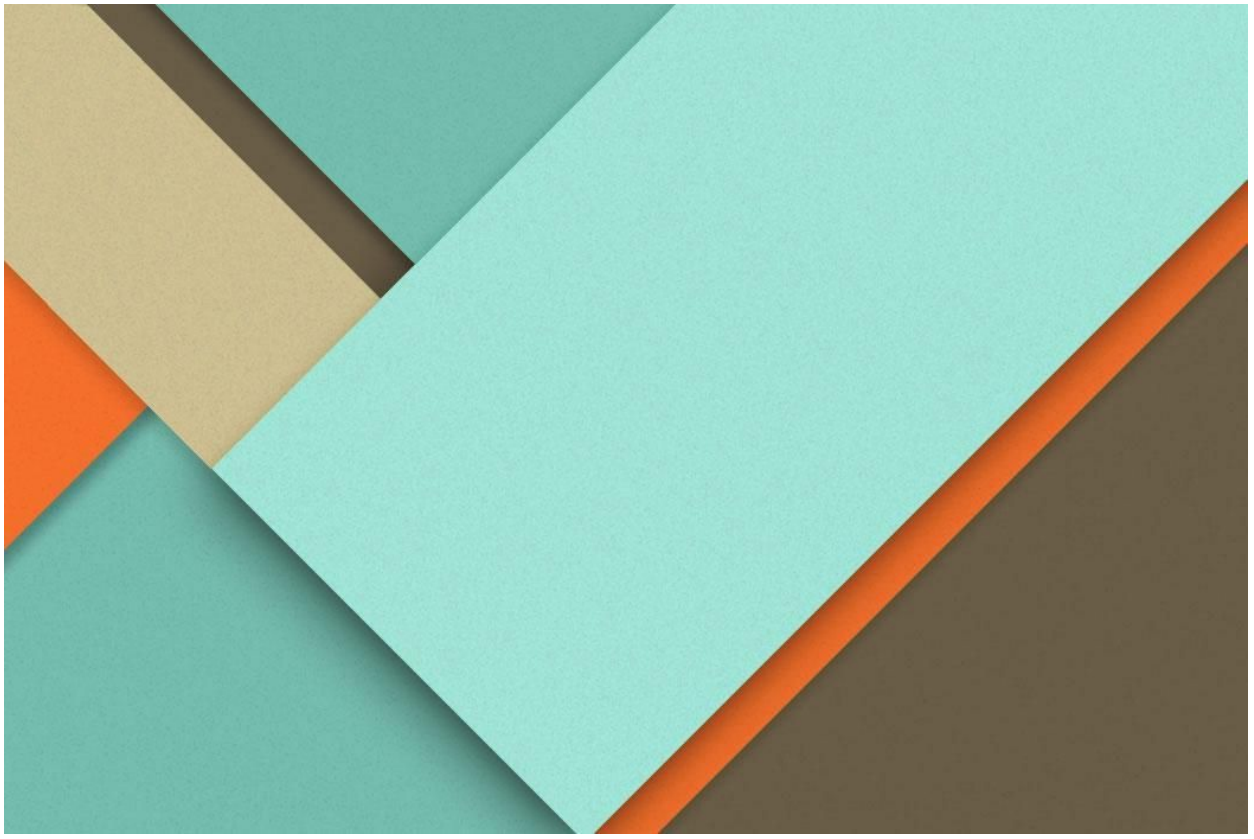
[Big Data ROMA TRE]

AA 2018-2019



Project 1: Historical Stock Analysis

Gruppo: bigdata-pals

Michele Ciaffarafà (#529060)

Lorenzo Guidaldi (#473014)

Codice del progetto: https://github.com/sullyD64/bigdata-2019/tree/master/project1

Premessa

Il nostro lavoro ha riguardato la progettazione e realizzazione di diversi job MapReduce, Hive e Spark per l'estrazione e il processamento di dati provenienti da uno storico di quotazioni di borsa.

Il dataset consiste di due file .csv: ***historical_stocks*** e ***historical_stock_prices***. Il primo contiene un elenco di titoli azionari, identificati da un simbolo e correlati con i dettagli dell'azienda proprietaria dell'azione. Il secondo è uno storico giornaliero dei titoli dal 1970 al 2018 per cui sono riportati i prezzi di apertura e chiusura, minimo e massimo raggiunto e numero di transazioni (volume).

Nel resto del report faremo riferimento ai due dataset rispettivamente con i nomi **legend** e **history**.

Tecnologie e strumenti

↘ Abbiamo scelto il linguaggio Python per realizzare l'implementazione dei job MapReduce e Spark. In particolare, per MapReduce abbiamo usato **hadoop-streaming**, una utility di Hadoop che consente di creare ed eseguire job utilizzando come mapper e reducer qualunque script o comando eseguibile.

In hadoop-streaming, il mapper e il reducer comunicano utilizzando lo standard input e lo standard output; tra le due fasi, Hadoop esegue automaticamente lo shuffle & sort in modo del tutto analogo ad un sort unix. L'utilizzo di interfacce standard consente di eseguire il job da riga di comando sfruttando la pipe Unix.

Ad esempio, per testare un wordcount è sufficiente eseguire da terminale la pipe:

```
~$ cat words.txt | python mapper.py | sort | python reducer.py
```

L'utilizzo di hadoop-streaming ci ha consentito di testare in modo agile il codice dei tre job MapReduce. L'unico "svantaggio" che questa tecnologia comporta è che l'operazione di shuffle (raggruppamento dei valori dopo la map) va implementata manualmente nel reducer; nonostante ciò, tutte le righe con stessa chiave sono presentate sequenzialmente al reducer, pertanto è facile processare i gruppi usando semplici strutture di controllo current-next.

↘ Per Hive abbiamo creato le due tabelle per legend e history utilizzando il SERDE (serializer-deserializer) **OpenCSVSerde**. Questo ci ha consentito di leggere senza errori i file .csv, passo critico specialmente per legend, dato che alcuni valori dei campi *name*, *sector* e *industry* contenevano al loro interno delle virgole ed erano racchiusi tra doppi apici. Specificando la property **skip.header.line.count =1** abbiamo inoltre potuto caricare direttamente i file senza doverli preprocessare rimuovendo gli header.

↘ Per Spark abbiamo utilizzato l'API **pyspark** per operare sugli RDD, e la **SparkSession** di *SparkSQL* per il parsing iniziale dei dati da .csv. Come per Hive, volevamo infatti caricare le tabelle senza dover modificare i file originali. L'unica interazione con SparkSQL è nel seguente snippet:

```
def load_data(spark_session, path_to_file):
    df = spark_session.read.format("csv") \
        .option("inferSchema", "true").option("header", "true") \
        .load(path_to_file)
    return df.rdd
```

Specificando l'opzione **inferSchema**, la session legge l'header e ritorna un DataFrame di oggetti Row, (ovvero un dizionario con l'header come chiave) da cui ricaviamo l'RDD. Grazie a questo caricamento

iniziale, abbiamo potuto operare sugli RDD in modo più elastico sfruttando l'indirizzabilità per attributo della struttura dati.

Testing e analisi delle prestazioni

Tutti i job sono stati eseguiti sui dataset originali (history: ~21 milioni di righe, legend: 6461 righe). Per poter testare il codice durante lo sviluppo, abbiamo creato diverse repliche in miniatura di *history*. I dataset di prova conservano la stessa struttura e contengono righe estratte in modo pseudo-casuale usando il comando Unix **shuf**.

Abbiamo quindi condotto l'analisi delle prestazioni su diversi input:

1) History originale ~21 milioni di righe;
2) History con 10 milioni di righe;
3) History con 5 milioni di righe.

In tutti e tre i casi abbiamo sempre utilizzato tutto il contenuto del dataset *legend*.

I test locali sono stati condotti su una macchina con otto core e 8 GB di memoria. I test in ambiente distribuito sono stati condotti sul Cluster di Roma Tre. Avevamo preparato anche un quarto dataset con 30 milioni di righe (ottenute duplicando e alterando alcune righe del dataset originale mediante un apposito job MapReduce, che avremmo utilizzato eventualmente sul cluster) ma per diversi motivi (*vedi Prestazioni sul cluster*) abbiamo desistito dal condurre anche questi test.

JOB 1

Generare, in ordine, le dieci azioni la cui quotazione (prezzo di chiusura) è cresciuta maggiormente dal 1998 al 2018, indicando, per ogni azione: (a) il simbolo, (b) l'incremento percentuale, (c) il prezzo minimo raggiunto, (e) quello massimo e (f) il volume medio giornaliero in quell'intervallo temporale.

L'input di questo job è la sola tabella history (storico giornaliero delle azioni), dalla quale estraiamo i seguenti campi: ***ticker, price_close, price_low, price_high, volume, date_created***. Siamo interessati solo alle entry nel periodo 1998-2018, pertanto le righe estratte vengono prefiltrate in base al valore del campo *date_created*.

Per ciascun ticker avremo N entry, che utilizziamo per calcolare le seguenti metriche:

- ***growth***: l'incremento percentuale dell'azione, calcolato mediante la formula:

$$\frac{\text{final\_price - initial\_price}}{\text{final\_price}} \cdot 100$$

  Dove *initial_price* corrisponde al prezzo di chiusura (*price_close*) dell'entry del ticker più vecchia (dal 1998 in poi), e *final_price* è il prezzo di chiusura dell'entry più recente. Se la lista delle entry di un ticker è ordinata per data, questi valori corrispondono rispettivamente al prezzo di chiusura del primo e dell'ultimo elemento della lista.

- ***min_price***: prezzo minimo raggiunto dell'azione, calcolato come minimo tra i valori del campo *price_low* (prezzo minimo giornaliero) e arrotondato alla quarta cifra decimale.
- ***max_price***: prezzo massimo raggiunto dell'azione, calcolato come massimo tra i valori del campo *price_high* (prezzo massimo giornaliero) e arrotondato alla quarta cifra decimale.
- ***avg_volume***: volume medio giornaliero, calcolato come media tra i valori del campo *volume* e arrotondato alla quarta cifra decimale.

L'output del job è una lista di 10 simboli, ordinati in modo decrescente per *growth*.

1.1 MapReduce

Per questo job abbiamo utilizzato una sola coppia di mapper/reducer.

Il mapper legge il file in input e mappa, per ciascuna riga, una tripla (**ticker**, **date_created**, **details**), dove details è una lista contenente i campi **price_close**, **price_low**, **price_high** e **volume**; inoltre filtra tutte le righe corrispondenti a entry che ricadono al di fuori dell'intervallo temporale scelto (1998-2018). Usiamo ticker e date_created come chiave doppia: in questo modo il sort ordina le righe per ticker e data.

Il reducer scorre le righe, che saranno raggruppate per ticker. Per ciascun ticker mantiene in memoria un oggetto **StockMetrics**, che aggiorna di volta in volta con i valori provvisori di min_price, max_price, initial_price e final_price; per il calcolo del volume medio, memorizza in una lista *volumes* ciascun volume giornaliero.

- Quando cambia il ticker, la StockMetrics viene finalizzata: growth viene calcolata come descritto sopra a partire da initial_price e final_price, mentre avg_volume viene calcolato facendo la media degli elementi di *volumes*.
- Mentre scorre le righe, il reducer costruisce dinamicamente la lista **TopStocks**, aggiornando la classifica ad ogni cambio di ticker e mantenendo in memoria solo i 10 ticker con la migliore growth. Alla fine del job, il reducer stampa il contenuto di TopStocks.

Pseudocodice: Mapper (Job 1)

```
map(csvFile):
        for row in csvFile:
                next() # Skip the header (first row)
                row = row.strip().split(',')

        if "1998-01-01" <= row.date <= "2018-12-31":
                print(row.ticker, row.date_created, row.details)
```

Pseudocodice: Reducer (Job 1)

```
reduce(mappedInput):
        # Initialize current ticker
        current_ticker = None

        # Initialize StockMetrics
        min_price = +inf
        max_price = -inf
        initial_price = None
        final_price = None
        volumes = []

        # Initialize TopStocks
        top_stocks = []

        # Main loop
        for row in mappedInput:
                ticker, date_created, details = splitBy(row, '\t')

        # Ticker has changed
        if current_ticker != ticker:
             if current_ticker != None:
                # Finalize StockMetrics
                growth = calculateGrowth(initial_price, final_price)
```

```
                min_price = round(min_price, 4)
                max_price = round(max_price, 4)
                avg_vol = round(mean(volumes),4)

                # Update TopStocks
                candidate = [ticker, growth, min_price, max_price, avg_volume]
                length = len(top_stocks)
                if length < 10:
                        top_stocks.append(candidate)
                else if length == 10 and growth > top_stocks[-1].growth:
                        top_stocks[-1] = candidate

                # Sort TopStocks
                top_stocks.sort(key=lambda x: x.growth, reverse=True)

                # Change current ticker
                current_ticker = ticker

                '''Re-initialize StockMetrics'''

        # Update StockMetrics
        min_price = details.min_price if details.min_price < min_price
        max_price = details.max_price if details.max_price > max_price
        last_closing_price = details.closing_price

        # Handle last line separately
        '''Finalize StockMetricks'''
        '''Update TopStocks'''

        # Print TopStocks
        for i, entry in enumerate(top_stocks):
        print(i+1, entry)
```

1.2 Hive

Per il primo Job abbiamo formulato una query a due livelli:

1. Selezioniamo tutte le colonne richieste da history per il calcolo delle metriche, filtrando per data. Mediante la funzione **over .. partition by**, siamo riusciti a prendere anche initial_price e final_price direttamente nel primo step;
2. Calcoliamo tutte le metriche come descritto sopra ordinando in base a growth, e mostriamo il risultato relativo ai migliori 10 ticker.

Hive (Job 1)

```
SET mapreduce.job.reduces = 1;

SELECT
    `ticker`,
    round(min(`price_low`),4) as `min_price`,
    round(max(`price_high`),4) as `max_price`,
    round(avg(`volume`),4) as `avg_volume`,
    round((`final_price` - `initial_price`) * 100 / `initial_price`) as `growth_percentage`
FROM (
    SELECT
        `ticker`, `price_close`, `price_low`, `price_high`, `volume`,
        first_value(`price_close`) over (partition by `ticker` order by `date_created`)
            as `initial_price`,
        first_value(`price_close`) over (partition by `ticker` order by `date_created` desc)
```

```
            as `final_price`
    FROM `history` h
    WHERE `date_created` between Date('1998-01-01') and Date('2018-12-31')) q1
GROUP BY `ticker`, `initial_price`, `final_price`
ORDER BY `growth_percentage` desc
LIMIT 10
```

1.3 Spark

Per tutti e tre i Job spark abbiamo scelto un approccio *divide-et-impera*, forti della possibilità di operare in-memory e di effettuare trasformazioni *lazy*; ogni volta che risulta necessario compiere diverse **action** sullo stesso RDD (composto da coppie chiave-valore) effettuamo in ordine i seguenti passi:

1. **Filter** iniziale delle righe al di fuori del periodo temporale interessato;
2. **Map**: split dell'RDD in diverse copie provvisorie lasciando inalterata la chiave e selezionando solo i valori interessati dalla action;
3. **Action** sui valori interessati;
4. **Join**: unione degli RDD provvisori sulla base della chiave.

Questa è la sequenza di operazioni effettuate nel primo Job:

- Filter del periodo 1998-2018;
- Map per estrarre la chiave **ticker** e i valori **price_close**, **price_low**, **price_high**, **volume**, **date_created**;
- Split dell'RDD nelle seguenti copie:
  - **minprice_rdd**: reduceByKey su price_low con la funzione min.
  - **maxprice_rdd**: reduceByKey su price_high con la funzione max.
  - **avgvol_rdd**: aggregateByKey **(*)** su volume, usando come accumulatore la coppia `(runningSum, runningCount)` per contare i record e simultaneamente sommare il volume, con una mapValues finale per calcolare il rapporto runningSum/runningCount.
  - **growth_rdd**: aggregateByKey sulle coppie (date_created, price_close). L'accumulatore usato è la seguente struttura: `((curr_initial_date, curr_initial_price),(curr_final_date,curr_final_price))` Mediante semplici confronti, ad ogni riga compariamo date_created con i valori dell'accumulatore ed eventualmente aggiorniamo i prezzi corrispondenti con il price_close della riga. Una volta finite le righe, con una mapValues calcoliamo growth con la solita formula a partire da curr_initial_price e curr_final_price.
- Join degli RDD di sopra due alla volta fino ad ottenere **metrics_rdd**, a cui applichiamo un mapValues per effettuare il *flatten* dei valori in una Row contenente **growth**, **min_price**, **max_price** e **avg_volume**.
- Ordinamento discendente di metrics_rdd con una sortBy(row.growth) per ottenere **ranked_rdd** ed estrazione dei primi 10 valori da quest'ultimo.

> **(*) *AggregateByKey, come funziona***
>
> *La funzione prende in input valori T e produce in output valori K. Per farlo, richiede tre parametri:*
>
> 1. *Un **accumulatore**, di tipo K.*
> 2. *Una funzione intra-partizione per aggiornare l'accumulatore K dato un valore T.*
> 3. *Una funzione inter-partizione per combinare due accumulatori K1 e K2.*

Pseudocodice Spark: metodi di supporto (tutti i Job)

```
'''Support methods for AggregateByKey for calculating growth
dp stands for date_price, aka (date_created, price), where price might be:
   - price_close (Job 1)
   - dailypricesum (Job 2 & 3)'''

# Compares two dates and returns the date-price pair associated to the date with the best value
# (default criterium is: earlier date wins, set initial=false to pick later date)
def compare_dateprices(dp_a, dp_b, initial=True):
    date_a, date_b = dp_a.date, dp_b.date
    if initial:
        best_dp = dp_a if date_a <= date_b else dp_b
    else:
        best_dp = dp_a if date_a > date_b else dp_b
    return best_dp

# Updates one growth accumulator
def update_growth_acc(acc, row_dp):
    initial_dp, final_dp = acc[0], acc[1]
    new_initial_dp = compare_dateprices(initial_dp, row_dp)
    new_final_dp = compare_dateprices(initial_dp, row_dp, initial=False)
    return (new_initial_dp, new_final_dp)

# Combines two growth accumulators into one with the lowest initial date and highest final date
def combine_growth_accs(acc_a, acc_b):
    initial_dp_a, final_dp_a = acc_a[0], acc_a[1]
    initial_dp_a, final_dp_b = acc_b[0], acc_b[1]
    combined_initial_dp = compare_dateprices(initial_dp_a, initial_dp_a)
    combined_final_dp = compare_dateprices(final_dp_a, final_dp_b, initial=False)
    return (combined_initial_dp, combined_final_dp)

# Calculates the growth given an accumulator
def calculate_growth(acc):
    initial_price, final_price = acc.initial_price, acc.final_price
    growth = math.floor((final_price - initial_price) * 100 / initial_price)
    return growth
```

Pseudocodice Spark: Job 1

```
def filter_period(row):
        return "1998-01-01" <= row.date <= "2018-12-31"

def select_columns(row):
        return (row.ticker,
                Row(price_close=row.close,
                price_low=row.low,
                price_high=row.high,
                volume=row.volume,
                date_created=row.date))

def run_job(rdd):
        # Prefilter and field extraction
        rdd = rdd.filter(filter_period) \
        .map(select_columns)

        ''' Divide et impera '''

        # Calculate min_price
        minprice_rdd = rdd.map(lambda row: (row[0], row[1].price_low)) \
        .reduceByKey(min) \
```

```
        .mapValues(lambda x: round(x, 4)) \
        .cache()

        # Calculate max_price
        maxprice_rdd = rdd.map(lambda row: (row[0], row[1].price_high)) \
        .reduceByKey(max) \
        .mapValues(lambda x: round(x, 4)) \
        .cache()

        # Calculate avg_volume
        # Initialize accumulator (runningSum, runningCount)
        avgvol_acc = (0, 0)
        avgvol_rdd = rdd.map(lambda row: (row[0], row[1].volume)) \
        .aggregateByKey(avgvol_acc,
                        lambda acc, x: (acc[0] + x, acc[1] + 1),
                        lambda acc_a, acc_b: (acc_a[0] + acc_b[0], acc_a[1] + acc_b[1])) \
        .mapValues(lambda acc: round(acc[0]/acc[1], 4)) \
        .cache()

        # Calculate growth
        # Initialize accumulator ((initial_date, initial_price),(final_date, final_price))
        growth_acc_initial = (MAX_DATE, 0)
        growth_acc_final = (MIN_DATE, 0)
        growth_rdd = rdd.map(lambda row: (row[0], (row[1].date_created, row[1].price_close))) \
        .aggregateByKey((growth_acc_initial, growth_acc_final),
                        utils.update_growth_acc,
                        utils.combine_growth_accs) \
        .mapValues(utils.calculate_growth) \
        .cache()

        ''' Join '''

        min_max_rdd = minprice_rdd.join(maxprice_rdd)
        min_max_avgvol_rdd = min_max_rdd.join(avgvol_rdd) \
        .mapValues(lambda value: (value[0][0], value[0][1], value[1]))  # First flatten
        metrics_rdd = min_max_avgvol_rdd.join(growth_rdd) \
        .mapValues(lambda value: # Final flatten
                Row(growth=value[1],
                min_price=value[0][0],
                max_price=value[0][1],
                avg_volume=value[0][2],
                )) \
        .cache()

        # Create ranking of tickers by growth
        ranked_rdd = metrics_rdd.sortBy(lambda row: row[1].growth, ascending=False) \
        .mapValues(pretty_print) \
        .take(10)
```

1.4 Output

Tutte le implementazioni hanno restituito il seguente output sul dataset originale:

```
[user33@node1 job2$]cat output/part-00000
1    SAB    ['+2629529%', 1.3655, 319600.0, 1608166.8736]
2    PJT    ['+296300%', 0.01, 11102.5, 71484.4493]
3    EAF    ['+267757%', 0.002, 24.364, 1245139.0385]
4    UVE    ['+226900%', 0.02, 45.9, 224226.4429]
5    ORGS   ['+217415%', 0.0031, 19.92, 8570.9623]
6    PUB    ['+179900%', 0.009, 138.0, 34449.4007]
7    MNST   ['+163340%', 0.0306, 70.22, 7347898.8208]
8    RMP    ['+121081%', 0.03, 79.8187, 120487.0477]
9    CCD    ['+111250%', 0.015, 25.98, 105416.8927]
10    KE    ['+99400%', 0.015, 22.45, 73249.8462]
```

Legenda: (rank, ticker, growth, min_price, max_price, avg_volume)

JOB 2

Generare, per ciascun settore, il relativo "trend" nel periodo 2004-2018, ovvero un elenco contenente, per ciascun anno nell'intervallo: (a) il volume complessivo del settore, (b) la percentuale di variazione annuale (differenza percentuale arrotondata tra la quotazione di fine anno e quella di inizio anno) e (c) la quotazione giornaliera media. N.B.: volume e quotazione di un settore si ottengono sommando i relativi valori di tutte le azioni del settore.

Questo job richiede l'impiego di entrambe le tabelle: da legend estraiamo i campi ***ticker*** e ***sector***, mentre da history selezioniamo ***ticker***, ***price_close***, ***volume*** e ***date_created***. Anche qui filtriamo le righe estratte da history in base alla data, stavolta selezionando il periodo 2004-2018.

Successivamente, uniamo i due dataset base al valore comune (*ticker*); qualora ad una riga in history non corrisponda una riga in legend, assegniamo il valore "N/A" come settore per quella riga.

Per ciascun settore e anno avremo N entry, da cui estraiamo le seguenti metriche:

- ***tot_volume***: volume annuale del settore, calcolato sommando i valori del campo *volume* di tutti i ticker del settore per quell'anno.
- ***growth***: incremento percentuale annuale, calcolato con la medesima formula del job precedente selezionando come prezzo iniziale, stavolta, il prezzo di chiusura del primo giorno dell'anno, e come prezzo finale il prezzo dell'ultimo giorno dell'anno. Dato che per ciascun giorno possono essere state registrate diverse azioni di quel settore, il prezzo di chiusura giornaliero è calcolato sommando tutti i prezzi di chiusura delle azioni del settore (***daily_price_sum***).
- ***avg_daily_price***: prezzo totale medio giornaliero, calcolato come media tra i valori di prezzo aggregati, calcolati per ciascun giorno dell'anno a partire dai *price_close* delle azioni del settore, e arrotondato alla quarta cifra decimale.

L'output del job è una lista di tuple **(*sector*, *year*, *tot_volume*, *growth*, *avg_daily_price*)**.

2.1 MapReduce

Per questo job abbiamo utilizzato due coppie di mapper-reducer; la prima serve ad effettuare il join tra i due dataset, mentre la seconda calcola le metriche annuali per ciascun settore.

Il primo mapper riceve in input il contenuto, concatenato, di entrambi i file csv, e mappa ciascuna riga in una tripla (**ticker**, **flag**, **value**), anche qui filtrando tutti i record fuori dall'intervallo temporale 2004-2018. Il campo flag è necessario a distinguere se la riga proviene dal primo o dal secondo dataset; in particolare flag vale 0 per righe di legend, e 1 per righe di history. Il campo value contiene **sector** per le righe provenienti da legend, mentre per le righe provenienti da history contiene una lista con i valori [**price_close**, **volume**, **date_created**]. Anche in questo caso usiamo una chiave doppia per far ordinare le righe per ticker e flag.

Dato che ad un ticker in legend corrispondono diverse righe in history, la prima riga letta dal reducer, per ciascun ticker, è quella proveniente da legend che contiene il settore del ticker. Il reducer realizza il join effettivo valorizzando tutte le righe successive per quel ticker con il valore di *sector* letto dalla prima. Dopodichè, emette in output la tripla (**sector**, **date_created**, **details**), dove details contiene la lista [*price_close*, *volume*].

Il secondo mapper effettua una semplice copia dell'output proveniente dal passo precedente, che stavolta verrà ordinato per settore e per data.

Il secondo reducer scorre le righe raggruppate per settore e ordinate per anno e - analogamente al primo job - mantiene in memoria un oggetto **YearMetrics**, dove memorizza i valori provvisori per il calcolo delle metriche di ciascun anno. In particolare somma il prezzo di chiusura di tutte le azioni dello stesso giorno in un accumulatore, che salva poi in una lista *daily_prices_sums*; quando cambia l'anno, YearMetrics viene finalizzata: **growth** viene calcolato come al solito, prendendo come *initial_price* e *final_price* il primo e l'ultimo elemento di *daily_prices_sums*, **avg_daily_price** come media di tutti gli elementi e **tot_volume** come somma dei volumi di tutte le azioni dell'anno. Il reducer stampa questi valori in output.

Pseudocodice: Mapper1 (Job 2)

```
map_join(csvFiles):
        for row in csvFiles:
        row = row.strip().split(',')

        # Skip headers
        if "ticker" in row.ticker:
                continue

        # Rows from legend have 5 fields, rows from history have 8
        if len(row) == 8 and (row.date < "2004-01-01" or row.date > "2018-12-31"):
                continue

        if len(row) == 5 :
                print(row.ticker, 0, row.sector)
        if len(row == 8):
                print(row.ticker, 1, [row.price_close, row.volume, row.date])
```

Pseudocodice: Reducer1 (Job 2)

```
reduce_join(mappedInput):
        # Initialize current ticker
        current_ticker = None

        for row in mappedInput:
                ticker, flag, value = row.split('\t')

                # Ticker has changed
                if current_ticker != ticker:
                        current_ticker = ticker
                        sector = value if flag==0 else "N/A" # Handle null values

                # Extract values and send output
                if flag==1:
                        print(sector, row.date_created, [row.price_close, row.volume])

        # Handle last line separately
        print(sector, row.date_created, [row.price_close, row.volume])
```

Pseudocodice: Mapper2 (Job 2)

```
map_copy(reducedInput):
        for row in reducedInput:
        print(row.strip())
```

Pseudocodice: Reducer2 (Job 2)

```
reduce(mappedInput):
        # Initialize current sector, year and date
        curr_sector = None
        curr_year = None
        curr_date = None

        # Initialize YearMetrics
        daily_prices_sums = []
        daily_price = 0
        tot_volume = 0

        # Main loop
        for row in mappedInput:
        sector, date_created, details = row.split('\t')
        year = date_created.year

           # Day has changed
           if curr_date != date_created:
                if curr_date:
                # Update YearMetrics at end of day
                daily_prices_sums.append(daily_price)
                daily_price = 0

                # Change current day
                curr_date = date_created

                # Year has changed
                if curr_year != year:
                    if curr_year:
                        # Finalize YearMetrics
                        initial_price = first(daily_prices_sums)
                        final_price = last(daily_prices_sums)
                        growth = calculateGrowth(initial_price, final_price)
                        avg_daily_price = round(mean(daily_prices_sums), 4)

                        # Print sector, year and YearMetrics
                        print(curr_sector, curr_year, [tot_volume, growth, avg_daily_price])

                # Change current year
                curr_year = year

                '''Re-initialize YearMetrics'''

                # Sector has changed
                if curr_sector != sector:
                       # Change current sector
                       curr_sector = sector

        # Update YearMetrics for each entry in day
        daily_price += details.price
        tot_volume += details.volume

        # Handle last line separately
        '''Finalize YearMetrics'''
        '''Print sector, year and YearMetrics'''
```

2.2 Hive

Per il secondo Job abbiamo formulato una query a quattro livelli:

1. Effettuiamo il join tra le tabelle filtrando per data ed estraendo settore, anno, data, prezzo e volume;
2. Calcoliamo le somme giornaliere dei prezzi per ciacun settore, anno e giornata;
3. Selezioniamo initial_price e final_price dalle somme giornaliere per ciascun settore e anno;
4. Calcoliamo growth e avg_daily_price per ciascun settore e anno.

Hive (Job 2)

```
SET mapreduce.job.reduces = 1;

SELECT
 `sector`, `year`,
 sum(`volume`) as `tot_volume`,
 round(avg(`daily_price_sum`),4) as `avg_daily_price`,
 round((`final_price` - `initial_price`) * 100 / `initial_price`) as `growth_percentage`
FROM (
 SELECT
   `sector`, `year`, `date_created`, `daily_price_sum`, `volume`,
   first_value(`daily_price_sum`) over (partition by `sector`, `year` order by `date_created`)
           as `initial_price`,
   first_value(`daily_price_sum`) over (partition by `sector`, `year` order by `date_created`
desc)
           as `final_price`
 FROM (
   SELECT
     `sector`, `year`, `date_created`, `volume`,
     sum(`price_close`) over (partition by `sector`, `year`, `date_created` order by
`date_created`)
           as `daily_price_sum`
   FROM (
     SELECT
       l.`sector`,
       year(h.`date_created`) as `year`,
       h.`date_created`, h.`price_close`, h.`volume`
     FROM `legend` l JOIN `history` h on l.`ticker`=h.`ticker`
     WHERE h.`date_created` between Date('2004-01-01') and Date('2018-12-31')) q1
   ) q2
) q3
GROUP BY `sector`, `year`, `initial_price`, `final_price`
ORDER BY `sector`, `year`
```

2.3 Spark

Questa è la sequenza di operazioni effettuate nel secondo Job:

- Filter del periodo 2004-2018 da **history_rdd** e map per estrarre la chiave **ticker** e i valori **price_close**, **volume** e **date_created**;
- Map di **legend_rdd** per estrarre la chiave **ticker** e il valore **sector**;
- Join dei due RDD in **sectoryear_rdd**: map per estrarre la chiave **(sector, year)**.
- Split di **sectoryear_rdd** in due copie:
    - **totvolume_rdd**: reduceByKey su volume con la funzione sum.
    - **dailypricessum_rdd**: map per estrarre la chiave **(sector, year, date_created)** e il valore price_close; reduceByKey su price_close con la funzione sum per ottenere dailypricesum. Ulteriore split di questo RDD in due copie:
        - **avgdailyprice_rdd**: aggregateByKey su dailypricesum, rimuovendo date_created dalla chiave e usando la coppia (runningSum, runningCount) con le stesse modalità del calcolo di avg_volume per il job precedente, ma con una chiave differente ((sector,year) invece di ticker).
        - **growth_rdd**: spostamento di date_created dalla chiave al valore e aggregateByKey sulle coppie (date_created, price_close), con le stesse modalità del calcolo di growth per il job precedente, ma con una chiave differente ((sector,year) invece di ticker).
    - Join dei tre RDD (totvolume_rdd, avgdailyprice_rdd, growth_rdd), che avranno tutti la stessa chiave (sector,year), anche qui unendoli due alla volta fino ad ottenere **metrics_rdd**, e successiva *flatten* dei valori in una Row contenente growth, tot_volume e avg_daily_price.
    - Ordinamento dei risultati per chiave, in modo da raggruppare per settore e ordinare per anno, dal più vecchio al più recente.

Pseudocodice Spark: Job 2

```
def filter_period(row):
        return "2004-01-01" <= row.date <= "2018-12-31"

def select_columns_history(row):
        return (row.ticker,
                Row(price_close=row.close,
                volume=row.volume,
                date_created=row.date))

def select_columns_legend(row):
        return (row.ticker,
                Row(sector=row.sector))

def run_job(history_rdd, legend_rddw):
        # Prefilter and field extraction
        rdd = rdd.filter(filter_period) \
        .map(select_columns)

def run_job(history_rdd, legend_rdd):
        # Prefilter and field extraction
        history_rdd = history_rdd.filter(filter_period) \
        .map(select_columns_history)
        legend_rdd = legend_rdd.map(select_columns_legend)
```

```
        ''' Join '''

        sectoryear_rdd = legend_rdd.join(history_rdd) \
        .map(lambda row: ((row[1][0].sector, row[1][1].date_created.year), row[1][1]))

        ''' Divide et impera '''

        # Calculate tot_volume
        totvolume_rdd = sectoryear_rdd.map(lambda row: (row[0], row[1].volume)) \
        .reduceByKey(lambda x, y: x + y) \
        .cache()

        # Calculate dailypricesum (for splitting again)
        dailypricessum_rdd = sectoryear_rdd \
        .map(lambda row: ((row[0][0], row[0][1], row[1].date_created), row[1].price_close)) \
        .reduceByKey(lambda x, y: x + y)

        ''' Divide et impera '''

        # Calculate avg_daily_price
        # initialize accumulator (runningSum, runningCount)
        avgdailyprice_acc = (0, 0)
        avgdailyprice_rdd = dailypricessum_rdd.map(lambda row: ((row[0][0], row[0][1]), row[1])) \
        .aggregateByKey(avgdailyprice_acc,
                        lambda acc, x: (acc[0] + x, acc[1] + 1),
                        lambda acc_a, acc_b: (acc_a[0] + acc_b[0], acc_a[1] + acc_b[1])) \
        .mapValues(lambda acc: round(acc[0]/acc[1], 4)) \
        .cache()

        # Calculate growth
        # initialize accumulator ((initial_date, initial_price),(final_date, final_price))
        growth_acc_initial = (MAX_DATE, 0)
        growth_acc_final = (MIN_DATE, 0)
        growth_rdd = dailypricessum_rdd.map(lambda row: ((row[0][0], row[0][1]), (row[0][2],
row[1]))) \
        .aggregateByKey((growth_acc_initial, growth_acc_final),
                        utils.update_growth_acc,
                        utils.combine_growth_accs) \
        .mapValues(utils.calculate_growth) \
        .cache()


        ''' Join '''

        avgdailyprice_growth_rdd = avgdailyprice_rdd.join(growth_rdd)
        metrics_rdd = avgdailyprice_growth_rdd.join(totvolume_rdd) \
        .mapValues(lambda value: Row(growth=value[0][1],     # Flatten
                                     avg_daily_price=value[0][0],
                                     tot_volume=value[1],
                                     )) \
        .mapValues(pretty_print) \
        .sortBy(lambda row: row[0]) \
        .collect()
```

2.4 Output

Tutte le implementazioni hanno restituito il seguente output sul dataset originale:
(non tutte le righe sono mostrate)

```
[user33@node1 job2]$ cat output/part-00000
BASIC INDUSTRIES       2004    [30767395827, '+26.0%', 2865.8737]
BASIC INDUSTRIES       2005    [37457588379, '+47.0%', 4367.6822]
BASIC INDUSTRIES       2006    [50413342778, '+55.0%', 7110.0708]
BASIC INDUSTRIES       2007    [67640775192, '+23.0%', 9211.1339]
BASIC INDUSTRIES       2008    [104336790359, '-58.0%', 7124.094]
BASIC INDUSTRIES       2009    [113161759706, '+36.0%', 4727.9192]
BASIC INDUSTRIES       2010    [96267427694, '+28.0%', 6126.0523]
BASIC INDUSTRIES       2011    [93277620675, '-25.0%', 8535.5519]
BASIC INDUSTRIES       2012    [79648935208, '+2.0%', 8694.819]
BASIC INDUSTRIES       2013    [81167036326, '+201.0%', 28486.425]
...
CAPITAL GOODS   2015   [72871030811, '-1.0%', 13598.0295]
CAPITAL GOODS   2016   [74995759977, '+19.0%', 13917.4226]
CAPITAL GOODS   2017   [67007410673, '+36.0%', 18508.0178]
CAPITAL GOODS   2018   [48146582330, '-2.0%', 21173.2494]
CONSUMER DURABLES   2004      [10057518399, '+21.0%', 1985.892]
CONSUMER DURABLES   2005      [10096843252, '-4.0%', 2111.8707]
...
N/A 2007       [158543844310, '+12.0%', 10651.1413]
N/A 2008       [152539834436, '-39.0%', 9147.4345]
N/A 2009       [179740098152, '+30.0%', 7831.8214]
N/A 2010       [257543462937, '+43.0%', 13890.6588]
N/A 2011       [315627354130, '+1.0%', 14267.345]
N/A 2012       [358328693664, '+15.0%', 14757.5146]
N/A 2013       [434889087533, '+10.0%', 16744.9203]
N/A 2014       [443615878897, '+4.0%', 18512.0124]
N/A 2015       [46770321800, '+0.0%', 18620.2086]
N/A 2016       [46511843051, '+1096.0%', 29556.0664]
N/A 2017       [40427789892, '-90.0%', 128263.749]
...
TRANSPORTATION  2017   [24750549271, '-88.0%', 8340.1206]
TRANSPORTATION  2018   [15782363800, '+2.0%', 4245.2039]
```

Legenda: (settore, anno, tot_volume, growth, avg_daily_price)

Osservazione: in tutto sono presenti 13 settori distinti su legend; in N/A abbiamo incluso i ticker di aziende per cui era presente uno storico (in history) ma non una entry in legend. Considerando il periodo 2004-2018 (14 anni), l'output conta intorno alle 200 righe prendendo come input il dataset originale per history.

JOB 3

Generare, coppie di aziende di settori diversi le cui azioni, negli ultimi 3 anni, hanno avuto lo stesso trend in termini di variazione annuale, indicando le aziende e il trend comune (es Apple, Fiat, 2016:-1%, 2017:+3%, 2018: +5%)

Anche questo job richiede l'impiego di entrambe le tabelle. Stavolta estraiamo da legend i campi ***ticker***, ***name*** e ***sector***, mentre da history selezioniamo ***ticker***, ***price_close*** e ***date_created***, anche qui filtrando le righe in base alla data, considerando solo il periodo 2016-2018. Analogamente al secondo Job, effettuiamo un join delle tabelle sulla base del ticker. Qualora ad una riga in history non corrisponda una riga in legend, assegniamo il valore "N/A" sia a *name* che *sector* per quella riga.

L'unica metrica da calcolare è ***growth***, stavolta in riferimento alla crescita annuale della compagnia (identificata da *name*) e non del settore; la differenza sta quindi nella scelta della chiave, mentre le modalità di calcolo dei valori restano identiche al Job precedente. Calcolata la growth annuale, costruiamo il ***trend*** di un'azienda unendo le crescite di ciascun anno (a cui affianchiamo l'anno di provenienza, estratto dal campo *date_created*.

Per ciascun trend avremo N compagnie che hanno avuto quel trend negli ultimi tre anni. All'interno di ciascun trend, generiamo tutte le **combinazioni** di coppie di aziende (descritte da nome e settore), escludendo quelle afferenti allo stesso settore e, naturalmente, le coppie di duplicati.

L'output del job è una lista di tuple **(*trend*, *company1:sector1*, *company2:sector2*)**

3.1 MapReduce

Questo job ha richiesto l'impiego di tre coppie di mapper-reducer.

La prima coppia effettua il join dei due dataset in modo del tutto analogo a quanto descritto per il Job precedente. Le uniche differenze sono i valori estratti (qui prendiamo anche **name** da legend e prendiamo solamente **price_close** da history, oltre a *date_created*), l'intervallo di tempo scelto (2016-2018) e l'output del reducer (una quadrupla **name**, **date_created**, **price_close**, **sector**).

La seconda coppia viene impiegata per il calcolo di **growth**, sfruttando stavolta il nome dell'azienda come chiave e ordinando per date_created. In questo passo, il mapper effettua una semplice copia e il reducer calcola la **YearMetrics** annuale dell'azienda, contenente appunto solo la growth. L'output di questo step è una tripla (**name**, **sector**, **year_growth**), dove *year_growth* è la concatenazione tra *year* e *growth* (con year estratto da *date_created*), ad esempio "2017:+10%".

Il terzo mapper effettua un raggruppamento per chiave dei valori year_growth: qui viene costruito il trend come descritto sopra, e viene prodotta in output la tripla (**trend**, **name**, **sector**). *Trend* è posto come chiave nel sort successivo, in modo da ottenere una lista dove tutte le aziende con lo stesso trend appaiono raggruppate.

Il terzo e ultimo reducer scorre le righe e raccoglie in una lista **SimilarTrendingCompanies** tutte le company che incontra. Quando cambia il trend, il reducer genera tutte le combinazioni di due elementi dal contenuto della lista (escludendo duplicati o ripetizioni) e filtra da questa tutte le coppie di company i cui settori corrispondono. Il reducer stampa in output tutte le coppie, svuota la lista e prosegue fino ad esaurire i trend.

Pseudocodice: Mapper1 (Job 3)

```
map_join(csvFiles):
    for row in csvFiles:
        row = row.strip().split(',')

        # Skip headers
        if "ticker" in row.ticker:
                continue

        # Rows from legend have 5 fields, rows from history have 8
        if len(row) == 8 and (row.date < "2004-01-01" or row.date > "2018-12-31"):
                continue

        if len(row) == 5 :
                print(row.ticker, 0, row.sector)
        if len(row == 8):
                print(row.ticker, 1, [row.price_close, row.date])
```

Pseudocodice: Reducer1 (Job 3)

```
reduce_join(mappedInput):
        # Initialize current ticker
        current_ticker = None

        for row in mappedInput:
            ticker, flag, value = row.split('\t')

                # Ticker has changed
                if current_ticker != ticker:
                        current_ticker = ticker
                        company_info = value if flag==0 else "['N/A', 'N/A']" # Handle null values

                # Extract values and send output
                if flag==1:
                        print(company_info.name, value.date_created,
                                value.price_close, company_info.sector)

        # Handle last line separately
        print(company_info.name, value.date_created,value.price_close, company_info.sector)
```

Pseudocodice: Mapper2 (Job 3)

```
map_copy(reducedInput):
   for row in reducedInput:
        print(row.strip())
```

Pseudocodice: Reducer2 (Job 3)

```
reduce_growth(mappedInput):
        # Initialize current company, sector, year and day (identified by date)
        curr_name = None
        curr_sector = None
        curr_year = None
        curr_date = None
        # Initialize YearMetrics (for calculating growth)
        daily_price_sum = 0
        initial_price = None
        final_price = None
        # Main loop
```

```
        for line in mappedInput:
            company, date_created, price_close, sector = line.strip().split('\t')
            year = getYear(date_created)

            # Day has changed
            if curr_date != date_created:
                 if curr_date:
                    # Update YearMetrics at end of day
                    if initial_price == None:
                        # Memorize the first price of the year (only once)
                        initial_price = daily_price_sum
                    # Save the current price as final price (every day)
                    final_price = daily_price_sum
                    daily_price_sum = 0

                    # Change current day
                    curr_date = date_created

                #Year has changed
                if curr_year != year:
                    if curr_year:
                        # Finalize YearMetrics (calculate only growth)
                        growth = calculateGrowth(final_price, initial_price)

                        # Print year:growth
                        year_growth = curr_year + ":" + growth
                        print(curr_name, curr_sector, year_growth)

                    # Change current year
                    curr_year = year

                    '''Re-initialize YearMetrics'''

                # Company name has changed
                if curr_name != company:
                    curr_name = company

                # Sector has changed
                if curr_sector != sector:
                    curr_sector = sector

                # Update YearMetrics for each entry in day
                daily_price_sum += price_close

        # Handle last year separately
        '''Finalize YearMetrics'''
        '''Print year:growth'''
```

Pseudocodice: Mapper3(Job 3)

```
map_trends(reducedInput):
        # Initialize current Company (name, sector, trend)
        curr_name = None
        curr_sector = None
        curr_trend = []

        for row in reducedInput:
                # Get next Company
                next_name, next_sector = row.name, row.sector
                next_trend = []
```

```
            # Company hasn't changed
            if curr_name == next_name and curr_sector == next_sector:
                    # Build the trend
                    curr_trend.append(row.year_growth)
            # Company has changed
            else:
                     if curr_name:
                    # Print trend, name and sector
                    print(curr_trend, curr_name, curr_sector)

            # Change current company and start building its trend
            next_trend.append(row.year_growth)
            curr_name = next_name
            curr_sector = next_sector
            curr_trend = next_trend

    # Handle last line separately
    print(trend, name, sector)
```

Pseudocodice: Reducer3 (Job 3)

```
reduce_combinations(mappedInput):
    # Initialize current SimilarTrendingCompanies
    curr_trend = None
    curr_trend_companies = []

    for line in input_file:
        trend, name, sector = line.strip().split('\t')

        # Trend has changed
        if curr_trend != trend:
                if curr_trend:
                        # Generate all couples of two companies from SimilarTrendingCompanies
                        for couple in combinations(curr_trend_companies,r=2):
                             # Pick only couples from different sectors
                             if not couple[0].sector == couple[1].sector:
                                 # Print common trend and couple names
                                print(curr_trend, couple[0].name, couple[1].name)
                # Change current trend and Re-initialize SimilarTrendingCompanies
                curr_trend = trend
                curr_trend_companies = []
        # Update SimilarTrendingCompanies
        curr_trend_companies.append(name)

    # Handle last trend separately
    '''Generate all all couples of two companies from SimilarTrendingCompanies'''
    '''Print only couples from different sectors'''
```

3.2 Hive

Abbiamo suddiviso il terzo Job in due query. Nella prima calcoliamo il trend annuale per ciascuna compagnia, mentre nella seconda estraiamo le coppie di compagnie di settori diversi col medesimo trend. Il risultato intermedio viene memorizzato in una tabella **company_trends**, con chiave il trend rappresentato da una lista di stringhe year:growth.

La growth viene calcolata in modo analogo al Job 2, stavolta per ogni azienda e ogni anno. I due step aggiuntivi vengono impiegati per formare la stringa year:growth e per raggrupparle utilizzando come chiave il nome dell'azienda.

Hive (Job 3, parte 1)

```
SET mapreduce.job.reduces = 1;
DROP TABLE IF EXISTS `company_trends`;
CREATE TABLE `company_trends` (
 `trend` array<STRING>,
 `name` STRING,
 `sector` STRING);

INSERT OVERWRITE TABLE `company_trends`
SELECT
 collect_set(q5.`year_trend`) as `trend`,
 q5.`name`, q5.`sector`
FROM (
 SELECT
   q4.`name`, q4.`sector`,
   concat_ws(':',cast(`year` as STRING), cast(`growth_percentage` as STRING))
         as `year_trend`
 FROM (
   SELECT
     q3.`name`, q3.`sector`, q3.`year`,
     round((q3.`final_price` - q3.`initial_price`) * 100 / q3.`initial_price`)
          as `growth_percentage`
   FROM (
     SELECT
       q2.`name`, q2.`sector`, q2.`year`, q2.`date_created`,
       first_value(`daily_price_sum`) over (partition by `name`, `year`
                                            order by `date_created`)
          as `initial_price`,
       first_value(`daily_price_sum`) over (partition by `name`, `year`
                                            order by `date_created` desc)
          as `final_price`
     FROM (
       SELECT
         q1.`name`, q1.`sector`, q1.`year`, q1.`date_created`,
         sum(`price_close`) over (partition by `name`, `year`, `date_created`
                                  order by `date_created`)
              as `daily_price_sum`
       FROM (
         SELECT
           l.`name`, l.`sector`,
           year(h.`date_created`) as `year`,
           h.`date_created`, h.`price_close`
         FROM `legend` l JOIN `history` h on l.`ticker`=h.`ticker`
         WHERE h.`date_created` between Date('2016-01-01') and Date('2018-12-31')
         ) q1
       ) q2
```

```
    ) q3
    GROUP BY `name`, `sector`, `year`, `initial_price`, `final_price`
  ) q4
 ) q5
GROUP BY `name`, `sector`
ORDER BY `trend`;
```

Nella query successiva abbiamo eseguito un self join di company_trends con trend come chiave; per fare ciò, abbiamo impostato la proprietà **hive.mapred.mode="nonstrict"** ed usato il LEFT JOIN. Dalla tabella risultante abbiamo filtrato le coppie di aziende con settori diversi e le coppie di duplicati.

A differenza di mapreduce, qui vengono generate le coppie simmetriche (A,B) e (B,A). Per rimuovere questi duplicati, abbiamo collassato le colonne relative a nomi e settori delle aziende in un'unico campo similar_trending_companies, costruito come array ordinato di due elementi. In questo modo, abbiamo potuto filtrare i duplicati con una SELECT DISTINCT. Dopodichè, abbiamo esploso di nuovo l'array in due colonne separate.

Hive (Job 3, parte 2)

```
SET mapreduce.job.reduces = 1;
SET hive.mapred.mode = "nonstrict";

SELECT
 `trend`,
 `similar_trending_companies`[0] as `company_A`,
 `similar_trending_companies`[1] as `company_B`
FROM (
 SELECT DISTINCT *
 FROM (
   SELECT
     concat_ws(', ',`trend`) as `trend`,
     sort_array(array(company_A, company_B)) as `similar_trending_companies`
   FROM (
     SELECT
       `trend`,
       concat_ws(' : ', `name_A`,`sector_A`) as `company_A`,
       concat_ws(' : ', `name_B`,`sector_B`) as `company_B`
     FROM (
       SELECT
         ct1.trend as `trend`,
         ct1.name as `name_A`,
         ct1.sector as `sector_A`,
         ct2.name as `name_B`,
         ct2.sector as `sector_B`
       FROM `company_trends` ct1 LEFT JOIN `company_trends` ct2
       ON concat_ws(',', ct1.trend)=concat_ws(',', ct2.trend)
       WHERE ct1.name <> ct2.name
       AND ct1.sector <> ct2.sector
       ORDER BY `trend`
       ) q1
     ) q2
   ) q3
 ORDER BY `trend`, `similar_trending_companies`
 ) q4
```

3.3 Spark

Queste sono le operazioni (sequenziali) effettuate nel terzo Job:

- Filter del periodo 2016-2018 da **history_rdd** e map per estrarre la chiave **ticker** e i valori **price_close** e **date_created**;
- Map di **legend_rdd** per estrarre la chiave **ticker** e i valori **name** e **sector**;
- Join dei due RDD: map per estrarre la chiave **(name, sector, year, date_created)** e il valore **price_close**.
- Calcolo di **dailypricesum** con una reduceByKey su price_close con funzione sum, in modo analogo al Job 2 ma stavolta sommando i prezzi giornalieri per azienda e non per settore.
- Calcolo di **growth** con una map per spostare date_created dalla chiave al valore (ottenendo il valore (date_created, dailypricesum)), e succesivamente aggregateByKey su tale valore con le stesse modalità del calcolo di growth per i job precedenti, ma con chiave differente (name, sector, year).
- Calcolo del **trend**: dopo aver ordinato le righe per growth, con una map spostiamo year dalla chiave al valore concatenando year e growth in un'unica stringa "year:growth"; dopodichè effettuiamo una groupByKey su growth, con la chiave che a questo punto conterrà solo (name, sector), ottenendo quindi il trend dell'azienda negli ultimi tre anni.
- Calcolo delle coppie **similartrendingcompanies**: per prima cosa effettuiamo il join dell'RDD precedente con sé stesso; a differenza di Hive, Spark produce combinazioni esatte (senza duplicati (A,B), (B,A)), pertanto resta solamente da filtrare le coppie di aziende con lo stesso nome (la stessa azienda) O con nome diverso ma dello stesso settore.

Pseudocodice Spark: Job 3

```
def filter_period(row):
        return "2016-01-01" <= row.date <= "2018-12-31"

def select_columns_history(row):
        return (row.ticker,
                Row(price_close=row.close,
                date_created=row.date))

def select_columns_legend(row):
        return (row.ticker,
                Row(name=row.name,
                sector=row.sector))

def run_job(history_rdd, legend_rdd):
        # Prefilter and field extraction
        history_rdd = history_rdd.filter(filter_period) \
        .map(select_columns_history)
         legend_rdd = legend_rdd.map(select_columns_legend)

        ''' Join '''
        # Here we operate sequentially on the same rdd.
        # We calculate dailypricesum first, then calculate growth
        # Initialize accumulator((initial_date, initial_price),(final_date, final_price))
        growth_acc_initial = (MAX_DATE, 0)
        growth_acc_final = (MIN_DATE, 0)
        yeargrowth_rdd = legend_rdd.join(history_rdd) \
        .map(lambda row: ((row[1][0].name, row[1][0].sector, row[1][1].date_created.year,
row[1][1].date_created), row[1][1].price_close)) \
        .reduceByKey(lambda x, y: x + y) \
```

```
        .map(lambda row: ((row[0][0], row[0][1], row[0][2]), (row[0][3], row[1]))) \
        .aggregateByKey((growth_acc_initial, growth_acc_final),
                        utils.update_growth_acc,
                        utils.combine_growth_accs) \
        .mapValues(utils.calculate_growth) \
        .sortBy(lambda row: row[0]) \
        .cache()

        # Calculate trend by mapping year:growth couples and then grouping by key
        trend_rdd = yeargrowth_rdd \
        .map(lambda row: ((row[0][0], row[0][1]), str(row[0][2]) + ":" + row[1])) \
        .groupByKey() \
        .map(lambda row: (row[1], (row[0]))) \
        .cache()

        ''' Combinations '''

        # Returns true when company_a and company_b are equal, or when their sectors are equal
        def filter_couples(row):
            company_a, company_b = row[1][0], row[1][1]
            a_name, a_sector = company_a[0], company_a[1]
            b_name, b_sector = company_b[0], company_b[1]
            return ((a_name != b_name) and (a_sector != b_sector))

        similartrendingcompanies_rdd = trend_rdd.join(trend_rdd) \
        .filter(filter_couples) \
        .collect()
```

3.4 Output

Tutte le implementazioni hanno restituito il seguente output sul dataset originale:
(non tutte le righe sono mostrate)

```
[user33@node1 job3]$ cat output/part-00000
['2016:+1%', '2017:-1%', '2018:+3%']    COMSTOCK RESOURCES, INC.:ENERGY    KAYNE ANDERSON ENERGY
DEVELOPMENT COMPANY:N/A
['2016:-1%', '2017:+4%', '2018:-1%']    GAP, INC. (THE):CONSUMER SERVICES    LAKELAND INDUSTRIES,
INC.:HEALTH CARE
['2016:+1%', '2017:+1%', '2018:+1%']    BLACK KNIGHT, INC.:TECHNOLOGY    POWERSHARES S&P SMALLCAP
CONSUMER DISCRETIONARY PORTFOLIO:N/A
['2016:+1%', '2017:+2%', '2018:-9%']    BROOKFIELD RENEWABLE PARTNERS L.P.:PUBLIC UTILITIES
INVESCO MORTGAGE CAPITAL INC:CONSUMER SERVICES
['2016:+1%', '2017:+9%', '2018:-1%']    HERSHEY COMPANY (THE):CONSUMER NON-DURABLES    SPECTRUM
BRANDS HOLDINGS, INC.:MISCELLANEOUS
['2016:+2%', '2017:-2%', '2018:+5%']    AZZ INC.:CONSUMER DURABLES    WESTERN GAS PARTNERS,
LP:PUBLIC UTILITIES
['2016:+2%', '2017:-2%', '2018:+1%']    COMMERCE BANCSHARES, INC.:FINANCE    GORMAN-RUPP COMPANY
(THE):CAPITAL GOODS
['2016:+2%', '2017:+1%', '2018:+3%']    EATON CORPORATION, PLC:TECHNOLOGY    POWERSHARES KBW BANK
PORTFOLIO:N/A
['2016:-2%', '2017:+7%', '2018:+4%']    COVANTA HOLDING CORPORATION:BASIC INDUSTRIES    UDR,
INC.:CONSUMER SERVICES
['2016:+2%', '2017:+2%', '2018:-4%']    FIRST TRUST RIVERFRONT DYNAMIC EUROPE ETF:N/A    FRESENIUS
MEDICAL CARE CORPORATION:HEALTH CARE
['2016:+3%', '2017:+2%']    NVE CORPORATION:TECHNOLOGY    TORCHMARK CORPORATION:FINANCE
['2016:-3%', '2017:+5%', '2018:+1%']    SYPRIS SOLUTIONS, INC.:CAPITAL GOODS    VIRTUS LIFESCI
BIOTECH CLINICAL TRIALS ETF:N/A
['2016:+3%', '2017:+1%', '2018:-1%']    CENTERPOINT ENERGY, INC.:PUBLIC UTILITIES    SCORPIO
BULKERS INC.:TRANSPORTATION
['2016:+3%', '2017:+2%', '2018:-3%']    NUVEN MORTGAGE OPPORTUNITY TERM FUND 2:N/A    PRICESMART,
INC.:CONSUMER SERVICES
['2016:+4%', '2017:+1%', '2018:-6%']    KELLY SERVICES, INC.:TECHNOLOGY    MONMOUTH REAL ESTATE
INVESTMENT CORPORATION:CONSUMER SERVICES
...
```

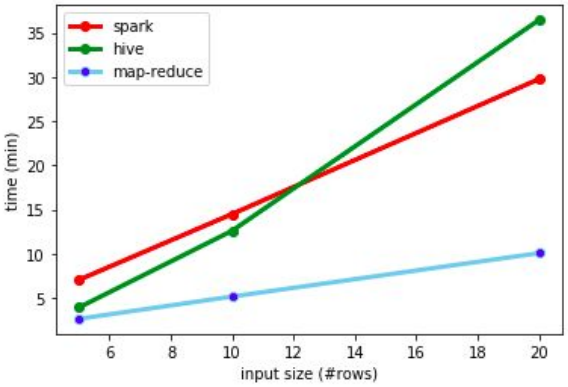
Legenda: (trend, nomeA:settoreA, nomeB:settoreB)

Osservazione: in questo caso, la dimensione dell'output diminuisce con l'aumentare della dimensione dell'input. Dal momento che il calcolo dei trend avviene con più dati a disposizione, questi sono più specifici, di conseguenza è più difficile trovare coppie di aziende con lo stesso trend, meno ancora aziende di settori diversi.

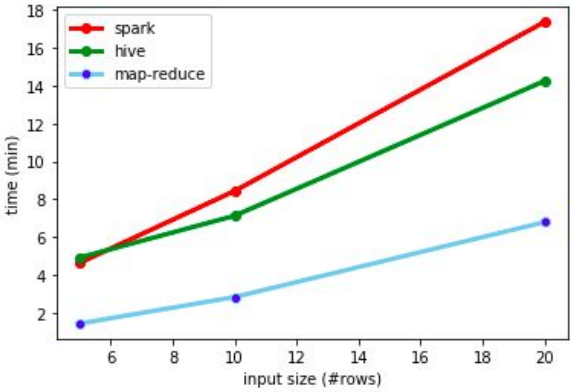
Prestazioni in locale (8 core, 8GB ram)

Abbiamo effettuato i test per i job sull'intero dataset, su metà e su un quarto, eseguendo le elaborazioni su MapReduce, Hive e Spark per un totale di 9 test per job. Di seguito sono riportate le prestazioni in termini di tempo di esecuzione in rapporto all'input size, mostrando per ciascun grafico l'esecuzione dei singoli job con le diverse tecnologie:

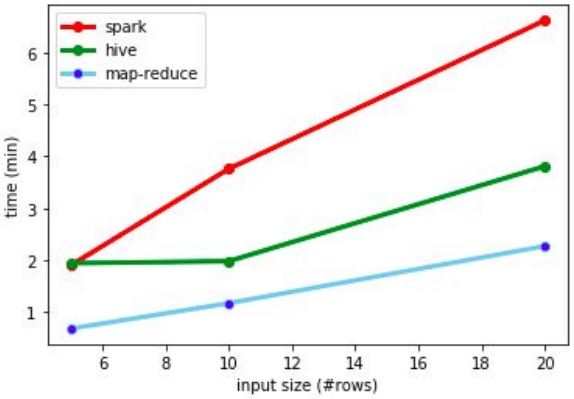
Job 1



Job 2



Job 3



Possiamo apprezzare una certa linearità nei tempi esecuzione al crescere del dataset. Su tutti i job, MapReduce risulta la soluzione più performante; abbiamo infatti riposto molta attenzione sull'ottimizzazione delle risorse nel codice dei vari mapper e reducer, aspettandoci questo risultato prima di condurre i test.

Le esecuzioni su Hive sono generalmente molto più lente, ma ciò dipende dalla natura stessa di Hive. Probabilmente con un approccio improntato alla materializzazione delle tabelle intermedie avremmo potuto ottenere prestazioni migliori, specialmente in un contesto come il nostro con poca memoria a disposizione. I risultati sono comunque in linea con quanto atteso.

Ci saremmo aspettati delle prestazioni migliori con Spark, visto che la grandezza dei dataset rientrava nella ram disponibile anche in locale. Probabilmente avremmo dovuto riservare maggiore attenzione al caching e alla persistenza dei risultati intermedi, specialmente per gli ultimi due Job.

Prestazioni sul Cluster

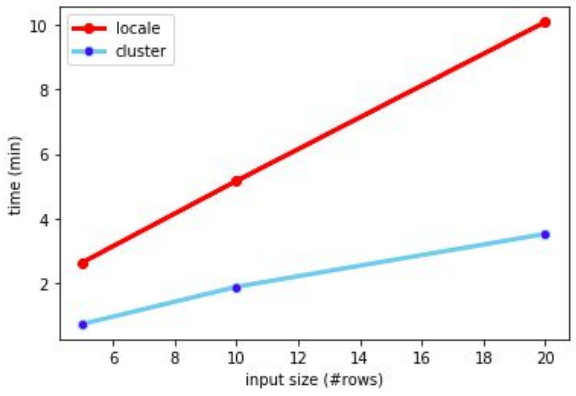
Di seguito riportiamo i grafici delle prestazioni su cluster di Roma Tre dei Job MapReduce confrontati on le prestazioni in locale.

Non abbiamo condotto i test per Hive e Spark; dopo aver condotto quelli su MapReduce, abbiamo constatato infatti che il cluster assegnava le risorse ai Job in maniera non deterministica; questo comportava una imprevedibilità delle perfomance, specialmente quando più gruppi di Big Data eseguivano contemporaneamente diversi job sul cluster. Avviando ripetutamente lo stesso job MapReduce, abbiamo infatti riscontrato tempi di esecuzione instabili,, notando come il sistema assegnasse al job una notevole quantità di memoria in alcuni casi (fino a 32 Gb circa) e poca (3 Gb) in altri.

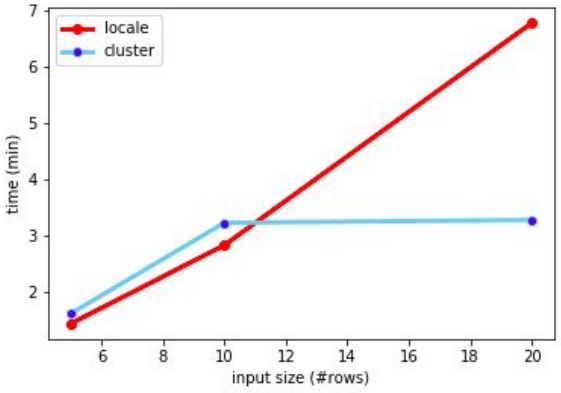
Nei grafici sottostanti emerge chiaramente il confronto tra la linearità dei tempi di esecuzione in locale e l'instabilità del cluster; su questo, escluso il primo job, abbiamo una variazione piuttosto casuale, dovuta appunto al variare delle risorse assegnate.

L'esecuzione del primo job sul cluster è forse quindi la più significativa per una valutazione. Ogni elaborazione è infatti avvenuta con un'assegnazione di circa 30 Gb per tutte le input size, registrando esecuzioni molto più rapide rispetto a quelle in locale.

Job 1



Job 2



Job 3