Primo Progetto Big Data:
Analisi comparativa di tecnologie per analisi di Big Data

Dipartimento Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Corso: Big Data AA. 2022/2023
Docente: Riccardo Torlone

Federico Bianchi, 534835 Matteo Wissel, 534693
Gruppo: FMBD
GitHub repository: https://github.com/magicWiss/BigData_Project1.git

25 Maggio 2023

Indice

1 Introduzione

La gestione e l'analisi efficiente dei Big Data rappresentano sfide cruciali nel contesto digitale attuale. Questa relazione si focalizza sul confronto di alcune delle più rinomate tecnologie per l'analisi dei Big Data: MapReduce, Hive, Spark Core e Spark SQL.

Nello specifico, la relazione pone l'attenzione sulla comparazione ed analisi delle performance rispetto a diversi task di analisi dati. Questi ultimi sono stati eseguiti su un dataset contentente recensioni di prodotti Amazon, memorizzato tramite HDFS di Apache Hadoop sia on-premise che in ambiente cloud utilizzando la piattaforma AWS. Le metriche scelte per la valutazione delle tecnologie sono velocità di elaborazione, scalabilità e facilità d'uso.

La relazione è suddivisa nel seguente modo:
nel *Capitolo 2* si introducono le tecnologie utilizzate, sottolineando per ognuna di esse gli aspetti più salienti;
nel *Capitolo 3* viene presentato il Dataset, descrivendone sia le caratteristiche principali che le operazioni di parsing operate; i *Capitoli 4,5,6* descrivono per ognuno dei job un analisi ad alto livello dell'implementazione, il relativo pseudocodice e un'analisi dei risultati ottenuti;
il *Capitolo 7* descrive l'analisi comparativa delle prestazioni delle teconologie per i job 1 e 2.

2 Tecnologie

In questo Capitolo si presentano in breve le tecnoogie utilizzate, ovvero MapReduce, Hive e Spark, descrivendo per ognuna di esse le caratteristiche principali ed i loro limiti.

2.1 MapReduce

MapReduce è un paradigma di programmazione e un framework di elaborazione distribuita progettato per garantire l'elaborazione efficiente in parallelo di volumi elevati di dati su cluster di server. Esso suddivide il processo di elaborazione in due *(più una intermedia)* fasi chiave:

- **Map**: Durante la fase di ***mappatura (map)***, il framework suddivide l'input in blocchi più piccoli e applica una funzione di mapping specifica a ciascun blocco. La fase di mapping converte l'input in coppie chiave-valore, in cui la chiave rappresenta un dato di interesse e il valore contiene i dettagli associati a quella chiave. I risultati intermedi delle funzioni di mappatura vengono raggruppati nella fase di ***Shuffle and Sort*** in base alle rispettive chiavi associate.
- **Reduce**: Nella fase di ***reduce*** viene applicata una funzione di riduzione a tutti gli insiemi di valori aventi una stessa chiave. Tale funzione può eseguire operazioni come somma, conteggio, media o altre operazioni definite dall'utente ad-hoc per i propri requisiti. Infine, i risultati finali della fase di reduce vengono restituiti come output dell'intero processo di elaborazione.
- **Shuffle and Sort**: La fase di Shuffle and Sort è responsabile dell'ordinamento e dell'aggregazione intermedia dei dati presenti su uno stesso blocco in base alla loro chiave, permettendo una riduzione del costo computazioniale nella fase di Reduce.
  Questa fase è trasparente al programmatore.

Questo approccio di elaborazione distribuita consente di sfruttare al massimo le capacità dei cluster di server, garantendo prestazioni scalabili per grandi volumi di dati. Tuttavia, è importante tenere presente che il modello MapReduce può risultare meno efficiente per operazioni complesse e iterazioni multiple, a causa della necessità di serializzare i dati tra le fasi di mappatura e riduzione.

2.2 Hive

Hive è un sistema di data warehouse basato su Hadoop che offre un'interfaccia SQL-like per l'interrogazione e l'analisi dei dati distribuiti. Hive semplifica l'analisi dei Big Data fornendo un approccio SQL-like familiare agli utenti.
Tuttavia, Hive può essere più lento rispetto a Spark per alcune operazioni complesse a causa delle operazioni MapReduce che permettono di eseguire le query.

2.3 Spark

Nel contesto di questa relazione, tra le varie tecnologie messe a disposizione da Spark, l'analisi sarà effettuata su due di esse: Spark Core e Spark SQL. È importante specificare che Spark può operare su una vasta gamma di sistemi di storage distribuito, tra cui HDFS di Hadoop, tecnologia utilizzata in questo progetto.

2.3.1 Spark Core

Alla base di questa tecnologia c'è il concetto di RDD (Resilient Distributed Dataset), che consiste in un insieme di oggetti distribuiti che possono essere memorizzati nella cache di diversi nodi del cluster, permettendo un'elaborazione molto veloce dei dati.
Gli operatori messi a disposizione da Spark possono infatti effettuare elaborazioni parallele su questi dati.
Si possono definire tre categorie di operatori:

1. **Transformations:** si occupano di gestire i dataset creandone di nuovi a partire da altri esistenti,
2. **Actions:** l'esecuzione di un operatore di questo tipo restituisce un valore o una collezione,
3. **Persistence:** questi operatori si occupano di gestire lo storage dei dati per agevolarne l'accesso o il salvataggio.

Data la natura a basso livello delle operazioni che Spark Core permette di eseguire, il framework offre una velocità di esecuzione molto alta ed una buona flessibilità a fronte di un approccio non immediato per un programmatore non esperto.

2.3.2 Spark SQL

Spark SQL è un modulo fornito da Spark per il processamento di dati strutturati o semi-strutturati. Il grande vantaggio che offre è la possibilità di scrivere query SQL o sfruttare quelle di Hive.
Offre ottime prestazioni su grandi moli di dati ed ha una buona predisposizione alla scalabilità.

3 Dataset

In questo Capitolo si presenta il dataset utilizzato per questo studio, descrivendo inoltre le operazioni di parsing implementate per permettere una migliore elaborazione dei dati nelle fasi successive.

3.1 Analisi del dataset

Il dataset utilizzato per questa relazione è **Amazon Fine Food Reviews** di Kaggle. Il dataset si presenta in formato *.csv* e contiene circa 500.000 recensioni di prodotti gastronomici rilasciate su Amazon dal 1999 al 2012 (https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews).
Le colonne presenti all'interno della sorgente dati sono:

- **Id**: id univoco associato alla singola recensione;
- **ProductId**: id univoco del prodotto recensito;
- **UserId**: id univoco associato all'utente autore della recensione;
- **ProfileName**: nome dell'utente autore della recensione;
- **HelpfulnessNumerator**: numero di utenti che hanno trovato la recensione utile;
- **HelpfulnessDenominator**: numero di utenti che hanno valutato la recensione;
- **Score**: lo score dato dall'utente al prodotto nella recensione;
- **Time**: timestamp della recensione in formato Unix Time;
- **Summary**: sintesi del testo associato alla recensione;
- **Text**: testo della recensione.

3.2 Pre-processing

3.2.1 Valori nulli

La prima analisi operata sul dataset è relativa all'individuazione di eventuali valori nulli presenti nelle righe.
Analizzando il dataset si notato come solamente due delle 10 colonne presentano valori nulli, ovvero **ProfileName** e **Summary**. Inoltre, il numero di *NaN values* presenti è estremamente più piccolo del numero di righe totali ($\approx$ *0.0005%*). La *Figura 1* mostra le statistiche relative a questa analisi.

In quanto il numero di valori nulli è molto piccolo, si è optato per sostituire questi con la stringa vuota.

3.2.2 Comma values

Uno dei punti critici nella fase di pre-processing del dataset è stato quello di individaure righe contenenti **virgole (",") extra** rispetto a quelle di separazione dei campi.
Questa operazione risulta essere fondamentale in quanto i framework MapReduce e Spark-Core lavorano su standard input splittando ogni riga sul carattere ",".
La presenza di eventuali *comma values* extra, rispetto a quelli di separazione dei campi, potrebbe comportare l'individuazione errata di un numero di colonne maggiore rispetto a quelle reali, generando potenziali fallimenti nell'esecuzione dei job.

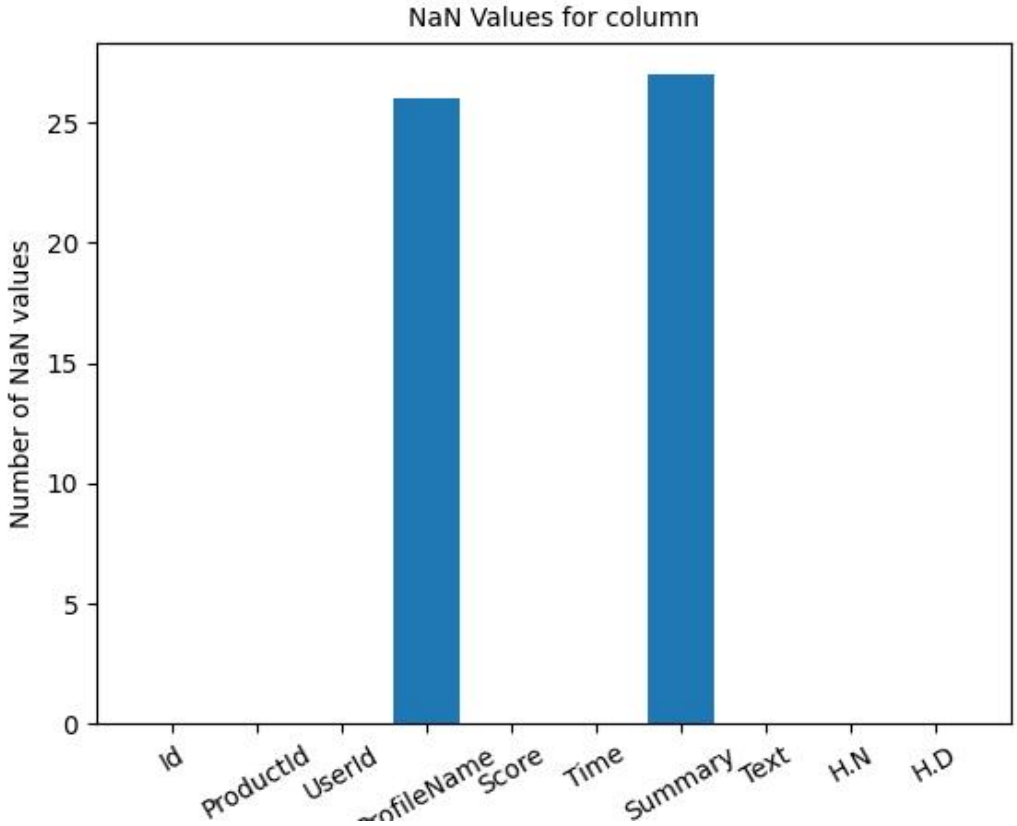

Figura 1: Valori NaN per colonna

Per questo task ci siamo concentrati solo sulle colonne di tipo *stringa* del dataset, ovvero **Profile Name, Summary, Text**.
Per ovviare alla problematica sopra descritta si è deciso di standardizzare il dataset eliminando tutte le virgole in eccesso (fatta eccezione per le virgole separatrici). La standardizzazione è stata implementata mediante sostituzione delle virgole extra con la ***stringa vuota*** o con lo ***spazio***.
Più in dettaglio:

1. **Stringa vuota**: è stata utilizzata nei casi in cui la virgola è seguita da uno spazio;
2. **Spazio**: è stata utilizzato nei casi in cui la virgola è preceduta e seguita da caratteri (in questo modo si evita di concatenare parole che originariamente erano divise).

La Tabella 1 mostra le percentuali di comma values extra prima e dopo il pre-processing.

Tabella 1: Percentuali comma values per colonne String (Before and After pre-processing)

Field	perc. Comma values Before parsing	perc. Comma values After parsing
ProfileName	0.95%	0%
Summary	9.45%	0%
Text	72.56%	0%

4 Job1

In questo Capitolo viene descritto il ***job 1*** e vengono presentate le implementazioni in Map Reduce, Hive, Spark Core e Spark SQL.

4.1 Task Overview

L'obiettivo del primo job è quello di restituire, per ogni anno, i 10 prodotti con più recensioni e, per ciascuno di essi, le 5 parole, con lunghezza maggiore di quattro, più utilizzate.
Per quanto riguarda l'implementazione, è stato necessario, prima di ogni elaborazione, convertire la data in anno, per ottenere la colonna di interesse. Questo è stato facilmente eseguito con una semplice funzione python nelle tecnologie Map Reduce, Spark SQL e Spark Core e con una query in Hive.
L'approccio generale è stato quello bottom-up: infatti il problema è stato studiato a partire dai passi essenziali, per poter poi ottenere uno schema risolutivo efficace ed efficiente.

4.2 MapReduce

La scelta è stata quella di scomporre il job in due fasi:

1. recupero, per ogni anno, dei 10 prodotti con più recensioni;
2. recupero, per ogni coppia (anno, prodotto), delle 5 parole con lunghezza maggiore di quattro più utilizzate.

Per fare ciò, si è scelto di eseguire una prima iterazione Map Reduce in cui:

- **Mapper**: Estrae le coppie (anno, id prodotto)
- **Reducer**: Conta per ciascun (anno, id prodotto) il numero di occorrenze, ordina i risultati in base al numero di occorrenze e restituisce la lista delle coppie (anno, id prodotto) dei 10 prodotti con il maggior numero di recensioni per ciascun anno.

Il risultato della prima iterazione viene poi dato in input alla seconda assieme al dataset completo. L'implementazione prevede quanto segue:

- **Mapper**: Filtra le recensioni relative alle coppie (anno, id prodotto) presenti nel file di output della precedente iterazione e, per ciascuna di esse restituisce la coppia ((anno, id prodotto), testo), in cui la chiave è a sua volta una coppia (anno, id prodotto)
- **Reducer**: Esegue uno split del testo e per ogni tripla (anno, id prodotto, parola) conta le occorrenze. Dopodiché ordina il risultato ottenuto e restituisce una serie di (anno, id prodotto, parola, numero occorrenze) relativa alle 5 parole con lunghezza maggiore di quattro relative, per ogni anno, ai 10 prodotti con il maggior numero di recensioni.

4.2.1 Pseudo-codice

Algorithm 1 Job1: Map Reduce

```
1: procedure GetTop10ProductsMapper(inputfile)                          ▷ First Iteration
2:     while line do                                                    ▷ For each line
3:         clean line
4:         fields ← splitline
5:         year, productid ← UnixTimeToYear(fields[time]), fields[productid]
6:         print year, productid                                        ▷ Map all the pairs
7:     end while
8: end procedure
9: procedure GetTop10ProductsReducer(pairs)                             ▷ First Iteration
10:    map < year, product >
11:    while line do                                                    ▷ For each line
12:        clean line
13:        year, product ← splitline
14:        map < (year, productid), count > += 1                        ▷ Increment the counter
15:    end while
16:    for year in map.keys[year] do                  ▷ Iterate over the year's values in the keys
17:        productCountMap < product, count >= GetTop10SortedProductCountMap(year)
18:        for product in productCountMap.keys do                       ▷ Iterate over the top 10 products
19:            print year, productid                                    ▷ Print the top 10 per year
20:        end for
21:    end for
22: end procedure
```

Algorithm 2 Job1: Map Reduce

```
1: procedure GetTop10ProductsMapper2(inputfile, topPairs)               ▷ Second Iteration
2:     while line do                                                    ▷ For each line
3:         fields ← split line
4:         year, productid, text ← UnixTimeToYear(fields[time]), fields[productid], fields[text]
5:         if year, productid in topPairs then                          ▷ Filter out the wanted pairs
6:             for word in text.split do
7:                 if len(word) greaterThan 4 then
8:                     print year, productid, word                      ▷ Map all the triples
9:                 end if
10:            end for
11:        end if
12:    end while
13: end procedure
14: procedure GetTop10ProductsReducer2(pairs)                           ▷ Second Iteration
15:    map < year, product >
16:    while line do                                                    ▷ For each line
17:        clean line
18:        year, product, word ← splitline
19:        map < (year, productid, word), count > += 1                  ▷ Increment the counter
20:    end while
21:    for year, productid in map.keys[year], map.keys[productid] do
22:        wordCountMap < (product, year, word), count >= GetTop10SortedWordCountMap(year, productid)
23:        for (word) in wordCountMap.keys do
24:            print year, productid, word, count   ▷ Print the top 5 words per each (year, productid)
25:        end for
26:    end for
27: end procedure
```

4.3 Hive

Anche in questo caso, il problema è stato affrontato con una metodologia bottom-up. Nello specifico si è sfruttata la possibilità di Hive di definire query SQL-Like per produrre i risultati intermedi da combinare per ottenere il risultato finale.
In particolare sono state utilizzate, dopo aver caricato il dataset in una tabella *reviews* quattro query distinte suddivise nel seguente modo:

1. La prima query estrae da ogni testo le parole con più di quattro caratteri;
2. La seconda query restituisce la lista delle coppie (anno, id prodotto) che identificano, per ogni anno, i 10 prodotti con il maggior numero di recensioni;
3. La terza query invece estrae le parole più utilizzate per ogni coppia (anno, id prodotto);
4. L'ultima query è quella che unisce i risultati delle query precedenti, in particolare delle ultime due, per ottenere, per ogni anno, le cinque parole maggiormente utilizzate nelle recensioni relative ai dieci prodotti più recensiti.

4.3.1 Pseudo-codice

Algorithm 3 Job1: Hive

```
CREATE TABLE reviews (Id STRING, ProductId STRING, UserId STRING,
ProfileName STRING, HelpfulnessNumerator STRING, HelpfulnessDenominator STRING,
Score STRING, Time INT, Summary STRING, Text STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
TBLPROPERTIES ("skip.header.line.count"="1");

LOAD DATA LOCAL INPATH "path" OVERWRITE INTO TABLE reviews;\\


CREATE TABLE reviews\_with\_year AS \\
(    SELECT ProductId, from\_unixtime(Time, 'yyyy') as year, Text \\
    FROM reviews);\\

CREATE TABLE words\_product\_per_year AS \\
(SELECT word, ProductId, year\\
    FROM reviews\_with\_year\\
    LATERAL VIEW explode(split(Text, '␣')) words\_table as word\\
    WHERE LENGTH(word) >= 4
);

CREATE TABLE top\_reviewed\_products\_per\_year AS \\
(SELECT year, ProductId, cont\_prods FROM \\
        (SELECT year, ProductId, COUNT(ProductId) as cont\_prods,\\
        ROW_NUMBER() OVER (PARTITION BY year ORDER BY COUNT(*) DESC) AS row\_num\\
        FROM reviews_with_year\\
        GROUP BY year, ProductId\\
        ) x WHERE row_num <= 10
);

CREATE TABLE top_words_per_product_per_year
    AS (SELECT year, ProductId, word, cont_words FROM
        (SELECT year, ProductId, word, COUNT(word) as cont_words,
        ROW_NUMBER() OVER
            (PARTITION BY year, ProductId ORDER BY COUNT(*) DESC) AS row_num
        FROM words_product_per_year
        WHERE ProductId IN (SELECT ProductId FROM top_reviewed_products_per_year)
        GROUP BY year, ProductId, word
    ) x WHERE row_num <= 5
);

SELECT tp.year, tp.ProductId, tw.word, tw.cont_words
FROM top_reviewed_products_per_year tp JOIN top_words_per_product_per_year tw
ON tp.year == tw.year AND tp.ProductId == tw.ProductId;
```

4.4 Spark SQL

Per quanto riguarda la tecnologia Spark SQL si è scelto di estrarre, in un primo momento i dieci prodotti più recensiti per ogni anno, usando delle ***Partition*** sulla colonna relativa all'anno.
Questi risultati sono stati utili per filtrare la tabella completa eliminando le righe corrispondenti a tutte le coppie non di interesse.

Dopo questa operazione, si è proceduto con lo split della colonna contenente il testo delle recensioni per ottenere le parole con più di quattro caratteri per ogni coppia (anno, id prodotto).
Quest'ultima struttura dati è stata poi utilizzata per ottenere il risultato finale, utilizzando ancora delle ***Partition*** sulle due colonne relative ad anno ed id prodotto.

I risultati vengono poi mostrati e salvati in un file.

4.4.1 Pseudo-codice

Algorithm 4 Job1: Spark SQL

```
custom_schema = StructType([
    StructField(name="id", dataType=StringType(), nullable=True),
    StructField(name="product_id", dataType=StringType(), nullable=True),
    StructField(name="user_id", dataType=StringType(), nullable=True),
    StructField(name="profile_name", dataType=StringType(), nullable=True),
    StructField(name="helpfullness_numerator", dataType=StringType(), nullable=True),
    StructField(name="helpfullness_denominator", dataType=StringType(), nullable=True),
    StructField(name="score", dataType=StringType(), nullable=True),
    StructField(name="time", dataType=IntegerType(), nullable=True),
    StructField(name="summary", dataType=StringType(), nullable=True),
    StructField(name="text", dataType=StringType(), nullable=True)])

original_DF = spark.read.csv(input_filepath, schema=custom_schema).cache()
new_DF = original_DF.withColumn("year", time_to_date_udf(original_DF["time"])).cache()

# GET TOP 10 PRODUCT ID PER YEAR
product_id_count_per_year = new_DF.groupBy("year", "product_id").count()
    .sort("count", ascending=False)
# Define the window specification for partitioning by year and ordering by count desc
window_spec = Window.partitionBy("year").orderBy(desc("count"))
# Add a new column that assigns a row number based on the window specification
df_with_row_number = product_id_count_per_year.
    withColumn("row_num", row_number().over(window_spec))
# Filter the rows with row number less than or equal to 10
top_10_productid_per_year = df_with_row_number.filter("row_num <= 10")
    .select("year", "product_id").orderBy("year", "row_num")

# GET REVIEWS FOR THE PRODUCT IDS FOUND ABOVE
new_DF = new_DF.join(top_10_productid_per_year, ["year", "product_id"], "inner")
    .select("year", "product_id", "text").cache()

# Split the text column into individual words
words_df = new_DF.select('year', 'product_id', explode(split('text', ' '))
    .alias('word')).filter("LENGTH(word) >= 4")

# Count the occurrence of each word per (product_id, year) and order by count desc
counts_df = words_df.groupby('year','product_id', 'word').agg(count('*').alias('count'))
    .orderBy('year','product_id', 'count', ascending=[True, True, False])

# Window function to rank the words by count within each (product_id, year) group
window = Window.partitionBy('year','product_id').orderBy(desc('count'))
counts_df_with_row_number = counts_df.withColumn("row_num", row_number().over(window))

# Add a rank column to the DataFrame and keep only the top 5 words per
#    (product_id, year) group
top_5_word_per_productid_year = counts_df_with_row_number.filter("row_num <= 5")
    .select("year", "product_id", "word", "count")
    .orderBy("year", "product_id", "row_num")
top_5_word_per_productid_year.show()
```

4.5 Spark Core

L'implementazione in Spark Core segue la falsa riga di Spark SQL. Si susseguono infatti le stesse operazioni di filtraggio, utilizzando però le funzioni native della tecnologia, permettendo di operare più a basso livello.
L'ordine delle operazioni è il seguente:

1. **(1)** filtraggio dei 10 prodotti con il maggior numero di recensioni relativi a ciascun anno;
2. **filtraggio** dell'RDD contenente l'intero dataset;
3. **split** del testo in parole, ciascuna con lunghezza maggiore o uguale a quattro;
4. **filtraggio** per ciascuna coppia anno, id prodotto, delle cinque parole più comuni;
5. **salvataggio** del risultato.

4.5.1 Pseudo-codice

Algorithm 5 Job1: Spark Core

procedure JOB1($inputfile$)
  $reviewsRDD \leftarrow dataset$
  $yearProductCountRDD \leftarrow reviewsRDD.map(((year, productId), 1).reduceByKey(a + b)$ ▷ Map the dataset as ((year,productId),count)
  $yearCountProductRDD \leftarrow yearProductCountRDD.map((year, count), productId)$
  $sortedRDD \leftarrow yearCountProductRDD.sortByKey().map((year), (productId, count))$ ▷ sort the rdd by (productId,count)
  $top10yearProductList \leftarrow sortedRDD.groupByKey().mapValues(sortByValue)[: 10].flatMap(year, productId).collect()$ ▷ Get the top10 products per year
  $filteredRDD \leftarrow reviewsRDD.filter((year, productId)intop10yearProductList).map((year, productId), text)$ ▷ Get the reviews associated to the values just obtained
  $wordRDD \leftarrow filteredRDD.flatMap((year, productId, word)forwordintext.split(), len(word) >= 4)$ ▷ Get all the words with length ¿= 4 in the text field
  $countRDD \leftarrow wordRDD.map((year, productId, word), 1).reduceByKey(a + b)$ ▷ count the occurrences
  $groupedRDD \leftarrow countRDD.map((year, productId, (word, count)).groupByKey()$
  $sortedRDD \leftarrow groupedRDD.mapValues(sortByValue)$ ▷ sort the rdd by values (word, count)
  $top5WordsPerYearProductRDD \leftarrow sortedRDD.mapValues(top5).flatMap((year, productId, word, count))$ ▷ Get the top 5 words per (year, productId)
end procedure

4.6 Analisi dei risultati

Per verificare l'effettiva correttezza delle soluzioni proposte nelle varie tecnologie è stato eseguito un semplice script python che, dati due file di testo contenenti righe del tipo (anno, id prodotto, parola, numero occorrenze), salva il set di tuple appartenenti al primo file e le confronta con quelle nel secondo file per trovare eventuali differenze.
I risultati sono coerenti tra le varie tecnologie, con l'unica sottile differenza nell'output di MapReduce, in cui le tuple non sono stampate in un particolare ordine, ma comunque sono tutte presenti.

5 Job2

Il secondo job consiste nella generazione di una lista di utenti ordinata sulla base del loro ***apprezzamento***. L'apprezzamento complessivo di un utente è ottenuto calcolando, per ciascuno di essi, la *media dell'utilità delle recensioni scritte*.
A sua volta l'utilità di una recensione viene calcolata come il rapporto tra i campi ***HelpfulnessNominator*** e ***HelpfulnessDenominator*** presenti nel dataset.
La logica base implementata per questo job può essere sintetizzata nei seguenti step:

1. Calcolare per ogni utente la media degli score di utilità relativa ad ogni singola recensione scritta. La formula per il calcolo di questo score è la seguente

$$Score(User_i) = \frac{\sum_j \frac{\text{HelpfulnessNominator}_j}{\text{HelpfulnessDenominator}_j}}{N_i}$$

   dove N_i è il numero totale di recensioni scritte dall'utente considerato.

2. Ordinare il risultato in ordine di score decrescente.

Un punto saliente di questo job è la gestione delle recensioni aventi valore di **HelpfulnessDenominator** nullo.
Si presentano adesso le diverse implementazioni del job in MapReduce, Hive, Spark Core e Spark SQL.

5.1 MapReduce

Come per il job 1 ed in linea con la descrizione di alto livello della logica sopradescritta, questo task è stato suddiviso in due fasi: (1) calcolo per ogni utente dello score associato; (2) ordinamento dei risultati ottenuto in base a score decrescente.
Nello specifico, le due iterazioni di Map Reduce effettuano le seguenti operazioni:

1. **Mapper**: estrae le coppie (UserId,HelpulnessNominator/HelpulnessDenominator);
2. **Reducer**: somma i rapporti associati ad uno stesso utente e calcola il numero di recensioni totali effettuate. Mediante questi due valori calcola lo score finale;
3. **Mapper**: prende in input l'output della prima passata e non fa nulla;
4. **Reducer**: ordina la struttura dati considerando lo score associato ad ogni utente.

5.1.1 Pseudo-codice

Algorithm 6 Job2: Map Reduce phase 1

```
1: procedure ComputeScoreForUserMapper(inputfile)                    ▷ First Iteration
2:     while line do                                                 ▷ For each line
3:         clean line
4:         fields ← split line (',')
5:         User_Id, HN, HD ← getData(fields)
6:         score ← 0
7:         if HD ≥ 0 then                          ▷ If HD==0 then the score remains 0
8:             score ← HN/ND
9:         end if
10:        print (User_Id, score)
11:    end while
12: end procedure
13: procedure ComputeAVGScore(output_mapper_1)
14:    user_to_value ←UserId,ratio    while doline                   ▷ For each line
16:    values ← line.split(\t)
17:    if thenlen(values) == 2
18:        user_id ← values[0]
19:        val ← float(values[1])
20:        if thenuser_id ∉ user_to_value
21:            user_to_value[user_id] ← empty list
22:        end if
23:        user_to_value[user_id].append(val)
24:    end if
25:
26:    for doeach k in user_to_value
27:        somma ← sum(user_to_value[k])
28:        N ← len(user_to_value[k])
29:        mean_utility ← somma/N
30:        print "k str(mean_utility)"
31:    end for
```

Algorithm 7 Process User Data

```
user_to_value ← {}
for each line in sys.stdin do
    line ← strip(line)
    values ← split(line)
    if length(values) = 2 then
        user_id ← values[0]
        mean_utility ← values[1]
        if user_id not in user_to_value then
            user_to_value[user_id] ← 0
        end if
        user_to_value[user_id] ← val
    end if
end for
user_to_value ← sorted(user_to_value.items(), key = lambdax : x[1], reverse = True)
for each k in keys(user_to_value) do
    print("%s%s" % (k, str(user_to_value[k])))
end for=0
```

5.2 Hive

La tabella utilizzata come sorgente risulta essere la stessa del job1, ovvero ***reviews***.
Il risultato è stato ottenuto mediante una semplice query annidata capitalizzando i metodi primitivi offerti da HiveQL come Coalesce, SUM e COUNT.
Va notato come, grazie alla possibilità di query che fornisce Hive, l'implementazione di un job più o meno complesso risulta essere semplice e coincisa.

5.2.1 Pseudo-codice

Si presenta una breve panoramica della query implementata.

```
CREATE TABLE Classjob2 (UserId FLOAT, Score FLOAT);


INSERT INTO Classjob2
SELECT UserId,
COALESCE(SUM(HelpfulnessNumerator/HelpfulnessDenominator),0)/ COUNT(*) AS avg_ratio
FROM (SELECT UserId, HelpfulnessNumerator,
        CASE WHEN HelpfulnessDenominator=0 THEN 0
        ELSE HelpfulnessDenominator END AS HelpfulnessDenominator FROM reviews)
        subquery
GROUP BY UserId ORDER BY avg_ratio DESC;

DROP TABLE reviews;
DROP TABLE Classjob2;
```

Si nota come la query interna permette di gestisce i casi in cui il fattore HelpulnessDenominator sia nullo.

5.3 Spark Core

In maniera analoga a quanto svolto nel job 1, sono stati utilizzati i metodi base offerti dal framework Spark come ***map, reduceByKey e mapValues*** utilizzando, per ognuno di essi, delle funzioni custom. Nello specifico:

1. **map ()** è stato utilizzato per creare per ogni riga la coppia (UserId, ratio);
2. **reduceByKey ()** è stato utilizzato per aggregare le coppie aventi UserId uguale ed aggiornare la somma degli score e del numero di recensioni totali effettuate da un utente;
3. **mapValues ()** è stato utilizzato per calcolare lo score;

Anche per l'ordinamento del risultato si è optato per l'utilizzo di un metodo proprio dell'oggetto RDD spark ovvero sortBy.

5.3.1 Pseudo-codice

Di seguito lo pseudocodice che sintetizza lo script python:

Algorithm 8 Spark Core for job 2

$USER\_ID \leftarrow 2$
$NUM \leftarrow 4$
$DENOM \leftarrow 5$
function CreateEntry($input$)
    $user\_id \leftarrow input[USER\_ID]$
    $numerator \leftarrow input[NUM]$
    $denom \leftarrow input[DENOM]$
    $ratio \leftarrow 0$
    if $denom \neq 0$ **then**
        $ratio \leftarrow \frac{numerator}{denom}$
    end if
    return $(user\_id, (ratio, 1))$
end function

function CustomReducer($vals1, vals2$)
    return $(vals1[0] + vals2[0], vals1[1] + vals2[1])$
end function

function ComputeRatio($vals$)
    return $\frac{vals[0]}{vals[1]}$
end function
$spark \leftarrow$ SparkSession.builder.appName("CSV as Text Processing").getOrCreate()

$input\_RDD \leftarrow$ spark.sparkContext.textFile($input\_filepath$)

$header \leftarrow$ input_RDD.first()
$input2\_RDD \leftarrow$ input_RDD.filter(*lambda* row: row != header)
$rows \leftarrow$ input2_RDD.map(*lambda* line: line.split(","))

$user2values \leftarrow$ rows.map(CreateEntry)
$user\_2\_data \leftarrow$ user2values.reduceByKey(CustomReducer)
$user\_2\_ratio \leftarrow$ user_2_data.mapValues(ComputeRatio)

5.4 Spark SQL

L'implementazione del task via Spark SQL segue la stessa logica definita in Hive. Nello specifico, invece di utilizzare i metodi offerti dal modulo Spark SQL quali sum(),count() si è optato per l'esecuzione diretta della query descritta nel paragrafo Hive sul DataFrame generato.

5.4.1 Pseudo-codice

Di seguito lo pseudocodice dello script python utilizzato.

Algorithm 9 Spark Application

1: $columns \leftarrow$ ["UserId", "HelpfulnessNumerator", "HelpfulnessDenominator"]

2: $spark \leftarrow$ SparkSession.builder.appName("UserAvgRatio").getOrCreate() ▷ Create SparkSession

3: $df \leftarrow$ spark.read.csv($input\_filepath, header = True$).$cache$() ▷ Read the CSV file into a DataFrame

4: df.createOrReplaceTempView($'data'$) ▷ Creating a view to store result

5: $query \leftarrow$ SELECT UserId, COALESCE(SUM(HelpfulnessNumerator/HelpfulnessDenominator),0)/COUNT(*)
FROM (SELECT UserId, Help.Num, CASE WHEN Help.Den=0 THEN 0
ELSE Help.Num END AS Help.Den FROM data) subquery
GROUP BY UserId ORDER BY avg_ratio DESC;

6: $output \leftarrow$ spark.RunQuery($query$)

5.5 Analisi dei risultati

In questa sezione si discute l'analisi effettuata per la verifica della bontà dei risultati ottenuti. I test di efficienza sono effettuati sia considerando la correttezza degli singoli output rispetto a risultati precalcolati in locale, sia confrontando i diversi output tra loro. La metrica utilizzata per il contronto è la ***MSE (Mean Squared Error)***.

In prima battuta si è analizzato l'andamento della ***Cumulative Distribution Probability*** **(CDF)** relativo ai punteggi calcolati dalle varie implementazioni. Questo ci è stato utile per prendere visione di eventuali discrepanze tra i risultati ottenuti. Dalla Figura 2 si evince come gli andamenti risultano essere i medesimi, indicando che la discribizione degli score risulta essere uguale per tutte le implementazioni.

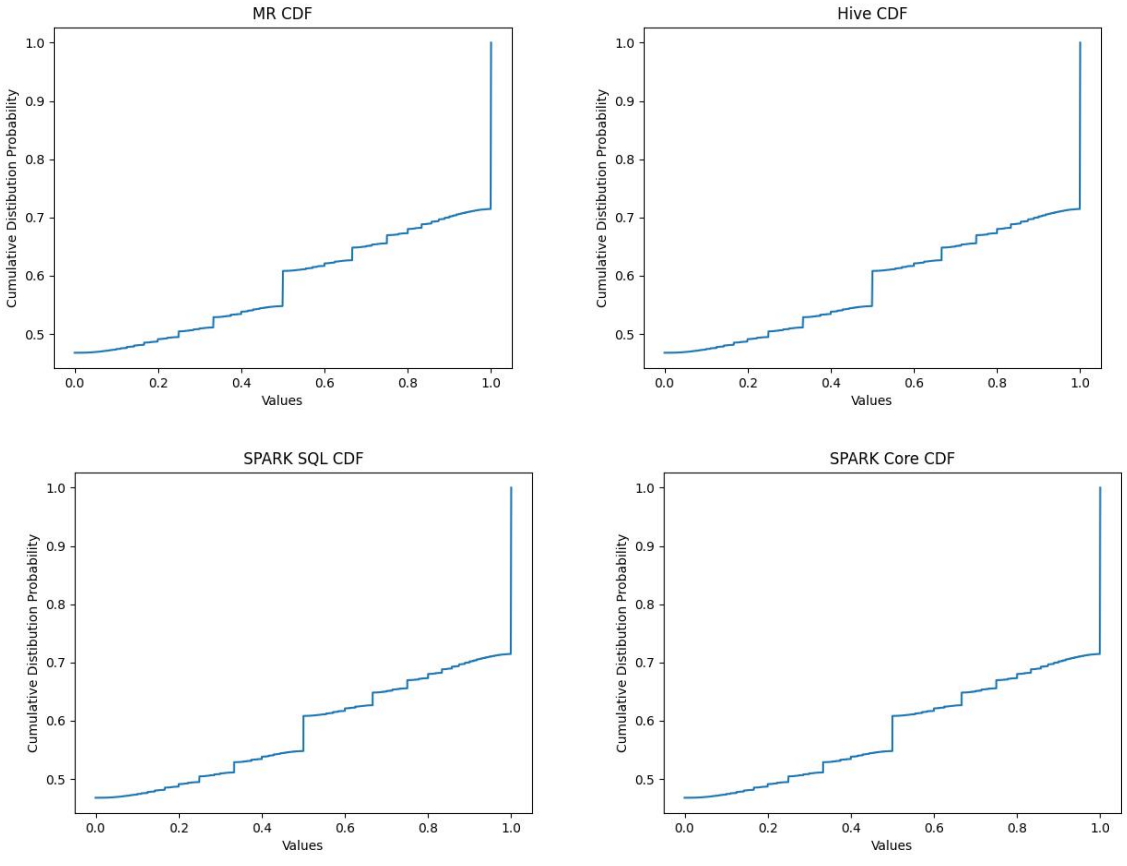

Figura 2: CDF risultati job 2

5.5.1 Efficienza locale

Per l'analisi dell'efficienza locale dei risultati si è optato per il calcolo del true score di affidabilità relativo a 100 user distinti. Questi score sono stati poi confrontati con gli score calcolati nelle quattro implementazioni del job. Di seguito la tabella in cui sono presenti gli *MSE* calcolati. L'origine degli

Tabella 2: MSE implementazioni JOB 2

Framework	MSE
MAPREDUCE	0
HIVE	1.11e-16
SPARK-CORE	1.13e-14
SPARK-SQL	1.11e-14

errori è data da una precision distinta utilizzata dai Framework per rappresentare i valori reali, come si può notare dal seguente confronto:

Tabella 3: Differenze tra i risultati

UserId	MAPREDUCE	HIVE	SPARK-CORE	SPARK-SQL	TrueValue
oc-R115TNMSPFT9I7	0.6666667	0.6666667	0.6666666666666667	0.6666666666666666	0.6666667

5.5.2 Efficienza comparativa

La stessa metrica è stata poi utilizzata per confrontare la distanza tra i risultati ottenuti dalle singole implementazioni. Di seguito la matrice che sintetizza i risultati ottenuti:

Tabella 4: MSE Comparativo

-	MAPREDUCE	HIVE	SPARK-SQL	SPARK-CORE
MAPREDUCE	0	-	-	-
HIVE	1.11e-14	0	-	-
SPARK-SQL	1.17e-05	1.17e-05	0	-
SPARK-CORE	7.83e-15	1.11e-14	1.17e-05	0

6 Job3

Svolti i primi due job ed avendo ancora del tempo a disposizione, si è scelto di implementare il job3 nella tecnologia Map Reduce.
Il job3 consiste nell'individuare gruppi di utenti con gusti affini. In questo caso specifico definiamo due utenti affini se hanno recensito, con score maggiore o uguale a 4, almeno 3 prodotti comuni. Per ognuno dei gruppi è richiesto di visualizzare i prodotti condivisi e di ordinare il risutlato in base allo UserId del primo elemento del gruppo.

La logica di implementazione del job può essere divisa in due fasi distinte:

1. Creazione di liste di adeacenza aventi come chiave lo UserId e come valore una lista composta da tutti i prodotti recensiti dal medesimo utente. In questa fase si effettua un duplice filtraggio sia relativo a recensioni aventi score minore di 4, sia relativo ad utenti che hanno recensito meno di 3 prodotti (o che hanno meno di 3 recensioni con almeno 4 di score);

2. Creazione dei gruppi mediante confronto iterativo della lista dei prodotti associati ad un utente.

La logica di individuazione dei gruppi può essere sintetizzata come segue:
dati due utenti U1 e U2 e dati gli insiemi dei prodotti ad essi associati P1 P2, U1 ed U2 appartengono allo stesso gruppo se e sole se la size(intersection(P1, P2)) è maggiore di 3.
Inoltre, se len(P1) è maggiore di len(P2) allora U1 apparterrà anche ad un gruppo in cui U2 non è presente.

6.1 MapReduce

L'implementazione MapReduce segue la logica descritta in precedenza. Questo è stato suddiviso in 2 iterazioni diverse, ognuna delle quali implementa gli step discussi.
Nello specifico, la prima iterazione:

1. **Mapper**: per ogni riga in input genera coppie (UserId,ProductId) se lo score associato alla recensione è maggiore o uguale di 4;

2. **Reducer**: per ogni User viene generata la lista di ProductId recensiti. Vengono filtrati eventuali User aventi associati meno di 3 prodotti;

Nella seconda iterazione, si prende in input l'output della prima fase. Più in dettaglio:

1. **Mapper**: inverte le liste associative in input passando da una configurazione UserId,Products ad una Product,UserId per faciliare l'ultimo Reducer della pipeline

2. **Reducer**: individua i gruppi applicando la logica descritta in precedenza. Inoltre in questa fase viene effettuato l'ordinamento dei gruppi in base al primo UserId presente all'interno del gruppo.

Si nota come nell'implementazione Python sono stati utiizzati ***FrozenSet*** per rappresentare l'insieme di prodotti associati ad un Utente. Questo ha permesso il loro utilizzo come chiavi del dizionario finale computato.

6.1.1 Pseudocodice

Algorithm 10 Job3

```
1: procedure MAPPER
2:     USER_ID_COL ← 2
3:     PRODUCT_COL ← 1
4:     SCORE_COL ← 6
5:     while lines do
6:         line ← line.strip()
7:         words ← line.split(",")
8:         if len(words) > 1 then
9:             user_id ← words[USER_ID_COL]                          ▷ userID
10:            product_id ← words[PRODUCT_COL]                       ▷ productID
11:            score ← words[SCORE_COL]                              ▷ score
12:            if score >= 4 then          ▷ print only if score greather or equals 4
13:                print "%s%s" %(user_id, product_id)
14:            end if
15:        end if
16:    end while
17: end procedure
18: procedure REDUCER
19:    user_to_product ← {}
20:    while lines do
21:        line ← getNextLine(lines)
22:        values ← line.split('\t')
23:        user_id ← values[0]
24:        products ← values[1]
25:        if user_id not in user_to_product then
26:            user_to_product[user_id] ← set()        ▷ Initialize an empty set for the user
27:        end if
28:        user_to_product[user_id].add(products)      ▷ Add the product to the user's set
29:
30:        for k in user_to_product do
31:            if len(user_to_product[k]) ≥ 3 then
32:                print "%s%s" %(k, str(user_to_product[k]))
33:            end if
34:        end for
35:
```

Algorithm 11 Job3

```
procedure MAPPER
    USER_ID_COL ← 2
    PRODUCT_COL ← 1
    SCORE_COL ← 6
    while lines do
        line ← line.strip()
        words ← line.split("\t")
        if len(words) > 1 then
            user_id ← words[0]
            product_string_list ← words[1]
            product_set ← eval(product_string_list)
            print "%s%s" %(str(product_set), user_id)
        end if
    end while
end procedure
procedure REDUCER
    group_to_products ← {}
    while lines do
        line ← getNextLine(lines)
        values ← line.split('\t')
        user_id ← values[1]
        products ← eval(values[0])
        products_fs ← frozenset(products)                ▷ Convert products to frozen set

        if len(group_to_products) == 0 then
            group_to_products[products_fs] ← [user_id]
        else
            inserted ← 0                                 ▷ Flag to check if the element is inserted
            for k in group_to_products.keys() do
                if products_fs == k then
                    inserted ← 1
                    group_to_products[k].append(user_id)
                end if
                if len(products_fs) > len(k) then
                    intersection ← products_fs & k
                    if len(intersection) >= 3 then
                        group_to_products[k].append(user_id)
                    end if
                end if
                if inserted == 0 then
                    group_to_products[products_fs] ← [user_id]
                end if
            end for
        end if
    end while
    group_to_products_filtered ← {}
    for each key k in group_to_products do
        if group_to_products[k] > 1 then
            Add to group_to_products_filtered
        end if
    end for
    ordered_dict ← orderByFirstUserInGroup(group_to_products[products_fs])
    printOutput (ordered_dict)
end procedure
```

=0

6.2 Analisi risultati

In questa Sezione si effettua una breve analisi dei risultati ottenuti. In prima battuta ci siamo accertati della correttezza dei risultati, verificando se nessuno dei gruppi si ripetesse e che la dimensione minima di onguno dei gruppi fosse di almeno 3 users.
Dall'analisi dell'output sono emerse le seguenti statistiche:

Tabella 5: Statistiche output

Numero gruppi totale	8885
Dimensione massima gruppo	506
Dimensione minima gruppo	2
Media dimensione gruppi	101
Media pesata dimensione gruppi	23.75
Massimo prodotti in gruppo	357
Minimo prodotti in gruppo	3
Media prodotti in gruppo	69

Il risultato del job può essere analizzato come un Grafo G composto da due tipologie di nodi (User, Product) ed archi non orientati che connettono le due tipologie di nodi
Nello specifico, le tipologie di nodi sono le seguenti

1. **Nodi User**: rappresentano il singolo utente;
2. **Nodi Prodotti**: rappresentano il singolo prodotto recensito

Gli archi collegano la coppia (user,product) se e solo se il prodotto è nella lista di prodotti ottenuta nel job *(associata ad un gruppo di cui l'utente fa parte)*.
In Figura 3 è mostrato una parte di grafo generato da tale analisi, mentre la Figura 4 è una visione più dettagliata della parte più densa. In rosso i nodi Prodotto, in blu i nodi User.

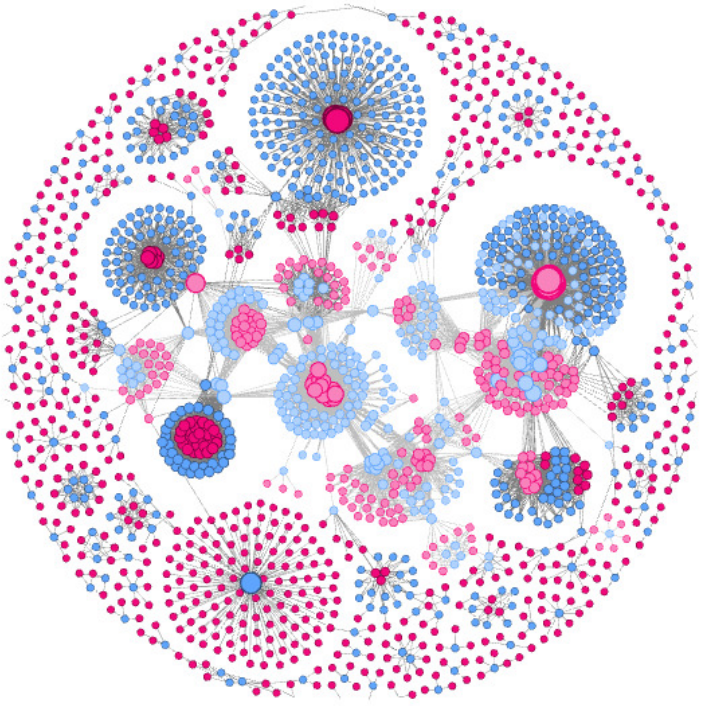

Figura 3: Grafo delle community/gruppi utente-prodotto

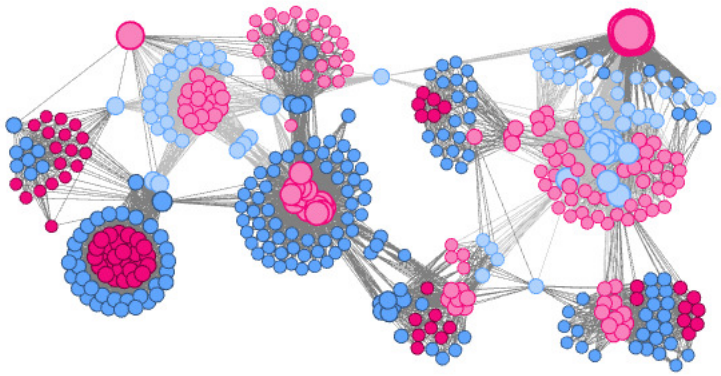

Figura 4: 10-Core del grafo

7 Tempi di esecuzione ed analisi

In questa sezione si effettua un confronto prestazionale tra le diverse tecnologie in merito al tempo di esecuzione dei Job 1 e 2. Per l'occorrenza sono stati generati 9 sample del dataset originale *(Rev_parsed.csv)*, ognungo di dimensione proporzionale alla dimensione del dataset originale rispetto ad una costante K. I dataset sono stati creati applicando la seguente logica:

1. Per $K > 1$ sono stati presi sample randomici del dataset originale;
2. Per K=1 è stato preso l'intero dataset;
3. Per $K < 1$ sono state duplicate righe del dataset originale, modificando alcuni campi, fino ad arrivare alla dimensione voluta

Le dimensioni dei sample sono le seguenti:

Tabella 6: Dimensioni Dataset

Dataset	Dimensione	Fattore Moltiplicativo [k]	Dimensione (MB)
Test_k_01	56845	0.1	31.3
Test_k_03	170536	0.3	93.5
Test_k_05	284227	0.5	155.7
Test_k_07	397917	0.7	218.4
Test_k_1	568454	1	405.2
Test_k_13	738990	1.3	267.4
Test_k_15	852681	1.5	405.2
Test_k_17	966371	1.7	529.9
Test_k_2	1136908	2	623.6

Per sfruttare al meglio il potenziale delle tecnologie in analisi, i test sono stati effettuati sia ***on-premise***, utilizzando un solo nodo master, sia in ***cloud***, mediante ***AWS***, utilizzando 1 Nodo Master e 2 nodi worker. I tempi indicati sono stati calcolati come tempo medio di esecuzione di 10 run distinti.

Le specifiche hardware relative agli environment utilizzati per i run sono le seguenti:

1. **Locale**:
   (a) **CPU**: Intel Core i5-8250u
   (b) **RAM**: 12GB
2. **Cloud** sia master che worker
   (a) **CPU**: 4 vCore
   (b) **RAM**: 16GB

7.1 Job1

Si presenta adesso un'analisi comparativa dei tempi di esecuzione del job1, on-premise e in cloud su dimensioni crescenti del dataset.

7.1.1 Tabella tempi

Tabella 7: Job1: Local vs Cluster Performance

Dataset	MR	HIVE	SPARK	SPARKSQL	MR	HIVE	SPARK	SPARKSQL
Test_k_01	15.650s	40.496s	9.583s	20.374s	7.890s	45.236s	12.191	31.239s
Test_k_03	20.798s	55.580s	11.824s	21.220s	8.340s	52.714s	16.134s	47.417s
Test_k_05	26.739s	115.187s	16.012s	25.507s	8.700s	52.578s	20.558s	54.282s
Test_k_07	27.889s	119.261s	18.426s	28.927s	9.010s	60.062s	24.235s	58.182s
Test_k_1	31.955s	135.070s	21.965s	35.502s	9.556s	61.814s	29.892s	62.871s
Test_k_13	32.738s	152.103s	25.213s	40.997s	9.754s	70.038s	34.264s	71.269s
Test_k_15	31.674s	159.95s	27.975s	42.927s	9.822s	71.352s	43.469s	78.339s
Test_k_17	33.753s	212.162s	29.687s	49.662s	10.210s	84.369s	41.074s	85.171s
Test_k_2	34.765s	220.862s	33.084s	56.289s	11.340s	153.849s	45.861s	97.232s

7.1.2 Andamento tempi

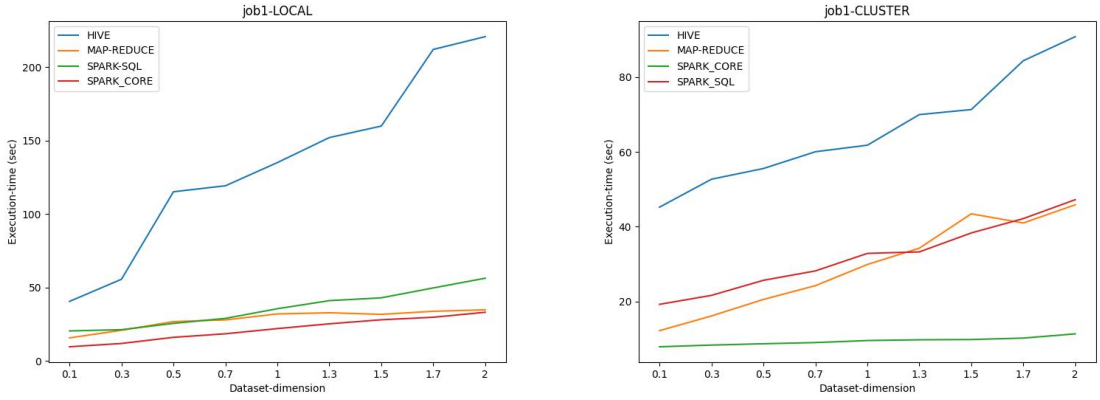

Figura 5: Andamento tempi job 1

7.1.3 Analisi

Nel job 1 i tempi risultano molto alti per l'implementazione in Hive. Questo può essere dovuto, in primo luogo, alla natura della tecnologia in quanto, fornendo un'interfaccia più ad alto livello rispetto ad esempio a Spark, ha bisogno di più tempo per eseguire le operazioni. In secondo luogo, la creazione di tabelle per le fasi intermedie potrebbe appesantire l'esecuzione poiché la quantità di dati in memoria aumenta. Potrebbe essere una soluzione virtualizzare le tabelle andando ad esempio ad annidare le diverse query o effettuare un preprocessing iniziale molto più profondo. Questo sarebbe però, forse, troppo ad-hoc e quindi potrebbe portare il sistema ad una minore flessibilità.
Per quanto riguarda le altre tecnologie, le più efficienti risultano essere Spark Core e Map Reduce, proprio a fronte di un livello più basso di programmazione che porta ad una implementazione più efficiente e diretta.
Questa analisi si addice sia per le prestazioni in locale che su cluster. Infatti, i tempi seguono lo stesso andamento, probabilmente a causa della non eccessiva dimensione del dataset che minimizza i vantaggi del cluster.

7.2 Job2

Si presenta adesso un'analisi comparativa dei tempi di esecuzione del job2, on-premise e in cloud su dimensioni crescenti del dataset.

7.2.1 Tabella

Tabella 8: Job2: Local vs Cluster Performance

Dataset	MR	HIVE	SPARK	SPARKSQL	MR	HIVE	SPARK	SPARKSQL
Test_k_01	3.058s	40.339s	7.117s	8.670s	22.662s	32.708s	10.166s	17.764s
Test_k_03	3.921s	40.729s	7.854s	8.763s	21.813s	34.155s	11.585s	21.174s
Test_k_05	4.065s	43.746s	10.562s	8.605s	20.505s	34.827s	14.495s	23.945s
Test_k_07	5.461s	45.957s	11.514s	8.744s	21.837s	41.324s	15.404s	25.076s
Test_k_1	5.770s	50.375s	12.986s	8.992s	23,705s	36.178s	18.873s	28.421s
Test_k_13	6.018s	57.766s	14.367s	9.059s	24.264s	38.993s	21.176s	31.496s
Test_k_15	6.861s	62.777s	15.415s	9.632	24.443s	37.938s	21.852s	31.957s
Test_k_17	7.267s	70.765s	16.591s	9.723s	22.312s	33.968s	22.737s	35.139s
Test_k_2	7.698s	75.734s	18.521s	9.838s	22.978s	43.668s	24.567s	37.612s

7.2.2 Andamento tempi

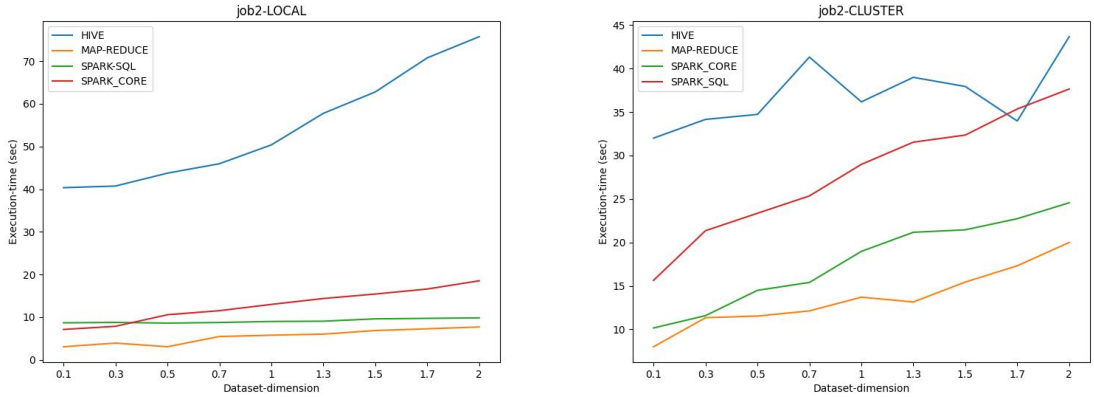

Figura 6: Andamento tempi job 2

7.2.3 Analisi

In generale dai grafici è possibile notare come i tempi di esecuzione in locale crescano proporzionalmente alla dimensione dei dataset considerati.
Come per il job 1, la tecnologia che risulta essere meno efficiente in termini di tempo di esecuzione, sia in locale sia in cluster, è Hive. Come spiegato nell'analisi del job 1, le motivazioni sono dettate dalla natura della tecnologia e dall'overhead generato dalle operazioni di scrittura dei risultati su una tabella.

Inoltre, dai numeri si evince come il tempo di esecuzione per dataset di piccole dimensioni non sia significativamente migliore su cluster, anzi in alcuni casi risulta essere anche leggermente più alto.
Come indicato per l'analisi del job 1, probabilemnte la dimensione ridotta del dataset non massimizza i vantaggi dell'utilizzo di cluster in cloud.

8 Conclusioni

Nella presente relazione abbiamo confrontato quattro tecnologie di elaborazione distribuita dei dati: Map Reduce, Spark SQL, Spark Core e Hive. Lo scopo principale del confronto è stato valutare i tempi di esecuzione e la facilità di utilizzo di queste tecnologie.

Dai risultati ottenuti, Spark Core si contraddistingue per i tempi di esecuzione più efficienti rispetto alle altre tecnologie prese in considerazione. Questo è dovuto alla sua capacità di elaborare i dati in memoria, consentendo un notevole aumento delle prestazioni rispetto all'approccio tradizionale di Map-Reduce. La sua architettura distribuita e l'ottimizzazione delle operazioni di trasformazione e azione contribuiscono a ridurre i tempi di esecuzione e migliorare le prestazioni complessive.

Per quanto riguarda la facilità di utilizzo, sia Hive che Spark SQL sono state valutate come le opzioni più semplici. Entrambe offrono un'interfaccia SQL-like che semplifica la query e l'analisi dei dati. Hive, in particolare, fornisce un'astrazione dei dati strutturati e un linguaggio SQL standard per interrogare e gestire grandi quantità di dati. Spark SQL, d'altra parte, si integra facilmente con Spark Core e offre un'interfaccia SQL unificata per lavorare con dati strutturati e non strutturati.

Tuttavia, nonostante Hive e Spark SQL siano considerate soluzioni user-friendly, è importante sottolineare che Spark Core permette di implementare gli stessi task in un unico script, garantendo prestazioni più efficienti grazie alla sua architettura in memoria e alle ottimizzazioni delle operazioni. Pertanto, se si richiede un'elevata efficienza e prestazioni ottimali, Spark Core rappresenta la scelta consigliata.

Map Reduce invece, data la sua architettura, è la tecnologia più limitata, in quanto non tutti i problemi presentano una forma immediatamente riconducibile al paradigma. Per questo, risulta sicuramente meno intuitiva rispetto alle due sopracitate, e meno efficiente rispetto a Spark Core. Questi fattori condizionano quindi la scelta di questa tecnologia.

In conclusione, l'analisi dei tempi di esecuzione e della facilità di utilizzo ha dimostrato che Spark Core si distingue come la tecnologia più efficiente in termini di tempi di esecuzione. D'altra parte, Hive e Spark SQL si sono rivelate le soluzioni più accessibili per gli utenti meno esperti o per coloro che preferiscono un'interfaccia SQL-like. La scelta della tecnologia più appropriata dipende dalle esigenze specifiche del progetto, considerando sia le prestazioni che la facilità di utilizzo.

9 Sample dell'output

9.1 Job1

Sono mostrate le prime 10 righe come da consegna, alcune righe centrali e le ultime 10 righe.

```
1999,0006641040,this,5
1999,0006641040,when,4
1999,0006641040,&amp;,3
1999,0006641040,book,3
1999,0006641040,books,2
1999,B00004CI84,twist,1
1999,B00004CI84,rumplestiskin,1
1999,B00004CI84,captured,1
1999,B00004CI84,film,1
1999,B00004CI84,starring,1
[...]
2000,B00002N8SM,flies,2
2000,B00002N8SM,bought,1
2000,B00002N8SM,these,1
2000,B00002N8SM,after,1
2000,B00002N8SM,apartment,1
[...]
2007,B0018KLPFK,bars,34
2007,B0018KLPFK,these,27
2007,B0018KLPFK,with,26
2007,B0018KLPFK,like,22
2007,B0018KLPFK,that,21
[...]
2012,B007JFMH8M,cookies,582
2012,B007JFMH8M,cookie,576
2012,B007JFMH8M,soft,491
2012,B007JFMH8M,this,424
2012,B007JFMH8M,these,422
2012,B007Y59HVM,coffee,734
2012,B007Y59HVM,this,474
2012,B007Y59HVM,that,355
2012,B007Y59HVM,have,288
2012,B007Y59HVM,like,266
```

9.2 Job2

```
A10005QPZS2TCW  1.0
A1000WA98BFTQB  1.0
A1001L6GILDZB5  1.0
A1001WMV1CL0XH  1.0
A100454KHKL9F4  1.0
A1005I3S2UN8UN  1.0
A1008ULQSWI006  1.0
A1009L47IQR00Q  1.0
A100B9UOLABJS8  1.0
A100BMGNNGK3V5  1.0
A100DOGJVFCY79  1.0
```

9.3 Job3

['#oc–R28I1AL1ZAZLXL', '#oc–R2W0C6DARSLJ0S', 'A1OHGX0LKS6QLP',
'#oc–R1VRD09DW4H2HI', '#oc–R2HWL8UHAIMFRS', '#oc–R34PQ2ORKQ6WCD',
'#oc–R2K9AJ2LWO7ZUJ', 'A2UGR7VULJBQ2N', '#oc–R12MGTQS5KZZRV',
'#oc–R31AI08Q9HLVF1', '#oc–RXFX0AFFBDSJN', '#oc–R1U8FAII0QYOT3',
'#oc–R3OS88C8I7GSS5', 'A161BVWC65FWE5', '#oc–R3F0UDHOQC1RNU',
'#oc–RXCJ97CMQTXVA', '#oc–RUCLMJ3IUSWPC', '#oc–RSZMLJGJRON5W',
'#oc–RS2VFRNYYRVUG', '#oc–R3RMG4F3JAN4CE', '#oc–R2B86BJE5FNKXX',
'#oc–RODMQLZAXFCUG', '#oc–RMBODWNVK1H1D', 'A174GR3NPUAPPN',
'A1W2EGUPW8OYH4', '#oc–RKU1BA6XFA3Q4', '#oc–R1UO6NAAGVBW7Z',
'#oc–R3TZTZQ7UHNM8Y', '#oc–R2M17G7NB9RAIV', '#oc–R19EJ3VEA88T6O',
'#oc–R3RQNMHS7481DE', '#oc–R3EDGA2893DM4Y', '#oc–R2YPVWM76O2TFX',
'#oc–R1K4OCJ8HEIEDY', '#oc–R1CF9LIM90SAB6', 'A10LDAXFO052F7',
'#oc–R11O5J5ZVQE25C', 'A2MGNLN42OKJHC', 'A38U7MGM8XLVI2',
'#oc–R3DERHJ8UWPZZ', '#oc–R2MG5C3DMRU8Q0', 'A11OBEH2ZCVAHX',
'#oc–R1GKUU1PTIDIZ7', '#oc–R163CP16SRRI50', '#oc–R3TXZAQ0JD85LR',
'#oc–R162D7S0A880MV', '#oc–R2W66Y63G88976', '#oc–R155JB2SA58E17',
'#oc–R11D9D7SHXIJB9', '#oc–R1GSBW9QIVY489', '#oc–R26DKYCO954ZWP',
'#oc–R3SRKE3YQ2BNES', '#oc–R1QHGBT11WAS7G', 'A27CK29BVQNGD9',
'#oc–R2R45EEG606NCJ', '#oc–R2XZVYL146WRFL', 'A30LSAC7UMZDWS',
'#oc–R14ZSRYW2YB41B', 'A3QTW5LIX5SB6F', 'ASB4QD6YZJ7EX',
'A3NOBH42C7UI5M', 'AC0HPFQVBZVGY', 'A1KXJCXS6HFRQZ', 'APP35M28G2U51',
'A2CRIQOX1SYA3I', 'A1E50L7PCVXLN4', 'A2HTPS0JV3Q8ZD', 'A1LA4K5JF78BER',
'ASJ0MKRFZC47B', 'A294SHLWPSG1BP', 'A1R1BFJCMWX0Y3', 'A25HYPL2XKQPZB',
'A2XIOXRRYX0KZY', 'A1JV4QKTEB7QBL', 'A2UW9WI22QKMZE', 'A1F7YU6O5RU432',
'A1ONZ8JRPLBNUI', 'A91TB0WX94MHP', 'A9KLAL1CXZ0W5', 'A3M06TE1J42O3T',
'A244CRJ2QSVLZ4', 'A1EVV74UQYVKRY', 'A3VJ27010XUWTF', 'A3CA3RWZYJDWXE',
'A3I1BJIFFM4S21', 'AHUT55E980RDR', 'A22KL4WOK6GTW2', 'A2TBAUW2W7J538',
'AEJAGHLC675A7', 'A37FFWZUGO8L7W', 'ATANE2SC44592', 'A3K91X9X2ARDOK',
'A2068BC3ZXAVJQ', 'A2PSC7LUNIDEAH', 'A1L1S42BOGPF96', 'AW7BIYHXUIZ62',
'A1VF5LN6SHFVFJ', 'A3OJX18B60PJR9', 'AYNRALJ4X1COS', 'A3VBXQKRM7A4JR',
'A1ITRGMT80D5TK', 'A3A7Y3TSPPZU9T', 'A22CW0ZHY3NJH8', 'A3LGT6UZL99IW1',
'A3MUSWDCTZINQZ', 'AN0U0GNJJPEUR', 'A2D7B5I7ZQ51XL', 'A2V0I904FH7ABY',
'A3BKNXX8QFIXIV', 'A1ODOGXEYECQQ8']->['B005HG9ESG', 'B005HG9ET0', 'B005HG9ERW']

['A1004703RC79J9', 'A5BJMAHZWGJ7N', 'AUV3OR951650C',
'A2RQOO8VYAEZZG', 'A1TPW86OHXTXFC', 'A336FE20YZZL3A',
'A1A1BM6N28X9J0', 'A37OYVYHR5U4NU', 'A2D9N670IKJ2BP',
'A3JH18T58CY65P', 'A25C2M3QF9G7OQ', 'AHKPZ11JT110F',
'A15OAS39003U5R', 'A2OJCTT5WLB2HR', 'ACJ9N7ED37HXS',
'A3QMIWXU2RGEMH', 'A1C139F5C7Y38P', 'A1ILWPH1GHUXE2',
'APP5MBH2BSX6I', 'A3EJ8XHVDC0CWH', 'A224KM22RQ5CD4',
'A1DRM3JV6T5O6P', 'A6B92PTV0Z1SU', 'A3FA9MUYT5D2C8']->
['B001E50THY', 'B001OCKIP0', 'B0090X8IPM', 'B003GTR8IO', 'B003XDH6M6']

['A1007PT85CIPMD', 'A3T8IK1VC5WZRC']->
['B0009F3POY', 'B0009Y8AGI', 'B0009ETA76', 'B0009ETA6W']

['A100IC7JRCQDUD', 'A2SJ4VQ7H3FFS5', 'AD7IBJCYSJ9EE',
'A25L4MK7LXYT50', 'AUHG8KSHI529U', 'A24ZGALKRTN4FF',
'A26F90M6FKBVGY', 'AROBLPGK3SR5P', 'A2EHNH3TPG8DV5',
'A1UV0UMP0V7L1N', 'A3FRW2T18C7W3D', 'AWNBIIKGDPB2E',
'A2VOMWM3O95P7O', 'AKN19A28XUWNZ', 'A1GT9XFMWFJ0I7',
'A25RVZC4ZNSXFW', 'AQXR5XB3URGK9', 'AZR55JSEMWB76',

'A32O98PFC3OP7K', 'AXXP9UHCSYQO5', 'A867X78IJFE8H',
'A3LQXP80NCRM5T', 'A2FMAJPZNVPDM5', 'AMXPD65PFMXOP',
'A2CREQ6E35MMO6', 'A1QRXEM0WSLGT5', 'A19RRTKHBS1MY7',
'A346YRPUO0OCCL', 'A21M3Y07V1F0EC', 'A36GEN9PV4OX1A',
'A1NMV9CRU9G8BS', 'AID3D20YLGU0U', 'A1THRXZXIM26NH',
'A1LWSBGXHLNVQ3', 'A3KSW8H581XS0N', 'A1Z4550X91KUW7',
'A2C9XE9I8RSKNX', 'A1CVN6FWUCZOMD', 'A2WKYCCXDLG9O3',
'A2QDOJFFLFGF18', 'AQLL2R1PPR46X', 'A3QA0BBQW08DLZ',
'AEC90GPFKLAAW', 'A1TMAVN4CEM8U8', 'A27NTHPTRXB766',
'AKOCTZQVAFCOZ', 'A1B05INWIDZ74O', 'A3OXHLG6DIBRW8']->
['B000CQID1A', 'B000CQID1K', 'B000CQE3IC', 'B000RI1W8E', 'B00016Q6BK', 'B000CQE3HS']

['A100WO06OQR8BQ', 'A3DOPYDOS49I3T', 'A3EPHBMU07LZ50',
'A13WOT3RSXKRD5', 'AKJHHD5VEH7VG', 'A1P2XYD265YE21',
'AYNAH993VDECT', 'A3EAP2VG0BVYWX', 'A3D9NUCR4RXDPY',
'A2TO2BN3P4C00L', 'A1LJR5IS0B6ADX', 'A16AXQ11SZA8SQ',
'A2GY5WCU9PKTMI', 'A27HB4L3I1WJUR', 'A2Y8IDC1FKGNJC',
'ACYR6O588USK', 'A2PSC7LUNIDEAH', 'A1P27BGF8NAI29',
'A3HAA7H8PBVM78', 'A3OXHLG6DIBRW8', 'A2YHXAZLCLDT8D',
'ALL9XFM0Q1N4E']-> ['B000H7LVKY', 'B006BXUY2Y', 'B0045XE32E',
'B004E4CCSQ', 'B001Q9EFW8', 'B001SB1F1I', 'B004YV80O4',
'B004E4EBMG', 'B003PFPFIE', 'B004JGQ15E', 'B004OQ257M',
'B001CFMGGI', 'B002LANN56', 'B007JT7ARQ', 'B004U43ZO0',
'B007KIOBMS', 'B004NSH6O8', 'B000MTM0WK', 'B007JT7AEY',
'B000H7ELTW', 'B001Q9EFVY', 'B006BXV14E']

['A1017Q5HHWNALE', 'AL5VAC89VKZ97', 'A8IKK5B480HXN',
'A15CIBTYRG20D', 'AG6FE6WBCDABN', 'A12I4M8WSDNDPF',
'A2EVUVG1MRLDK4', 'A10CMEPEV19WAB', 'A3LNCVKOGNUDB6',
'AO0HPTFIKQE9B', 'AGW1F5N8HV3AS', 'AVDM0ICCRC5BD',
'A39PUPTNFPYXH5', 'A1UJ6I72N22HT7', 'A2PMACL5GLNHI6',
'ABUE0ALHKWKHC', 'A3RYMO6S22FMM5', 'A1ABORTGN181GP',
'A154MF74R6T4II']->['B0016CMVZI', 'B000FNH1C2', 'B000FNB3AI', 'B000FNEX8C']

['A101EVFXW8G49L', 'A14AWP8V95R6SZ', 'A2IPAQZ8GTU7RC',
'A39U8Z3OTYPHFC', 'ADB6MLJ02UXWJ', 'A33Y8C4818EJL0']->
['B00384GGGC', 'B00384GGLC', 'B001IQ1ITW', 'B0016687F2', 'B00384ABT0']

['A102LH0KD8SYHX', 'A2OUNVRPRWH0', 'A2G9M92C9FBUEP',
'A29T9EWY6GISOZ', 'A1XHDMS5TGMKQP', 'A14RZUPW44KCPF',
'A1K82R24ROO2I7', 'A3977OZLNA8LCB']-> ['B001PMDYXW', 'B001PMDYV4',
'B000VK4K3W', 'B000VK8HJ0', 'B001PMC3O8', 'B000WSHV1Q',
'B00139C3P2', 'B001397WV2', 'B001397WYY', 'B000VK6TKO']

['A102TGNH1D915Z', 'A3PPFW1RHMAULY', 'A3I93LKN51G8YX']->
['B000RHXKC6', 'B000XJK7UG', 'B000SP1CWW', 'B0002DHNXC',
'B0009YD7P2', 'B00008DFK5']

['A102UXGLDF76G1', 'A216NSW58Q3SCJ', 'AY1EF0GOH80EK',
'A35BSF26M75AR2', 'AMZWWP4LFT932', 'A345CMJ7NABJ3',

'A2WSAGFJSQC1YF', 'A3CFN1DW7YHUDP', 'ACAIEIV03NBHY',
'A37CENQP0ZNKVE', 'A1ZPY91VE3IDN1', 'A1J5HIF41ENSMZ',
'A2Y8IDC1FKGNJC']–> ['B001EO5U88', 'B000CQY378', 'B000CR008I',
'B0012AQN8U', 'B001HTJ49G', 'B00153C4B4', 'B001HTP04E',
'B000CR41D8', 'B000CR00FQ', 'B001HTC17S']