Primo Progetto Big Data — 22 Aprile 2021

Francesco Foresi FRA.FORESI@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli studi Roma Tre

Pier Vincenzo De Lellis PIE.DELELLIS@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli studi Roma Tre

Progetto: Analizzare le azioni di mercato sulla borsa di New York (NYSE) e sul NASDAQ dal 1970 al 2018 (Corso di Big Data 2020/2021, Università degli Studi Roma Tre, Dipartimento di Ingegneria)

Link gitHub: `https://github.com/bigdata2021-projects/Project_first`

1. Dataset e Preprocessamento

Il dataset considerato come input consiste in due file formato *csv*:

- **historical_stock_prices.csv:** descrive l'andamento delle azioni nel mercato. I campi di ogni riga presi in considerazione sono *ticker* (simbolo univoco dell'azione), *open* (prezzo di apertura), *close* (prezzo di chiusura), *lowThe* (prezzo minimo), *highThe* (prezzo massimo), *volume* (numero di transazioni) e *data* (giorno, mese e anno dell'azione).
- **historical_stocks.csv:** contiene le informazioni relative all'azione in riferimento all'azienda che l'ha emessa. I campi di ogni riga presi in considerazione sono *ticker* (simbolo univoco dell'azione), *name* (nome dell'azienda) e *sector* (settore dell'azienda).

I file ricevuti prima di essere utilizzati per la fase di data analysis , sono stati pre-processati in locale per due ragioni: Il file *historical_stocks.csv* conteneva alcuni record non formattati correttamente, infatti, oltre a delle virgole di troppo in alcun valori del campo *name*, abbiamo notato che alcuni record contenevano una colonna in più rispetto agli altri. Quest'ultimo problema è stato riscontrato durante l'esecuzione del join tra i due file *csv* nei job2 e job3. In secondo luogo il file *historical_stocks_prices.csv* è stato impiegato per crearne copie di diverse dimensioni per favorire il testing del codice dei job e verificarne la scalabilità, a fronte di una variazione crescente dell'input. Questo inoltre è stato necessario poiché la prima versione della macchina virtuale creata su Oracle Virtual Box non era in grado di elaborare dati di queste dimensioni a causa della scarsità di risorse di cui disponevamo. In seguito sono proposte le soluzioni per i job 1, 2, 3 per *Map Reduce*, *Hive* e *Spark*.

2. Soluzione - job1

In questa sottosezione saranno proposte alcune soluzioni del job1 per ogni tecnologia: Map Reduce, Hive e Spark. Il task richiesto nel job1 è quello di generare un report contenente, per ciascuna azione: (a) la data della prima quotazione, (b) la data dell'ultima quotazione, (c) la variazione percentuale della quotazione (differenza percentuale tra il primo e l'ultimo prezzo di chiusura presente nell'archivio), (d) il prezzo massimo e quello minimo.

Map Reduce: L'implementazione del primo job in Map Reduce prevede un singolo stage *Map-Reduce* durante il quale nel mapper si prende l'input *historical_stock_prices* (nelle varie dimensioni) riga per riga e, dopo averlo riorganizzato, si emette come input per la fase di reducer.

Algorithm 1 Mapper job1

```
1: Begin function mapper(historical_stock_price.csv)
2: for line in input do
3:     ticker,open,close,lowThe,highThe,volume,date = line.split()   ▷ values from splitted
4:                                                                    ▷ line
5:     print(ticker,open,close,lowThe,highThe,volume,date)
6: end for
7: end of function
```

Dopo aver riorganizzato ogni riga del file *csv*, nel reducer si impiegano cinque dizionari di supporto per i task, in cui si salva (inizializzando tali dizionari) la coppia chiave-valore *(ticker, value)*, mappando ogni ticker con la data minima(righe 6-9), la data massima (righe 11-12), prezzo massimo (righe 13-14) e prezzo minimo (righe 15-16) nei rispettivi dizionari Python.
Infine si calcola la variazione percentuale relativa alle chiusure nelle date minime e massime per ogni ticker (righe 21-22) e si aggregano i dati dei vari dizionari in un unico dizionario di output (ordinato per il valore di data minima) che verrà restituito in output dal reducer (righe 29-30). Si omettono per brevità le trasformazioni di tipo effettuate su ogni variabile prima di inserirle nei vari dizionari e la riorganizzazione dell'output per una stampa più comprensibile. Tutto ciò che è stato omesso è comunque disponibile nel codice GitHub.

Algorithm 2 Reducer job1

```
1: Begin function reducer(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker,open,close,lowThe,highThe,volume,date = line.split()   ▷ values from splitted
5:                                                                    ▷ line
6:     if ticker not in dateMinDic.keys() then
7:         put date,close in dateMinDic[ticker]                       ▷ initialize dictionary
8:     elif date ≤ dateMinDic[ticker][date] then
9:         put date,close in dateMinDic[ticker]                       ▷ update dictionary
10:
11:    if ticker not in dateMaxDic.keys() then
12:        ....                                                       ▷ initialize or update dateMaxDic
13:    if ticker not in highTheDic.keys() then
14:        ....                                                       ▷ initialize or update highTheDic
15:    if ticker not in lowTheDic.keys() then
16:        ....                                                       ▷ initialize or update lowTheDic
17: end for
18:
19: for ticker in dateMinDic.keys() do
20:     if ticker in dateMaxDic.keys() then
21:         get closeMin,closeMax from dateMinDic and DateMaxDic
22:         variationPercentage=(closeMax − closeMin) / CloseMin * 100
23:       put variationPercentage in actionDictionary[ticker]
24:     if ticker in highTheDic.keys() then
25:         put highThe in actionDictionary[ticker]
26:     if ticker in lowTheDic.keys() then
27:         put lowThe in actionDictionary[ticker]
28: end for
29: for ticker in sorted(actionDictionary.keys()) do
30:     print(ticker,actionDictionary[ticker])
31: end for
32: end of function
```

Hive: Nell'implementazione di Hive del job1 sono state create più tabelle per rendere il codice il più lineare possibile e per distribuire i record nel in modo da ottimizzarne l'aggregazione nella vista finale. Inizialmente è stata creata una tabella *action_temp* per importare i record dal file historical_stock_prices.csv. Successivamente sono state create ed utilizzate altre tabelle per eseguire i vari punti del task: in particolare *max_date_action* e *min_date_action* per il punto A e B, *first_close _action* e *last_close_action* per calcolare la variazione percentuale di una azione per il punto C, *min_action* e *max_action* per calcolare il prezzo massimo e quello minimo dell'azione per il punto D. Infine per ottenere il report complessivo del job1 abbiamo utilizzato una vista con i join tra le varie tabelle per gettare i record necessari e ordinato il report per la data dell'ultima quotazione in ordine decrescente.

Spark: La soluzione di Spark proposta per il primo job consiste nel salvare l'input in due RDD(riga 3) mediante una funzione di *map* che costruisce la tupla di valori corrispondente ad un record dell'input. Successivamente è eseguita un'operazione *reduceByKey* per produrre un RDD per ogni punto dell'esercizio, ossia per calcolare data minima del ticker, data massima, prezzo massimo e prezzo minimo (righe 4-7). Viene infine effettuato un join tra gli RDD riguardanti le date minime e massime per calcolare la variazione percentuale (riga 8). Nella riga 9 si aggregano gli RDD facendo il join per la chiave ticker, ed ordinandoli per il campo *minDate*.

Algorithm 3 Spark job1

```
1: Begin function job1 (datasetRRD1)
2: inputRDD1 = datasetRDD1
3: rdd1 = inputRDD1.map(f=lambda → (tuple))      ▷ (ticker,(open,close,low,high,date))
4: rdd2 = rdd1.reduceByKey(f=min(date1, date2))
5: rdd3 = rdd1.reduceByKey(f=max(date1, date2))
6: rdd4 = rdd1.reduceByKey(f=max(high1, high2))
7: rdd5 = rdd1.reduceByKey(f=min(low1, low2))
8: rdd6 = rdd2.join(rdd3).mapValues(f=varperc(close1, close2))                ▷ var.perc.
9: rdd_output = rdd6.join(rdd4.join(rdd5)).mapValues(f=act_output).sortBy(minDate)
10: return output
11: end of function
```

Alla fine, l'ultimo RDD viene salvato su un file di testo. Tutto ciò che è stato omesso è comunque presente nel codice GitHub.

3. Soluzione - job2

In questa sottosezione saranno proposte alcune soluzioni del job2 per ogni tecnologia: Map Reduce, Hive e Spark. Il task richiesto nel job2 è quello di generare un report contenente, per ciascun settore e per ciascun anno del periodo 2009-2018: (a) la variazione percentuale della quotazione del settore1 nell'anno, (b) l'azione del settore che ha avuto il maggior incremento percentuale nell'anno (con indicazione dell'incremento) e (c) l'azione del settore che ha avuto il maggior volume di transazioni nell'anno (con indicazione del volume). Il report deve essere ordinato per nome del settore.

Map Reduce:
Il job2 MapReduce è stato suddiviso in due stage per la natura del job stesso. Infatti nel primo stage si effettua un *mapper* ed un *reducer* solo per permettere il join tra i due file di input *historical_stock_prices.csv* e *historical_stocks.csv*, utilizzando una variabile temporanea binaria che tiene traccia del fatto che la riga proviene da un file piuttosto che da un altro. Si effettua un controllo sulla lunghezza della linea e, in entrambi i casi, si effettua un filtraggio considerando solo le ennuple che presentato rispettivamente la data compresa tra il 2009 ed il 2018 (righe 6-9) ed il settore non nullo (righe 10-13). Si restituiscono poi in output le righe contenente anche il flag 0 o 1 per il reducer.

Algorithm 4 Mapper_join job2

```
1:  Begin funcion mapper_join(historical_stock_prices.csv, historical_stocks.csv)
2:  for line in input do
3:      line = line.split()
4:      initialize temporary variables
5:      if len(line) == 8 then                                  ▷ line belong to first file
6:          if '2009-01-01' ≤ date ≤ '2018-12-31' then                          ▷ date is ok
7:              set ticker,close,date,volume,temp from line     ▷ get values from splitted line
8:              temp = 0                                                    ▷ flaggin to 0
9:              print(ticker,close,date,volume,temp)
10:     elif sector != 'N/A' then          ▷ line belong to second file and sector is not null
11:         set ticker,sector from line                         ▷ get values from splitted line
12:         temp = 1                                                    ▷ flagging to 1
13:         print(ticker,sector,temp)
14: end for
15: end of function
```

Successivamente, nel reducer si utilizza il flag 0 o 1 per riconoscere la provenienza della riga e, dunque, riuscire a recuperare i valori di interesse che saranno poi inseriti in due dizionari separati: uno per il prezzo di chiusure ed il volume, uno per il titolo del settore (righe 5-9). Successivamente si riorganizzano i valori dei due dizionari in un unico dizionario che per ogni ticker e data contiene il prezzo di chiusura, il volume ed il rispettivo volume (righe 11-13). Infine viene restituito in output il contenuto di quest'ultimo dizionario (righe 16-18), che costituirà l'input per il secondo stage di *map-reduce*.

Algorithm 5 Reducer_join job2

```
1: Begin funcion reducer_join(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker, close, volume, sector, date, temp = line.split()
5:     end of function
6:     if temp == 0 then                                  ▷ line belong to first file
7:         put close, volume in closeDictionary[ticker, date]
8:     else                                               ▷ line belong to second file
9:         put sector in sectorDictionary[ticker]
10: end for
11:
12: for ticker, date in closeDictionary.keys() do
13:     if ticker in sectorDictionary.keys() then
14:         put close, volume, sector in actionDictionary[ticker, date]   ▷ merging into one
15:                                                                        ▷ dictionary
16: end for
17: for ticker, date in actionDictionary.keys() do
18:     print(ticker + date + actionDictionary[ticker, date])
19: end for
20: end of function
```

Nel secondo stage di *map-reduce* risiede il cuore della logica applicativa necessaria per la risoluzione del job2. Il *mapper* è simile a quelli già proposti, riorganizzando semplicemente ogni linea e restituendola in output per il reducer. Nonostante siano stati riscontrati risultati soddisfacenti, si nota come il reducer del secondo stage sia leggermente sovraccarico anche se computazionalmente ancora sostenibile anche per grandi moli di dati in input. Una soluzione alternativa può essere, dunque, quello di spezzare ulteriormente il secondo stage in due sotto-stage per aumentare l'efficacia dell'algoritmo *map-reduce* il quale, come noto, si comporta più efficientemente su piccoli task. Per una migliore formattazione il reducer è stato suddiviso in due algoritmi.

Algorithm 6 Mapper job2

```
1: Begin funcion mapper1(input)
2: for line in input do
3:     ticker, date, close, volume, sector = line.split()
4:     print(ticker, date, close, volume, sector)
5: end for
6: end of function
```

Il reducer del secondo stage inizializza ed (eventualmente) aggiorna i dizionari utilizzati per tenere traccia della data minima, di quella massima, della somma dei volumi per ogni ticker, di ogni settore, per ogni anno (righe 5-11). Successivamente si calcolano le quotazioni di ogni settore utilizzando i dizionari con minima/massima data e relative chiusure, salvando le quotazioni in minima data e quelle in massima data per ogni settore in due dizionari differenti (righe 20-29 e righe 32-35).

Algorithm 7 Reducer job2 PART1

```
1: Begin funcion reducer1(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker, date, close, volume, sector = line.split()          ▷ values from splitted
5:     if sector,ticker,YEAR(date) not in minDateDictionary.keys() then
6:         put date,close in minDateDictionary[sector, ticker,YEAR(date)]
7:     elif sector,ticker,YEAR(date) not in maxDateDictionary.keys() then
8:         ....                                          ▷ same as minDateDictionary
9:     else
10:        update minDateDictionary[sector,ticker,YEAR(date)] if date is lower
11:        update maxDateDictionary[sector,ticker,YEAR(date)] if date is higher
12:                                          ▷ summing volumes per ticker in one year
13:    if sector,ticker,YEAR(date) not in sumVolumeDictionary.keys() then
14:        put volume into sumVolumeDictionary[sector, ticker, YEAR(date)]
15:    else                                                ▷ update summing volume
16:    update sumVolumeDictionary[sector, ticker, YEAR(date)] with sum volume
17: end for
18:
19:                              ▷ using minDate dictionary to calculate first quotations sum
20: for sector, ticker, year in minDateDictionary.keys() do
21:    if sector, year not in minQuotationDictionary.keys() then
22:        put close from minDateDictionary in minQuotationDictionary[sector, year]
23:    elif sector is different then                              ▷ meet new sector
24:        update minQuotationDictionary[sector, year] with close from
25:                                                              minDateDictionary
26:    else          ▷ if is not new, sum the first closes to calculate percentage variation
27:        update minQuotationDictionary[sector, year] with sum close from
28:                                                              minDateDictionary
29: end for
30:
31:                              ▷ using maxDate dictionary to calculate last quotations sum
32: for sector, ticker, year in maxDateDictionary.keys() do
33:    if sector, year not in maxQuotationDictionary.keys() then
34:    ....                                      ▷ same for minQuotationDictionary(rows )
35: end for
36: .... continue in Reducer job2 PART2....
```

Viene poi calcolata la variazione percentuale della quotazione del settore, inserita in un dizionario dedicato (righe 3-6). Analogamente si calcola la variazione percentuale dei prezzi di ogni ticker nelle righe 8-11. Si utilizza il risultato intermedio appena descritto per trovare i ticker del settore con massima variazione percentuale in un certo anno, inizializzando o aggiornando (in caso di variazione percentuale maggiore) un dizionario dedicato (righe 14-19). Analogo procedimento per trovare il ticker che in un anno ha riscontrato il maggior numero di volumi di transazioni (righe 21-26). Infine si prendono i valori di tutti i dizionari coinvolti e si restituiscono in output, ordinati per il nome del settore. Si omettono per brevità le trasformazioni di tipo effettuate su ogni variabile, la riorganizzazione dell'output per una stampa più comprensibile e i *get* dei valori dai vari dizionari.

Algorithm 8 Reducer job2 PART2

```
1: ....
2:                                          ▷ calculating percentage variation of sector quotation
3: for sector, year in maxQuotationDictionary.keys() do
4:     if sector, year in minQuotationDictionary.keys() then
5:         put (maxQuotation − minQuotation) / minQuotation * 100 in varDictionary[sector, ticker, year]
6: end for
7:                                          ▷ calculating percentage variation of ticker price
8: for sector, ticker, year in minDateDictionary.keys() do
9:     if sector, ticker, year in maxQuotationDictionary.keys() then
10:        put (lastClose − firstClose) / firstClose * 100 in
11:                                          tickerVarDictionary[sector, ticker, year]
12: end for
13:                                          ▷ finding ticker with max variation
14: for sector, ticker, year in tickerVarDictionary.keys() do
15:    if sector, year not in tickerVarMaxDictionary.keys() then
16:        put ticker,tickerVar from tickerVarDictionary[sector,ticker, year] in tickerVar-
    MaxDictionary[sector, year]
17:    elif tickerVar is higher then
18:        update tickerVarMaxDictionary[sector, year] with tickerVar
19: end for
20:                                          ▷ finding ticker with max sum of volumes
21: for sector, ticker, year in sumVolumeDictionary.keys() do
22:    if sector, year not in maxVolumeDictionary.keys() then
23:        put sumVolume in maxVolumeDictionary
24:    elif sumVolume is higher
25:        update maxVolumeDictionary[sector, year] with sumVolume
26: end for
27:                                          ▷ print output joining interesting dictionaries
28: print(sector, year, varQuotation, ticker, maxQuotation, ticker, maxVolume) from
29:                                          sorted(dictionaries)
30: end of function
```

Hive: Nell'implementazione di Hive del job2, in prima battuta sono stati prelevati i dati da entrambi i file *csv* descritti nella sezione 1, poichè richiesto esplicitamente di mostrare nel report finale il settore relativo ad un ticker. Infatti la prima tabella creata è stata *historical_stock_join*, ottenuta eseguendo il join fra le due tabelle *historical_stock_prices e historical_stock* sulla chiave *ticker_id*. Con la condizione *WHERE* nel join sono stati selezionati, dalla tabella *historical_stock_prices*, soltanto i record riguardanti gli anni compresi dal 2008 al 2018 ed esclusi i settori con valore *null* come richiesto dal report. Successivamente come nel job1 sono state create due tabelle per gettare la data minima e la data massima raggruppandole per ticker , settore e anno. Queste due tabelle sono state poi oggetto di join con la tabella *historical_stock_join* per costruire rispettivamente le tabelle *MaxDateAndCloseForSectorAndYear e MinDateAndCloseForSectorAndYear* le quali sono state necessarie per ottenere il prezzo di chiusura di ogni ogni azione relative ad un settore ed ad un anno. Infine per concludere il punto A del job2 sono state create due ulteriori tabelle *SumMinSector e SumMaxSector* che hanno ricevuto come input rispettivamente le ultime due tabelle citate e come output restituiscono la somma dei prezzi di chiusura delle varie azioni raggruppate per settore e per anno. Per il punto B invece sono state create 4 ulteriori tabelle *actionFirstCloseForYearAndSector , actionLasttCloseForYearAndSector, varPercentActForYearAndSector, maxVarPercentActForYearAndSector* . Queste tabelle di cui le prime due sono di appoggio, servono semplicemente per calcolarsi la massima variazione percentuale per ogni azione raggruppandole però sempre per settore ed anno, in questo modo per esempio avremo per il settore INC nell'anno 2015 soltanto l'azione che possiede la massima variazione percentuale andando a scartare le altre azioni relative allo stesso anno e allo stesso settore. Infine per il punto C del job2 sono state create soltanto due tabelle *sumVolume e MaxVolumeSectorYear* che servono rispettivamente per calcolare la somma dei volumi delle transazioni raggruppandole per settore, anno e ticker, e come nel caso di prima per prendere soltanto le azioni che hanno possiedono il volume massimo. L'ultima tabella *finalSectorHistoricalStocks* è stata creata per calcolare la variazione percentuale descritta precedentemente e per stampare i risultati del report come richiesto, considerando anche l'ordinamento per settore.

Spark: La soluzione Spark proposta per il secondo job è da subito più complessa poichè è necessario, dopo aver salvato e filtrato adeguatamente i due input(righe 5-6 e 8-9), farne il join e costruire adeguatamente la tupla oggetto di lavoro (riga 11). Dalla riga 13 alla riga 16 sono impiegati 3 RDD per calcolare la somma dei volumi per ogni ticker, settore ed anno (coppia *map* e *reduceByKey*), per poi considerare il valore massimo appena calcolato (altra coppia *map* e *reduceByKey*). Discorso analogo per le righe 18-29 in cui vengono impiegati altri otto RDD per il calcolo delle date minime, date massime e somme delle chiusure in tali date, così da poter trovare la variazione percentuale del settore e quella dell'azione con massima variazione percentuale per quell'anno. Vengono eseguite quattro funzioni *map* e cinque funzioni di *reduceByKey*, oltre a un join tra RDD per il calcolo della variazione percentuale. Infine viene costruito l'output RDD costruendo la tupla con un *map* del join tra i vari RDD intermedi, e ordinandolo per il nome del settore.

Algorithm 9 Spark job2

```
1: Begin function job2 (datasetRRD1,datasetRDD2)
2: inputRDD1 = datasetRDD1
3: inputRDD2 = datasetRDD2
4:
5: rdd1 = inputRDD1.map(f=lambda → (tuple))                    ▷ (ticker,(close,volume,date))
6: rdd1 = rdd1.filter(f=lambda → 2009 ≤ YEAR(date) ≤ 2018)     ▷ filtro sulla data
7:
8: rdd2 = inputRDD2.map(f)lambda → (tuple))                    ▷ (ticker,(sector))
9: rdd2 = rdd2.filter(f=lambda → sector ≠ "N/A")               ▷ filtro sul settore
10:
11: rdd3 = rdd1.join(rdd2).map(f=build_RDD)    ▷ build tupla del join con chiave ticker
12:
13: rdd4 = rdd3.map(f=lamda→ (tuple))          ▷ ((sector,YEAR(date),ticker), volume)
14: rdd5 = rdd4.reduceByKey(sum(volume1, volume2))          ▷ somma dei volumi
15: rdd6 = rdd5.map(f=lambda→ (tuple))          ▷ (sector, year), (ticker, volume)
16: rdd6 = rdd6.reduceByKey(max_volume)                     ▷ sum max vol
17:
18: rdd7 = rdd3.map(f=lambda→ (tuple))          ▷ ((sector,year,ticker), (date,close))
19: rdd8 = rdd7.reduceByKey(min(date1, date2))              ▷ minDate
20: rdd9 = rdd7.reduceByKey(max(date1, date2))              ▷ maxDate
21: rdd10 = rdd8.map(f=lambda→ (tuple))          ▷ (sector, year ), first_close)
22: rdd10 = rdd10.reduceByKey(sum(close1, close2))          ▷ sum first close
23: rdd11 = rdd9.map(f=lambda→ (tuple))          ▷ ((sector, year ), last_close))
24: rdd11 = rdd11.reduceByKey(sum(close1, close2))          ▷ sum last close
25: rdd12 = rdd10.join(rdd.11).mapValues(f=varperc_sector(close1, close2))   ▷ var.perc.
26:                                                                          ▷ settore
27: rdd13 = rdd8.join(rdd.9).mapValues(f=varperc_ticker(close1, close2))     ▷ var.perc.
28:                                                                          ▷ ticker
29: rdd14 = rdd13.map(f=lambda→ (tuple))    ▷ ((sector, year), (ticker, max_varperc))
30: rdd14 = rdd14.reduceByKey(max(varperc1, varperc2))      ▷ max var.perc. ticker
31: rdd_output = rdd14.join(rdd.12).mapValues(f=lambda→ (tuple))                  ▷
32:                                  ▷ ((sector,year),(ticker,max_varperc),sector_varperc)
33: rdd_output = rdd_output.join(rdd6).map(f=lambda→ (tuple))                     ▷
34:                          ▷ ((sector,year),(sector_varperc, ticker_varperc,max_varperc,
35:                                                  ▷ ticker_volume,max_volume))
36: rdd_output = rdd_output.sortBy(sector)                  ▷ sort by sector name
37: rdd_output = rdd_output.map(pretty_print)                       ▷ print output
38: return output
39: end of function
```

Alla fine, l'ultimo RDD viene salvato su un file di testo. Tutto ciò che è stato omesso è comunque presente nel codice gitHub.

4. Soluzione - job3

In questa sottosezione saranno proposte alcune soluzioni per ogni tecnologia: Map Reduce, Hive e Spark. Il task richiesto nel job3 è quello di generare un report contenente, le coppie di aziende che si somigliano (sulla base di una soglia scelta a piacere) in termini di variazione percentuale mensile nell'anno 2017 mostrando l'andamento mensile delle due aziende (es. Soglia=1% , coppie: 1:{Apple, Intel}: GEN: Apple +2%, Intel +2,5%, FEB: Apple +3%, Intel +2,7%, MAR: Apple +0,5%, Intel +1,2%, ...; 2:{Amazon, IBM}: GEN: Amazon +1%, IBM +0,5%, FEB: Amazon +0,7%, IBM +0,5%, MAR: Amazon +1,4%, IBM +0,%, ..)

Map Reduce: La soluzione proposta di Map Reduce per il job3 è stata suddivisa in tre stage per via della complessità dello stesso job. In effetti, analogamente al secondo job, è necessario un primo stage in cui viene effettuato un join tra i due file di input *historical_stock_prices.csv*, utilizzando ancora il flag a 0 o 1, ma filtrando questa volta le ennuple provenienti dal primo file riguardanti l'anno 2017 (riga 6).

Algorithm 10 Mapper_join job3

```
1:  Begin funcion mapper_join(historical_stock_prices.csv, historical_stocks.csv)
2:  for line in input do
3:      line = line.split()
4:      initialize temporary variables
5:      if len(line) == 8 then                          ▷ line belong to first file
6:          if '2017-01-01' ≤ date ≤ '2017-12-31' then   ▷ date is ok
7:              set ticker,close,date,volume,temp from line
8:              temp = 0                                ▷ flaggin to 0
9:              print(ticker,close,date,volume,temp)
10:     else                                            ▷ line belong to second file
11:         set ticker,name from line
12:         temp = 1                                    ▷ flagging to 1
13:         print(ticker,name,temp)
14: end for
15: end of function
```

Nel reducer del primo stage, così come per il secondo job, è riconosciuta la provenienza della riga per salvare in due dizionari separati i valori di interesse: *close* per il primo, *name* del settore per il terzo. I valori dei due dizionari vengono poi aggregati in un dizionario che sarà restituito come output dal primo stage *map-reduce*. Come al solito, per brevità si omettono le manipolazioni di tipo delle variabili e la riorganizzazione dell'output.

Algorithm 11 Reducer_join job3

```
1: Begin funcion reducer_join(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker, close, name, date, temp = line.split()
5:     end of function
6:     if temp == 0 then                          ▷ line belong to first file
7:         put close in closeDictionary[ticker, date]
8:     else                                       ▷ line belong to second file
9:         put name in nameDictionary[ticker]
10: end for
11:
12: for ticker, date in closeDictionary.keys() do
13:     if ticker in nameDictionary.keys() then
14:         put close, name in actionDictionary[ticker, date]   ▷ merging into one
15:                                                             ▷ dictionary
16: end for
17: for ticker, date in actionDictionary.keys() do
18:     print(ticker,date,actionDictionary[ticker, date])
19: end for
20: end of function
```

Nel secondo stage *map-reduce* del job3 si entra più nel vivo della loggica applicativa del task. Il mapper è sostanzialmente uguale a quelli già mostrati in precedenza, ossia riorganizza la linea di input da passare al secondo reducer.

Algorithm 12 Mapper1 job3

```
1: Begin funcion mapper1(input)
2: for line in input do
3:     ticker,date,close,name = line.split()      ▷ values from splitted line
4:     print(ticker,date,close,name)
5: end for
6: end of function
```

Più in particolare, il reducer del secondo stage si occupa di inizializzare o aggiornare i dizionari che contengono le chiusure in data minima (righe 7-10), quelle in data massima (righe 13-15), per poi aggregarle calcolando la variazione percentuale per un ticker in un mese (righe 17-22), salvando il tutto in un dizionario che verrà restituito come output per il terzo stage (righe 23-25).

Algorithm 13 Reducer1 job3

```
1: Begin funcion reducer1(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker, name, close, date = line.split()                    ▷ values from splitted
5:
6:                                                  ▷ initialize or update minCloseDictionary
7:     if ticker, MONTH(date) not in minCloseDictionary.keys() then
8:       put name, date, close in minCloseDictionary[ticker, MONTH(date)]
9:     elif DAY(date) ≤ minCloseDictionary[ticker, MONTH(date)] then
10:      put name, date, close in minCloseDictionary[ticker, MONTH(date)]
11:
12:                                                  ▷ initialize or update maxCloseDictionary
13:    if ticker, MONTH(date) not in maxCloseDictionary.keys() then
14:      ....                          ▷ same as minCloseDictionary with ≥ for the DAY
15: end for
16:                    ▷ aggregate into one dictionary, computing the percentage variation
17: for ticker, month in minCloseDictionary.keys() do
18:    if ticker, month in maxCloseDictionary.keys() then
19:      set name, closeFirst, closeLast from minCloseDictionary and maxCloseDictionary
20:
21:      put name, (closeMax − closeMin) / CloseMin * 100 in varMonthDictionary[ticker, month]
22: end for
23: for ticker, month in varMonthDictionary.keys() do
24:    print(ticker,month,varMonthDictionary[ticker, month])
25: end for
26: end of function
```

Arrivati al terzo stage, il mapper usualmente riorganizza le linee prese in input dallo stage precedente e le inoltra al reducer che si occuperà dei calcoli finali da restituire in output.

Algorithm 14 Mapper2 job3

```
1: Begin funcion mapper2(input)
2: for line in input do
3:     ticker,month,name,varper = line.split()          ▷ values from splitted line
4:     print(ticker,month,name,varper)
5: end for
6: end of function
```

L'ultimo reducer merita una discussione leggermente più attenta. In effetti, per la natura del task, è necessario creare, per ogni mese, tutte le coppie di ticker che si comportano similarmente in base ad una *threshold*, a nostra scelta uguale a 1.00%. Per permettere ciò è stato necessario utilizzare un dizionario a più livelli poiché, altrimenti, la computazione

diventava insostenibile con il più grande numero possibile di output. Inizialmente si riempie ed inizializza tale dizionario con la funzione *dict()* (righe 6-10). Successivamente si creano tutte le possibili coppie tra le quali si scelgono quelle con i due ticker che mostrano similarità (righe 12-24). Infine si riorganizza tale dizionario in un altro di appoggio per la stampa (righe 27-34), così da permettere che l'output sia come nell'esempio descritto all'inizio di questa sezione.

Algorithm 15 Reducer2 job3

```
1: Begin funcion reducer2(input)
2: Initialize dictionaries
3: for line in input do
4:     ticker,month,name,varper = line.split()                    ▷ values from splitted
5:                 ▷ initialize multilevel dictionary to create ticker couples for every month
6:     if month not in varMonthDictionary.keys() then
7:         initialize varMonthDictionary[month]                                ▷ dict()
8:     if ticker not in varMonthDictionary[month] then
9:         put name, varper in varMonthDictionary[month][ticker]
10: end for
11:                                   ▷ create all possible couples of ticker per month
12: for month in varMonthDictionary.keys() do
13:     for tickerItem in varMonthDictionary[month].items() do
14:         for ticker1Item in varMonthDictionary[month].items() do
15:             if tickerItem != ticker1Item then
16:                 set name, name1, varper, varper1 from tickerItem, ticker1Item
17:                         ▷ verify the tickers are similar in this month (threshold = 1.00%)
18:                 if abs(varper - varper1) ≤ 1 and (ticker, ticker1, month) not in
19:                                                   similarityDictionary.keys() then
20:                     put name,name1,varper,varper1 in
21:                                         similarityDictionary[ticker, ticker1, month]
22:         end for
23:     end for
24: end for
25:
26:                         ▷ aggregate similar couples of ticker in one dictionary for output
27: for ticker, ticker1, month in similarityDictionary.keys() do
28:     if (ticker, ticker1) not in printDictionary.keys() then
29:         put similarityDictionary[ticker,ticker1,month] concat month in
30:                                     printDictionary[ticker, ticker1]
31:     else
32:         update printDictionary[ticker, ticker1] with
33:                         similarityDictionary[ticker, ticker1, month] concat month
34: end for
35: print(printDictionary)                          ▷ return all the values of the dictionary
36: end of function
```

Hive: Nell'implementazione di Hive del job3, come nel job2, sono stati prelevati i dati da entrambi i file *csv* descritti nella sezione 1. A differenza del job2, il join tra le due tabelle, sempre sul ticker_id, è stato effettuato per ricavare il nome dell'azienda relativa all'azione emessa. Inoltre nel join sono state selezionate soltanto le date relative all'anno 2017 e sono state scartati i campi "name" nulli. Inizialmente sono state create due tabelle *mindateAction e maxdateAction* per ricavare rispettivamente la minima data e la massima data raggrupate per ticker _id e il mese. In seconda battuta sono state crete due ulteriori tabelle *firsCloseAction2017 e lastCloseAction2017* per ricavare rispettivamente il prezzo di chiusura della prima data e il prezzo di chiusura dell'ultima data per ogni azione. Successivamente sono state prodotte altre due tabelle *varPercentForTicker e varPercentActionWithTreshold*, la prima per riceve in input le tabelle *firstClose e lastClose* e calcola la variazione percentuale per ogni azione in relazione ad ogni mese del 2017, mentre la seconda crea tutte le possibili coppie di azioni, scartando tutte le coppie per cui *ticker_id1 = ticker_id2* e calcola la differenza tra le variazione percentuali dei due elementi della coppia. Infine nell'ultima tabella *outputJob3* viene generato l'output come richiesto dal report, considerando solo le coppie che possiedono una differenza della variazione percentuale minore e uguale ad una soglia scelta arbitrariamente (nel nostro caso threshold=1).

Spark: La soluzione Spark proposta per il terzo job è, nella prima parte, simile a quella del secondo job, in quanto necessario il salvataggio dei due input in due RDD diversi, il filtraggio del primo input per l'anno 2017 ed il join tra i due RDD con relativo build della tupla tramite funzione *map* (righe 2-10). Successivamente si usa l'RDD del join per creare tramite *reduceByKey*, due RDD: uno per la data minima e uno per la data massima (righe 13-14). Questi vengono poi aggregati tramite join in modo da poter calcolare la variazione percentuale per quel ticker in quel mese, tramite funzione *map* (righe 15-16). Vengono poi mappati i valori in una lista per permettere l'impiego di una *flatMap* con la quale si possono raggruppare per ogni coppia di ticker, la lista dei mesi in cui essi sono simili per variazione percentuale, sulla base di una threshold = 1.00% (righe 20-22). Infine l'output viene riorganizzato con una funzione *map* per una stampa migliore.

Algorithm 16 Spark job3

```
1: Begin function job3 (datasetRRD1,datasetRDD2)
2: inputRDD1 = datasetRDD1
3: inputRDD2 = datasetRDD2
4:
5: rdd1 = inputRDD1.map(f=lambda → (tuple))                    ▷ (ticker,(close,date))
6: rdd1 = rdd1.filter(f=lambda → YEAR(date) == 2017)           ▷ filtro sulla data
7:
8: rdd2 = inputRDD2.map(f=lambda → (tuple))                    ▷ (ticker,(name))
9:
10: rdd3 = rdd1.join(rdd2).map(f=month_ticker)                 ▷ ((MONTH(date), ticker),
11:                                                            ▷ (close, date, name))
12:
13: rdd4 = rdd3.reduceByKey(min(date1, date2))                 ▷ minDate
14: rdd5 = rdd3.reduceByKey(max(date1, date2))                 ▷ maxDate
15: rdd6 = rdd4.join(rdd5)                                     ▷ minDate e maxDate
16: rdd7 = rdd6.map(f=var_percent_act)                         ▷ (month ,(ticker, name ,varperc))
17: rdd8 = rdd7.groupByKey().mapValues(list)                   ▷ valori in lista
18: rdd9 = rdd8.mapValues(f=similar_pairs)                     ▷
19:                          ▷ (month, [((ticker1,name,varperc),(ticker2,name,varperc)), ...])
20: rdd10 = rdd9.flatMap(f=to_print).map(f=lambda → (tuple))   ▷
21:                          ▷ ((ticker1,ticker2), (month,name1,varperc1,name2,varperc2))
22: rdd11 = rdd10.groupByKey()
23: output_rdd = rdd11.map(f=pretty_print)                     ▷ output for print
24: return output
25: end of function
```

Alla fine, l'ultimo RDD viene salvato su un file di testo. Tutto ciò che è stato omesso è comunque presente nel codice gitHub.

5. Ambiente di esecuzione

L'esecuzione dei tre job per ogni tecnologia è avvenuta in due ambienti diversi: macchina locale e cluster su cloud AWS. Per il primo ambiente è stata creata una VM Oracle Virtual Box con Ubuntu-18.04(64-bit) a cui sono stati assegnati 8GB di RAM, 60GB di disco fisso SSD e processore host Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. Per il secondo ambiente è stato creato un cluster con il servizio di AWS EMR v-5.32.0, hardware m5.xlarge con 3 numeri di istanze(1 master-2 nodi principali).

L'hardware utilizzato per la sperimentazione è un DELL Inspiron 15 7000 con 16GB di RAM, , scheda grafica NVIDIA GeForce GTX 1060Ti.

6. Risultati

In questa sezione sono analizzate le performance dei vari job richiesti sia in forma grafica, sia in forma tabellare. Inoltre, in entrambe le visualizzazioni sono state messe a paragone i tempi di esecuzione dei vari job in locale con quelli del cluster AWS m5.xlarge su 3 contenitori. Come si può notare dai risultati riportati le performance su cluster rispetto a quelle in locale sono migliorate di media più del 50 % soprattutto al variare dell'input, in particolare sui run da 20 mln, infatti abbiamo notato come al variare dell'input, le risorse assegnate alla macchina virtuale descritta precedentemente non siano state sufficienti per avere le stesse prestazioni del cluster.

6.1 Risultati Tabellari

Input Size	MapReduce	Hive	Spark
10k	28	241	46
100k	28	273	46
1 mln	37	312	60
10 mln	240	421	188
20 mln	422	726	374

Table 1: Tempi Job1 in Locale in sec

Input Size 1	MapReduce	Hive	Spark
10k	24	28	29
100k	24	28	29
1 mln	35	40	42
10 mln	92	115	61
20 mln	149	183	91

Table 2: Tempi Job1 su Cluster in sec

Input Size	MapReduce	Hive	Spark
10k	47	375	57
100k	49	387	60
1 mln	97	500	83
10 mln	322	606	423
20 mln	864	747	904

Table 3: Tempi Job2 in Locale in sec

Input Size	MapReduce	Hive	Spark
10k	36	28	33
100k	36	28	33
1 mln	56	39	45
10 mln	151	66	187
20 mln	315	121	375

Table 4: Tempi Job2 su Cluster in sec

Input Size	MapReduce	Hive	Spark
10k	44	205	47
100k	50	212	48
1 mln	73	251	60
10 mln	279	634	257
20 mln	813	2188	797

Table 5: Tempi Job3 in Locale in sec

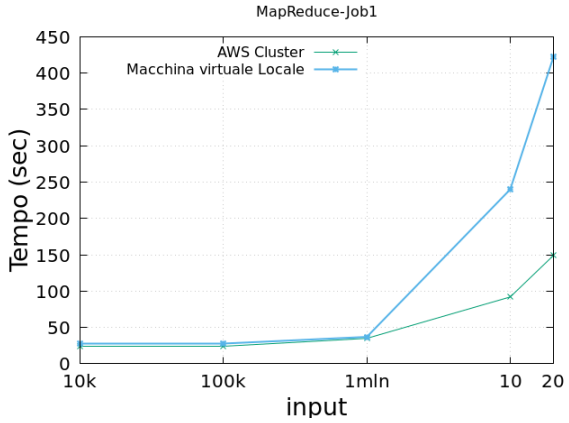
Input Size 1	MapReduce	Hive	Spark
10k	26	35	28
100k	26	35	28
1 mln	42	180	40
10 mln	199	444	135
20 mln	384	808	271

Table 6: Tempi Job3 su Cluster in sec

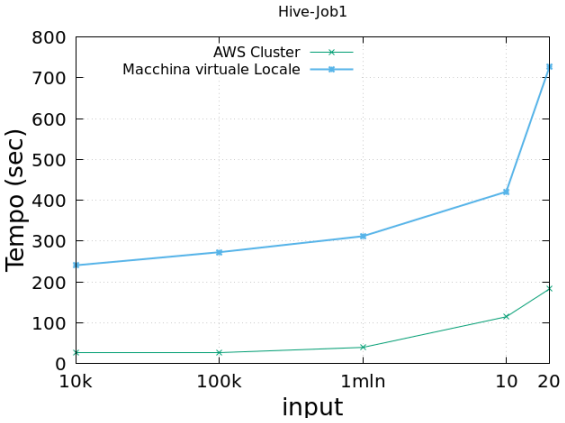
6.2 Risultati Grafici

La sperimentazione delle varie soluzioni proposte ha permesso uno studio comparativo dal punto di vista dell'ambiente di esecuzione dei job. In questa sotto-sezione sono riportati i grafici tempo/input di ogni esecuzione dei job sulle 3 diverse tecnologie, per la realizzazione dei grafici è stato utilizzato un programma di tipo command-driven chiamato *GnuPlot*.

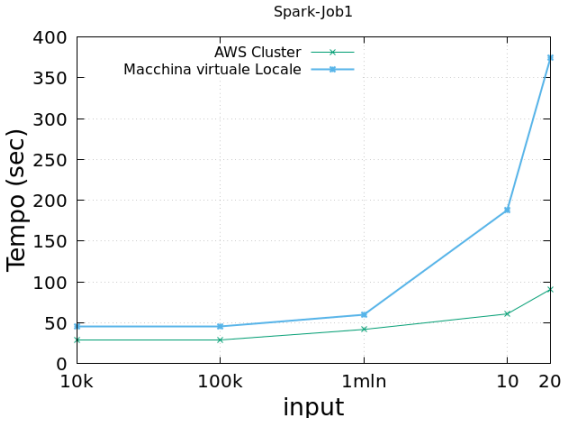
6.2.1 Job1:



(a) Grafico MapReduce



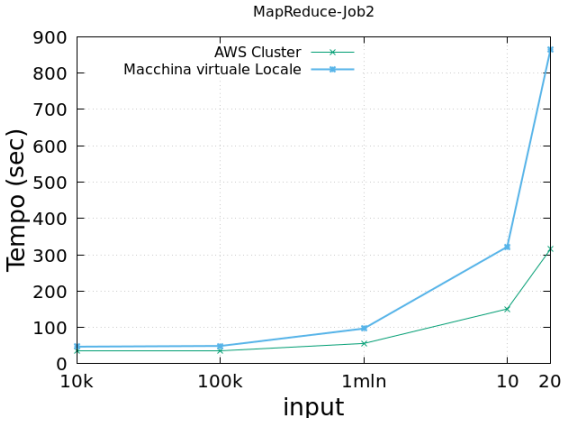
(b) Grafico Hive



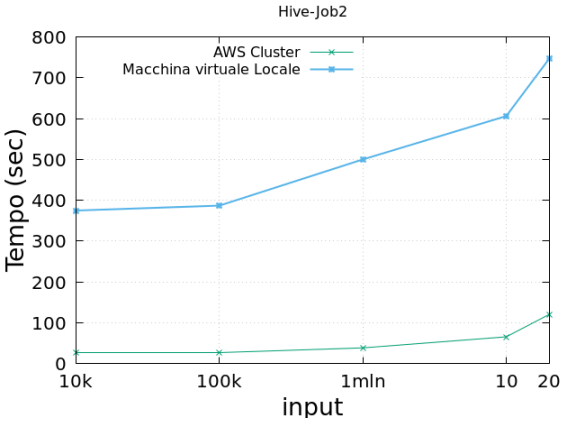
(c) Grafico Spark

Figure 1: Risultati grafici a paragone del job1 tra cluster e macchina virtuale delle 3 tecnologie utilizzate

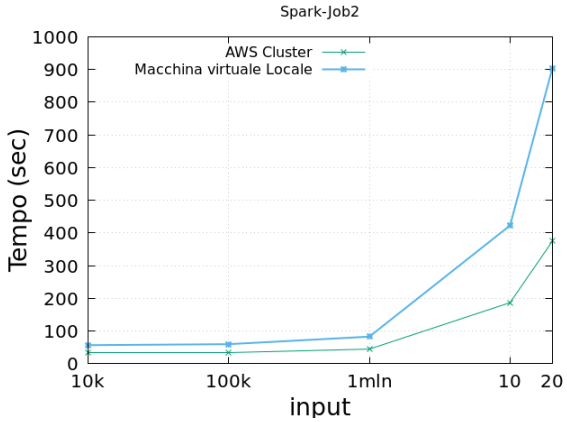
6.2.2 Job2



(a) Grafico MapReduce



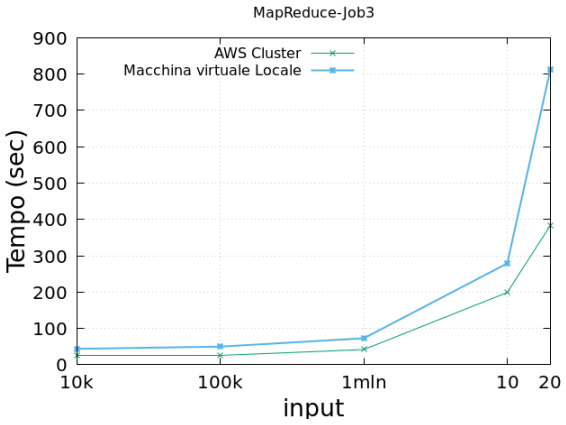
(b) Grafico Hive



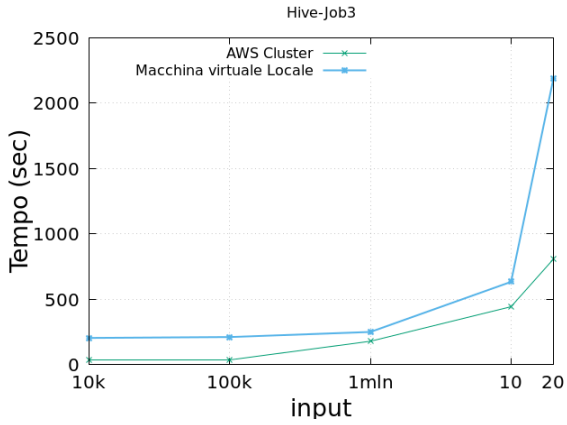
(c) Grafico Spark

Figure 2: Risultati grafici a paragone del job2 tra cluster e macchina virtuale delle 3 tecnologie utilizzate

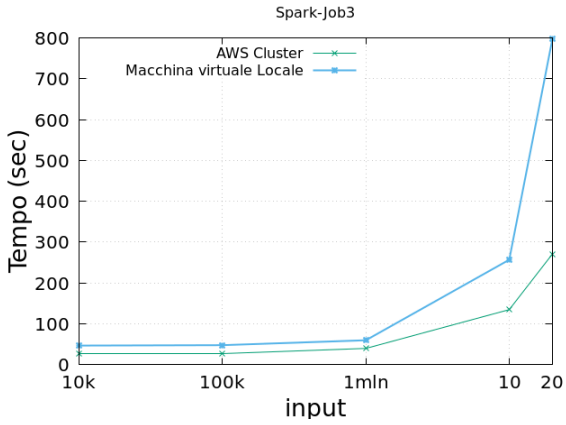
6.2.3 Job3



(a) Grafico MapReduce



(b) Grafico Hive



(c) Grafico Spark

Figure 3: Risultati grafici a paragone del job3 tra cluster e macchina virtuale delle 3 tecnologie utilizzate