Big Data, Primo Progetto

Relazione Finale: Menta Vanessa, Silvestri Corrado

Gruppo The Organizer Team, Link repository: https://github.com/konrad169/Big_Data_project_19-20

1)Analisi del dataset

Sono stati analizzati i due dataset per il progetto, per ricercare se eventualmente presenti stringhe vuote, record duplicati o errori nella formattazione del dataset.

1.1)Historical_stock_prices.csv

Il dataset contiene 20973889 righe ed 8 colonne per una dimensione di 2 GB.Ogni record è composto dai seguenti campi:

- Ticker:simbolo azionario della società
- Open: prezzo di apertura
- Close : prezzo di chiusura
- Adj_close:prezzo di chiusura modificato
- Low:prezzo minimo
- High: prezzo massimo
- Volume:numero di transizioni
- Data:nel formato aaaa-mm-gg

La prima attività effettuata sul dataset HSP è stata quella di cercare eventualmente dati nulli su tutte le righe.Tramite la libreria pandas di python utilizzando il metodo .isnull().any(), è stato effettuato un controllo che ha dato esito negativo.
Successivamente abbiamo accertato, con successo, che non ci siano record duplicati nel dataset, ovvero righe che contenessero lo stesso ticker e la stessa data sempre tramite l'utilizzo della libreria di pandas.

1.1.1)HSP Data cleaning

È stata prodotta una coppia di script in Python che in maniera preventiva eliminasse tutte le righe e colonne non rilevanti ai fini del progetto, chiamati clean_data_job12.py & clean_data_job3.py presenti nella cartella Cleaning_Increase_Data del repository.

Lo script clean_data_job12.py esegue una pulizia ad hoc sul dataset HSP originale,utile per un esecuzione leggera dei primi due job, riducendone notevolmente le dimensioni. Ciò viene fatto, con l'utilizzo di pandas, tramite l'eliminazione della colonna Adj_close e quella dei record che non rientrassero nell'intervallo temporale [2008-2018] portando la dimensione dai 2 GB a circa 970MB.
Lo script clean_data_job3.py esegue una pulizia utile soltanto per il job 3 su HSP. Infatti come prima è stata eliminata interamente la colonna Adj_close e l'intervallo temporale è stato ridotto al triennio [2016-2018]. Ciò ha portato ad una riduzione sostanziale del dataset dai 2GB sopra citati a circa 300 MB.
Questi script diminuirebbero notevolmente i tempi di esecuzione per i job su tutte le tecnologie , ma non sono stati utilizzati, sia nell'esecuzione in locale che in quella sul cluster, perché avrebbero portato ad una riduzione della complessità del progetto e dei relativi job.

1.1.2)HSP Data Increase

Ai fini del progetto, sono stati prodotti invece dei dataset aumentati , cosi da poter constatare l'eventuale aumento delle tempistiche d'esecuzione dei job al crescere dei record da analizzare. Tramite lo script hsp_increase.py presente nella cartella Cleaning_Increase_Data del repository vengono generati dei dataset di dimensioni molto maggiori a quello originale. Ciò viene fatto, utilizzando sempre la libreria pandas di pyhton , effettuando l'append del dataset in coda a quello originale, togliendo la prima riga che contiene i nomi delle colonne. Lo script presenta una variabile iterations che è possibile impostare per ottenere un aumento del dataset desiderato. Lo script è stato eseguito 3 volte, generando i seguenti dataset:

1) X3, in cui l'append su HSP viene effettuato per tre volte, passando da 2 GB a 8.2 GB. Il numero delle righe del nuovo file è 83895556. , il cui rapporto tra le righe del dataset aumentato con quello originale è circa 4.
2) X5, in cui l'append su HSP viene effettuato per cinque volte, passando da 2 GB a 12,3GB. Il numero delle righe del nuovo file è 125843334, il rapporto righe dataset aumentato con quello originale è circa 6

3) X10, in cui l'append su HSP viene effettuato per dieci volte, passando da 2 GB a 22.6 GB. Il numero delle righe del nuovo file è 230712779 , il rapporto righe dataset aumentato con quello originale è circa 11

I Dataset generati non sono presenti nel repository di progetto per ovvi motivi di spazio.

1.2)Historical_stocks.csv

Il dataset contiene 6460 righe e 5 colonne.Ogni record è composto dai seguenti campi:

- Ticker:simbolo azionario della società
- Exchange: Mercato in cui è quotata la società, NYSE o NASDAQ
- Name : nome della società
- Sector: settore in cui opera la società
- Industry: Industria di riferimento per la società

Anche su questo dataset è stato effettuato un controllo tramite la libreria pandas come nel caso precedente per verificare l'eventuale presenza di valori nulli e o duplicati.La ricerca restituito la presenza di valori nulli sui campi sector ed industry mentre non ha evidenziato la presenza di valori duplicati nel dataset. I campi nulli presenti in questo dataset saranno scartati in ogni job per ogni tecnologia.

1.3)Unione tra Historical_stock_prices.csv Historical_stock.csv

Analizzando le consegne del progetto, nei job2 e 3 è necessario effettuare il join tra i due csv per ottenere il settore e il nome dell'azienda presenti soltanto nel file Historical_stocks.csv.
Dall'unione dei due dataset possiamo notare la presenza di aziende con lo stesso ticker operanti in settori diversi oppure di aziende che operano in entrambi i mercati.
Un esempio è dato dalle compagnie :

- [('ENERGIZER HOLDINGS, INC.', 'CONSUMER NON-DURABLES'), ('ENERGIZER HOLDINGS, INC.', 'MISCELLANEOUS')] che è presente in più settori
- [('AMTRUST FINANCIAL SERVICES, INC.', 'NASDAQ'), ('AMTRUST FINANCIAL SERVICES, INC.', 'NYSE')] che è quotata in entrambe le borse.

2)Primo job

2.1)Specifica

Un job che sia in grado di generare le statistiche di ciascuna azione tra il 2008 e il 2018 indicando, per ogni azione: (a) il simbolo, (b) la variazione della quotazione (differenza percentuale arrotondata tra i prezzi di chiusura iniziale e finale dell'intervallo temporale), (c) il prezzo minimo, (e) quello massimo e (f) il volume medio nell'intervallo, ordinando l'elenco in ordine decrescente di variazione della quotazione.

2.2)Map reduce

Viene creata una classe *startJob1_local_hql* che consente di prendere i dati dal csv *historical_stock_prices* e di eseguire il job.
Viene definita una classe *Mapper.py*, per la fase di mapping, in essa vengono estrapolati i campi di interesse per ogni record, ovvero: *ticker, close, low, high, volume* e *date*, il tutto nell'intervallo d'interesse 2008-2018. In particolare, ciò è possibile definendo una chiave costituita da due campi:

- *Ticker*, la chiave principale
- *Date*, la chiave secondaria.

I valori perciò risulteranno essere aggregati in base al campo *ticker* e ordinati in base al campo *date* in modo da facilitare la fase di reduce.
Segue lo pseudocodice dell'implementazione:

```
class mapper:
    map(key, record):
        ticker, _, close, _, low, high, volume, date = specifics
        year = getYear(date)
        if year in range from 2008 to 2018:
            key = ticker, date
            value = close, low, high, volume
            print(key, value)
```

Successivamente, viene definita una classe *Reducer.py* nella quale si definisce una variabile globale *output*, contenente una lista di strutture di dati, degli item, ognuno dei quali risulta essere composto dai campi: *ticker, percentageChange, minLowPrice, maxHighPrice, avgVolume.*
Ogni item è stato generato da una coppia <ticker,valore> che era stata passata in input dopo la fase di mapping. In particolare, per calcolare:

- *percentageChange,* ovvero la variazione della quotazione, viene creata una lista *insertItemToLista()* da cui vengono estratti il campo *close* del primo e dell'ultimo elemento della lista per il calcolo della differenza percentuale, processo reso possibile grazie all'ordinamento per data fatto in fase di mapping;
- *avgVolume*, il volume medio, vengono estratti i valori di *volume* associati al ticker corrente, si sommano e successivamente il risultato ottenuto viene diviso per il numero di occorrenze riscontrato;
- *minLowPrice,* il prezzo minimo, viene estratto il campo *low*;
- *maxHighPrice,* il prezzo massimo, viene estratto il campo *high.*

A questo punto viene effettuato un controllo sull'anno degli item trovati per vedere che essi ricadano nell'intervallo d'interesse, in caso di esito positivo essi vengono memorizzati nella *listaOrdinata*; tale lista viene ordinata in base alla variazione della quotazione in maniera decrescente.
Segue lo pseduocodice dell'implementazione:

```
class reducer:
  setup():
        stats = empty list
  reduce(ticker, records):
         totalVolume = 0
         countVolume = 0
         minLowPrice = infinity
         maxHighPrice = - infinity
    # get price variation
         closePriceStartingValue = values.getFirstElement().getClose()
         closePriceFinalValue = values.getLastElement.getClose()
         percentageChange = (closeFinalPrice - closeStartingPrice)/closeStart ingPrice
    # compute remaining values
          for each value in valueList:
                totalVolume += volume for current value
                countVolume += 1
                minLowPrice = min(minLowPrice, low price for current value)
                maxHighPrice = max(maxHighPrice, high price for current value)
    # get volume
     avgVolume = sommaVolume/contatoreVolume
    # add this item to stats list
     startingDate = values.getFirstElement().getYear()
     endingDate = values.getLastElement().getYear()
     if startingDate == 2008 and endingDate == 2018:
              obj = {ticker, percentageVariation, minLowPrice, maxHighPrice, avgV olume}
               stats.append(obj)
       cleanup()
               listaOrdinata = sortByPercentageChange(output, reverse=True)
               for i in range(10):
                     print(sortedStats.getItem(i))
```

2.3)Hive

I dati dal csv vengono caricati nella table: *historical_stock_prices1 (ticker STRING, open float, close float, adj_close float, low float, high float, volume float, dates date).*
Inizialmente viene definita una prima vista *ticker_min_max_avg* che per ogni simbolo (*ticker*) nell'intervallo d' interesse trova: il prezzo minimo (*min_price*) come minimo di *low*, il prezzo massimo (*max_price*) come massimo di *high* e il volume medio (*avg_volume*).
Con le due viste *min2008* e *max2008* invece si calcolano rispettivamente: la data meno recente (*min_data*) e la data più recente (*max_data*).

Invece, con le due viste *ticker_chiusura_iniziale* e *ticker_chiusura_finale* si calcolano per ciascun *ticker* il valore di chiusura (*close*) per la data meno recente (*min_data*) e per la data più recente (*max_data*), riprese dalle due viste precedenti.
Infine, la vista *variazione_quotazione* calcola la differenza percentuale arrotondata tra i prezzi di chiusura iniziale e finale nell'intervallo temporale per ogni *ticker* basandosi su *ticker_chiusura_iniziale* e *ticker_chiusura_finale*.
Attraverso il *join* tra le viste *ticker_min_max_avg* e *variazione_quotazione* si ottiene il risultato nella tabella *risultati1*.
Segue lo pseduocodice dell'implementazione:

```
CREATE VIEW IF NOT EXISTS ticker_min_max_avg AS
SELECT ticker, min(low) AS min_price, max(high) AS max_price, avg(volume) AS avg_volume
FROM historical_stock_prices1
WHERE dates>='2008-01-01' AND dates<='2018-12-31'
GROUP BY ticker;

CREATE VIEW IF NOT EXISTS min2008 AS
SELECT ticker, min(dates) AS min_data
FROM historical_stock_prices1
WHERE dates>='2008-01-01' AND dates<='2018-12-31'
GROUP BY ticker;

CREATE VIEW IF NOT EXISTS max2018 AS
SELECT ticker, max(dates) AS max_data
FROM historical_stock_prices1
WHERE dates>='2008-01-01' AND dates<='2018-12-31'
GROUP BY ticker;

CREATE VIEW IF NOT EXISTS ticker_chiusura_iniziale AS
SELECT hsp.ticker, hsp.dates, hsp.close
FROM min2008 AS min, historical_stock_prices1 AS hsp
WHERE hsp.ticker=min.ticker AND hsp.dates=min.min_data;

CREATE VIEW IF NOT EXISTS ticker_chiusura_finale AS
SELECT hsp.ticker, hsp.dates, hsp.close
FROM max2018 AS max, historical_stock_prices1 AS hsp
WHERE hsp.ticker=max.ticker AND hsp.dates=max.max_data;

CREATE VIEW IF NOT EXISTS variazione_quotazione AS
SELECT ci.ticker, FLOOR(((cf.close-ci.close)/ci.close)*100) AS variazione
FROM ticker_chiusura_finale AS cf join ticker_chiusura_iniziale AS ci on cf.ticker=ci.ticker;

CREATE TABLE IF NOT EXISTS risultati1 AS
SELECT a.ticker, b.variazione, a.min_price, a.max_price, a.avg_volume
FROM ticker_min_max_avg AS a join variazione_quotazione AS b on a.ticker=b.ticker
ORDER BY b.variazione DESC limit 10;
```

2.4)Spark

Sono specificati i seguenti RDD per l'esecuzione del job:
ticker_close_minimo, contiene per ciascun ticker il prezzo minimo di close nell'intervallo
*ticker_close_massimo,*contiene per ciascun ticker il prezzo massimo di close nell'intervallo
ticker_volume_medio, contiene per ciascun ticker il volume medio giornaliero
massimo_close_assoluto, contiene per ciascun ticker il prezzo di chiusura relativo alla data più recente
minimo_close_assoluto, contiene per ciascun ticker il prezzo di chiusura relativo alla data meno recente

Su questi due ultimi RDD viene effettuato il join per ottenere la *join_variazione_percentuale*, su cui sarà calcolata successivamente la variazione percentuale per ogni ticker tramite formula presente nell'RDD *variazione_percentuale.*
Una volta ottenuti tutti i dati viene effettuato il join tra gli RDD *ticker_close_minimo, ticker_close_massimo, ticker_volume_medio, variazione_percentuale* per ottenere il risultato. L'RDD del risultato sarà poi ordinato

in base alla variazione percentuale , in ordine decrescente, come richiesto nella specifica e saranno presi i primi 10 record prodotti.
Segue lo pseduocodice dell'implementazione:

funzione massimo(x,y)=valuta il massimo tra due valori
funzione minimo(x,y)=valuta il minimo tra due valori

input = legge tutte le righe del file historical_stock_prices.csv con data compresa tra il 2008 e il 2018
persist input in memory (ed opzionalmente vengono memorizzati sul disco se RDD non è adeguato alla memoria)

```
ticker_close_minimo = input.map(linea -> (ticker, low))
                           .reduceByKey(min(low1, low2))
ticker_close_massimo = input.map(linea -> (ticker, high))
                           .reduceByKey(max(high1, high2))
ticker_volume_medio = input.map(linea -> (ticker, (volume,1)))
                           .reduceByKey((volume1+volume2, count+1))
                           .map(linea -> (ticker, TotVolume/count))
minimo_close_assoluto = input.map(linea -> (ticker, (close, data)))
                           .reduceByKey(minimo((data1,close1), (data2,close2)))
massimo_close_assoluto = input.map(linea -> (ticker, (close, data)))
                           .reduceByKey(massimo((data1,close1), (data2,close2)))
join_variazione_percentuale = ticker_close_minimo.join(ticker_close_massimo)

variazione_percentuale = join_variazione_percentuale
                           .map(linea -> (ticker, (maxclose-minclose)/minclose))

output = ticker_close_massimo.join(ticker_close_minimo)
                           .join variazione_percentuale)
                           .join(ticker_volume_medio)
                           .sortBy(variazione percentuale decrescente) .take(10)
```

2.5)Risultati ottenuti

ticker	Variazione %	Prezzo_min	Prezzo_Max	Volume_Medio
EAF	267757.13458927936	0.0020000000949949	24.364000320434602	1245139.0384615385
ORGS	217415.9261280251	0.00313999992795289	19.9200000762939	8570.962343096235
PUB	179900.0040233133	0.008999999612569809	138.0	34449.40070921986
RMP	121081.60190125568	0.0299999993294477	79.818733215332	120487.04767063922
CTZ	120300.00230968006	0.00499999988824129	26.8400001525879	52050.27308066084
CCD	111250.00477768485	0.0149999996647239	25.9799995422363	105416.8926615553
SAB	110428.76995446288	1.36549997329712	319600.0	1695378.0163179915
KE	99400.00031664963	0.0149999996647239	22.4500007629395	73249.84615384616
LN	96699.99920527148	0.0149999996647239	51.4799995422363	394344.92964071856
GHG	64850.000307336515	0.00499999988824129	25.100000381469698	607659.2937293729

3)Secondo Job

3.1)Specifica

Un job che sia in grado di generare, per ciascun settore, il relativo "trend" nel periodo 2008-2018 ovvero un elenco contenete, per ciascun anno nell'intervallo: (a) il volume annuale medio delle azioni del settore, (b) la variazione annuale media delle aziende del settore e (c) la quotazione giornaliera media delle aziende del settore.

3.2)Map Reduce

Per leggere il file *historical_stocks.csv* utilizziamo la cache distribuita: (https://hadoop.apache.org/docs/r3.0.0/api/org/apache/hadoop/filecache/DistributedCache.html) la quale permette di mettere a disposizione dei worker i file che gli occorrono per svolgere le loro attività.

Viene creata una classe Mapper.py, la quale contenente la struttura dati *tickerToSectorMap* che verrà usata in fase di mapping per effettuare il join tra i due csv, ovvero, ad ogni ticker del file *historical_stock_prices.csv* sarà associato il settore presente in *historical_stocks.csv.*
Successivamente vengono estrapolati i campi: *ticker, close, volume, date* e si verifica che il record sia relativo all'intervallo temporale preso in esame.
Si otterranno delle coppie <chiavi,valore> per ogni record in cui:

- la chiave risulterà essere composta da tre campi: *sector,* la chiave primaria, *ticker* e *date*, le chiavi secondarie utilizzate per fare l'ordinamento, in questo modo i valori appartenenti allo stesso settore saranno ordinati in base al *ticker* e alla *date.* Tale ordinamento gerarchico consente di semplificare notevolmente la fase di reduce, in modo che si possa procedere direttamente al calcolo delle differenze percentuali.
- Il valore risulterà composto dai campi *volume* e *close.*

Segue lo pseduocodice dell'implementazione:

```
class mapper:
    setup():
        read the historical_stocks.csv file from the Distributed Cache
        tickerToSectorMap = map which associate a (non null) sector for each ticker in  historical_stocks.csv
    map(key, record):
        ticker, _, close, _, _, _, volume, date = record
        year = getYear(date)
        if year in range from 2008 to 2018 and
        ticker has a corresponding sector in tickerToSectorMap:
                sector = ticker's corresponding sector
                key = sector, ticker, date
                value = close, volume
                print(key, value)
```

Successivamente viene implementata una classe *Reducer.py* per effettuare la fase di reduce nella quale, per ciascun settore, verranno definite delle mappe <chiave,valore> finalizzate al calcolo del volume annuale medio delle azioni del settore, della quotazione giornaliera e la variazione annuale media delle aziende del settore. In particolare, per calcolare:

- Il volume annuale medio, si controllano i record di ogni settore e quando il relativo volume viene sommmato solo se i record appartengono allo stesso anno. Il calcolo del volume annuale è dato dal rapporto tra il volume sommato per uno specifico anno e il conteggio delle occorrenze (*entireVolume/countVolume).*
- Per il calcolo della quotazione giornaliera in un certo anno si sommano i prezzi di chiusura dei ticker relativi a quella data e in seguito se ne calcola la media.
- La differenza percentuale, si sfrutta l'ordine associato ad ogni settore creato in fase di mapping per estrarre il prezzo di chiusura più recente e quello meno recente per ogni ticker di ogni anno. Dopo di che, tali valori verranno sommati per ottenere il prezzo di apertura dell'anno e il prezzo di chiusura dell'anno e grazie a questi due nuovi valori, che definiamo come la quotazione iniziale e la quotazione finale, si potrà procedere al conteggio della differenza percentuale.

Segue lo pseduocodice dell'implementazione:

```
class reducer:
        reduce(sector, records):
        yearToStartingCloseValueMap = empty map
        yearToEndingCloseValueMap = empty map
        yearToTotalVolumeMap = empty map
        yearToTotalCloseValue = empty nested map
        for record in records:
                day = get date field from the current record
                year = get year value from the day variable
                if the record corresponding ticker has changed from the previous record:
```

```
                yearToEndingCloseValueMap[previous year] += close field taken from the previous record
                  yearToStartingCloseValueMap[year] += close field taken from the current record
                  yearToTotalVolumeMap[year] += volume field taken from the current record
        countVolume[year] += 1
        yearToTotalCloseValue[year][date] += close
        percentChange = empty map
        for each year from 2008 to 2018:
            difference = yearToEndingCloseValueMap[year] - yearToStartingC loseValueMap[year]
            percentChange[year] = difference/yearToStartingCloseValueMap[ year]
            count = number of date keys in yearToTotalCloseValue[year][date]
            averageClosePrice[year] = yearToTotalCloseValue[year][date]/count
            annualMeanVolume[year] = yearToTotalVolumeMap[year]/countVolume[year]
            print(sector, year, annualMeanVolume[year], percentChange[year], averageClosePrice[year])
```

3.3)Hive

I dati dai csv vengono caricati nelle tabelle:
historical_stock_prices (ticker STRING, open float, close float, adj_close float, low float, high float, volume float, dates STRING)
historical_stock (ticker STRING, exchanges STRING, name STRING, sector STRING, industry STRING).
Inizialmente viene effettuato il join dei due csv nella tabella *join_csv2* per avere a disposizione i ticker nell'intervallo di interesse che appartengono ad un settore, di conseguenza vengono scartati tutti quelli che non hanno un settore di appartenenza.
Per calcolare il volume annuale medio vengono usate due viste:

- nella prima, *volume_settore,* viene calcolata la somma dei volumi di un settore (*somma_volume*) in tutti i giorni dell'anno e si fa un conteggio dei ticker relativi ad ogni settore (*contatore*);
- nella seconda, *volume_medio_settore,* viene calcolato il volume annuale medio delle azioni del settore (*avg_volume*) come il rapporto tra la somma dei volumi (*somma_volume*) e il conteggio dei ticker (*contatore*).

Nella vista *dates_min_max* vengono calcolate per ciascun ticker e per ciascun anno la data meno recente (*min_data*) e quella più recente (*max_data*).
Successivamente vengono definite due viste: *quotazione_inizio_anno2* e *quotazione_fine_anno2* che calcolano rispettivamente il prezzo di chiusura associato alla data meno recente e quello associato alla data più recente per ciascun settore e per ogni anno, basandosi sulla vista *dates_min_max* creata precedentemente.
A questo punto, nella tabella *variazione settore* si calcola la variazione annuale media (*differenza_percentuale*) come differenza percentuale tra la quotazione di fine anno e quella di inizio anno, utilizzando le due viste *quotazione_inizio_anno2* e *quotazione_fine_anno2*.
Infine, per calcolare la quotazione giornaliera media delle aziende del settore, vengono usate due viste:

- nella prima, *prezzo_chiusura*, si calcola la somma dei prezzi di chiusura di uno stesso giorno per tutti i ticker appartenenti allo stesso settore;
- nella seconda, *quotazione_giornaliera_media*, si fa la media della somma precedente.

Nella tabella *risultati2* viene effettuato il join tra *quotazione_giornaliera_media*, *variazione_settore2* e *volume_medio_settore* per ottenere l'output.
Segue lo pseduocodice dell'implementazione:

```
CREATE TABLE IF NOT EXISTS join_csv2 AS
SELECT hs.sector, hsp.ticker, hsp.dates, hsp.close, hsp.volume
FROM historical_stock AS hs JOIN historical_stock_prices AS hsp ON hsp.ticker=hs.ticker
WHERE YEAR(hsp.dates)>=2008 AND YEAR(hsp.dates)<=2018 AND hs.sector!='N/A';

CREATE VIEW IF NOT EXISTS volume_settore AS
SELECT sector, YEAR(dates) AS anno, SUM(volume) AS somma_volume, COUNT(ticker) AS contatore
FROM join_csv2
GROUP BY sector, YEAR(dates);

CREATE VIEW IF NOT EXISTS volume_medio_settore AS
SELECT sector, anno, ROUND(somma_volume/contatore) AS avg_volume
FROM volume_settore;
```

```
CREATE VIEW IF NOT EXISTS dates_min_max AS
SELECT sector, ticker, min(TO_DATE(dates)) AS min_data, max(TO_DATE(dates)) AS max_data
FROM join_csv2
GROUP BY sector, ticker, YEAR(dates);

CREATE VIEW IF NOT EXISTS quotazione_inizio_anno2 AS
SELECT d.sector, YEAR(d.min_data) AS anno, SUM(j.close) AS min_close
FROM join_csv2 AS j, dates_min_max AS d
WHERE j.sector=d.sector AND j.dates=d.min_data AND d.ticker=j.ticker
GROUP BY d.sector, YEAR(d.min_data);

CREATE VIEW IF NOT EXISTS quotazione_fine_anno2 AS
SELECT d.sector, YEAR(d.max_data) AS anno, SUM(j.close) AS max_close
FROM join_csv2 AS j, dates_min_max AS d
WHERE j.sector=d.sector AND j.dates=d.max_data AND d.ticker=j.ticker
GROUP BY d.sector, YEAR(d.max_data);

CREATE TABLE IF NOT EXISTS variazione_settore2 AS
SELECT mi.sector, mi.anno, ROUND(AVG(((ma.max_close - mi.min_close) / mi.min_close) * 100), 2) AS differenza_percentuale
FROM quotazione_inizio_anno2 AS mi, quotazione_fine_anno2 AS ma
WHERE mi.sector=ma.sector AND mi.anno=ma.anno
GROUP BY mi.sector, mi.anno;

CREATE VIEW IF NOT EXISTS prezzo_chiusura AS
SELECT sector, dates, SUM(close) AS somma
FROM join_csv2
GROUP BY sector, dates;

CREATE VIEW IF NOT EXISTS quotazione_giornaliera_media AS
SELECT sector, YEAR(dates) AS anno, AVG(somma) AS media
FROM prezzo_chiusura
GROUP BY sector, YEAR(dates);

CREATE TABLE IF NOT EXISTS risultati2 AS
SELECT a.sector, a.anno, c.avg_volume, b.differenza_percentuale, a.media
FROM quotazione_giornaliera_media AS a, variazione_settore2 AS b, volume_medio_settore AS c
WHERE a.sector=b.sector AND b.sector=c.sector AND a.anno=b.anno AND c.anno=b.anno
ORDER BY sector, anno;
```

3.4)Spark

Sono specificati i seguenti RDD per l'esecuzione del job:
volume_medio_annuale, contiene per ciascun settore, per ogni anno dell'intervallo, il volume medio
somma_medie_close, contiene per ciascun settore, per ogni anno dell'intervallo, la quotazione giornaliera media
*data_close_minimo,*contiene per ciascun settore, per ogni anno dell'intervallo, la quotazione di close con data meno recente
data_close_massimo, contiene per ciascun settore, per ogni anno dell'intervallo, la quotazione di close con data più recente

Su questi due ultimi RDD viene effettuato il join per ottenere la *join_variazione_percentuale*, su cui sarà calcolata successivamente la *variazione percentuale* media per ogni anno dell'intervallo per ciascun settore, tramite formula presente nell'RDD *variazione_percentuale.*
Infine viene effettuato il join tra *somma_medie_close, volume medio_annuale e variazione percentuale* per ottenere l'output richiesto dal job , ordinato alfabeticamente per settore.
Segue lo pseduocodice dell'implementazione:

funzione massimo(x,y)=valuta il massimo tra due valori
funzione minimo(x,y)=valuta il minimo tra due valori
funzione parsing_linea= effettua il parsing delle righe del csv
funzione get_anno=estrae dal campo date l'anno del record

hsp = RDD che contiene il dataset historical_stock_prices.csv con i record relativi agli anni nell'intervallo 2008-2018

hs = RDD contenente il dataset historical_stocks.csv, filtrando i record in cui il campo sector assume valori nulli o 'N/A'

join_hsp_hs = join tra gli RDD hs e hsp
persist join_hsp_hs in memory (ed opzionalmente vengono memorizzati sul disco se RDD non è adeguato alla memoria)

```
volume_medio_annuale  = join_hsp_hs.map(linea -> ((sector, anno),Volume)
                                        .map(linea->((sector,anno),volume, 1)
                                        .reduceByKey(volume1+volume2, count volume +1)
                                        .map(linea->(sector(somma volume/count volume)

somma_medie_close = join_hsp_hs
                                         .map(linea: ((sector, date), close))
                                        .reduceByKey(close1+close2)
                                        .map(linea->: ((sector, anno), (somma_close, 1)))
                                        .reduceByKey(linea -> (somma_close1+somma_close2, count close +1))
                                       .map(linea -> (sector, (somma_close/count close))

Data_close_minimo = join_hsp_hs.map(linea -> ((ticker, sector, anno), (close, data)))
                                    .reduceByKey(minimo((data1,close1), (data2,close2)))
                                    .map(linea -> ((settore, anno), close_minimo))
                                    .reduceByKey(close_minimo1+close_minimo2)

Data_close_massimo = join_hsp_hs.map(linea -> ((ticker, sector, anno), (close, data)))
                                     .reduceByKey(massimo((data1,close1), (data2,close2)))
                                     .map(linea -> ((settore, anno), close_massimo))
                                     .reduceByKey(close_massimo1+close_massimo2)

join_variazione_percentuale = data_close_minimo.join(data_close_massimo )

variazione_percentuale = join_variazione_percentuale.map(linea -> (ticker, (massimo close-minimo close)/minimo close * 100 r
ound 2))

output = variazione_percentuale.join(somma_medie_close)
                                    .join(volume_medio_annuale)
                                    .sortBy(sector)
```

3.5)Risultati ottenuti

A causa della grandezza dell'output prodotto, sono riportati i record relativi ai primi 10 settori dell'anno 2018. L'output completo è presente nella cartella risultati_jobs.

settore	anno	Variazione %	Quotazione giornaliera	Volume medio annuale
BASIC INDUSTRIES	2018	-3.08	39532.54847019547	1428835.1726318246
CAPITAL GOODS	2018	-1.46	84692.99770890722	856807.472994857
CONSUMER DURABLES	2018	7.59	21840.274999514586	510230.143462197
CONSUMER NON-DURABLES	2018	7.47	35786.986083468284	1223517.1754731748
CONSUMER SERVICES	2018	-63.07	141030.16139430113	1169981.7828834706
ENERGY	2018	8.26	31663.262133251603	2295534.999301849
FINANCE	2018	4.14	138889.96390886858	702434.5736126429
HEALTH CARE	2018	15.12	95122.84587634986	901967.2047827973
MISCELLANEOUS	2018	10.4	34051.4778331569	1108339.1960214572
PUBLIC UTILITIES	2018	100.5	51917.52065874509	1242742.8820936237

4)Terzo Job

4.1)Specifica

Un job in grado di generare gruppi di aziende le cui azioni hanno avuto lo stesso trend in termini di variazione annuale nell'ultimo triennio disponibile, indicando le aziende e il trend comune (es. {Apple, Intel, Amazon}: 2016:-1%, 2017:+3%, 2018:+5%).

4.2)Map Reduce

Per realizzare questo job, a differenza dei precedenti, è stato necessario definire due mapper e due reducer.

In FirstMapper.py viene letto il file *historical_stocks.csv* sempre utilizzando la *Distributed Cache*, col fine di definire una struttura dati *TickerToCompanyNameMap* che assocerà a ciascun ticker il nome della compagnia corrispondente. In questa classe inoltre, vengono estrapolati da ciascun record i campi *ticker, close, date* per il nuovo intervallo temporale preso in esame, 2016-2018.

Per ogni record si verrà a formare una coppia <chiave,valore> in cui:

- la chiave risulterà essere composta da tre campi: *name,* la chiave primaria, *ticker* e *date*, le chiavi secondarie utilizzate per fare l'ordinamento, in questo modo i valori appartenenti allo stessa compagnia saranno ordinati in base al *ticker* e alla *date.*Tale ordinamento gerarchico consente di semplificare notevolmente la fase di reduce, in modo che si possa procedere direttamente al calcolo delle differenze percentuali.
- Il valore risulterà composto dal campo *close.*

Segue lo pseudocodice dell'implementazione:

```
class mapper:
   setup():
        read the historical_stocks.csv file from the Distributed Cache
        tickerToCompanyNameMap = map which associate the respective company name for each ticker in
historical_stocks.csv
    map(key, record):
         ticker, _, close, _, _, _, _, date = input
          year = getYear(date)
          if year in range from 2016 to 2018 and
          ticker has a corresponding sector in tickerToCompanyNameMap:
                  sector = ticker's corresponding sector
                   key = sector, ticker, date
                   value = close
                   print(key, value)
```

Viene create una classe *Reducer.py* che utilizza il metodo della *Distributed Cache* per leggere il file *historical_stocks.csv,* in questo modo sarà possibile definire la struttura dati *YearToCompanyTrend* che permette di calcolare per ogni nome di compagnia la variazione annuale per ogni anno del triennio 2016-2018. In particolare, per svolgere tale calcolo, verrà sfruttato l'ordinamento dei valori associati a ciascun nome di compagnia, in questo modo sarà possibile estrarre i prezzi di chiusura più recenti e quelli meno recenti relativi a ciascun ticker di ogni anno. La somma dei prezzi di chiusura meno recenti costituirà la quotazione iniziale, mentre la somma dei prezzi di chiusura più recenti costituirà la quotazione finale e a partire da questi ultimi due valori calcolati sarà possibile individuare la differenza percentuale.

Segue lo pseudocodice dell'implementazione:

```
class reducer:
     setup():
       read the historical_stocks.csv file from the Distributed Cache
       reduce(companyName, records):
        yearToStartingCloseValueMap = empty map
        yearToEndingCloseValueMap = empty map
        for record in records:
              day = get date field from the current record
              year = get year value from the day variable
              if the record corresponding ticker has changed from the previous record:
                     yearToEndingCloseValueMap[previous year] += close field taken from the previous record
                       yearToStartingCloseValueMap[year] += close field taken from the current record
         percentChange = empty map
         for each year from 2016 to 2018:
               difference = yearToEndingCloseValueMap[year] - yearToStartingC loseValueMap[year]
              percentChange[year] = difference/yearToStartingCloseValueMap[ year]
             sortedPercentChangeMap = sorted(percentChangeMap)
         for each year in sortedPercentChangeMap:
                print(percentChangeMap[2016], percentChangeMap[2017], percentC hangeMap[2018], companyName)
```

Viene definite una classe *SecondMapper.py* che estrae da ciascun record i campi prodotti dal precedente *Reducer.py* e produce in output una coppia <chiave,valore> dove:

- La chiave corrisponde alla tripla delle differenze percentuali relative al 2016, 2017 e al 2018;
- Il valore corrisponde al nome della compagnia.

In questo modo verranno successivamente aggregate nella classe *SecondReducer.py* tutte le compagnie che avranno lo stesso trend in termini di differenza percentuale durante il triennio preso in esame.

Segue lo pseudocodice dell'implementazione:

```
class mapper:
        map(key, record):
            percentChange2016, percentChange2017, percentChange2018, companyName = record
            key = percentChange2016, percentChange2017, percentChange2018
            value = companyName
            print(key, value)
```

Infine viene implementata la classe *SecondReducer.py,* con il compito di effettuare di scandire i record e stampare i gruppi di aziende che avranno la stessa tripla di valori di differenza percentuale.

Segue lo pseudocodice dell'implementazione:

```
class reducer:
    reduce(percentTriplet, records):
     companyList = list of company which have the same "trend" (same triplet for 2016, 2017, 2018)
    for each company in companyList:
            print(company, trend)
```

4.3)Hive

I dati dai csv vengono caricati nelle tabelle:

historical_stock_prices (ticker STRING, open float, close float, adj_close float, low float, high float, volume float, dates STRING)

historical_stock (ticker STRING, exchanges STRING, name STRING, sector STRING, industry STRING).

Inizialmente viene effettuato il join dei due csv nella tabella *join_csv3* per avere a disposizione i ticker nel nuovo intervallo di interesse che appartengono ad un settore, di conseguenza vengono scartati tutti quelli che non hanno un settore di appartenenza.

Come nel job precedente, si procede a calcolare la variazione annuale, con la differenza che in questo job si terrà conto anche del *name*. Si definiscono:

- una vista *name_dates_min_max* dove si calcolano la data meno recente e quella più recente per ogni ticker per ogni anno;
- una vista *quotazione_inizio_anno3* per il prezzo di chiusura associato alla data meno recente per ogni compagnia per ogni anno;
- una vista *quotazione_fine_anno3* per il prezzo di chiusura associato alla più recente per ogni coppia per ogni anno;
- la tabella *variazione_settore3* che contiene il calcolo della differenza percentuale per ciascuna compagnia per ogni anno a partire dalle viste precedenti.

Successivamente si definisce la tabella *groups*, dove saranno contenuti i gruppi di compagnie di differenti settori che hanno la stessa differenza percentuale per un certo anno.

Infine, nella tabella *risultati3*, si selezionano le aziende della tabella *groups* che hanno lo stesso trend per ogni anno del triennio preso in esame.

Segue lo pseudocodice dell'implementazione:

```
CREATE TABLE IF NOT EXISTS join_csv3 AS
SELECT hs.name, hs.sector, hsp.dates, hsp.close, hsp.ticker
FROM historical_stock AS hs JOIN historical_stock_prices AS hsp ON hsp.ticker=hs.ticker
WHERE YEAR(dates)>=2016 AND YEAR(dates)<=2018 AND hs.sector!='N/A';

CREATE VIEW IF NOT EXISTS name_dates_min_max AS
SELECT name, ticker, sector, YEAR(dates) AS anno, min(TO_DATE(dates)) AS min_data, max(TO_DATE(dates)) AS max_data
```

```
FROM join_csv3
GROUP BY name, ticker, sector, YEAR(dates);

CREATE VIEW IF NOT EXISTS quotazione_inizio_anno3 AS
SELECT b.name, a.ticker, YEAR(b.min_data) AS anno, SUM(a.close) AS min_close
FROM join_csv3 AS a, name_dates_min_max AS b
WHERE a.ticker=b.ticker AND a.dates=b.min_data
GROUP BY b.name,a.ticker, YEAR(b.min_data);

CREATE VIEW IF NOT EXISTS quotazione_fine_anno3 AS
SELECT b.name, a.ticker, YEAR(b.max_data) AS anno, SUM(a.close) AS max_close
FROM join_csv3 AS a, name_dates_min_max AS b
WHERE a.dates=b.max_data AND a.ticker=b.ticker
GROUP BY b.name, a.ticker, YEAR(b.max_data);

CREATE TABLE IF NOT EXISTS variazione_settore3 AS
SELECT qi.name, qi.ticker,qi.anno, ROUND((qf.max_close - qi.min_close) / qi.min_close *100, 0) AS differenza_percentuale
FROM quotazione_inizio_anno3 AS qi, quotazione_fine_anno3 AS qf
WHERE qi.name=qf.name AND qi.anno=qf.anno AND qi.ticker=qf.ticker
ORDER BY  ticker, anno;

CREATE TABLE IF NOT EXISTS groups AS
SELECT name, concat_ws(',', collect_list(cast (differenza_percentuale AS STRING))) AS quotazione, anno
FROM variazione_settore3
GROUP BY name, anno;

CREATE TABLE IF NOT EXISTS risultati3 AS
SELECT collect_set(g1.name), g1.quotazione AS variazione16, g2.quotazione AS variazione17, g3.quotazione AS variazione18
FROM groups g1, groups g2, groups g3
WHERE g1.name = g2.name AND g1.anno != g2.anno != g3.anno AND g1.name = g3.name AND g1.anno = '2016' AND g2.anno =
'2017' AND g3.anno = '2018'
GROUP BY g1.quotazione, g2.quotazione, g3.quotazione
HAVING count(*)>1;
```

4.4)Spark

Sono specificati i seguenti RDD per l'esecuzione del job:
data_close_minimo, contiene per ciascuna compagnia, per ogni anno dell'intervallo, la quota relativa alla data meno recente
data_close_massimo, contiene per ciascuna compagnia , per ogni anno dell'intervallo, la quota relativa alla data più recente
Su questi RDD viene effettuato il join per ottenere la *join_variazione_percentuale*, su cui sarà calcolata successivamente la *variazione percentuale* o trend , per ogni anno dell'intervallo, per ogni azienda.
variazione percentuale avrà come chiave il nome e come valori l'anno e la relativa variazione percentuale.
In seguito sarà definito un RDD detto *tripletta_trend* che andrà a scorrere *variazione_percentuale* estraendo e raggruppando per ogni azienda i trend calcolati. Successivamente, in *tripletta_trend_pulito*, ogni record presente nel RDD precedentemente definito, sarà ripulito da tutte le aziende che non hanno valori di trend anche solo per uno dei 3 anni dell'intervallo.
Nella fase successiva viene applicata una funzione chiamata *ordinamento_tripletta* che si occupa di ordinare in modo crescente , gli anni presenti *tripletta_trend_pulito* per ottenere un risultato uniforme.
Infine in *output* verranno prese ed aggregate tutte le aziende che avranno lo stesso trend per il triennio, da RDD *tripletta_trend_pulito_ordinato,*restituendo il risultato desiderato nella forma (nomi aziende, trend uguale triennio).

Segue lo pseudocodice dell'implementazione:
funzione massimo(x,y)=valuta il massimo tra due valori
funzione minimo(x,y)=valuta il minimo tra due valori
funzione parsing_linea= effettua il parsing delle righe del csv
funzione get_anno=estrae dal campo date l'anno del record
funzione ordinamento_tripletta= ordina i trend per anno nella sequenza 2016,2017, 2018 tramite un dizionario.

hsp = RDD contenente il dataset historical_stock_prices.csv, con i record riguardanti i soli anni nell'intervallo 2016-2018
hs = RDD contenente il dataset historical_stocks.csv, filtrando i record in cui il campo sector assume valore "N/A"

```
join_hsp_hs = join tra gli RDD hs e hsp
persist join_hsp_hs in memory (ed opzionalmente vengono memorizzati sul disco se RDD non è adeguato alla memoria)

data_close_minimo = join_hsp_hs.map(linea -> ((ticker, anno, close), (ticker, nome )))
                                    .reduceByKey(minimo((data1,close1), (data2,close2)))
                                   .map(linea -> ((name, anno), close_minimo))
                                   .reduceByKey(close_minimo1+close_minimo2)

Data_close_massimo = join_hsp_hs.map(linea -> ((ticker, anno, close), (ticker,nome)))
                                  .reduceByKey(massimo((data1,close1), (data2,close2)))
                                  .map(linea -> ((name, anno), close_massimo))
                                  .reduceByKey(close_massimo1+close_massimo2)


join_variazione_percentuale = data_close_minimo.join(data_close_massimo)
variazione_percentuale = join_variazione_percentuale.map(linea -> (nome, (data_close_minimo –
data_close_massimo)/data_close_minimo * 100 ,round 0))

tripletta_trend = variazione_percentuale.map(linea -> ((name), ([(anno,variazione percentuale)])))
                                          .reduceByKey(([(anno1,variazione percentuale1) concat (anno2,variazione percentuale2)]))

Tripletta_trend_pulito = tripletta_trend.filter(linea -> filtro le righe che contengono valori non nulli di incremento percentuali in
almeno uno degli anni 2016 2017 e 2018)

Tripletta_trend_pulito_ordinato = tripletta_trend_pulito.map(linea ->ordinamento_tripletta(linea)])

output = Tripletta_trend_pulito_ordinato.
                                  .map(linea -> ([anno1,trend1,anno2,trend2,anno3,trend3]),((name)))
                                  .reduceByKey((name1)+(name2)).filter(len([name])>=2
                                  .map(linea->(name, anno1:trend1,anno2:trend2,anno3:trend3)
                                  .sortBy(anno1, trend crescente)
```

4.5)Risultati ottenuti

Nomi Compagnie	2016	2017	2018
STURM, RUGER & COMPANY, INC., LINCOLN EDUCATIONAL SERVICES CORPORATION	-14.0	4.0	8.0
COMPASS MINERALS INTERNATIONAL, INC, SIGNATURE BANK	3.0	-8.0	-15.0
LEAR CORPORATION, PENNYMAC FINANCIAL SERVICES, INC.	10.0	33.0	-8.0
SPECTRUM BRANDS HOLDINGS, INC., HERSHEY COMPANY (THE)	18.0	9.0	-11.0
CALERES, INC., GENTEX CORPORATION	25.0	3.0	11.0
DUNKIN' BRANDS GROUP, INC., TERRENO REALTY CORPORATION	28.0	23.0	9.0
DENNY'S CORPORATION, WINMARK CORPORATION	35.0	4.0	14.0
MIDDLESEX WATER COMPANY, INTERNATIONAL BANCSHARES CORPORATION	64.0	-3.0	18.0

5)Specifiche hardware e software

5.1)Macchina locale

5.1.1)Hardware

- Intel i7 9700k 3.6 GHz, 8 core, 12Mb Cache L3
- 32 Gb di RAM , DDR4 3000 Mhz
- 500Gb hard disk SATA, 7200 RPM

5.1.2)Software

- Ubuntu 20.04 LTS
- Java 1.8.0_252
- Python 3.8.2
- Hadoop 3.2.1
- Apache Hive 3.1.2
- Spark 3.0.0 con Hadoop 3.2.X

5.2)Cluster

Il cluster è stato realizzato utilizzando il servizio AWS educate gentilmente offerto dalla classroom BigData. E' stato configurato con le specifiche Hardware presenti nel sottoparagrafo 5.2.1.Per motivi di spazio sul Cluster e di tempistiche e dato il divario a livello di prestazioni con la macchina locale utilizzata, sul cluster è stato eseguito ogni job soltanto sul dataset originale e non sulle versioni Increased del dataset.

5.2.1)Hardware

- 4 vCore, emr 6.0.0 x5-large
- 1 Nodo master, 3 Nodi slave
- 16 RAM
- Storage EBS 64 Gb

5.2.2)Software

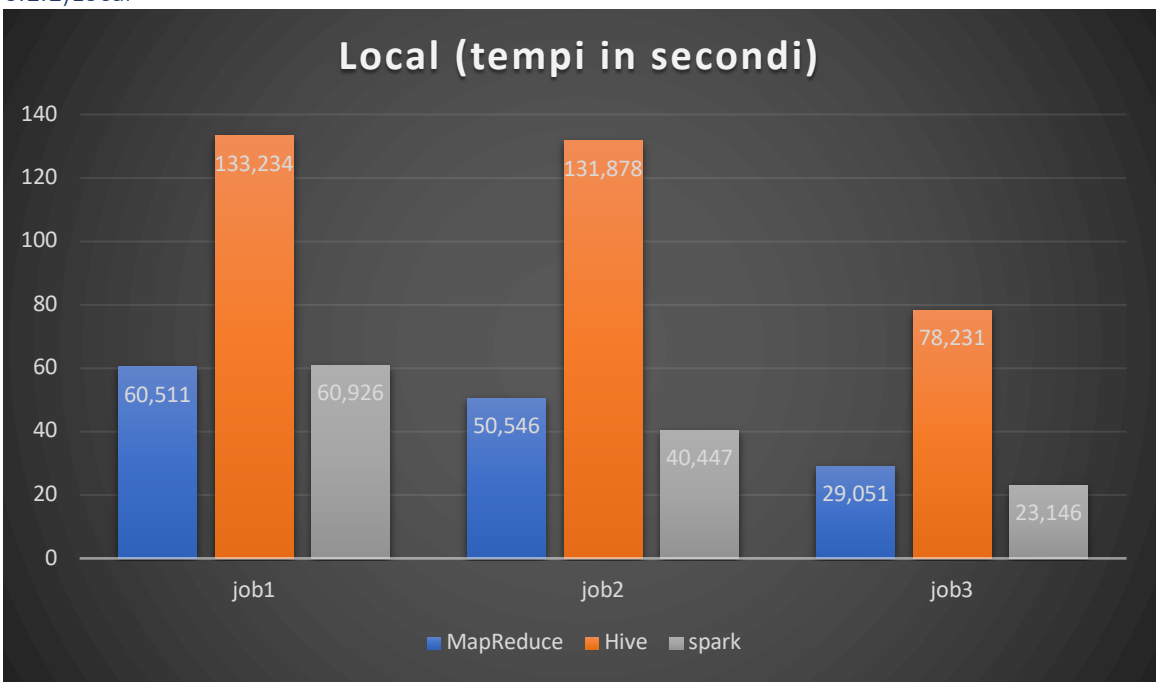
- Java 1.8.0_242
- Hadoop 3.2.1
- Python 3.7.4
- Hive 3.1.2
- Spark 2.4.4

6)Grafici e tempi

In questo paragrafo sono riportati i grafici relativi ai tempi d'esecuzione per ognuno dei 3 job di MapReduce, Hive e Spark.
Tutti i tempi riportati nei grafici sono stati prodotti effettuando la media dei tempi ottenuti, per ognuna delle tecnologie, per 3 volte.Tutti i tempi sono stati campionati tramite il comando di bash 'time'.Il confronto tra locale e cluster, come già accennato nel paragrafo 5.2, viene effettuato soltanto sul dataset originale di dimensione di 2.0 Gb. Nel paragrafo 6.1.2 viene spiegato prima di mostrare i grafici come sono stati effettuati, per ognuno delle tecnologie , il caricamento dei dati e degli script e il relativo avvio del job. Nel paragrafo 6.3 invece viene mostrato per singolo job , l'andamento dei tempi in locale sul dataset originale e su quelli aumentati.

6.1)Original dataset

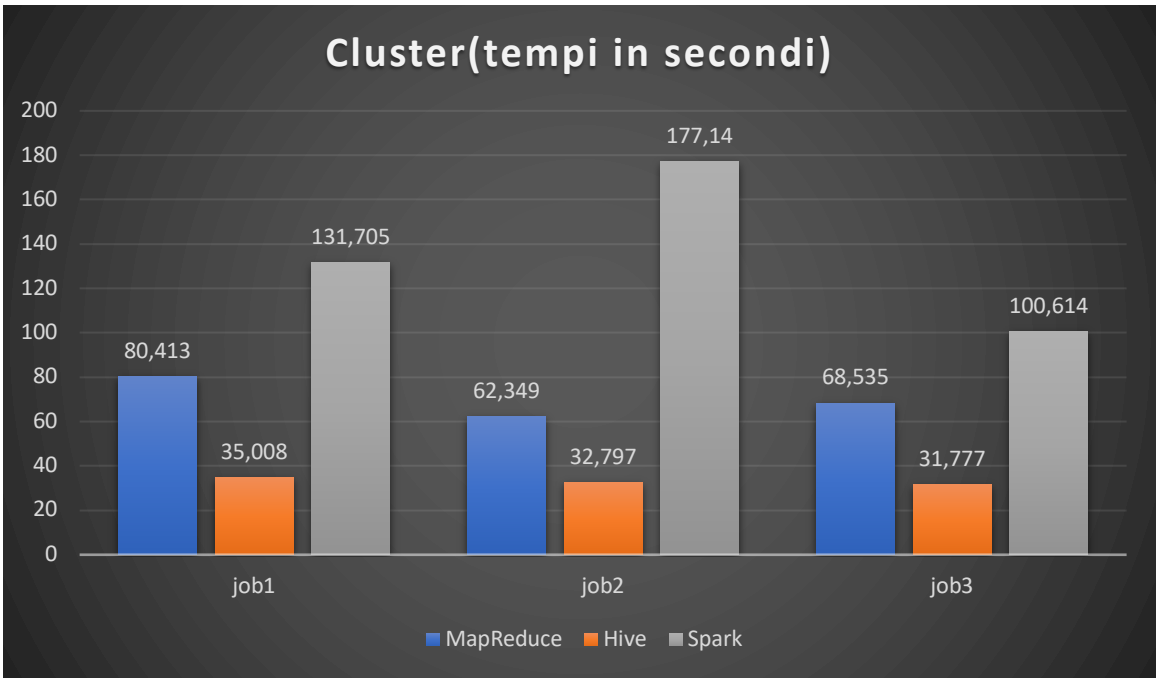
6.1.1)Local



N.d.A.:Su Hive bisogna tener conto dei tempi per la creazione delle tabelle utili all'esecuzione del job, che vengono create all'inizio tramite lo script presente nella cartella Hive_script.Il tempo per la creazione delle tabelle sul dataset originale è di circa 31,716 secondi

6.1.2)Cluster

Prima di mostrare il grafico relativo all'esecuzione dei job su cluster analizzeremo come sono stati eseguiti gli script e come è stato fatto il caricamento del file.
Ogni file è stato caricato manualmente sul cluster tramite il comando 'scp' ed i job sono stati stati avviati singolarmente. L'accesso al cluster è effettuato tramite ssh.La procedura per ognuna delle tecnologie è stata la seguente:
MAP REDUCE: vengono caricati sul cluster il file .py relativo al mapper e reducer, lo script .sh è configurato dopo che sono state create manualmente le cartelle sull'HDFS /input/ , /output/ e le relative sottocartelle specificate nei file .sh. Inoltre viene fatto il put sull'HDFS del dataset historical_stock_prices.csv per ognuno dei tre job, mentre historical_stocks.csv è lasciato nella cartella /home/hadoop del cluster insieme al file .sh, mapper e reducer.Viene avviato lo script .sh tramite il comando "bash XXXX.sh" che si occuperà dell'esecuzione del job su hadoop.
HIVE: I file sono caricati manualmente nel path /home/hadoop del cluster di aws. Successivamente viene creata una cartella /dataset dove saranno collocati HSP e HS. Viene avviato lo script .hql che crea le table sul dataset e successivamente sono avviati gli script relativi ai singoli job in maniera sequenziale tramite "hive -f /path/to/file.hql", così da ottenere tutti e tre i risultati in una sola passata senza dover eliminare nessuna tabella o vista. Infine , nella cartella Hive_script è stato preparato uno script che svuota le tabelle di hive.
SPARK: Dopo aver caricato il dataset sul cluster ed averli spostati nella cartella /dataset creata precedentemente , viene caricato lo script relativo al job e creata la cartella /output dove saranno collocati i risultati dei vari job. Successivamente questi, saranno avviati tramite il comando spark-submit.
Di seguito sono riportati i tempi di esecuzione su cluster dei job:



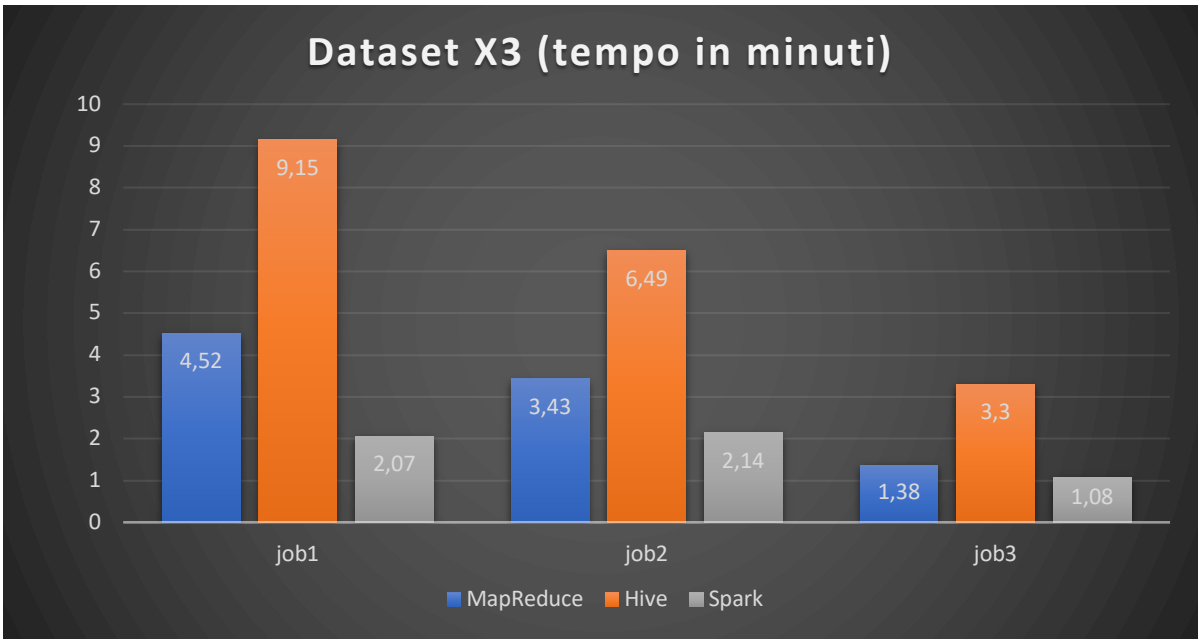
N.d.A: Come nel caso precedente u Hive bisogna tener conto dei tempi per la creazione delle tabelle utili all'esecuzione del job, che vengono create all'inizio tramite lo script presente nella cartella Hive_script.Il tempo per la creazione delle tabelle sul dataset originale è di circa 10,705 secondi.

Possiamo notare che rispetto all'esecuzione locale la situazione tra Hive e Spark è praticamente invertita. Infatti questa volta Hive ottiene dei tempi migliori nell'esecuzione dei job mentre Spark si è mostrato lento e macchinoso. Ciò è dovuto al fatto che la versione presente sul cluster emr di Spark è una versione non aggiornata, rispetto a quella utilizzata in locale. Inoltre la gestione dei metadati di Hive su cluster hadoop nativo è risultata decisamente migliore rispetto a quella in locale rispetto alle tecnologie utilizzate. Hadoop Map reduce si è dimostrato molto più efficiente in locale, ma questo è sicuramente dovuto alla differenza di prestazioni tra le due macchine.

6.2)Increased dataset

Successivamente al confronto cluster- locale, ci siamo posti l'obiettivo di analizzare la crescita dei tempi all'aumentare della mole di dati analizzati sul singolo job per valutare l'efficienza di quanto realizzato. Il dataset è stato aumentato, come già spiegato nel paragrafo 1.1.2 generando 3 copie aggiuntive di dimensioni via via sempre maggiori.Questa volta, considerando i tempi ottenuti i dati saranno espressi in minuti, per ottenere una lettura più facile dei grafici.

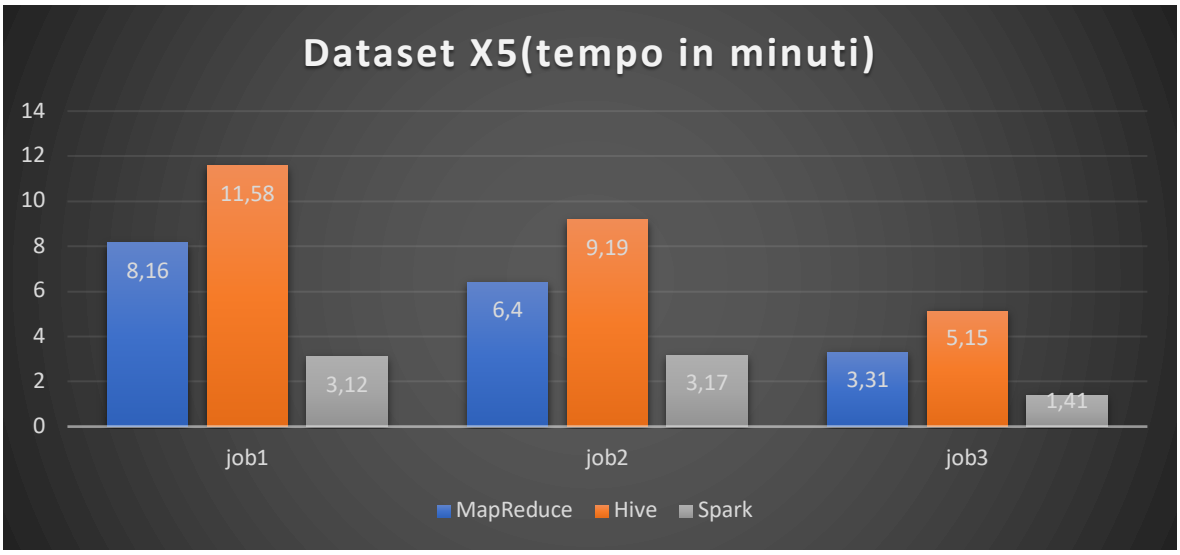
Dataset x3:



NdA: Tempo per creare le tabelle in Hive 4 minuti e 39 secondi

Da questa prima analisi si evince come l'aumento del dataset influisca poco sulle prestazioni di Spark, i cui tempi sono quasi raddoppiati nonostante una quadruplicazione(come spiegato nel paragrafo 1.1.2) del dataset. Mentre Map Reduce rispecchia a pieno il lavoro di append effettuato sul dataset, dato che i tempi sono 4 volte quelli ottenuti sul dataset originale.Invece Hive , all'aumentare del dataset peggiora notevolmente le prestazioni.

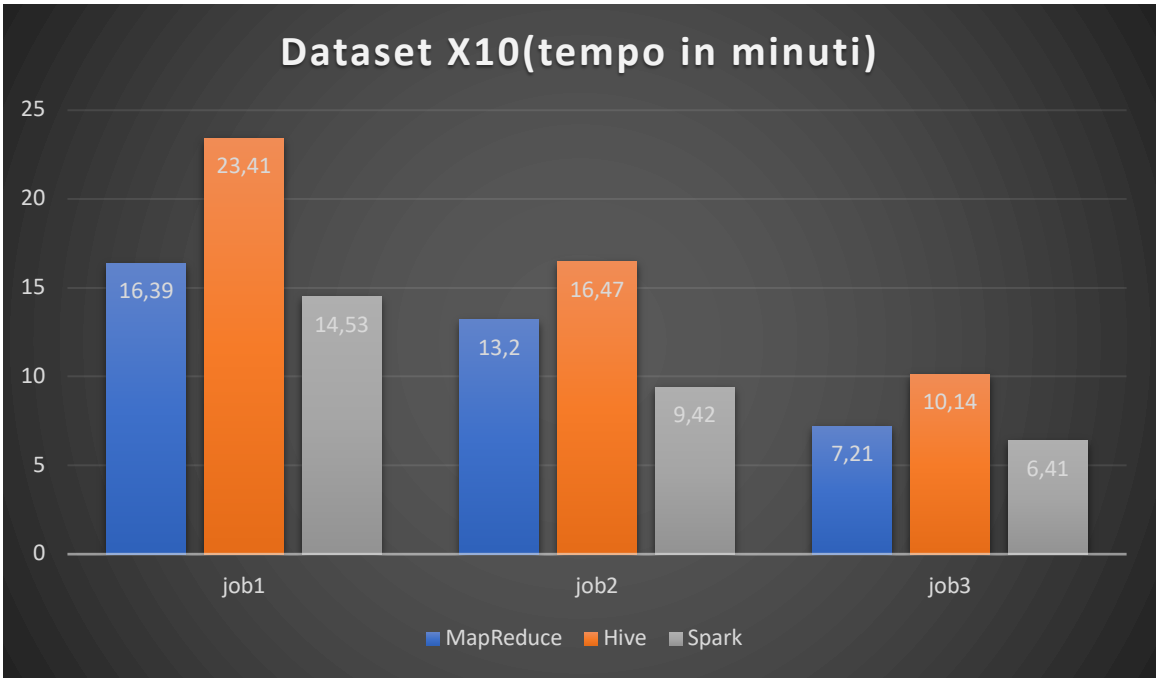
Dataset x5:



NdA: Tempo per creare le tabelle in Hive 9minuti e 54 secondi

Mentre Spark mantiene ancora bassi le tempistiche con il dataset x5 (contenente 6 volte il numero di righe rispetto al dataset originale) , MapReduce accusa un calo notevole delle prestazioni, venendo raddoppiate nel job 1 e 2 rispetto alla variante x3 e quasi triplicate sul job 3. Hive invece continua a dimostrare quanto sia su moli di dati sempre maggiori sia tempisticamente il meno efficiente dei tre, almeno in locale.

Dataset x10:



NdA: Tempo per creare le tabelle in Hive 19 minuti e 53 secondi

Con un dataset delle dimensioni di circa 23Gb(circa 11 volte il numero delle righe del dataset originale) si ha un sostanziale aumento delle tempistiche su tutte e tre le tecnologie. In paricolare, Spark che si era dimostrato molto efficiente nei casi x3 e x5 , subisce una quintuplicazione delle tempistiche nel primo job ed una triplicazione nei job 2 e 3, diminuendo il gap con Map reduce evidenziato nei casi precedenti. Map reduce infatti, ha un aumento costante alla crescita della mole dei dati da analizzare e si potrebbe ipotizzare che con un ulteriore raddoppiamento del dataset potrebbe essere la tecnologia su cui puntare con l'hardware utilizzato in questi test. Hive presenta una situazione di duplicazione delle tempistiche sui job 1 e 3 mentre sul job 2 l'aumento è inferiore a due.

7) Conclusioni

Analizzando le tempistiche prodotte nella comparazione in locale, per quanto riguarda le tecnologie utilizzate,sia sul dataset aumentato che su quello originale, Spark è risultato molto più efficiente rispetto ad Hive e Map Reduce. Quest'ultimo ha generato tempistiche paragonabili a quelle di Spark solo nel caso del dataset da 2 GB e in quello da 23GB, mentre nei casi da 7GB e 12 GB i tempi prodotti sono sempre stati almeno il doppio. Hive invece ha mostrato evidenti limiti e la causa potrebbe ricondursi a come il job sia stato pensato e programmato. Si potrebbe operare un partizionamento basato sull'anno della tabella del file HSP, che potrebbe portare ad una soluzione più efficiente ed a tempistiche più basse. Purtroppo per problemi di tempo questa soluzione non è stata implementata.

Nel caso del cluster invece Hive è stato il migliore rispetto alle prestazioni ottenute in locale. Avremmo voluto effettuare i test riguardanti i dataset aumentati anche sul cluster, ma per problemi di tempo e spazio disponibile sul cluster messo a disposizione, abbiamo dovuto rinunciare a questa opzione.

Dai nostri esperimenti abbiamo potuto però constatare che sul cluster la soluzione migliore sia Hive, mentre con la forte potenza di calcolo presente in locale ci siamo potuti accertare che Spark sia la soluzione più efficiente, offrendo tempistiche eccezionali per molti dei casi analizzati. MapReduce invece ha prodotto dei risultati accettabili in tutti i casi , mostrando una crescita lineare con l'aumento della mole dei dati.