Final-Report

May 22, 2021

1 Relazione Finale

Gruppo - Dig Data

Componenti Gruppo - Alexandru Pavel, Simone Garzarella

2 Indice

3 Introduzione

Il dataset “Daily Historical Stock Prices” contiene l’andamento delle azioni sulla borsa di New York (NYSE e NASDAQ) dal 1970 al 2018.

Due file CSV compongono il dataset: - historical_stock_prices.csv - historical_stocks.csv

Il primo contiene i valori dei prezzi e volumi che variano nel tempo per ogni ticker. Il secondo i dati relativi ad ogni ticker, come il settore e l'exchange in cui è quotato.

3.1 Descrizione Problema

Dopo una fase iniziale di analisi e processamento di dati si vogliono eseguire 3 job (descritti nel dettaglio più avanti) con le diverse tecnologie affrontate nel corso (Hadoop, Hive e Apache Spark).

3.2 Specifiche Hardware

I test sono stati eseguiti in locale e su cluster con macchine con queste caratteristiche: - **Locale:** Ubuntu 20.04, CPU i5 2.5GHZ, 8GB Ram e 256GB SSD - **Cluster:** AWS EMR con 1 Master Node e 5 DataNode. Istanze m5.xlarge con 16GB RAM, 4 vCPU e 64GB di spazio.

4 Analisi Dataset

Di seguito vengono analizzati i due file del dataset per individuare eventuali preprocessamenti da effettuare. Inoltre viene anche descritto il processo per creare dataset più piccoli o grandi (con sampling) per effettuare i successivi test.

[1]:
```
import pandas as pd
```

4.1 Historical Stock Prices

I campi di questo dataset sono:

- `ticker`: simbolo univoco dell'azione (https://en.wikipedia.org/wiki/Ticker_symbol)
- `open`: prezzo di apertura
- `close`: prezzo di chiusura
- `adj_close`: prezzo di chiusura "modificato"
- `lowThe`: prezzo minimo
- `highThe`: prezzo massimo
- `volume`: numero di transazioni
- `date`: data nel formato aaaa-mm-gg

[2]:
```
stock_prices = pd.read_csv('dataset/historical_stock_prices.csv')
```

[3]:
```
stock_prices
```

[3]:

	ticker	open	close	adj_close	low	high	volume	date
0	AHH	11.50	11.58	8.493155	11.25	11.68	4633900	2013-05-08
1	AHH	11.66	11.55	8.471151	11.50	11.66	275800	2013-05-09
2	AHH	11.55	11.60	8.507822	11.50	11.60	277100	2013-05-10
3	AHH	11.63	11.65	8.544494	11.55	11.65	147400	2013-05-13
4	AHH	11.60	11.53	8.456484	11.50	11.60	184100	2013-05-14
…	…	…	…	…	…	…	…	…
20973884	NZF	14.60	14.59	14.590000	14.58	14.62	137500	2018-08-20
20973885	NZF	14.60	14.58	14.580000	14.57	14.61	151200	2018-08-21
20973886	NZF	14.58	14.59	14.590000	14.57	14.63	185400	2018-08-22

```
20973887    NZF  14.60  14.57  14.570000  14.57  14.64   135600  2018-08-23
20973888    NZF  14.60  14.69  14.690000  14.59  14.69   180900  2018-08-24

[20973889 rows x 8 columns]
```

Ci sono ~21milioni di record per questo file

[4]:
```
stock_prices.isna().sum()
```

[4]:
```
ticker       0
open         0
close        0
adj_close    0
low          0
high         0
volume       0
date         0
dtype: int64
```

Non sono presenti valori nulli per nessuna delle colonne

[5]:
```
stock_prices.nunique()
```

[5]:
```
ticker          5685
open          807688
close         835181
adj_close    9235753
low           815237
high          821044
volume        385849
date           12274
dtype: int64
```

In totale ci sono 5685 `ticker` univoci nel dataset

[6]:
```
stock_prices[stock_prices.duplicated(subset=['ticker','date'])].shape
```

[6]:
```
(0, 8)
```

Non ci sono record distinti con valori duplicati di (ticker, data)

4.1.1 Creazione di dataset di varie dimensioni

Sono stati generati dataset di dimensioni (approssimativamente) di 256/512/1024MB e ~4GB, oltre al dataset originale che ha dimensioni ~2GB.

I file generati (con relativa dimensione precisa) hanno nome historical_stock_prices[size].csv

- historical_stock_prices256.csv (239.75MB)
- historical_stock_prices512.csv (479.51MB)

- historical_stock_prices1024.csv (959.03MB)
- historical_stock_prices.csv (1909.97MB)
- historical_stock_prices4096.csv (3835.92MB)

La scelta dei record da includere è effettuata con un sampling randomico (con un seed preimpostato, per la ripetibiltà)

```
def sample_all_sizes(historical_stock_prices_df):
    for size in [0.125, 0.25, 0.5, 2]:
        sample_n_rows = dataset_row_count * size
        sampled_df = dataset.sample(sample_n_rows)
        filename = 'dataset/historical_stock_prices[SIZE].csv'
        sampled_df.to_csv(filename)
```

4.2 Historical Stocks

Il dataset con le informazioni sui ticker è così strutturato:

- `ticker`: simbolo dell'azione
- `exchange`: NYSE o NASDAQ
- `name`: nome dell'azienda
- `sector`: settore dell'azienda
- `industry`: industria di riferimento per l'azienda

[7]:
```
stocks = pd.read_csv("dataset/historical_stocks.csv")
```

[8]:
```
stocks
```

[8]:

	ticker	exchange	name	\
0	PIH	NASDAQ	1347 PROPERTY INSURANCE HOLDINGS, INC.	
1	PIHPP	NASDAQ	1347 PROPERTY INSURANCE HOLDINGS, INC.	
2	TURN	NASDAQ	180 DEGREE CAPITAL CORP.	
3	FLWS	NASDAQ	1-800 FLOWERS.COM, INC.	
4	FCCY	NASDAQ	1ST CONSTITUTION BANCORP (NJ)	
…	…	…	…	
6455	ZOES	NYSE	ZOE'S KITCHEN, INC.	
6456	ZTS	NYSE	ZOETIS INC.	
6457	ZTO	NYSE	ZTO EXPRESS (CAYMAN) INC.	
6458	ZUO	NYSE	ZUORA, INC.	
6459	ZYME	NYSE	ZYMEWORKS INC.	

	sector	industry
0	FINANCE	PROPERTY-CASUALTY INSURERS
1	FINANCE	PROPERTY-CASUALTY INSURERS
2	FINANCE	FINANCE/INVESTORS SERVICES
3	CONSUMER SERVICES	OTHER SPECIALTY STORES
4	FINANCE	SAVINGS INSTITUTIONS
…	…	…
6455	CONSUMER SERVICES	RESTAURANTS

```
6456        HEALTH CARE                    MAJOR PHARMACEUTICALS
6457     TRANSPORTATION        TRUCKING FREIGHT/COURIER SERVICES
6458         TECHNOLOGY  COMPUTER SOFTWARE: PREPACKAGED SOFTWARE
6459        HEALTH CARE                    MAJOR PHARMACEUTICALS

[6460 rows x 5 columns]
```

[9]:
```
stocks.nunique()
```

[9]:
```
ticker      6460
exchange       2
name        5462
sector        13
industry     136
dtype: int64
```

Sono presenti 6460 `ticker` univoci, come il numero di righe del dataset. Il ticker può essere considerato una chiave di questo dataset, il nome dell'azienda `name` invece no, ha delle ripetizioni.

[10]:
```
stocks[stocks.duplicated(subset=['name'])].shape
```

[10]:
```
(998, 5)
```

In particolare sono presenti 998 nomi di azienda duplicati. Nel resto del progetto non si considererà questo campo per identificare record (in particolare per il job3).

[11]:
```
stocks['sector'].unique()
```

[11]:
```
array(['FINANCE', 'CONSUMER SERVICES', 'TECHNOLOGY', 'PUBLIC UTILITIES',
       'CAPITAL GOODS', 'BASIC INDUSTRIES', 'HEALTH CARE',
       'CONSUMER DURABLES', nan, 'ENERGY', 'MISCELLANEOUS', 'SECTOR',
       'TRANSPORTATION', 'CONSUMER NON-DURABLES'], dtype=object)
```

Visualizzando i possibili valori di `sector` si può notare la presenza di un valore nullo.

[12]:
```
stocks.isna().sum()
```

[12]:
```
ticker         0
exchange       0
name           0
sector      1440
industry    1440
dtype: int64
```

Il campo `sector` presenta 1440 valori nulli, che vengono eliminati durante il preprocessing di questo dataset.

[13]:
```
stocks_clean = stocks.loc[(stocks['sector'].notna())]
```

[14]:
```
stocks_clean.shape
```

[14]:
```
(5020, 5)
```

Il dataset pulito dai valori nulli del campo `sector` ha 5020 record. Verrà salvato come `historical_stocks_clean.csv`

```
stocks_clean.to_csv('dataset/historical_stocks_clean.csv')
```

5 Job 1

Deve generare un report contenente, per ciascuna azione:

- data prima quotazione (a)
- data ultima quotazione (b)
- variazione percentuale della quotazione (tra primo e ultimo prezzo di chiusura nel dataset) (c)
- prezzo massimo (d)
- prezzo minimo (e)

Il report deve essere ordinato per valori decrescenti del secondo punto (dalla data di quotazione più recente alla più vecchia).

5.1 MapReduce

Durante la fase di Map dapprima si estraggono i campi (mediante split e parsing) di ticker, closePrice, minPrice, maxPrice e date. Queste righe filtrate vengono mandate al Reducer che farà altre operazioni.

```
class mapper:

    for row in INPUT:
        # split the current row into fields (ignoring not needed ones)
        ticker, closePrice, minPrice, maxPrice, date = row

        # write the separated fields to standard output
        print(ticker, closePrice, minPrice, maxPrice, date)
```

Nel Reducer dapprima si definisce una struttura dati dizionario (results) che conterrà, per ogni ticker, un dizionario con i valori richiesti dal job.

`results` ha il seguente formato per le sue entry: (ticker): (first_quot_date, last_quot_date, perc_var, min_price, max_price)

```
class reducer:

    # maps each ticker to the required values to calculate
    # for example: {'AAPL': {'min': 1, 'max': 2, ..},
    #               'AMZN': {'min': 0.5, 'max': 5, ..}}
    results = {}
```

Vengono parsati i valori provenienti dal Mapper, e per ogni ticker se esso non è già presente nel dizionario si inizializzano i suoi valori.

```
    for row in INPUT:
        # split the current row into fields
        ticker, closePrice, minPrice, maxPrice, date = row

        # if the ticker hasn't been seen before, initialize its values in the dictionary
        if ticker not in results:
            results[ticker] = {
                'first_quot_date': date,
                'last_quot_date': date,
                'first_quot_price': closePrice,
                'last_quot_price': closePrice,
                'perc_var': 0,
                'min_price': minPrice,
                'max_price': maxPrice
            }
            continue
```

I ticker già contenuti in results verranno aggiornati solo se necessario (ad esempio se si trova un ticker con una data di quotazione antecedente a quella salvata).

```
# gets the input ticker's current saved values from the dictionary
        currTicker = results[ticker]

        # update the saved ticker values with the ones from the input data
        if date < currTicker['first_quot_date']:
            currTicker['first_quot_date'] = date
            currTicker['first_quot_price'] = closePrice

        if date > currTicker['last_quot_date']:
            currTicker['last_quot_date'] = date
            currTicker['last_quot_price'] = closePrice

        if minPrice < currTicker['min_price']:
            currTicker['min_price'] = minPrice

        if maxPrice > currTicker['max_price']:
            currTicker['max_price'] = maxPrice
```

Infine i risultati ottenuti vengono ordinati in senso decrescente sul campo della data dell'ultima quotazione.

Vengono poi mandati in output calcolando anche la variazione percentuale del prezzo dell'azione tra la prima e l'ultima data di quotazione.

```
    # sort the results from the most to the least recent quotation dates
    sortedResults = sort(results.items(), key='last_quot_date', reverse=True)

    # result is in the format ('TickerName', {'min': 1, 'max': 2}), a tuple
```

```
    for result in sortedResults:
        perc_var = calculate_percent_variation(first_quot_price, last_quot_price)

        print(ticker, first_quot_date, last_quot_date, perc_var, min_price, max_price)
```

5.2 Hive

Per eseguire questo job sono state create prima due tabelle esterne:

- Una che, per ogni ticker, etrae il prezzo di chiusura alla prima data disponibile per quel ticker nel database:

```
create table ticker_to_minDate as
select d.ticker as min_ticker, d.close_price as min_close_price
from historical_stock_prices(size) d
join (select ticker as min_ticker, min(price_date) as min_price_date
from historical_stock_prices(size) group by ticker) min_table
on (d.ticker = min_table.min_ticker and d.price_date <= min_table.min_price_date);
```

- L'altra che, per ogni ticker, etrae il prezzo di chiusura allultima data disponibile per quel ticker nel database:

```
create table ticker_to_maxDate as
select d.ticker as max_ticker, d.close_price as max_close_price
from historical_stock_prices(size) d
join (select ticker as max_ticker, max(price_date) as max_price_date
from historical_stock_prices(size) group by ticker) as max_table
on (d.ticker = max_table.max_ticker and d.price_date >= max_table.max_price_date);
```

- Successivamente esse sono state utilizzate per la query finale, in cui si estrae, per ogni ticker, la data della prima quotazione, la data dell'ultima quotazione, la variazione percentuale della quotazione, il prezzo massimo e quello minimo.

```
CREATE TABLE job1_hive ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n' as
select ticker,
    min(price_date) as first_price_date,
    max(price_date) as last_price_date,
    max((
        (max_table.max_close_price - min_table.min_close_price)
        / min_table.min_close_price) * 100) as variation,
    max(high) as max_price, min(low) as min_price
from historical_stock_prices(size) d
join ticker_to_maxDate max_table on d.ticker = max_table.max_ticker
join ticker_to_minDate min_table on d.ticker = min_table.min_ticker
group by ticker
order by ticker, last_price_date desc;
```

5.3 Spark

Spark caricherà i dati del dataset `historical_stock_prices` solo nel momento in cui deve effettuare una azione

```
historical_stock_prices = loadDataFromCsv()
```

Viene creato un nuovo RDD con chiave = ticker e valore = (prezzo di chiusura, data)

```
ticker_date = historical_stock_prices
 .map(x -> (ticker,(close_price, price_date)))
```

Vengono poi creati due RDD per calcolare la data minima e massima della quotazione

```
first_quot_date = ticker_date.reduceByKey(min_date(a, b))
last_quot_date = ticker_date.reduceByKey(max_date(a, b))
```

Viene fatto il join tra i due precedenti RDD e poi si mappa per avere (ticker) -> (primo prezzo di chiusura, prima data, ultimo prezzo di chiusura, ultima data). Infine si mappa, calcolando la variazione percentuale, per ottenere (ticker) -> (prima data, ultima data, variazione percentuale)

```
percent_variation = first_quot_date
    .join(last_quot_date)
    .map((ticker, ((first_close, first_date), (last_close, last_date)))
         -> (ticker, (first_close, first_date, last_close, last_date))
    .map((ticker, (first_close, first_date, last_close, last_date)
         -> (ticker, (first_date, last_date, percent_variation(first_close, last_close)))))
```

Viene mappato l'RDD iniziale per ottenere, per ogni ticker, il prezzo minimo ed il prezzo massimo

```
min_price = historical_stock_prices
    .map(x -> (ticker, min_price))
    .reduceByKey(min(a,b))

max_price = historical_stock_prices
    .map(x -> (ticker, max_price))
    .reduceByKey(min(a,b))
```

Vengono joinati insieme i precedenti risultati per ottenere l'RDD finale ordinato per data, nella forma (ticker) -> data prima quotazione, data ultima quotazione, variazione percentuale, valore minimo e valore massimo

```
results = percent_variation
    .join(min_price)
    .join(max_price)
    .map((ticker, ((first_date, last_date, perc_var), min_price, max_price))
         -> (ticker, (first_date, last_date, percent_var, min_price, max_price))
    .sortByDate()
```

6 Job 2

Generare un report contenente, per ciascun settore e per ciascun anno del periodo 2009-2018:

- variazione percentuale della quotazione del settore nell'anno (somma prezzi di chiusura di tutte le azioni del settore, considerando la prima e l'ultima data di ogni azione aggregate) (a)
- azione del settore con incremento percentuale maggiore nell'anno (col valore) (b)
- azione del settore con maggior volume di transazioni nell'anno (con valore) (c)

Il report deve essere ordinato per nome del settore.

6.1 MapReduce

Nel Mapper si definisce una struttura `ticker_to_sector` (dizionario) che conterrà, per ogni ticker, il valore del suo settore.

In seguito il Mapper invierà al Reducer queste informazioni più quelle sui prezzi, volumi e date relativi al periodo 2009-2018.

```
class mapper:

    # will contain (sector, year): (results) pairs
    ticker_to_sector = {}
```

Per poter ottenere le informazioni sui settori dei ticker (che sono nel secondo file del dataset) si deve effettuare un join. Si è scelto di utilizzare la Distributed Cache di Hadoop per leggere il file `historical_stocks` già preprocessato (senza le righe con settore nullo).

```
    with open('historical_stocks_clean.csv') as hs_file:
        for row in hs_file:
            ticker, sector = row
            ticker_to_sector[ticker] = sector
```

A questo punto si processano i dati in input provenienti da `historical_stock_prices`, controllando se il ticker della riga abbia un settore corrispondente. Se non lo ha la riga verrà ignorata.

Il join viene effettuato dal lato del Mapper, semplicemente aggiungendo l'informazione del settore alle righe del ticker corrispondente.

Infine i dati vengono mandati uno per uno al Reducer.

```
    for row in INPUT:

        ticker, closePrice, volume, date = row

        # the ticker had a null sector, ignore it
        if ticker not in ticker_to_sector:
            continue

        if 2009 <= date.year <= 2018:
            # the join adds a column sector
            sector = ticker_to_sector[ticker]
            print(sector, ticker, date, closePrice, volume)
```

Nel Reducer vengono dapprima definite due strutture dati (dizionari) che serviranno per aggregare i risultati.

In particolare esse hanno il seguente formato:

- `tickerDataBySectorYear(ticker, sector, year)`
  - 'first_close_date': 2012-01-01,
  - 'first_close_value': 50.5,
  - 'last_close_date': 2012-12-31,
  - 'last_close_value': 240,
  - 'total_volume': 300000
- `aggregatedSectorYearData(sector, year)`
  - 'sum_initial_close': 4000,
  - 'sum_final_close': 6000,
  - 'max_perc_var_ticker': 'AAPL',
  - 'max_perc_var_value': 75,
  - 'max_total_volume_ticker': 'AAPL',
  - 'max_total_volume_value': 3000000

```
class reducer:

    tickerDataBySectorYear = {}
    aggregatedSectorYearData = {}
```

Per ogni riga proveniente dal Mapper si salvano le informazioni di ogni (ticker, settore e anno) in `tickerDataBySectorYear`. Anche qui se la tripla (ticker, settore e anno) non è mai stata vista essa verrà inizializzata, oppure aggiornata se erano già presenti dei valori.

```
for line in INPUT:

    sector, ticker, date, closePrice, volume = line

    # save (in memory) the info of each ticker per year and sector (inefficient)
    if (ticker, sector, date.year) not in tickerDataBySectorYear:
        newTicker = {'first_close_date': date,
                     'first_close_value': closePrice,
                     'last_close_date': date,
                     'last_close_value': closePrice,
                     'total_volume': volume}
        tickerDataBySectorYear[(ticker, sector, date.year)] = newTicker

    # the ticker in that year (with that sector) has been seen, update it
    else:
        currTicker = tickerDataBySectorYear[(ticker, sector, date.year)]
        if date < currTicker['first_close_date']:
            currTicker['first_close_date'] = date
            currTicker['first_close_value'] = closePrice
        if date > currTicker['last_close_date']:
            currTicker['last_close_date'] = date
            currTicker['last_close_value'] = closePrice
        currTicker['total_volume'] += volume
```

In modo analogo, iterando su `tickerDataBySectorYear`, si popolerà il dizionario

`aggregatedSectorYearData`:

- aggregando i valori iniziali e finali di prezzo di chiusura
- conservando il ticker con la variazione percentuale massima
- conservando il ticker con il maggior volumi di transazioni

Anche qui si inizializzerà la entry non presente nel dizionario se necessario.

```
# aggregate the single ticker and year data by sector
for (ticker, sector, year) in tickerDataBySectorYear:
    currTicker = tickerDataBySectorYear[(ticker, sector, year)]
    initialClose = currTicker['first_close_value']
    finalClose = currTicker['last_close_value']
    volume = currTicker['total_volume']

    percVar = calculatePercVar(initialClose, finalClose)

    # create a new dict to save the data
    if (sector, year) not in aggregatedSectorYearData:
        newData = {'sum_initial_close': initialClose,
                   'sum_final_close': finalClose,
                   'max_perc_var_ticker': ticker,
                   'max_perc_var_value': percVar,
                   'max_total_volume_ticker': ticker,
                   'max_total_volume_value': volume}
        aggregatedSectorYearData[(sector, year)] = newData

    # update the existing data
    else:
        currData = aggregatedSectorYearData[(sector, year)]
        currData['sum_initial_close'] += initialClose
        currData['sum_final_close'] += finalClose
        if percVar > currData['max_perc_var_value']:
            currData['max_perc_var_ticker'] = ticker
            currData['max_perc_var_value'] = percVar
        if volume > currData['max_total_volume_value']:
            currData['max_total_volume_ticker'] = ticker
            currData['max_total_volume_value'] = volume
```

Si ordinano i risultati ottenuti per nome del settore.

Infine, iterando su `aggregatedSectorYearData`, si calcola la variazione percentuale del settore nell'anno e si stampa ogni riga in output.

```
sortedResults = sorted(aggregatedSectorYearData.items(), key='sector', reverse=False)

for result in sortedResults:
    sector = result[0][0]
    year = result[0][1]

    currResult = aggregatedSectorYearData[(sector, year)]
```

```
    initialCloseSum = currResult['sum_initial_close']
    finalCloseSum = currResult['sum_final_close']
    currResult['total_perc_var'] = calculatePercVar(initialCloseSum, finalCloseSum)

    print(
        sector,
        year,
        currResult['total_perc_var'],
        currResult['max_perc_var_ticker'],
        currResult['max_perc_var_value'],
        currResult['max_total_volume_ticker'],
        currResult['max_total_volume_value'])
```

6.2 Hive

Per questo job sono state create diverse tabelle esterne, per chiarezza divise a seconda del task per cui esse hanno un'utilità:

UTILI PER IL TASK A

- Per ogni settore estrare l'anno dalle date dei prezzi

```
create table sector_2_date as
select distinct d2.sector, extract(year from d1.price_date)
from historical_stock_prices(size) as d1
    left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
order by d2.sector, `_c1`;
alter table sector_2_date change `_c1` year int;
```

- Per ogni settore, anno e ticker estrae la data della prima e dell'ultima quotazione dell'anno.

```
create table sector_ticker_min_max as
select d2.sector, sd.year, d1.ticker,
    min(d1.price_date) as first_date,
    max(d1.price_date) as last_date
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
left join sector_2_date as sd
    on d2.sector = sd.sector
    and sd.year = extract(year from d1.price_date)
where sd.year >=2009 and sd.year <= 2018
group by d2.sector, sd.year, d1.ticker
order by sector, year, d1.ticker;
```

- Per ogni settore e anno calcola la somma di tutte e quotazioni nella prima data dell'anno per quel settore.

```
create table sector_to_min_quot as
select d2.sector, sm.year, sum(d1.close_price) as first_quot
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
```

```
join sector_ticker_min_max as sm
    on d2.sector = sm.sector
    and sm.year = extract(year from d1.price_date)
    and d1.ticker = sm.ticker
where d1.price_date = sm.first_date
group by d2.sector, sm.year
order by d2.sector, sm.year;
```

- Per ogni settore e anno calcola la somma di tutte e quotazioni nell'ultima data dell'anno per quel settore.

```
create table sector_to_max_quot as
select d2.sector, sm.year, sum(d1.close_price) as last_quot
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
join sector_ticker_min_max as sm
    on d2.sector = sm.sector
    and sm.year = extract(year from d1.price_date)
    and d1.ticker = sm.ticker
where d1.price_date = sm.last_date and  d2.sector != "N/A"
group by d2.sector, sm.year
order by d2.sector, sm.year;
```

UTILI PER IL TASK B

- Per ogni settore e anno estrae il ticker con la sua prima quotazione per quel settore e in quell'anno.

```
create table sector_year_to_tickerFirstQuotation as
select d2.sector, sm.year, d1.ticker, close_price as first_quotation
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
left join sector_ticker_min_max as sm
    on d2.sector = sm.sector
    and d1.ticker = sm.ticker
where d1.price_date = sm.first_date
order by d2.sector, sm.year;
```

- Per ogni settore e anno estrae il ticker con la sua ultima quotazione per quel settore e in quell'anno.

```
create table sector_year_to_tickerLastQuotation as
select d2.sector, sm.year, d1.ticker, close_price as last_quotation
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
left join sector_ticker_min_max as sm
    on d2.sector = sm.sector
    and d1.ticker = sm.ticker
where d1.price_date = sm.last_date
order by d2.sector, sm.year;
```

- Per ogni settore e anno estrae il ticker con la sua prima e ultima quotazione per quel settore e in quell'anno (Join delle due tabelle precedenti).

```
create table sector_year_to_tickerFirstLastQuotation as
select s1.sector, s1.year, s1.ticker, s1.first_quotation, s2.last_quotation
from sector_year_to_tickerFirstQuotation as s1
left join sector_year_to_tickerLastQuotation as s2
    on (s1.sector = s2.sector and s1.year = s2.year and s1.ticker = s2.ticker)
order by s1.sector, s1.year;
```

- Per ogni settore, anno e ticker calcola la variazione percentuale del ticker in quell'anno per quel settore.

```
create table sector_year_to_variation as
select sector,
    year,
    ticker,
    max(((last_quotation - first_quotation)/first_quotation)*100) as variation
from sector_year_to_tickerFirstLastQuotation
group by sector, year, ticker;
```

- Per ogni settore e anno calcola la variazione massima avuta in quell'anno e per quel settore.

```
create table sector_year_to_maxVariation as
select sector, year, max(variation) as max_variation
from sector_year_to_variation
group by sector, year;
```

- Per ogni settore e anno estrae il ticker che ha avuto la variazione percentuale massima in quell'anno e per quel settore, con l'indicazione di tale variazione.

```
create table sector_year_to_maxTicker as
select smax.sector, smax.year, sv.ticker, smax.max_variation
from sector_year_to_maxVariation as smax
left join sector_year_to_variation as sv
    on smax.sector = sv.sector and smax.year = sv.year
where max_variation = variation;
```

UTILI PER IL TASK C

- Per ogni settore, anno e ticker calcola la somma dei volumi dei ticker in quell'anno e per quel settore.

```
create table sector_year_ticker_to_volumeSum as
select d2.sector,
    year(d1.price_date) as price_year,
    d1.ticker, sum(d1.volume) as volume
from historical_stock_prices(size) as d1
join historical_stocks_clean as d2 on d1.ticker = d2.ticker
group by d2.sector, year(d1.price_date), d1.ticker;
```

- Per ogni settore e anno estrae la somma di volumi massima in quell'anno e per quel settore.

```
create table sector_year_to_maxVolume as
select sector, price_year, max(volume) as maxVolume
from sector_year_ticker_to_volumeSum
group by sector, price_year
order by sector, price_year;
```

- Per ogni settore e anno estrae il ticker che ha la somma di volumi massima in quell'anno e per quel settore, con indicazione di tale somma.

```
create table sector_year_toMaxVolumeTicker as
select ayt.sector, ayt.price_year, ayt.ticker as v_ticker, ayt.volume
from sector_year_ticker_to_volumeSum as ayt
left join sector_year_to_maxVolume as aym
    on ayt.sector = aym.sector and ayt.price_year = aym.price_year
where volume = maxVolume;
```

QUERY FINALE

- Mette insieme tutte le precedenti tabelle per estrarre, per ogni settore e anno, la variazione percentuale della quotazione del settore nell'anno, l'azione del settore che ha avuto il maggior incremento percentuale nell'anno (con indicazione dell'incremento), l'azione del settore che ha avuto il maggior volume di transazioni nell'anno(con indicazione del volume).

```
create table job2_hive as
select d2.sector,
    smin.year,
    min(((smax.last_quot - smin.first_quot)/smin.first_quot)*100) as variation,
    max(sy.ticker), max(sy.max_variation), min(v_ticker), max(syv.volume)
from historical_stock_prices(size) as d1
left join historical_stocks_clean as d2 on d1.ticker = d2.ticker
left join sector_to_min_quot as smin
    on d2.sector = smin.sector and smin.year = extract(year from d1.price_date)
left join sector_to_max_quot as smax
    on d2.sector = smax.sector and smax.year = extract(year from d1.price_date)
left join sector_year_to_maxTicker sy
    on d2.sector = sy.sector and sy.year = extract(year from d1.price_date)
left join sector_year_toMaxVolumeTicker as syv
    on d2.sector = syv.sector and syv.price_year = extract(year from d1.price_date)
where smin.year >=2009
    and smin.year <= 2018
    and smax.year >=2009
    and smax.year <= 2018
    and d2.sector != "N/A"
group by d2.sector, smin.year
order by d2.sector, smin.year;
```

6.3 Spark

Spark caricherà i dati dei dataset `historical_stock_prices` o `historical_stocks` solo nel momento in cui deve effettuare una azione

```
historical_stock_prices = loadDataFromCsv()
historical_stocks = loadDataFromCsv()
```

Filtra i dati per estrarre soltanto quelli relativi al periodo compreso tra 2009 e 2018

Mappa per ottenere (ticker) -> (prezzo di chiusura, volume, data)

```
historical_stock_prices_filtered = historical_stock_prices
    .filter(2009 <= year <= 2018)
    .map(x -> (ticker, (close_price, volume, date)))
```

Mappa il secondo dataset per ottenere ticker -> settore (gli unici dati che ci interessano)

```
hs = historical_stocks
    .map(x -> (ticker, sector))
```

Viene effettuato il join tra i due dataset e viene crato un nuovo RDD mappato per avere (settore, year, ticker) -> (prezzo di chiusura, volume, data)

```
hsp_sector = historical_stock_prices_filtered.join(hs)
    .map((ticker, ((close_price, volume, date), sector))
         -> ((sector, year, ticker),(close, volume, date)))
```

Vengono create due RDD che per ogni settore, anno e ticker restituiscono il prezzo di chiusura alla prima e all'ultima data dell'anno per quel settore

```
first_quotation_close = hsp_sector
    .reduceByKey_minDate()
    .map(((sector, year, ticker),(close, volume, date)) -> ((sector, year, ticker),close))

last_quotation_close = hsp_sector
    .reduceByKey_maxDate()
    .map(((sector, year, ticker),(close, volume, date)) -> ((sector, year, ticker),close))
```

Viene effettuato il join tra i precedenti due RDD e una map per calcolare e aggiungere, per ogni settore, anno e ticker, la variazione percentuale in quell'anno per quel settore

```
ticker_percent_variation = first_quotation_close
     .join(last_quotation_close)
     .map(((sector, year, ticker),(first_close, last_close))
          -> ((sector, year, ticker),(first_close,
                                      last_close,
                                      percent_variation(first_close, last_close))
```

Viene effettuata prima una map per avere per ogni settore e anno il ticker e la variazione percentuale

Viene poi effettuata una reduce by key per avere il ticker con la variazione percentuale massima per quell'anno e in quel settore con indicazione di tale variazione

```
ticker_max_percent_var = ticker_percent_variation
    .map(((sector, year, ticker),(first_close, last_close, percent_var ))
         -> ((sector, year), (ticker, percent_var)))
    .reduceByKey(max_value(a, b))
```

Per ogni settore, anno e ticker calcola la somma dei volumi e poi prende il ticker con il massimo valore di somma di volumi per settore e anno

```
ticker_max_volume = hsp_sector
    .map(((sector, year, ticker),(close, volume, date)) -> ((sector, year, ticker), volume))
    .reduceByKey(volume_a + volume_b)
    .map(((sector, year, ticker), max_volume) -> ((sector, year), (ticker, max_volume)))
    .reduceByKey(max_value(a, b))
```

Viene effettuata una map rimuove la variazione precentuale dal valore e il ticker dalla chiave

Poi una seconda map per avere per ogni settore e anno la variazione percentuale totale

```
sector_year_percent_variation = ticker_percent_variation
    .map(((sector, year, ticker),(first_close, last_close, percent_var ))
         -> ((sector, year), (first_close, last_close)))
    .reduceByKey(sum_tuple(a, b))
    .map(((sector, year), (sum_first_close, sum_last_close))
         -> ((sector, year),
             (total_percent_variation = (sum_last_close - sum_first_close)
              /sum_first_close)*100))
```

Aggrega tutti i risultati intermedi e calcola i record finali per ottenere, per ogni settore e anno, la massima variazione della quotazione del settore nell'anno, l'azione con incremento maggiore nel settore e l'azione del settore con il maggior volume, con indicazione di tali valori.

```
results = sector_year_percent_variation
    .join(ticker_max_percent_var)
    .map(((sector, year), (total_percent_variation, (max_ticker, max_percent_var))
          -> ((sector, year), (total_percent_variation, max_ticker, max_percent_var))))
    .join(ticker_max_volume)
    .map(((sector, year),
          ((total_percent_variation, max_ticker, max_percent_var), (ticker, max_volume)))
         -> ((sector, year),
             (total_percent_variation, max_ticker, max_percent_var, ticker, max_volume)))
    .sortBySectorYear()
```

7 Job 3

Generare le coppie di aziende che si somigliano (sulla base di una soglia = 1%) in termini di variazione percentuale mensile nell'anno 2017.

Mostrare l'andamento mensile delle due aziende nel formato:

- 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 +0,7%, ...

7.1 MapReduce

Nel primo Mapper si fa il parsing dei campi necessari per il job, e si conservano solo i record con data relativa al 2017.

```
class first_mapper:

    for row in INPUT:

        ticker, closePrice, date = row

        # filter out all the years but 2017
        if date.year != 2017:
            continue

        print(ticker, closePrice, date)
```

Come già detto nella sezione sull'analisi del dataset, la scelta dell'utilizzo del `ticker` come chiave invece del `company_name` è per evitare di dover gestire proprio questi ultimi valori duplicati. Sarebbe stato possibile effettuando un preprocessamento, e non sarebbe stato banale aggregare i dati dei ticker con lo stesso nome di azienda.

Il Reducer definisce un dizionario `tickerToMonthVar` che conterrà, per ogni ticker, i valori di chiusura iniziali e finali di ogni mese dell'anno. Il formato è indicato nel commento di seguito.

```
class first_reducer:

# saves the monthly first and last close price for each ticker
# (along with their dates for comparing)
# tickerToMonthVar = {'AAPL':
#                   { GEN: {'first_close': 15, 'last_close': 20,
#                    'first_date': .., 'last_date': ..},
#                     FEB: {'first_close': 20, 'last_close': 2, ...}
#                     ...
#                     DIC: {'first_close': 5, 'last_close': 2, ...} ...}
tickerToMonthVar = {}
```

A questo punto si leggono le righe provenienti dal primo Mapper, inserendo il mese e i corrispondenti valori per ogni ticker.

Anche qui, se i dati non sono presenti nel dizionario vanno inizializzati, oppure aggiornati se erano presenti.

```
for row in INPUT:

    ticker, closePrice, date = row

    # the ticker and month are already in the dict, update them
    if (ticker in tickerToMonthVar) and (date.month in tickerToMonthVar[ticker]):
```

```
            currTickerMonth = tickerToMonthVar[ticker][date.month]
            if date < currTickerMonth['first_date']:
                currTickerMonth['first_close'] = closePrice
                currTickerMonth['first_date'] = date
            if date > currTickerMonth['last_date']:
                currTickerMonth['last_close'] = closePrice
                currTickerMonth['last_date'] = date

        # insert ticker data in the dict
        else:
            currTickerMonth = {'first_close': closePrice,
                               'last_close': closePrice,
                               'first_date': date,
                               'last_date': date}
            if ticker not in tickerToMonthVar:
                tickerToMonthVar[ticker] = {date.month: currTickerMonth}
            else:
                tickerToMonthVar[ticker][date.month] = currTickerMonth
```

Per mandare i dati al prossimo Mapper si è fatta una separazione, per ogni ticker, dei mesi corrispondenti. Si itera su tutti i ticker, e poi su tutti i mesi dei ticker, per stampare la variazione percentuale associata.

Ciò introduce una inefficienza, poiché nel secondo Reducer essi verranno di nuovo aggregati. Tuttavia ciò è stato necessario per via del paradigma di programmazione di MapReduce.

```
# print the data structure calculating the monthly percent variation
for ticker in tickerToMonthVar:
    yearData = tickerToMonthVar[ticker]

    # iterate over all months for the ticker
    for month in yearData:
        initialClose = yearData[month]['first_close']
        finalClose = yearData[month]['last_close']

        # prints ('AAPL', 3, 25.3) separating each month for the same ticker
        print(ticker, month, calculatePercVar(initialClose, finalClose))
```

Il secondo Mapper restituisce l'identità, non effettuando nessuna operazione.

```
class second_mapper:

    for line in INPUT:
        ticker, month, percent_variation = line
        print(ticker, month, percent_variation)
```

Nel secondo Reducer si definiscono due dizionari con le seguenti chiavi e valori:

- `tickerToMonthsVar(ticker)` contiene un dizionario, per ogni ticker, coi seguenti valori:
  - GEN: percent_variation
  - FEB: percent_variation

  - ...
  - DIC: percent_variation
- `crossProduct(ticker_1, ticker_2)` contiene un dizionario, per ogni coppia di ticker validi, con:
  - GEN: (percent_variation_ticker_1, percent_variation_ticker_2)
  - FEB: (percent_variation_ticker_1, percent_variation_ticker_2)
  - ...
  - DIC: (percent_variation_ticker_1, percent_variation_ticker_2)

Per semplicità si è adottato un formato più semplice rispetto alle specifiche, che richiedevano di inserire il ticker delle aziende su ogni coppia di variazioni percentuali. Nell'implementazione sono stati inseriti solo all'inizio.

Inoltre la soglia di variazione percentuale entro la quale considerare due aziende che si somigliano è stata impostata a THRESHOLD = 1%

```
class second_reducer:

    THRESHOLD = 1

    # the dict aggregates all months for each ticker
    # tickerToMonthsVar = {'AAPL': {GEN: 13.5, FEB: 12.0, ... , DIC: -5.3}, ...}
    tickerToMonthsVar = {}

    # structure to contain the cross product (without duplicates or inverted pairs)
    # crossProduct = {('AAPL', 'AMZN'): {GEN: (2, 2.6), FEB: (-1, 3.7), ... },
    #                 ('AAPL', 'BTP'): {GEN: (2, -1), FEB: (-1, 3.4), ...}}
    crossProduct = {}
```

E' stata anche definita una funzione `mergeTickerPair` per combinare due entry di `tickerToMonthsVar`, ottenendo le coppie di variazioni percentuali per ogni mese.

La funzione restituisce solo le coppie di aziende che si somigliano, controllando che la soglia dell'1% sia rispettata per tutti i mesi.

Una scelta effettuata esclusivamente nell'implementazione MapReduce è stata quella di considerare anche le coppie di aziende che non hanno tutti e 12 i mesi, purché abbiano dati relativi agli stessi mesi. Ad esempio vengono scartate le coppie dove ticker_1 ha solo gennaio e ticker2 solo febbraio, mentre se entrambe avessero gennaio verrebbero conservate.

Nelle altre implementazioni (Hive, Spark) si considerano solo le aziende che hanno tutti e 12 i mesi, riducendo il numero di record di output (da circa 600 a 450).

```
# generates a merged pair to insert in the crossProduct data structure
# it will return None if the pair of tickers is not similar enough (or months not consistent)

    def mergeTickerPair(ticker1, ticker2, tickerData):
        result = {}
        ticker1Data = tickerData[ticker1]
        ticker2Data = tickerData[ticker2]

        # the comparison will fail if the months data are not consistent
```

```
    if ticker1Data.keys() != ticker2Data.keys():
        return None

    for month in ticker1Data:
        percVar1 = ticker1Data[month]
        percVar2 = ticker2Data[month]
        percent_difference = abs(percVar1 - percVar2)

        if percent_difference <= THRESHOLD:
            result[month] = (percVar1, percVar2)
        else:
            return None
    return result
```

A questo punto il secondo Reducer riceve le coppie (ticker: month_percent_variation) dal secondo Mapper.

Come già anticipato esse verranno aggregate in `tickerToMonthsVar`, che conterrà tutti i mesi per ogni ticker.

```
# each month is unique for a given ticker (we assume no duplicates)
for row in sys.stdin:

    ticker, month, percVar = row

    if ticker not in tickerToMonthsVar:
        tickerToMonthsVar[ticker] = {month: percVar}
    else:
        tickerToMonthsVar[ticker][month] = percVar
```

Si itera su ogni coppia di ticker effettuando un prodotto cartesiano. Vengono conversate solo le coppie di ticker che sono simili secondo le specifiche, usando la funzione `mergeTickerPair`, che controlla i singoli valori di variazione percentuale e poi unisce i due dizionari.

Durante l'iterazione si ignorano le coppie già presenti nel dizionario (anche invertite) e le coppie di un ticker con se stesso.

```
for ticker_1 in tickerToMonthsVar:
    for ticker_2 in tickerToMonthsVar:
        if (ticker_1, ticker_2) in crossProduct
        or (ticker_2, ticker_1) in crossProduct
        or ticker_1 == ticker_2:
            continue
        else:
            mergedPair = mergeTickerPair(ticker_1, ticker_2, tickerToMonthsVar)
            # the pair was not valid
            if mergedPair is None:
                continue
            else:
                crossProduct[(ticker_1, ticker_2)] = mergedPair
```

```
    for (ticker_1, ticker_2) in crossProduct:
        pair = (ticker_1, ticker_2)
        print(pair, crossProduct[pair])
```

7.2 Hive

Per l'ultimo job vengono create diverse tabelle esterne che saranno poi utilizzate per costruire la tabella finale:

- Filtra il database per mostrare solo i dati del 2017

```
create table 2017_data as
select ticker, price_date, extract(month from price_date), close_price
from historical_stock_prices(size)
where extract(year from price_date) = 2017
order by ticker, price_date;
alter table 2017_data change `_c2` month int;
```

- Per ogni ticker e mese estrae il primo e l'ultimo prezzo di chiusura del ticker in quel mese

```
create table ticker_month_to_max_min_date as
select ticker, month, min(price_date) as min_date, max(price_date) as max_date
from 2017_data
group by ticker, month;
```

- Per ogni ticker e mese estrae la prima quotazione di quel ticker in quel mese

```
create table ticker_to_first_month_quotation as
select d.ticker, d.month, d.close_price
from 2017_data as d
left join ticker_month_to_max_min_date as tm on d.ticker = tm.ticker
where price_date = min_date;
```

- Per ogni ticker e mese estrae l'ultima quotazione di quel ticker in quel mese

```
create table ticker_to_last_month_quotation as
select d.ticker, d.month, d.close_price
from 2017_data as d
left join ticker_month_to_max_min_date as tm on d.ticker = tm.ticker
where price_date = max_date;
```

- Per ogni ticker e mese estrae la prima e l'ultima quotazione del ticker in quel mese (join tra le due tabelle precedenti)

```
create table ticker_to_first_last_month_quotation as
select first.ticker, first.month, first.close_price as first_quotation,
    last.close_price as last_quotation
from ticker_to_first_month_quotation as first
left join ticker_to_last_month_quotation as last
on first.ticker = last.ticker and first.month = last.month
order by ticker, month;
```

- Per ogni ticker e mese calcola la variazione percentuale di quel ticker in quel mese

```
create table ticker_month_to_variation as
select ticker, month, (((last_quotation - first_quotation)/first_quotation)*100) as variation
from ticker_to_first_last_month_quotation
order by ticker, month;
```

- Per ogni coppia di ticker e per ogni mese estrae la variaizone percentuale del primo e del secondo ticker per quel mese

```
create table variations_comparison as
select t1.ticker as ticker_1,
    t2.ticker as ticker_2,
    t1.month, cast(t1.variation as decimal(10,2)) as variation_1,
    cast(t2.variation as decimal(10,2)) as variation_2
from ticker_month_to_variation as t1, ticker_month_to_variation as t2
where t1.ticker > t2.ticker and t1.month = t2.month and (abs(t1.variation - t2.variation) <= 1)
order by ticker_1, ticker_2, t1.month;
```

- Crea la tabella dei risultati raggruppando per coppie di ticker e trasformando i valori della colonna "month" in nuove colonne, ognuna per ogni mese, popolate dai rispettivi valori per quel mese.

```
create table raw_results as
select ticker_1 as t1, ticker_2 as t2,
max(case when month="1"
    then "GEN:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as gen,
max(case when month="2"
    then "FEB:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as feb,
max(case when month="3"
    then "MAR:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as mar,
max(case when month="4"
    then "APR:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as apr,
max(case when month="5"
    then "MAG:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as mag,
max(case when month="6"
    then "GIU:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as giu,
max(case when month="7"
    then "LUG:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as lug,
max(case when month="8"
    then "AGO:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as ago,
max(case when month="9"
    then "SET:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as sep,
max(case when month="10"
    then "OTT:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as ott,
max(case when month="11"
    then "NOV:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as nov,
max(case when month="12"
    then "DIC:"||" "||"("||variation_1||"%"||", "||variation_2||"%"||")" else "" end) as dic
from variations_comparison
group by ticker_1, ticker_2;
```

- Nel mostrare i risutati si filtra la tabella finale affinché mostri soltanto le coppie di ticker che si somigliano in tutti i mesi.

```
create table job3_hive as
select * from raw_results
where (gen!="" and feb!="" and mar!="" and apr!="" and mag!="" and giu!=""
       and lug!="" and ago!="" and sep!="" and ott!="" and nov!="" and dic!="");
```

7.3 Spark

I dati contenuti in `historical_stock_prices` verranno caricati dal .csv in una RDD (e partizionati a seconda dei thread) solo nel momento in cui un'azione viene richiesta.

```
historical_stock_prices = loadDataFromCsv()
```

Filtra i dati per ottenere soltanto quelli relativi all'anno 2017

```
ticker_close_date = historical_stock_prices
    .filter(x -> year(price_date) == 2017)
    .map(x -> (ticker, close_price, price_date))
```

Mappa i valori per avere, per ogni ticker e mese, la tupla originale corrispondente

```
ticker_month_to_list = ticker_close_date
    .map((ticker, close_price, price_date)
         -> ((ticker, month(price_date), (ticker, close_price, price_date))))
```

Reduce By Key per ottenere, per ogni ticker e mese, la prima data e il primo prezzo disponibili in quel mese per quel ticker

Map per ottenere, per ogni ticker e mese, il prezzo minimo

```
ticker_month_to_mindate = ticker_month_to_list
    .reduceByKey(min_price_and_date)
    .map(((ticker, month), (ticker, min_close_price, min_price_date))
         -> ((ticker, month), min_close_price))
```

Reduce By Key per ottenere, per ogni ticker e mese, l'ultima data e l'ultimo prezzo disponibili in quel mese per quel ticker

Map per ottenere, per ogni ticker e mese, il prezzo massimo

```
ticker_month_to_maxdate = ticker_month_to_list
    .reduceByKey(max_price_and_date)
    .map(((ticker, month), (ticker, max_close_price, max_price_date))
         -> ((ticker, month), max_close_price))
```

Viene effettuato il join tra i due RDD precedenti e una map per ottere, per ogni ticker e mese, la variazione percentuale per quel ticker in quel mese

```
ticker_month_variation = ticker_month_to_mindate
    .join(ticker_month_to_maxdate)
    .map(((ticker, month), (min_close_price, max_close_price))
         -> ((ticker, month), percent_variation(min_close_price, max_close_price))
```

Raggruppa le variazioni percentuali di tutti i mesi per ogni ticker

Filtra i record non relativi ad un intero anno

Ordina per mese

```
ticker_aggregate_months = ticker_month_variation
    .map(((ticker, month), perc_variation) -> (ticker, (month, variation)))
    .groupByKey()
    .filter(length(list of (month, variation)) == 12)
    .map((ticker, list of (month, variation)) -> (ticker, sorted list by month))
```

Effettua il prodotto cartesiano per individuare tutte le possibili coppie di ticker

Filtra i record univoci e quelli che si somigliano in base ad una soglia (1%) in termini di variazione percentuale mensile

```
ticker_pairs_threshold = ticker_aggregate_months
    .cartesian(ticker_aggregate_months)
    .filter(ticker_1 < ticker_2 and abs(variation_1 - variation_2) < 1)
    .map(
        ((ticker_1, sorted list by month),(ticker_2, sorted list by month))
        -> ((ticker_1, ticker_2), merged list of months))
```

8 Risultati

Di seguito vengono illustrati i risultati ottenuti eseguendo le diverse implementazioni dei job. Inoltre vengono confrontati i tempi di esecuzione al variare delle dimensioni del dataset di input (descritte nella prima sezione).

[15]:
```
import seaborn as sns
%matplotlib inline
import matplotlib.pyplot as plt
```

[16]:
```
sns.set()
```

[17]:
```
job1_data = pd.read_csv('./benchmarks/job1_all_data.csv')
job2_data = pd.read_csv('./benchmarks/job2_all_data.csv')
job3_data = pd.read_csv('./benchmarks/job3_all_data.csv')
```

[18]:
```
def plot_job_benchmark(job_data, job_n):
    plt.figure(figsize=(14,8))
    fig = sns.lineplot(x='variable', y='value', hue='tech', data=job_data,␣
 ↪marker="o", linewidth = 2.5,
                       palette=["C0", "C1", "C2", "C0", "C1", "C2"],␣
 ↪style='tech',
                       dashes=["", "", "", (2, 2), (2, 2), (2, 2)])
    plt.xlabel("Dataset Size (MB)")
    plt.ylabel("Time (Minutes)")
    plt.title("Job"+str(job_n)+" Execution Times")
```

```
    fig.legend(title='Tecnology')
    plt.show(fig)
```

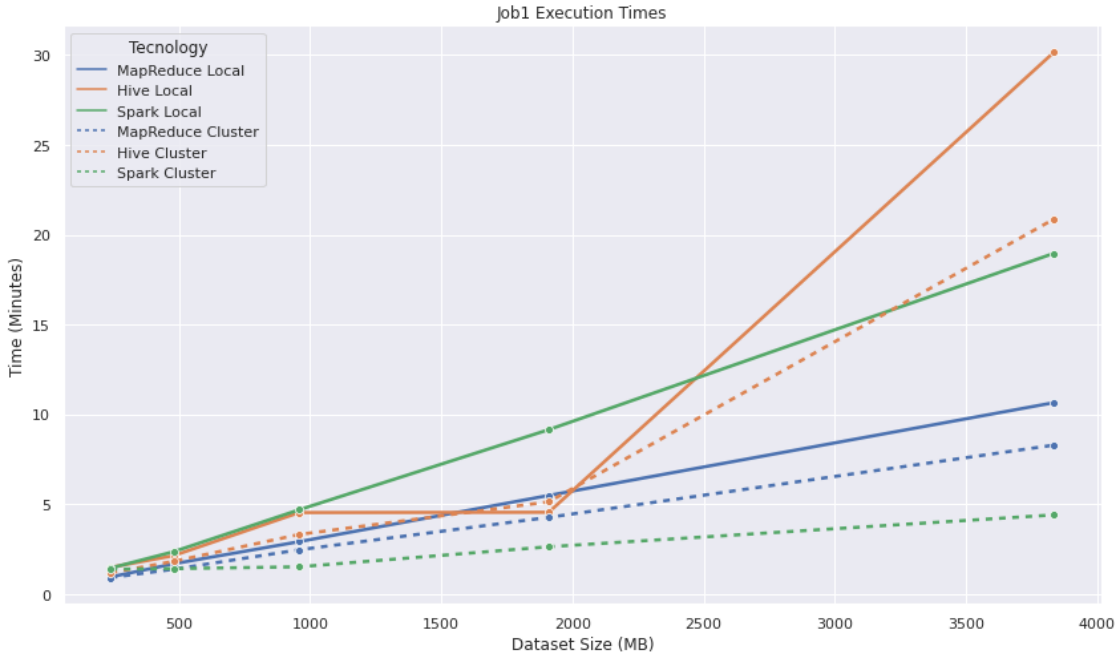
8.1 Job 1

Risultati del primo job

Ticker	Prima Quot.	Ultima Quot.	Var. Perc.	Prezzo Min	Prezzo Max
A	1999-11-18	2018-08-24	109.636%	7.510	115.879
AA	1970-01-02	2018-08-24	508.325%	3.604	117.194
AABA	1996-04-12	2018-08-24	4910.909%	0.645	125.031
AAC	2018-01-16	2018-08-24	4.856%	7.789	12.960
AAL	2005-09-27	2018-08-24	101.139%	1.450	63.270
AAME	1980-03-17	2018-08-24	-29.870%	0.375	15.800
AAN	1987-01-20	2018-08-24	4683.263%	0.481	51.529
AAOI	2013-09-26	2018-08-24	330.421%	8.079	103.410
AAON	1992-12-16	2018-08-24	41348.203%	0.089	43.299
AAP	2001-11-29	2018-08-24	1084.149%	12.329	201.240

Di seguito il grafico che confronta i tempi di esecuzione al variare della dimensione del dataset, sia in cluster (linee tratteggiate) che in locale (linee piene).

[19]:
```
plot_job_benchmark(job1_data, 1)
```



Si può notare un andamento generalmente lineare di tutte le tecnologie usate sia in locale che su cluster.

In particolare si nota come Spark in locale sia più onerosa computazionalmente rispetto a MapReduce. Su cluster invece per via della grande disponibilità di risorse hardware è con un buon margine la tecnologia più prestante.

Per quanto riguarda MapReduce e Hive si notano miglioramenti minori rispetto alle esecuzioni in locale.

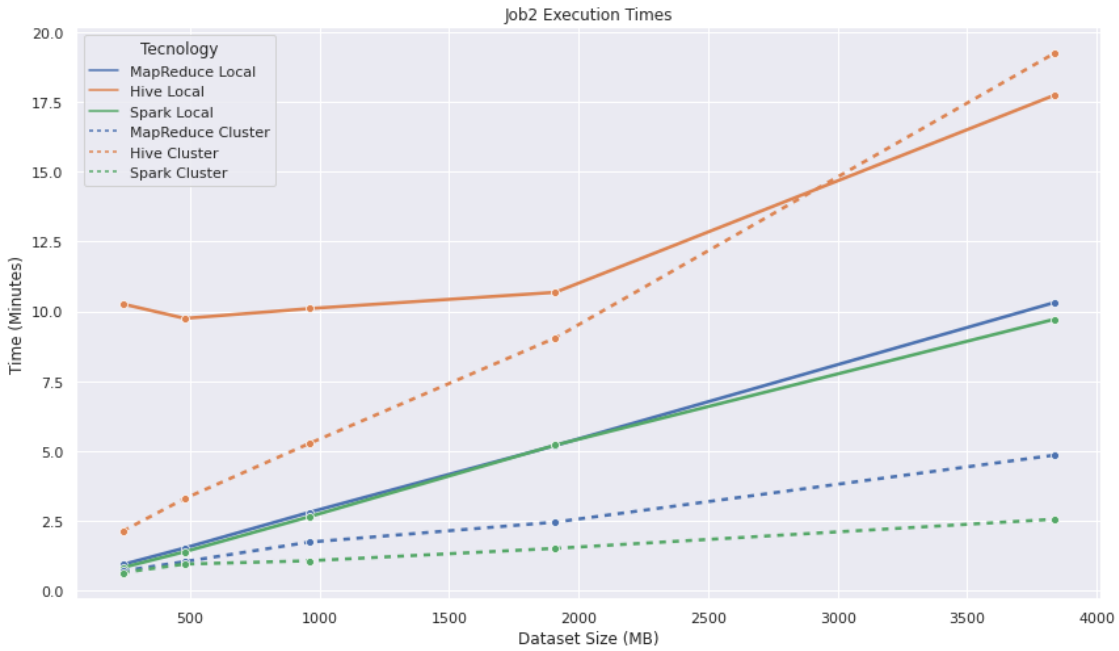
8.2 Job 2

Risultati del secondo job

Settore	Anno	Tot. Var.	Ticker	Max. Var.	Ticker	Max. Volume
BASIC INDUSTRIES	2009	3.482	GURE	709.722	FCX	9141685400.0
BASIC INDUSTRIES	2010	21.790	BLD	519.802	FCX	6891808600.0
BASIC INDUSTRIES	2011	-58.600	ROAD	188.704	FCX	5150807800.0
BASIC INDUSTRIES	2012	-68.788	PATK	261.860	VALE	4659766700.0
BASIC INDUSTRIES	2013	10.322	XRM	416.927	VALE	4428233700.0
BASIC INDUSTRIES	2014	-71.902	BLD	884.599	VALE	5660183200.0
BASIC INDUSTRIES	2015	-48.101	SUM	35191.629	FCX	7286761300.0
BASIC INDUSTRIES	2016	13.829	TECK	451.790	FCX	10464699500.0
BASIC INDUSTRIES	2017	15.279	OPNT	310.178	VALE	7023267600.0
BASIC INDUSTRIES	2018	-3.079	XRM	213.817	VALE	3710091900.0

Di seguito il grafico che confronta i tempi di esecuzione al variare della dimensione del dataset, sia in cluster (linee tratteggiate) che in locale (linee piene).

[20]:
```
plot_job_benchmark(job2_data, 2)
```



In questo grafico si può apprezzare in modo migliore l'andamento dei tempi di esecuzione al variare della dimensione di input.

Hive è di gran lunga la tecnologia più onerosa, e i tempi su cluster risultano essere più del doppio della controparte MapReduce e Spark. In locale mantiene per ogni dimensione di input dei tempi estremamente alti, poco abbattibili per via del prodotto cartesiano che bisogna effettuare per il job.

In locale MapReduce e Spark hanno un andamento pressoché identico. Su cluster hanno significativi miglioramenti, dimezzando i tempi di esecuzione di MapReduce, e mostrando prestazioni ancora migliori per Spark.

8.3 Job 3

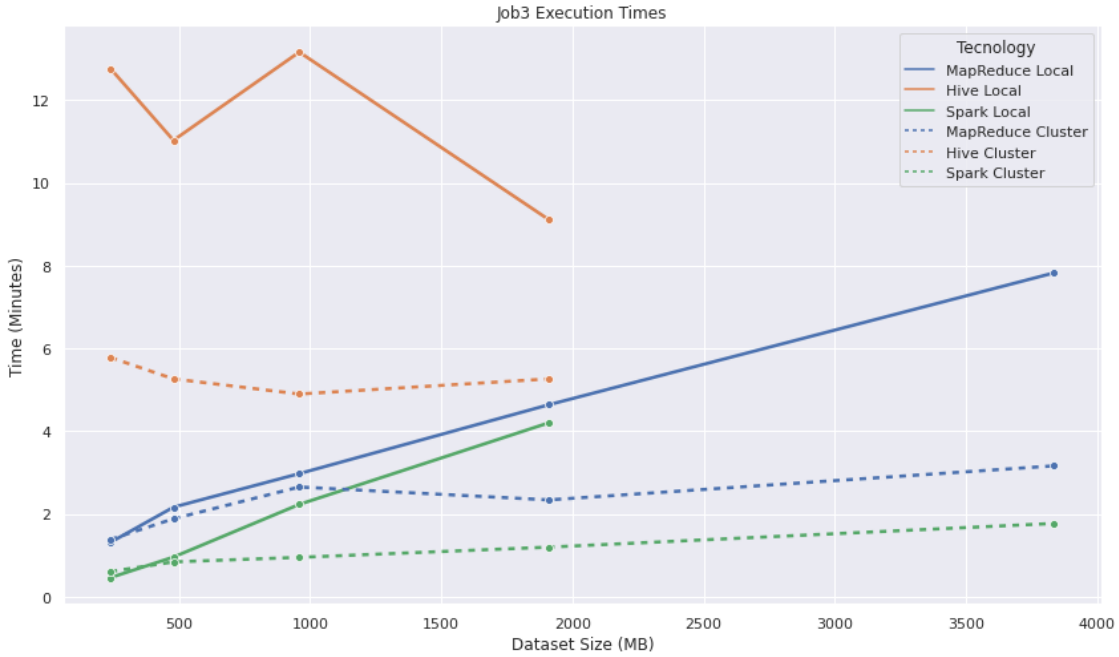
This is a sub paragraph, formatted in heading 3 style

```
(Ticker1, Ticker2) {Mese: (Var. Perc. Mese 1, Var. Perc. Mese 2), ...}

('OSBCP', 'TCRZ'){
'GEN': (1.678, 1.768), 'FEB': (0.389, 0.077), 'MAR': (0.0, 0.735),
'APR': (0.875, 1.124), 'MAG': (-0.095, -0.192), 'GIU': (1.156, 0.578),
'LUG': (0.190, -0.307), 'AGO': (-0.382, 0.269), 'SET': (-0.286, -0.546),
'OTT': (0.673, 0.387), 'NOV': (-0.095, -0.424), 'DIC': (-0.382, -0.276)}

('OSBCP', 'ISF'){
'GEN': (1.678, 0.832), 'FEB': (0.389, 0.512), 'MAR': (0.0, -0.117),
'APR': (0.875, 0.698), 'MAG': (-0.095, -0.885), 'GIU': (1.156, 0.310),
'LUG': (0.190, 0.426), 'AGO': (-0.382, -0.385), 'SET': (-0.286, -0.541),
'OTT': (0.673, 0.194), 'NOV': (-0.095, -0.233), 'DIC': (-0.382, 0.038)}

('OXLCO', 'VRIG'){
'GEN': (0.078, 0.247), 'FEB': (0.980, 0.159), 'MAR': (-0.583, 0.0),
'APR': (0.352, 0.433), 'MAG': (-0.039, -0.067), 'GIU': (-0.313, -0.118),
'LUG': (0.668, -0.079), 'AGO': (-0.078, -0.015), 'SET': (-0.859, 0.075),
'OTT': (-0.937, -0.051), 'NOV': (-0.828, -0.019), 'DIC': (0.474, 0.003)}

('OXLCO', 'VGSH'){
'GEN': (0.078, 0.214), 'FEB': (0.980, 0.049), 'MAR': (-0.583, 0.197),
'APR': (0.352, 0.115), 'MAG': (-0.039, 0.164), 'GIU': (-0.313, -0.016),
'LUG': (0.668, 0.214), 'AGO': (-0.078, 0.164), 'SET': (-0.859, -0.131),
'OTT': (-0.937, -0.098), 'NOV': (-0.828, -0.148), 'DIC': (0.474, -0.198)}

('OXLCO', 'VCSH'){
'GEN': (0.078, 0.441), 'FEB': (0.980, 0.364), 'MAR': (-0.583, 0.163),
'APR': (0.352, 0.288), 'MAG': (-0.039, 0.501), 'GIU': (-0.313, 0.087),
'LUG': (0.668, 0.463), 'AGO': (-0.078, 0.261), 'SET': (-0.859, -0.012),
'OTT': (-0.937, 0.062), 'NOV': (-0.828, -0.250), 'DIC': (0.474, -0.276)}

('OXLCO', 'CIU'){
'GEN': (0.078, 0.462), 'FEB': (0.980, 0.775), 'MAR': (-0.583, 0.322),
```

```
'APR': (0.352, 0.513), 'MAG': (-0.039, 0.704), 'GIU': (-0.313, 0.009),
'LUG': (0.668, 0.813), 'AGO': (-0.078, 0.408), 'SET': (-0.859, -0.054),
'OTT': (-0.937, 0.018), 'NOV': (-0.828, -0.218), 'DIC': (0.474, -0.128)}

('OXLCO', 'ECCB'){
'GEN': (0.078, 0.196), 'FEB': (0.980, 0.765), 'MAR': (-0.583, 0.308),
'APR': (0.352, 0.193), 'MAG': (-0.039, 0.613), 'GIU': (-0.313, -0.644),
'LUG': (0.668, 1.006), 'AGO': (-0.078, 0.723), 'SET': (-0.859, -0.643),
'OTT': (-0.937, -0.950), 'NOV': (-0.828, -0.076), 'DIC': (0.474, 0.114)}

('OXLCO', 'FTSM'){
'GEN': (0.078, 0.033), 'FEB': (0.980, 0.083), 'MAR': (-0.583, -0.016),
'APR': (0.352, 0.0), 'MAG': (-0.039, -0.016), 'GIU': (-0.313, 0.008),
'LUG': (0.668, 0.016), 'AGO': (-0.078, 0.016), 'SET': (-0.859, 0.0),
'OTT': (-0.937, 0.049), 'NOV': (-0.828, -0.016), 'DIC': (0.474, -0.049)}

('OXLCO', 'HYLS'){
'GEN': (0.078, 0.144), 'FEB': (0.980, 1.049), 'MAR': (-0.583, -0.832),
'APR': (0.352, 0.429), 'MAG': (-0.039, 0.386), 'GIU': (-0.313, -0.606),
'LUG': (0.668, 0.567), 'AGO': (-0.078, -0.685), 'SET': (-0.859, -0.406),
'OTT': (-0.937, -0.346), 'NOV': (-0.828, -0.654), 'DIC': (0.474, 0.020)}

('OXLCO', 'HYXE'){
'GEN': (0.078, 0.434), 'FEB': (0.980, 1.196), 'MAR': (-0.583, -1.427),
'APR': (0.352, 0.868), 'MAG': (-0.039, 0.428), 'GIU': (-0.313, 0.283),
'LUG': (0.668, 0.767), 'AGO': (-0.078, -0.319), 'SET': (-0.859, -0.286),
'OTT': (-0.937, -0.009), 'NOV': (-0.828, -0.440), 'DIC': (0.474, -0.351)}
```

Di seguito il grafico che confronta i tempi di esecuzione al variare della dimensione del dataset, sia in cluster (linee tratteggiate) che in locale (linee piene).

[21]:
```
plot_job_benchmark(job3_data, 3)
```



Il tempo di esecuzione per MapReduce e Spark è lineare al crescere del dataset, e la curva ha una pendenza molto inferiore su cluster, dove si dispone di più risorse hardware.

Si può notare come manchi un datapoint per Spark in locale con il dataset ~4GB. Pur tentando di aumentare la memoria riservata a Spark non si è riusciti ad evitare di sforare il Java Heap size. Si può ipotizzare con confidenza che il risultato sarebbe stato in linea con l'andamento lineare.

Il tempo di esecuzione di Hive per questo job, seppur molto variabile in locale, ha un andamento che sembra quasi costante al variare delle dimensioni del dataset. In realtà, come si è visto negli altri job, il tempo di esecuzione aumenta linearmente con la dimensione del dataset, ma per questo job al diminuire della dimensione del dataset il tempo non diminuisce.

Inoltre non è stato possibile ottenere risultati con il dataset ~4GB per via dei tempi estremamente lunghi (superiori a un'ora). Si ipotizza che avendo scelto un sampling casuale per generare questo dataset possa esserci qualche problema nel gestire righe duplicate.

9 Conclusioni

Per quanto riguarda le tecnologie utilizzate si è riscontrato come MapReduce e Spark siano più prestanti di Hive sia in locale che su cluster.

In locale i tempi per eseguire i Job in MapReduce e Spark rislta essere abbastanza simile per i tre Job. Su cluster, dove le risorse hardware sono ben maggiori, si possono apprezzare i benefici di performance di Spark, che fa uso di RDD con caching in memoria.

Inoltre Spark non vincola il programmatore ad utilizzare un paradigma di tipo Map-Reduce, e consente di effettuare operazioni come il join o il prodotto cartesiano più agevolmente.

Per l'esecuzione su dataset più piccoli (<1GB) non c'è molta differenza tra l'esecuzione in locale e su cluster, mentre solo su dimensioni più grandi si possono apprezzare i benefici di un'architettura distribuita.