hanno una lista di prodotti condivisi di almeno 3 elementi.

### NOTA

l'ordine rispetto a utente1 è garantito dall'operazione di groupBy

### ESEMPIO

UserId	ProductId	Score
89	001	4
51	030	5
51	001	4
89	170	5



DF <sub>i</sub>
(prodotto, user1, user2)
(001, 89, 51)
(003, 51)
(170, 89)

### PSEUDOCODICE

```
// DF con forma (prodotto, utente, voto, anno, recensione) dove voto>3
input_DF = caricaInCache(file_di_input).filter()

// DF con forma (utente, prodotto)
utente_voto_DF = input_DF.drop()

// DF con forma (prodotto, utente1, utente2)
prodotto_utente1_utente2 = utente_voto_DF.join(utente_voto_DF, "prodotto")

// DF con forma (utente1, utente2, lista_prodotti)
utente1_utente2_lstProdotti = utente_voto_DF.groupBy( (utente1,utente2) )

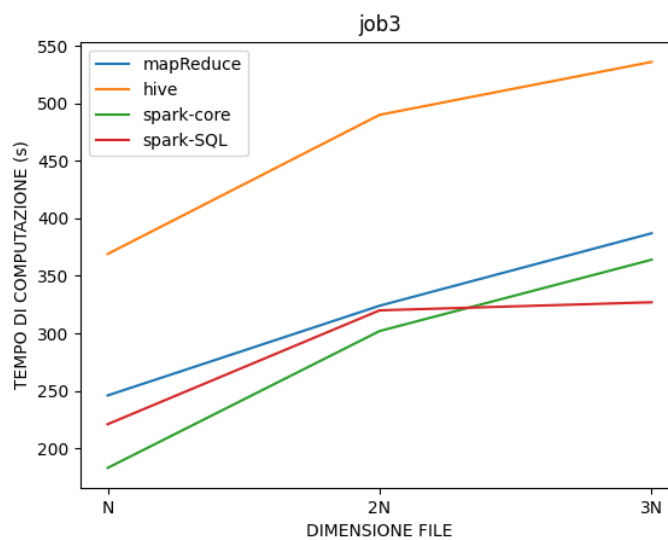
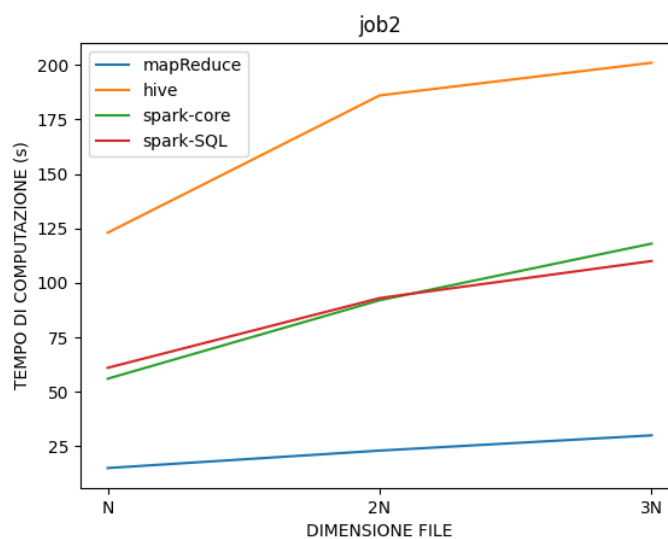
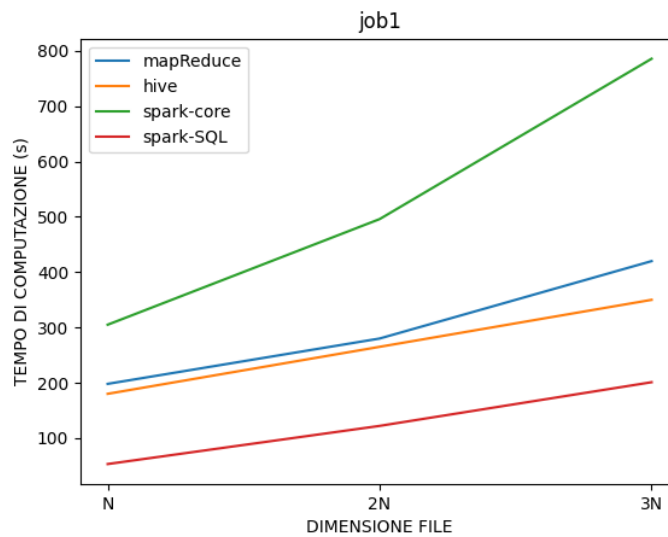
// stampa le sole ennuple tali per cui len(lista_prodotti)>3
stampa(utente1_utente2_lstProdotti.filter())
```

## Output

#oc-R11D9D7SHXIJB9 B005HG9ERW B005HG9ET0	#oc-R3SRKE3YQ2BNES	B005HG9ESG
#oc-R11D9D7SHXIJB9 B005HG9ET0 B005HG9ESG	#oc-R3RQNMHS7481DE	B005HG9ERW
#oc-R11D9D7SHXIJB9 B005HG9ET0 B005HG9ERW	#oc-R3OS88C8I7GSS5	B005HG9ESG
#oc-R11D9D7SHXIJB9 B005HG9ESG B005HG9ET0	#oc-R34PQ2ORKQ6WCD	B005HG9ERW
#oc-R11D9D7SHXIJB9 B005HG9ET0 B005HG9ERW	#oc-R2R45EEG606NCJ	B005HG9ESG
#oc-R11D9D7SHXIJB9 B005HG9ET0 B005HG9ESG	#oc-R2HWL8UHAIMFRS	B005HG9ERW
#oc-R11D9D7SHXIJB9 B005HG9ET0 B005HG9ERW	#oc-R2B86BJE5FNKXX	B005HG9ESG
#oc-R11D9D7SHXIJB9 B005HG9ERW B005HG9ESG	#oc-R28I1AL1ZAZLXL	B005HG9ET0

## RISULTATI SPERIMENTALI – IN LOCALE

Di seguito è mostrato come varia il tempo di computazione dei vari job in funzione della dimensione.





Come si può notare il job2 è un task che si adatta bene al paradigma map-reduce ed è intuitivo sviluppare soluzioni che sfruttano al meglio l'approccio lo stage intermedio di *shuffle and sort*.

Per quanto riguarda il job1 si ottengono dei risultati paragonabili con l'approccio map-reduce e hive. La spiegazione a questo fenomeno è che ho utilizzato lo stesso modo operandi in map-reduce e in hive che come sappiamo in realtà implementa il tutto con stage map-reduce.

A primo impatto pensavo che la soluzione in spark sarebbe stata quella ottimale ma andando a implementare una possibile soluzione per questo job ho dovuto utilizzare 2 trasformazioni wide che hanno avuto un impatto sulle performance in spark-core.

Per il job3 la scelta ottima è quella dell'utilizzo di spark.

Gli alti tempi di esecuzione di hive sono dovuti alla scelta di effettuare una operazione di join dei dati in input con se stessi.

I tempi espressi in precedenza sono stati acquisiti considerando una computazione in modalità standalone utilizzando una CPU amd a9 radeon a 2 soli core.

A solo titolo di test ho eseguito la computazione del job1 con Spark-core su un servizio cloud Microsoft – Azure Synapse Analytics – utilizzando una macchina con 24 core. Il tempo di esecuzione è stato dell'ordine di poche decine di secondi, poco più di mezzo minuto.

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	Job group for statement 38 <a href="#">collect at &lt;stdin&gt;:37</a> <a href="#">+details</a>	2022/05/05 18:29:51	0.3 s	<div><div></div></div> 2/2			2.9 MiB	
2	Job group for statement 38 <a href="#">reduceByKey at &lt;stdin&gt;:31</a> <a href="#">+details</a>	2022/05/05 18:29:30	20 s	<div><div></div></div> 2/2			5.5 MiB	2.9 MiB
1	Job group for statement 38 <a href="#">reduceByKey at &lt;stdin&gt;:25</a> <a href="#">+details</a>	2022/05/05 18:29:05	25 s	<div><div></div></div> 2/2	159.5 MiB			5.5 MiB