PROGETTO BIG DATA

Università degli studi Roma Tre

Link github:

Big-Data-Project

Fabio Letizia 546590

Contents

1 Introduzione

Questa relazione riguarda il secondo progetto del corso di Big Data.
I **Big Data** rappresentano un'enorme quantità di dati, generati a velocità elevata e con una varietà notevole, che non possono essere facilmente gestiti e analizzati con i tradizionali strumenti e tecniche di gestione dei dati.
Lo scopo del progetto è quello di riuscire a realizzare delle applicazioni basate su diverse tecnologie per generare delle statistiche su datasets di grandi dimensioni.

Il dataset utilizzato è **Daily Historical Stock Prices**. Esso contiene l'andamento giornaliero di una selezione di azioni sulla borsa di New York (NYSE) e sul NASDAQ dal 1970 al 2018.
In particolare il mio lavoro è stato realizzare tre job con quattro diverse tecnologie: **Hadoop** (MapReduce), **Hive**, **Spark Core** e **Spark SQL**:
(Job1) Un'applicazione che sia in grado di generare le statistiche di ciascuna azione dall'anno in cui è entrata in borsa indicando, per ogni azione: (a) il simbolo, (b) il nome dell'azienda, (c) una lista con l'andamento dell'azione in ciascun anno della presenza dell'azione in borsa indicando, per ogni anno: (i) la variazione percentuale della quotazione nell'anno (differenza percentuale arrotondata tra il primo prezzo di chiusura e l'ultimo prezzo di chiusura dell'anno), (ii) il prezzo minimo nell'anno, (iii) quello massimo nell'anno e (iv) il volume medio dell'anno.
(Job2) Un'applicazione che sia in grado di generare un report contenente, per ciascun'industria e per ciascun anno: (a) la variazione percentuale della quotazione dell'industria nell'anno, (b) l'azione dell'industria che ha avuto il maggior incremento percentuale nell'anno (con indicazione dell'incremento) e (c) l'azione dell'industria che ha avuto il maggior volume di transazioni nell'anno (con indicazione del volume). Nel report le industrie sono raggruppate per settore e ordinate per ordine decrescente di variazione percentuale.
(Job3) Un job in grado di generare gruppi di aziende le cui azioni hanno avuto lo stesso trend in termini di variazione annuale per almeno tre anni consecutivi a partire dal 2000, indicando le aziende e il trend comune.

Tutti i job sono stati eseguiti sia sulla mia macchina in locale, sia su AWS (Amazon Web Services), una piattaforma di cloud computing che offre una vasta gamma di servizi e risorse scalabili. Questa doppia esecuzione ha permesso di confrontare le prestazioni e l'efficienza delle operazioni in due ambienti differenti.
Un aspetto significativo del lavoro riguarda il pre-processing del dataset di input. In particolare, è stato eseguito il **Data Profiling** per analizzare le caratteristiche e le statistiche del dataset. Questo processo è stato implementato utilizzando il tool **Y-Data Profiling** in combinazione con **Apache Spark**.
Maggiori dettagli sul progetto, inclusi i metodi utilizzati, le sfide affrontate e i risultati ottenuti, sono riportati nel resto della relazione.

2 Dataset

Il dataset di input è **Daily Historical Stock Prices**, scaricabile da Kaggle (https://www.kaggle.com/) all'indirizzo https://www.kaggle.com/dataset. Il dataset è formato da due file CSV.

Il primo (**historical_stock_prices**) ha i seguenti campi:

- 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"
- low: prezzo minimo
- high: prezzo massimo
- volume: numero di transazioni
- date: data nel formato aaaa-mm-gg

Il secondo (**historical_stocks**) ha questi campi:

- ticker: simbolo dell'azione
- exchange: NYSE o NASDAQ
- name: nome dell'azienda
- sector: settore dell'azienda (per esempio "technology")
- industry: industria di riferimento per l'azienda (per esempio "semiconductors")

Il primo dataset ha come chiave combinata i campi ticker e date, ogni record rappresenta diverse informazioni di prezzi relative ad una certa azione (ticker) in una data precisa (date). Ecco alcune caratteristiche della prima tabella:

- numero di righe: 20973889
- numero di colonne: 8
- numero di righe con valori nulli: 0
- dimensione: 1.87 GB

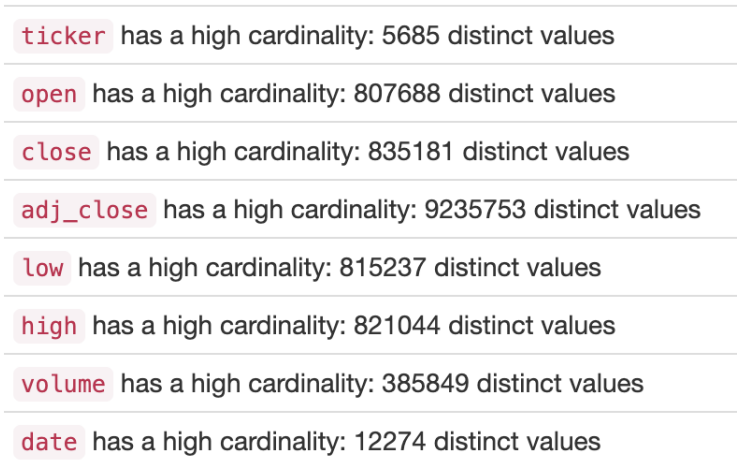


Figure 1: YData Profiling report

Il secondo dataset ha come chiave il campo ticker e ogni record contiene diverse informazioni relative ad una certa azione (ticker). Ecco alcune caratteristiche della seconda tabella:

- numero di righe: 6460
- numero di colonne: 5
- numero di righe con valori nulli: 1440
- dimensione: 0,442 MB

Figure 2: YData Profiling report

Tutto il codice relativo a YData Profiling si trova nel notebook **dataset_stats.ipynb** presente nella cartella data cleaning.

2.1 Pre-processing: YData Profiling

Un contributo significativo del lavoro riguarda il pre-processing del dataset di input. In particolare, è stato effettuato Data Profiling sul dataset per analizzarne le caratteristiche principali. Questo processo è stato realizzato utilizzando il tool **Y-Data Profiling** in combinazione con Apache Spark. È una fase cruciale nel pre-processing dei dati, che permette di comprendere meglio la struttura, la qualità e le peculiarità del dataset. Attraverso il Data Profiling, siamo in grado di identificare valori mancanti, anomalie, distribuzioni di valori, e altre metriche statistiche che sono essenziali per una corretta pulizia e preparazione dei dati.

Y-Data Profiling è un potente strumento che fornisce una panoramica dettagliata del dataset. Questo tool offre numerose funzionalità, tra cui: analisi della distribuzione dei dati, dentificazione di valori nulli e duplicati, rilevamento di outlier, calcolo di statistiche descrittive come media, mediana, deviazione standard, visualizzazioni grafiche per una comprensione immediata delle caratteristiche del dataset.
Per gestire dataset di grandi dimensioni e ottimizzare le prestazioni del Data Profiling, è stato integrato Y-Data Profiling con **Apache Spark**, un framework di calcolo distribuito noto per la sua velocità e capacità di elaborare dati su larga scala. Spark ha permesso di eseguire operazioni di analisi e pulizia dei dati in modo efficiente e scalabile.
Basandomi sulle carattestiche dei due file csv e sui requisiti dei job da realizzare le operazioni preliminari di pulizia eseguite riguardano la rimozione della colonna '**adj_close**', la divisione dei campi con '**;**' al posto di ',' e la fusione dei due file in un unica tabella (csv).
Il campo 'adj_close' è stato ritenuto poco utile ai fini delle applicazioni e il separatore ';' è stato introdotto per evitare di separare i campi in maniera errata data la presenza di ',' all'interno ad esempio del campo 'nome'. La tabella finale realizzata prende il nome di **historical_stocks_data** e nasce dal join delle due tabelle sul campo 'ticker'.
Questo dataset ha come chiave combinata i campi ticker e date, ogni record rappresenta diverse informazioni relative ad una certa azione (ticker) in una data precisa (date).
Ecco alcune caratteristiche della tabella finale:

- numero di righe: 20973889
- numero di colonne: 11
- numero di righe con valori nulli: 2549449
- dimensione: 2.81 GB

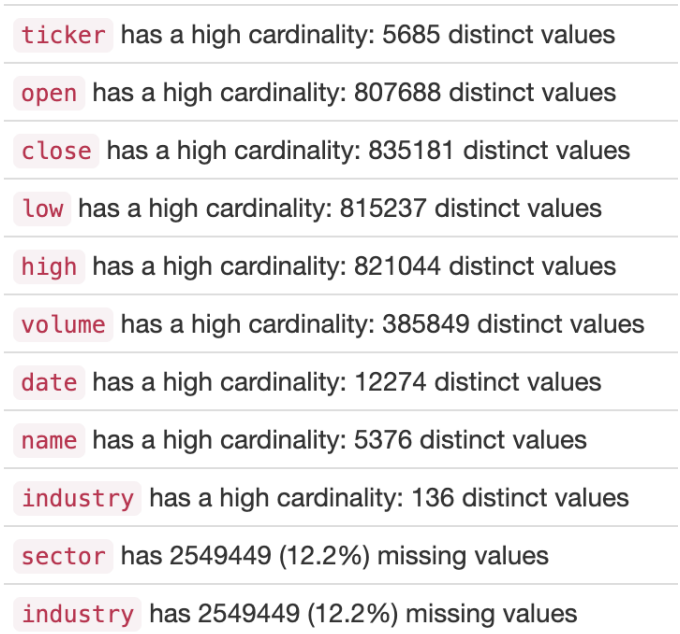


Figure 3: YData Profiling report

Per tutto il resto del progetto il dataset di riferimento sarà quest'ultimo (historical_stocks_data).
Il codice relativo al pre-processing del dataset si trova nel notebook **cleaning.ipynb** presente nella cartella data cleaning.

2.2 Variazione dimensione dell'input

In conclusione, è stato sviluppato uno script Python che consente di modificare le dimensioni dell'input aggiungendo record fittizi. L'obiettivo di questa operazione è testare le tecnologie e le applicazioni su diverse dimensioni di input.
A partire dal dataset di riferimento **historical_stocks_data** di 2.81 GB sono stati realizzati:

- **historical_stocks_data_50**: dataset dimezzato: 1.40 GB
- **historical_stocks_data_150**: dataset aumentato del 50%: 4.21 GB
- **historical_stocks_data_200**: dataset raddoppiato: 5.62 GB

Tutti gli esperimenti sono quindi stati realizzati su 4 dataset di dimensioni diverse.

Il codice relativo alla generazione di questi dataset si trova nel notebook **input_variation.ipynb** presente nella cartella data cleaning.

3 Job 1

Il job è stato realizzato a partire dal dataset pre-processato (historical_stocks_data) e con quattro diverse tecnologie: Hadoop, Hive, Spark Core e Spark SQL.
Il risultato ottenuto è stato memorizzato sul repository github, ecco riportate le prime dieci righe (relative ai primi 10 ticker) dell'**output**:

Ticker	Name	Year	Percentage Change	Min Price	Max Price	Avg Volume
A	AGILENT TECHNOLOGIES, INC.	1999	75.71	28.48	57.22	5,739,950.00
AA	ALCOA CORPORATION	1970	-19.85	4.71	7.41	119,964.96
AABA	ALTABA INC.	1996	-48.48	0.65	1.79	6,108,502.73
AAC	AAC HOLDINGS, INC.	2018	4.86	7.79	12.96	171,227.10
AAL	AMERICAN AIRLINES GROUP, INC.	2005	92.44	19.10	38.80	897,358.21
AAME	ATLANTIC AMERICAN CORPORATION	1980	40.26	3.10	5.50	11,181.54
AAOI	APPLIED OPTOELECTRONICS, INC.	2013	50.70	9.07	16.61	99,426.87
AAON	AAON, INC.	1992	40.00	0.09	0.15	50,366.67
AAP	ADVANCE AUTO PARTS INC	2001	19.48	13.23	16.58	233,659.09
AAPL	APPLE INC.	1980	18.70	0.45	0.65	25,862,523.08

Table 1: Output job 1

3.1 Hadoop - MapReduce

Hadoop è un framework open-source sviluppato da Apache per l'elaborazione e l'archiviazione di grandi quantità di dati in un ambiente distribuito. È progettato per scalare da un singolo server a migliaia di macchine, ognuna delle quali offre storage locale e potenza di calcolo.
MapReduce è un modello di programmazione usato per elaborare grandi quantità di dati. Il processo è diviso in due fasi principali: Map e Reduce.
Map: il dataset viene suddiviso in blocchi più piccoli. Ogni blocco viene elaborato in parallelo da un "mapper", un'unità che esegue una funzione di mappatura. La funzione di mappatura prende in input una coppia chiave-valore e produce una lista di coppie chiave-valore intermedie.
Fase **Shuffle and Sort**: le coppie chiave-valore intermedie prodotte dai mapper vengono riordinate (shuffled) e raggruppate (sorted) per chiave.
Reduce: le chiavi ordinate e i relativi valori vengono passati ai "reducers", che applicano una funzione di riduzione. La funzione di riduzione

aggrega i valori per ogni chiave, producendo l'output finale.

Il codice è stato realizzato in python in due classi mapper.py e reducer.py nella cartella **job1/HADOOP** del progetto.

- Mapper (**mapper.py**) ha l'obiettivo di preparare i dati di input per il reducer, estraendo solo i campi necessari e riorganizzandoli:

Algorithm 1 Mapper

```
1: for ogni record di input do
2:     RIMUOVI gli spazi bianchi dalla linea
3:     SALTA l'header
4:     SEPARA la linea in campi usando il carattere ';' come delimitatore
5:     if il numero di campi è 11 then
6:         ASSEGNA i campi alle variabili: ticker, open, close, low, high,
               volume, date, exchange, name, sector, industry
7:         ESTRAPOLA l'anno dalla data
8:         CREA la chiave come combinazione di ticker, name e year
9:         CREA il valore come combinazione di close, low, high, volume
               e date
10:        STAMPA la chiave e il valore separati da '\t'
11:    end if
12: end for
```

- Reducer (**reducer.py**) ha l'obiettivo di calcolare le statistiche annuali per ogni titolo:

Algorithm 2 Reducer

```
1: function CALCULATE_PERCENTAGE_CHANGE(start, end)
2:     if start ≠ 0 then
3:         return ((end - start) / start) * 100
4:     else
5:         return 0
6:     end if
7: end function
8:
9: function PARSE_INPUT_LINE(line)
10:    DIVIDI la linea usando il delimitatore '\t'
11:    ASSEGNA i valori estratti a key, value
12:    DIVIDI key usando il delimitatore ';'
13:    ASSEGNA i valori estratti a ticker, name, year
14:    DIVIDI value usando il delimitatore ';'
15:    ASSEGNA i valori estratti a close, low, high, volume, date
16:    CONVERTI close, low, high in float
17:    CONVERTI volume in int
18:    return ((ticker, name, year), (close, low, high, volume, date))
19: end function
20:
21: function MAIN
22:    INIZIALIZZA stock_data come un dizionario di liste
23:    for ogni line in sys.stdin do
24:        ASSEGNA key, value usando parse_input_line(line)
25:        AGGIUNGI value a stock_data[key]
26:    end for
27:    INIZIALIZZA result_data come una lista vuota
28:
29:    for ogni (ticker, name, year), daily_values in stock_data.items() do
30:        ORDINA daily_values per date
31:        ASSEGNA first_close a daily_values[0][0]
32:        ASSEGNA last_close a daily_values[-1][0]
33:        ASSEGNA min_price al minimo di low in daily_values
34:        ASSEGNA max_price al massimo di high in daily_values
35:        ASSEGNA avg_volume alla media di volume in daily_values
36:        ASSEGNA percentage_change a calculate_percentage_change
               (first_close, last_close)
37:        AGGIUNGI (ticker, name, year, percentage_change, min_price,
               max_price, avg_volume) a result_data
38:    end for
39:
40:    STAMPA i risultati
41:
```

3.2 Hive

Hive è un framework di data warehousing sviluppato da Apache basato su Hadoop, che fornisce un'interfaccia SQL-like per interrogare, analizzare e gestire grandi dataset immagazzinati in Hadoop Distributed File System (HDFS) o in altri sistemi di archiviazione compatibili. Hive traduce le query SQL in job MapReduce, semplificando l'analisi dei big data per gli utenti familiari con il linguaggio SQL.
Il codice è stato realizzato in due file hql nella cartella **job1/HIVE** del progetto.

- **datasetLoading.hql** ha l'obiettivo di caricare il dataset di input su hive:

```
-- Create a Hive table for the input data
CREATE TABLE IF NOT EXISTS stock_data (
    `ticker` STRING,
    `open` DOUBLE,
    `close` DOUBLE,
    `low` DOUBLE,
    `high` DOUBLE,
    `volume` INT,
    `date` STRING,
    `exchange` STRING,
    `name` STRING,
    `sector` STRING,
    `industry` STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ';'
STORED AS TEXTFILE;

-- Load data into Hive table from hdfs
LOAD DATA INPATH '/input/historical_stocks_data.csv' INTO TABLE
   stock_data;
```

- **query.hql** ha l'obiettivo di calcolare le statistiche annuali per ogni titolo facendo delle interrogazioni SQL:

```
-- Set number of reducer
SET mapreduce.job.reduces=2;

-- This part of the code creates a CTE (Common Table Expression) called
   stock_yearly_stats,
-- which calculates yearly statistics for each stock
WITH stock_yearly_stats AS (
    SELECT
        ticker,
        `name`,
        YEAR(FROM_UNIXTIME(UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd'))) AS `
   year`,
        FIRST_VALUE(close) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
   UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd'))) ORDER BY `date`) AS first_close
   ,
        LAST_VALUE(close) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
   UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd'))) ORDER BY `date` ROWS BETWEEN
   UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_close,
        MIN(low) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
   UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd')))) AS min_price,
        MAX(high) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
   UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd')))) AS max_price,
        AVG(volume) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
   UNIX_TIMESTAMP(`date`, 'yyyy-MM-dd')))) AS avg_volume
    FROM stock_data
)

-- Final table with all statistics
-- (including the calculation of the percentage change in the closing
   price)
CREATE TABLE output AS
SELECT
    ticker,
    `name`,
    `year`,
    ROUND(((last_close - first_close) / first_close) * 100, 2) AS
   percent_change,
    min_price,
    max_price,
```

```
    ROUND(avg_volume, 2) AS avg_volume
FROM stock_yearly_stats
GROUP BY ticker, `name`, `year`, first_close, last_close, min_price,
   max_price, avg_volume
ORDER BY ticker, `year`;

SELECT * FROM output;
```

3.3 Spark Core

Spark Core è il motore di esecuzione di Apache Spark, un framework open-source per l'elaborazione parallela e distribuita di grandi set di dati. È responsabile della gestione dei task, della distribuzione dei dati in memoria e della coordinazione tra i nodi di un cluster Spark. Consente agli sviluppatori di scrivere applicazioni complesse per l'analisi dei dati in modo efficiente e scalabile.

Il codice è stato realizzato in python nel file job1.py che nella cartella **job1/SPARKcore** del progetto.

Algorithm 3 SPARKCore/job1.py

1: **function** PARSE_LINE(linea)
2: ESTRAI i campi dalla linea usando il delimitatore ';'
3: ASSEGNA i valori estratti alle variabili: *ticker*, *date*, *close*, *low*, *high*, *volume*, *name*
4: CONVERTI la data nel formato *datetime* e ottieni l'anno
5: CONVERTI *close*, *low*, *high*, *volume* in *float*
6: **return** ((*ticker*, *year*), (*date*, *close*, *low*, *high*, *volume*, *name*))
7: **end function**
8:
9: **function** CALCULATE_STATS(valori)
10: ORDINA i valori per *date*
11: ESTRAPOLA i campi: *date*, *close_prices*, *low_prices*, *high_prices*, *volumes*, *name*
12: OTTIENI il primo e l'ultimo valore di *close_prices*
13: CALCOLA la variazione percentuale arrotondata a 2 decimali
14: CALCOLA il prezzo massimo (*high*) arrotondato a 2 decimali
15: CALCOLA il prezzo minimo (*low*) arrotondato a 2 decimali
16: CALCOLA il volume medio arrotondato a 2 decimali
17: **return** (*name*[0], *percentual_variation*, *min_low*, *max_high*, *mean_volume*)
18: **end function**
19:
20: CREA una sessione Spark
21: LEGGI il dataset da *dataset_filepath* e memorizzalo in *lines*
22: MAPPA ogni linea usando *parse_line* e FILTRA i valori *None*
23: RAGGRUPPA i valori per chiave (*ticker*, *year*)
24: CALCOLA le statistiche annuali per ogni gruppo usando *calculate_stats*
25: MAPPA i risultati in una tupla (*chiave*, *statistiche*)
26: ORDINA i risultati per chiave
27: FERMA la sessione Spark =0

3.4 Spark SQL

Spark SQL è un modulo di Apache Spark progettato per l'elaborazione di dati strutturati. Fornisce un'interfaccia SQL-like per interrogare e analizzare dati in formato tabellare, rendendo più facile per gli sviluppatori e gli analisti lavorare con dati strutturati all'interno di un ambiente Spark. Spark SQL supporta l'integrazione di dati strutturati con altri componenti di Spark, consentendo agli utenti di eseguire query SQL su dati distribuiti in modo efficiente e scalabile.

Il codice è stato realizzato in python nel file job1.py nella cartella **job1/SPARKSQL** del progetto.

```
# Read data from hdfs
stock_data = spark.read.csv("/input/historical_stocks_data.csv", header=
    True, sep=';', inferSchema=True)

# Creates the temporary table that allows you to query the stock_data
    dataframe
stock_data.createOrReplaceTempView("stock_data")

# Query (very similar to HIVE)
query = "
WITH stock_yearly_stats AS (
    SELECT
        ticker,
        name,
        YEAR(FROM_UNIXTIME(UNIX_TIMESTAMP(date, 'yyyy-MM-dd'))) AS year,
        FIRST_VALUE(close) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
    UNIX_TIMESTAMP(date, 'yyyy-MM-dd'))) ORDER BY date) AS first_close,
        LAST_VALUE(close) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
    UNIX_TIMESTAMP(date, 'yyyy-MM-dd'))) ORDER BY date ROWS BETWEEN
    UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_close,
        MIN(low) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
    UNIX_TIMESTAMP(date, 'yyyy-MM-dd')))) AS min_price,
        MAX(high) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
    UNIX_TIMESTAMP(date, 'yyyy-MM-dd')))) AS max_price,
        AVG(volume) OVER (PARTITION BY ticker, YEAR(FROM_UNIXTIME(
    UNIX_TIMESTAMP(date, 'yyyy-MM-dd')))) AS avg_volume
```

```
    FROM stock_data
)
SELECT
    ticker,
    name,
    year,
    ROUND(((last_close - first_close) / first_close) * 100, 2) AS
   percentage_change,
    min_price,
    max_price,
    ROUND(avg_volume, 2) AS avg_volume
FROM stock_yearly_stats
GROUP BY ticker, name, year, first_close, last_close, min_price,
   max_price, avg_volume
ORDER BY ticker, year
"

result = spark.sql(query)
```

4 Job 2

Come per l'applicazione precedente, il job è stato realizzato a partire dal dataset pre-processato (historical_stocks_data) e con quattro diverse tecnologie: Hadoop, Hive, Spark core e Spark SQL.
Il risultato ottenuto è stato memorizzato sul repository github, ecco riportate le prime dieci righe dell'**output**:

Sector	Industry	Year	Industry Change	Max Increment Ticker	Max Volume Ticker
BASIC INDUSTRIES	SPECIALTY CHEMICALS	2008	-33.23%	('VERU', 39.21022036959007)	('WST', 337887150)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1990	-32.90%	('CSL', -14.440433212996389)	('WST', 32039600)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1999	-21.92%	('WST', -11.607142857142858)	('WST', 84262700)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1994	-20.10%	('WST', 12.244897959183673)	('WST', 32482600)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	2007	-10.76%	('VERU', 71.71052156888285)	('WST', 305445400)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1981	-8.28%	('WST', 69.83240223463687)	('WST', 67376000)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1995	-7.21%	('CSL', 12.937062937062937)	('WST', 34430800)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	2011	-5.77%	('CSL', 9.40972437800593)	('WST', 263308468)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1984	-4.41%	('CSL', 20.982142857142858)	('WST', 44765600)
BASIC INDUSTRIES	SPECIALTY CHEMICALS	1992	-4.13%	('WST', 23.28767123287671)	('WST', 42648500)

Table 2: Output job 2

4.1 Hadoop - MapReduce

Come per il job 1, il codice è stato realizzato in python in due classi mapper.py e reducer.py, nella cartella **job2/HADOOP** del progetto.

- Mapper (**mapper.py**) ha l'obiettivo di preparare i dati di input per il reducer, estraendo solo i campi necessari e riorganizzandoli:

Algorithm 4 Mapper

```
1:  for ogni record di input do
2:      for ogni record di input do
3:          RIMUOVI gli spazi bianchi dalla linea
4:          SALTA l'header
5:          SEPARA la linea in campi usando il carattere ';' come
              delimitatore
6:          if il numero di campi è 11 then
7:              ASSEGNA i campi alle variabili: ticker, _, close, _, _, volume,
                 date, _, name, sector, industry
8:              ESTRAPOLA l'anno dalla data
9:              if sector e industry non sono vuoti then
10:                 CREA la chiave come (sector, industry, year)
11:                 CREA il valore come (ticker, close, volume, date)
12:                 STAMPA la chiave e il valore separati da '\t'
13:             end if
14:         end if
15:     end for
```

- Reducer (**reducer.py**) ha l'obiettivo di calcolare le statistiche annuali per ogni settore e industria:

Algorithm 5 Reducer: PART 1

```
1: function CALCULATE_PERCENTAGE_CHANGE(start, end)
2:     if start ≠ 0 then
3:         return ((end - start) / start) * 100
4:     else
5:         return 0
6:     end if
7: end function
8:
9: function MAIN
10:    CREA un dizionario data per memorizzare i dati
11:    for ogni record di input do
12:        DIVIDI la linea in chiave e valore usando '\t' come delimitatore
13:        CONVERTI chiave e valore da stringa a tuple
14:        AGGIUNGI il valore alla lista associata alla chiave in data
15:    end for
16:    CREA una lista result per memorizzare i risultati
17:    for ogni chiave (sector, industry, year) e lista di records in data do
18:        ORDINA i records per ticker e date
19:        CREA dizionari per memorizzare i primi e gli ultimi valori di
           close per ogni ticker
20:        INIZIALIZZA variabili per il massimo incremento percentuale
                e il volume totale
21:        CREA un dizionario ticker_records per memorizzare i records
           associati a ciascun ticker
22:        for ogni record in records do
23:            ASSEGNA i campi del record alle variabili ticker, close,
               volume, date
24:            CONVERTI close in float e volume in int
25:            AGGIUNGI il record alla lista associata a ticker in
               ticker_records
26:        end for
27:
```

Algorithm 6 Reducer: PART 2

for ogni *ticker* e lista di records in *ticker_records* **do**
    ORDINA i records per *date*
    OTTIENI il primo e l'ultimo valore di *close*
    CALCOLA l'incremento percentuale tra il primo e l'ultimo valore di *close*
    AGGIUNGI il volume totale di transazioni per il *ticker*
    if l'incremento è maggiore del massimo incremento registrato **then**
        AGGIORNA il massimo incremento e il relativo *ticker*
    end if
    if il volume totale del *ticker* è maggiore del massimo volume registrato **then**
        AGGIORNA il massimo volume e il relativo *ticker*
    end if
end for
CALCOLA la variazione percentuale per l'industria sommando i primi e gli ultimi valori di *close* dei *ticker*
AGGIUNGI i risultati alla lista *result* come una tupla (*sector*, *industry*, *year*, *industry_change*, *max_increment_ticker*, *max_ticker_volume*)

CREA un dizionario *grouped_results* per raggruppare i risultati per *sector* e *industry*
for ogni record in *result* **do**
    AGGIUNGI il record alla lista associata alla chiave (*sector*, *industry*) in *grouped_results*
end for
ORDINA i risultati raggruppati per la variazione percentuale dell'industria in ordine decrescente
for ogni gruppo di risultati in *grouped_results* **do**
    ORDINA i risultati del gruppo per *sector* e *industry*
    for ogni record nel gruppo **do**
        STAMPA i risultati in formato tabellare
    end for
end for=0

4.2 Hive

Il codice è stato realizzato in due file hql e si trova nella cartella **job2/HIVE** del progetto.

- **datasetLoading.hql** ha l'obiettivo di caricare il dataset di input su hive:

```
-- Create a Hive table for the input data
CREATE TABLE IF NOT EXISTS stock_data (
    `ticker` STRING,
    `open` DOUBLE,
    `close` DOUBLE,
    `low` DOUBLE,
    `high` DOUBLE,
    `volume` INT,
    `stock_date` STRING,
    `exchange` STRING,
    `name` STRING,
    `sector` STRING,
    `industry` STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ';'
STORED AS TEXTFILE;

-- Load data into Hive table from hdfs
LOAD DATA INPATH '/input/historical_stocks_data.csv' INTO TABLE
   stock_data;
```

- **query.hql** ha l'obiettivo di calcolare le statistiche annuali per ogni settore e industria facendo delle interrogazioni SQL:

```
-- Table with addition of the year field
CREATE TABLE stock_data_year AS
SELECT
    ticker,
    open,
    close,
    low,
```

```
    high,
    volume,
    `date` AS stock_date,
    `exchange`,
    name,
    sector,
    industry,
    YEAR(TO_DATE(`date`)) AS year
FROM
    stock_data;

-- Table containing the ticker, year, first closing price, and last
   closing price for each stock in each year
CREATE TABLE stock_first_last_close AS
SELECT
    ticker,
    year,
    FIRST_VALUE(close) OVER (PARTITION BY ticker, year ORDER BY
   stock_date) AS first_close,
    LAST_VALUE(close) OVER (PARTITION BY ticker, year ORDER BY stock_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
   last_close
FROM
    stock_data_year;

-- Table in which the following are calculated for each ticker: the
   annual percentage change, the annual minimum, the annual maximum and
   the annual volume.
CREATE TABLE stock_statistics AS
SELECT
    sdy.ticker,
    sdy.name,
    sdy.year,
    sdy.sector,
    sdy.industry,
    ROUND(((sflc.last_close - sflc.first_close) / sflc.first_close) *
   100, 2) AS percent_change,
    ROUND(MIN(sdy.low), 2) AS min_low,
    ROUND(MAX(sdy.high), 2) AS max_high,
    ROUND(SUM(sdy.volume), 2) AS total_volume
FROM
```

```
    stock_data_year sdy
JOIN
    stock_first_last_close sflc
ON
    sdy.ticker = sflc.ticker AND sdy.year = sflc.year
GROUP BY
    sdy.ticker, sdy.name, sdy.year, sdy.sector, sdy.industry, sflc.
   first_close, sflc.last_close;

-- Summing up first and last close prices for each industry in each year
CREATE TABLE industry_first_last_close AS
SELECT
    sector,
    industry,
    year,
    SUM(open) AS industry_first_close,
    SUM(close) AS industry_last_close
FROM
    stock_data_year
GROUP BY
    sector, industry, year;

-- Calculating the industry annual percentage change
CREATE TABLE industry_annual_change AS
SELECT
    sector,
    industry,
    year,
    ROUND(((industry_last_close - industry_first_close) /
   industry_first_close) * 100, 2) AS industry_percent_change
FROM
    industry_first_last_close;

-- Table to calculate the ticker with the highest percentage increase for
    each industry
CREATE TABLE industry_max_increase AS
SELECT
    sector,
    industry,
    year,
    FIRST(ticker) AS ticker,
```

```
    FIRST(name) AS name,
    MAX(percent_change) AS highest_percent_change
FROM
    stock_statistics
GROUP BY
    sector, industry, year;

-- Table to calculate the ticker with the highest volume for each
   industry
CREATE TABLE industry_max_volume AS
SELECT
    sector,
    industry,
    year,
    FIRST(ticker) AS ticker,
    FIRST(name) AS name,
    MAX(total_volume) AS highest_volume
FROM
    stock_statistics
GROUP BY
    sector, industry, year;

-- Final table with all statistics
CREATE TABLE output AS
SELECT
    iac.sector,
    iac.industry,
    iac.year,
    iac.industry_percent_change,
    imh.name AS highest_increase_stock,
    imh.highest_percent_change,
    imv.name AS highest_volume_stock,
    imv.highest_volume
FROM
    industry_annual_change iac
JOIN
    industry_max_increase imh
ON
    iac.sector = imh.sector AND iac.industry = imh.industry AND iac.year
   = imh.year
JOIN
```

```
    industry_max_volume imv
ON
    iac.sector = imv.sector AND iac.industry = imv.industry AND iac.year
   = imv.year
ORDER BY
    iac.sector, iac.industry, iac.year DESC;

SELECT * FROM output;
```

4.3 Spark Core

Il codice è stato realizzato in python nel file job2.py nella cartella **job2/SPARKcore** del progetto.

Algorithm 7 SPARKcore/job2.py: PART 1

```
1:  function CALCULATE_PERCENTAGE_CHANGE(start, end)
2:      if start ≠ 0 then
3:          return ((end−start)/start) × 100
4:      else
5:          return 0
6:      end if
7:  end function
8:
9:  function PARSE_LINE(linea)
10:     RIMUOVI gli spazi bianchi dalla linea
11:     SALTA l'header e le linee malformate
12:     SEPARA la linea in parti usando il delimitatore ';'
13:     ASSEGNA i campi alle variabili: ticker, close, volume, date, name,
                sector, industry
14:     ESTRAPOLA l'anno dalla data
15:     if sector e industry non sono vuoti then
16:         CREA la chiave come (sector, industry, year)
17:         CREA il valore come (ticker, float(close), int(volume), date)
18:         return (key, value)
19:     end if
20:     return None
21: end function
```

Algorithm 8 SPARKcore/job2.py: PART 2

```
1: function AGGREGATE_RECORDS(records)
2:     CREA ticker_records come dizionario di liste
3:     CREA industry_tickers_first_close come dizionario
4:     CREA industry_tickers_last_close come dizionario
5:     CREA total_ticker_volume come dizionario di interi
6:     for record in records do
7:         ASSEGNA i campi del record alle variabili ticker, close, volume,
               date
8:         AGGIUNGI il record alla lista associata a ticker in
               ticker_records
9:     end for
10:    INIZIALIZZA max_increment a −∞
11:    INIZIALIZZA max_increment_ticker a None
12:    INIZIALIZZA max_ticker_volume a (None, −∞)
13:    for ogni ticker e lista di records in ticker_records do
14:        ORDINA i records per date
15:        OTTIENI il primo valore di close come first_close
16:        OTTIENI l'ultimo valore di close come last_close
17:        CALCOLA l'incremento percentuale tra first_close e last_close
18:        for record in records do
19:            AGGIUNGI il volume totale di transazioni per il ticker in
                   total_ticker_volume[ticker]
20:        end for
21:        if l'incremento è maggiore del massimo incremento registrato
       then
22:            AGGIORNA il massimo incremento e il relativo ticker
23:        end if
24:        if il volume totale del ticker è maggiore del massimo volume
       registrato then
25:            AGGIORNA il massimo volume e il relativo ticker
26:        end if
27:        if ticker non è in industry_tickers_first_close then
28:            AGGIUNGI first_close a industry_tickers_first_close
29:        end if
30:        AGGIORNA last_close in industry_tickers_last_close
31:    end for
32:    CALCOLA il totale di first_close dei tickers dell'industria
33:    CALCOLA il totale di last_close dei tickers dell'industria
34:    CALCOLA la variazione percentuale per l'industria
35:    return (industry_change, max_increment_ticker,
       max_ticker_volume)
36: end function
```

Algorithm 9 SPARKcore/job2.py: PART 3

CREA una sessione Spark
LEGGI il dataset da *dataset_filepath* e memorizzalo in *lines*
MAPPA ogni linea usando *parse_line* e FILTRA i valori *None*
RAGGRUPPA i valori per chiave e trasformali in lista
CALCOLA i risultati aggregati usando *aggregate_records*
COLLEZIONA i risultati
CREA un dizionario *grouped_results* per memorizzare i risultati raggruppati
for ogni record in *result* **do**
    ASSEGNA *key* e *values* dal record
    AGGIUNGI il record alla lista associata a (*sector*, *industry*) in *grouped_results*
end for
ORDINA i risultati raggruppati per la variazione percentuale dell'industria in ordine decrescente
for ogni gruppo di risultati in *grouped_results* **do**
    ORDINA i risultati del gruppo per *sector* e *industry*
    for ogni record nel gruppo **do**
        STAMPA i risultati in formato tabellare
    end for
end for
FERMA la sessione Spark

4.4 Spark SQL

Il codice è stato realizzato in python nel file job2.py nella cartella **job2/SPARKSQL** del progetto.

```
# Read data from hdfs
stock_data = spark.read.csv('/input/historical_stocks_data.csv', header=
    True, sep=';', inferSchema=True)

# Creates the temporary table that allows you to query the stock_data
    dataframe
stock_data.createOrReplaceTempView("stock_data")

# Table with addition of the year field
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_data_year AS
SELECT
    ticker,
    open,
    close,
    low,
    high,
    volume,
    `date` AS stock_date,
    `exchange`,
    name,
    sector,
    industry,
    YEAR(TO_DATE(stock_date, 'yyyy-MM-dd')) AS year
FROM
    stock_data
")

# Summing up first and last close prices for each industry in each year
spark.sql("
CREATE OR REPLACE TEMP VIEW industry_first_last_close AS
SELECT
    sector,
    industry,
    year,
    SUM(open) AS industry_first_close,
```

```
    SUM(close) AS industry_last_close
FROM
    stock_data_year
GROUP BY
    sector, industry, year
")

# Calculating the industry annual percentage change
spark.sql("
CREATE OR REPLACE TEMP VIEW industry_annual_change AS
SELECT
    sector,
    industry,
    year,
    ROUND(((industry_last_close - industry_first_close) /
   industry_first_close) * 100, 2) AS industry_percent_change
FROM
    industry_first_last_close
")

# Table containing the ticker, year, first closing price, and last
   closing price for each stock in each year
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_first_last_close AS
SELECT
    ticker,
    year,
    FIRST_VALUE(close) OVER (PARTITION BY ticker, year ORDER BY
   stock_date) AS first_close,
    LAST_VALUE(close) OVER (PARTITION BY ticker, year ORDER BY stock_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
   last_close
FROM
    stock_data_year
")

# Table in which the following are calculated for each ticker: the annual
    percentage change, the annual minimum, the annual maximum and the
   annual volume.
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_statistics AS
```

```
SELECT
    sdy.ticker,
    sdy.name,
    sdy.year,
    sdy.sector,
    sdy.industry,
    ROUND(((sflc.last_close - sflc.first_close) / sflc.first_close) *
   100, 2) AS percent_change,
    ROUND(MIN(sdy.low), 2) AS min_low,
    ROUND(MAX(sdy.high), 2) AS max_high,
    ROUND(SUM(sdy.volume), 2) AS total_volume
FROM
    stock_data_year sdy
JOIN
    stock_first_last_close sflc
ON
    sdy.ticker = sflc.ticker AND sdy.year = sflc.year
GROUP BY
    sdy.ticker, sdy.name, sdy.year, sdy.sector, sdy.industry, sflc.
   first_close, sflc.last_close
")

# Table to calculate the ticker with the highest percentage increase for
   each industry
spark.sql("
CREATE OR REPLACE TEMP VIEW industry_max_increase AS
SELECT
    sector,
    industry,
    year,
    FIRST(ticker) AS ticker,
    FIRST(name) AS name,
    MAX(percent_change) AS highest_percent_change
FROM
    stock_statistics
GROUP BY
    sector, industry, year
")

# Table to calculate the ticker with the highest volume for each industry
spark.sql("
```

```
CREATE OR REPLACE TEMP VIEW industry_max_volume AS
SELECT
    sector,
    industry,
    year,
    FIRST(ticker) AS ticker,
    FIRST(name) AS name,
    MAX(total_volume) AS highest_volume
FROM
    stock_statistics
GROUP BY
    sector, industry, year
")

# Final table with all statistics
spark.sql("
CREATE OR REPLACE TEMP VIEW output AS
SELECT
    iac.sector,
    iac.industry,
    iac.year,
    iac.industry_percent_change,
    imh.name AS highest_increase_stock,
    imh.highest_percent_change,
    imv.name AS highest_volume_stock,
    imv.highest_volume
FROM
    industry_annual_change iac
JOIN
    industry_max_increase imh
ON
    iac.sector = imh.sector AND iac.industry = imh.industry AND iac.year
   = imh.year
JOIN
    industry_max_volume imv
ON
    iac.sector = imv.sector AND iac.industry = imv.industry AND iac.year
   = imv.year
ORDER BY
    iac.sector, iac.industry, iac.year DESC
")
```

```
output = spark.sql("SELECT * FROM output")
```

5 Job 3

Come per le applicazioni precedenti, il job è stato realizzato a partire dal dataset pre-processato (historical_stocks_data) e con quattro diverse tecnologie: Hadoop, Hive, Spark core e Spark SQL.
Il risultato ottenuto è stato memorizzato sul repository github, ecco riportate le prime dieci righe dell'**output**:

Tickers	Trends	Years
AA, ARNC	-36.2, 61.4, -16.4	2002, 2003, 2004
AA, ARNC	61.4, -16.4, -4.6	2003, 2004, 2005
AA, ARNC	-16.4, -4.6, 0.4	2004, 2005, 2006
AA, ARNC	-4.6, 0.4, 24.6	2005, 2006, 2007
AA, ARNC	0.4, 24.6, -68.8	2006, 2007, 2008
AA, ARNC	24.6, -68.8, 33.1	2007, 2008, 2009
AA, ARNC	-68.8, 33.1, -7.6	2008, 2009, 2010
AA, ARNC	33.1, -7.6, -45.3	2009, 2010, 2011
AA, ARNC	-7.6, -45.3, -6.0	2010, 2011, 2012
AA, ARNC	-45.3, -6.0, 18.2	2011, 2012, 2013

Table 3: Output job 3

5.1 Hadoop - MapReduce

Come per i primi due job, il codice è stato realizzato in python in due classi mapper.py e reducer.py, nella cartella **job3/HADOOP** del progetto.

- Mapper (**mapper.py**) ha l'obiettivo di preparare i dati di input per il reducer, estraendo solo i campi necessari e riorganizzandoli:

Algorithm 10 Mapper

```
1: CREA un dizionario stock_data con valori predefiniti contenenti una
   lista di prezzi di chiusura e un nome di compagnia vuoto
2: for ogni linea nel file di input do
3:     RIMUOVI spazi bianchi dalla linea
4:     SALTA l'header
5:     SEPARA la linea in campi usando ';' come delimitatore
6:     if il numero di campi è uguale a 11 then
7:         ESTRAPOLA i campi: ticker, open, close, low, high, volume,
               date, exchange, name, sector, industry
8:         ESTRAPOLA l'anno dalla data
9:         CONVERTI l'anno in un intero
10:        if l'anno è maggiore o uguale a 2000 then
11:            AGGIUNGI il prezzo di chiusura al dizionario stock_data
                 con chiave (year, ticker)
12:            ASSEGNA il nome della compagnia al dizionario stock_data
13:        end if
14:    end if
15: end for
16: for ogni chiave nel dizionario stock_data do
17:    ORDINA i prezzi di chiusura per data
18:    CREA una lista di prezzi di chiusura
19:    if la lista di prezzi di chiusura ha più di 2 elementi then
20:        ESTRAPOLA il prezzo iniziale e finale
21:        CALCOLA la variazione percentuale tra prezzo finale e prezzo
               iniziale
22:        STAMPA il ticker, anno, variazione percentuale e nome della
             compagnia
23:    end if
24: end for
```

- Reducer (**reducer.py**) ha l'obiettivo di calcolare e confrontare i trend delle aziende negli anni:

Algorithm 11 Reducer

```
1: function MAIN
2:     CREA un dizionario trend_data per memorizzare i dati delle tendenze
3:     CREA un dizionario ticker_year_data per memorizzare i dati delle tendenze per anno e ticker
4:     for ogni linea nel file di input do
5:         SEPARA la linea in campi usando '\t' come delimitatore
6:         CONVERTI l'anno in un intero
7:         CONVERTI la variazione percentuale in float
8:         AGGIUNGI i dati delle tendenze al dizionario trend_data
9:         AGGIUNGI la variazione percentuale al dizionario ticker_year_data per anno e ticker
10:    end for
11:    CREA un dizionario trend_patterns per memorizzare i ticker con gli stessi schemi di tendenza
12:    for ogni ticker e dati annuali in ticker_year_data do
13:        ORDINA gli anni
14:        for ogni set di 3 anni consecutivi do
15:            if i 3 anni sono consecutivi then
16:                CREA una chiave unica per il pattern di tendenza
17:                AGGIUNGI il ticker al dizionario trend_patterns
18:            end if
19:        end for
20:    end for
21:    for ogni chiave e ticker in trend_patterns do
22:        if ci sono più di un ticker then
23:            STAMPA i ticker e il pattern di tendenza con gli anni
24:        end if
25:    end for
26: end function
```

5.2 Hive

Il codice è stato realizzato in due file hql nella cartella **job3/HIVE** del progetto.

- **datasetLoading.hql** ha l'obiettivo di caricare il dataset di input su hive:

```
-- Create a Hive table for the input data
CREATE TABLE IF NOT EXISTS stock_data (
    `ticker` STRING,
    `open` DOUBLE,
    `close` DOUBLE,
    `low` DOUBLE,
    `high` DOUBLE,
    `volume` INT,
    `date` STRING,
    `exchange` STRING,
    `name` STRING,
    `sector` STRING,
    `industry` STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ';'
STORED AS TEXTFILE;

-- Load data into Hive table from hdfs
LOAD DATA INPATH '/input/historical_stocks_data.csv' INTO TABLE
   stock_data;
```

- **query.hql** ha l'obiettivo di calcolare i trend delle azioni negli anni (accorpando ticker con stessi trend) facendo delle interrogazioni SQL:

```
-- Add the year field to the stock_data table
CREATE TABLE stock_data_year AS
SELECT
    `ticker`,
    `open`,
    `close`,
    `low`,
```

```
    `high`,
    `volume`,
    `date`,
    `exchange`,
    `name`,
    `sector`,
    `industry`,
    YEAR(TO_DATE(`date`, 'yyyy-MM-dd')) AS `year`
FROM stock_data
WHERE CAST(YEAR(TO_DATE(`date`, 'yyyy-MM-dd')) AS INT) >= 2000;

-- Table to store intermediate results
CREATE TABLE stock_changes AS
SELECT
    `ticker`,
    `year`,
    MIN(CAST(`close` AS DOUBLE)) AS `start_price`,
    MAX(CAST(`close` AS DOUBLE)) AS `end_price`,
    `name`
FROM stock_data_year
GROUP BY `ticker`, `year`, `name`;

-- Calculation of percentage changes
CREATE TABLE stock_percentage_changes AS
SELECT
    `ticker`,
    `year`,
    ((`end_price` - `start_price`) / `start_price`) * 100 AS `
   percentage_change`,
    `name`
FROM stock_changes
WHERE `start_price` IS NOT NULL AND `end_price` IS NOT NULL;

-- Find 3 consecutive year patterns
CREATE TABLE stock_trends AS
SELECT
    s1.`ticker`,
    s1.`year` AS `year1`,
    s2.`year` AS `year2`,
    s3.`year` AS `year3`,
    s1.`percentage_change` AS `change1`,
```

```
    s2.`percentage_change` AS `change2`,
    s3.`percentage_change` AS `change3`,
    s1.`name`
FROM
    `stock_percentage_changes` s1
JOIN
    `stock_percentage_changes` s2 ON s1.`ticker` = s2.`ticker` AND s1.`
   year` + 1 = s2.`year`
JOIN
    `stock_percentage_changes` s3 ON s1.`ticker` = s3.`ticker` AND s1.`
   year` + 2 = s3.`year`
ORDER BY
    s1.`ticker`, s1.`year`;

-- Select tickers with the same 3-year trend patterns
CREATE TABLE output AS
SELECT
    COLLECT_SET(s1.`ticker`) AS `tickers`,
    CONCAT('[', s1.`percentage_change`, ', ', s2.`percentage_change`, '
   ', s3.`percentage_change`, ']') AS `changes`,
    CONCAT('[', s1.`year`, ', ', s2.`year`, ', ', s3.`year`, ']') AS `
   years`
FROM
    stock_percentage_changes s1
JOIN
    stock_percentage_changes s2 ON s1.`ticker` = s2.`ticker` AND s1.`year
   ` + 1 = s2.`year`
JOIN
    stock_percentage_changes s3 ON s1.`ticker` = s3.`ticker` AND s1.`year
   ` + 2 = s3.`year`
GROUP BY
    s1.`year`, s2.`year`, s3.`year`, s1.`percentage_change`, s2.`
   percentage_change`, s3.`percentage_change`
HAVING SIZE(COLLECT_SET(s1.`ticker`)) > 1;

SELECT * FROM output;
```

5.3 Spark Core

Il codice è stato realizzato in python nel file job3.py nella cartella **job3/SPARKcore** del progetto.

Algorithm 12 SPARKCore/job3.py: PART 1

```
1: function PARSE_LINE(linea)
2:     ESTRAI i campi dalla linea usando il delimitatore ';'
3:     SALTA l'header e le linee malformate
4:     ASSEGNA i valori estratti alle variabili: ticker, open, close, low,
           high, volume, date, exchange, name, sector, industry
5:     OTTIENI l'anno dalla data
6:     CONVERTI year e close in int e float rispettivamente
7:     if year ≥ 2000 then
8:         return (ticker, (year, close, name))
9:     else
10:        return None
11:    end if
12: end function
13: function CALCULATE_PERCENTAGE_CHANGE(data)
14:    ORDINA i dati per date
15:    if la lunghezza dei dati > 1 then
16:        OTTIENI il prezzo iniziale e finale
17:        CALCOLA la variazione percentuale
18:        OTTIENI l'anno e il nome della compagnia
19:        return (year, percent_change, name)
20:    else
21:        return None
22:    end if
23: end function
24: function PREPARE_PATTERN_DATA(data)
25:    ASSEGNA i dati a ticker e year_percentage
26:    OTTIENI year, percentage_change, name da year_percentage
27:    return (ticker, (year, percentage_change, name))
28: end function
```

Algorithm 13 SPARKCore/job3.py: PART 2

```
1: function FIND_PATTERNS(data)
2:     ASSEGNA i dati a ticker e year_data
3:     ORDINA year_data per year
4:     INIZIALIZZA una lista patterns
5:     for ogni i da 0 a (lunghezza di year_data - 3) do
6:         OTTIENI year1, change1, name1 da year_data[i]
7:         OTTIENI year2, change2, name2 da year_data[i + 1]
8:         OTTIENI year3, change3, name3 da year_data[i + 2]
9:         if year3 - year1 == 2 then
10:            AGGIUNGI (year1, year2, year3, change1, change2,
                 change3, name1) a patterns
11:        end if
12:    end for
13:    return patterns
14: end function
15:
16: function GROUP_PATTERNS(pattern)
17:    ASSEGNA pattern a years, changes, name
18:    return (years, changes), [name]
19: end function
20:
21: function FORMAT_OUTPUT(data)
22:    ASSEGNA data a (years, changes), tickers
23:    return una stringa formattata con tickers, changes, years
24: end function
25: CREA una sessione Spark
26: LEGGI i dati di input da input_path
27: RIMUOVI l'intestazione e FILTRA le linee non valide
28: RAGGRUPPA per ticker e year, e raccogli i prezzi di chiusura
29: CALCOLA le variazioni percentuali e FILTRA i valori None
30: PREPARA i dati per l'analisi dei pattern
31: RAGGRUPPA per ticker
32: TROVA i pattern e FILTRA i valori non validi
33: RAGGRUPPA per pattern e trova i tickers con lo stesso pattern
34: FERMA la sessione Spark
```

5.4 Spark SQL

Il codice è stato realizzato in python nel file job3.py nella cartella **job3/SPARKSQL** del progetto.

```
# Read data from hdfs
stock_data = spark.read.csv('/input/historical_stocks_data.csv', header=
    True, sep=';', inferSchema=True)

# Creates the temporary table that allows you to query the stock_data
    dataframe
stock_data.createOrReplaceTempView("stock_data")

# Table with addition of the year field
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_data_year AS
SELECT
    `ticker`,
    `open`,
    `close`,
    `low`,
    `high`,
    `volume`,
    `date` AS stock_date,
    `exchange`,
    `name`,
    `sector`,
    `industry`,
    YEAR(TO_DATE(stock_date, 'yyyy-MM-dd')) AS `year`
FROM
    stock_data
")

# Table to store intermediate results
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_changes AS
SELECT
    `ticker`,
    `year`,
    MIN(CAST(`close` AS DOUBLE)) AS `start_price`,
    MAX(CAST(`close` AS DOUBLE)) AS `end_price`,
    `name`
```

```
FROM stock_data_year
WHERE CAST(`year` AS INT) >= 2000
GROUP BY `ticker`, `year`, `name`
")

# Calculation of percentage changes
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_percentage_changes AS
SELECT
    `ticker`,
    `year`,
    ((`end_price` - `start_price`) / `start_price`) * 100 AS `
   percentage_change`,
    `name`
FROM stock_changes
WHERE `start_price` IS NOT NULL AND `end_price` IS NOT NULL
")

# Find 3 consecutive year patterns
spark.sql("
CREATE OR REPLACE TEMP VIEW stock_trends AS
SELECT
    s1.`ticker`,
    s1.`year` AS `year1`,
    s2.`year` AS `year2`,
    s3.`year` AS `year3`,
    s1.`percentage_change` AS `change1`,
    s2.`percentage_change` AS `change2`,
    s3.`percentage_change` AS `change3`,
    s1.`name`
FROM
    stock_percentage_changes s1
JOIN
    stock_percentage_changes s2 ON s1.`ticker` = s2.`ticker` AND s1.`year
   ` + 1 = s2.`year`
JOIN
    stock_percentage_changes s3 ON s1.`ticker` = s3.`ticker` AND s1.`year
   ` + 2 = s3.`year`
ORDER BY
    s1.`ticker`, s1.`year`
")
```

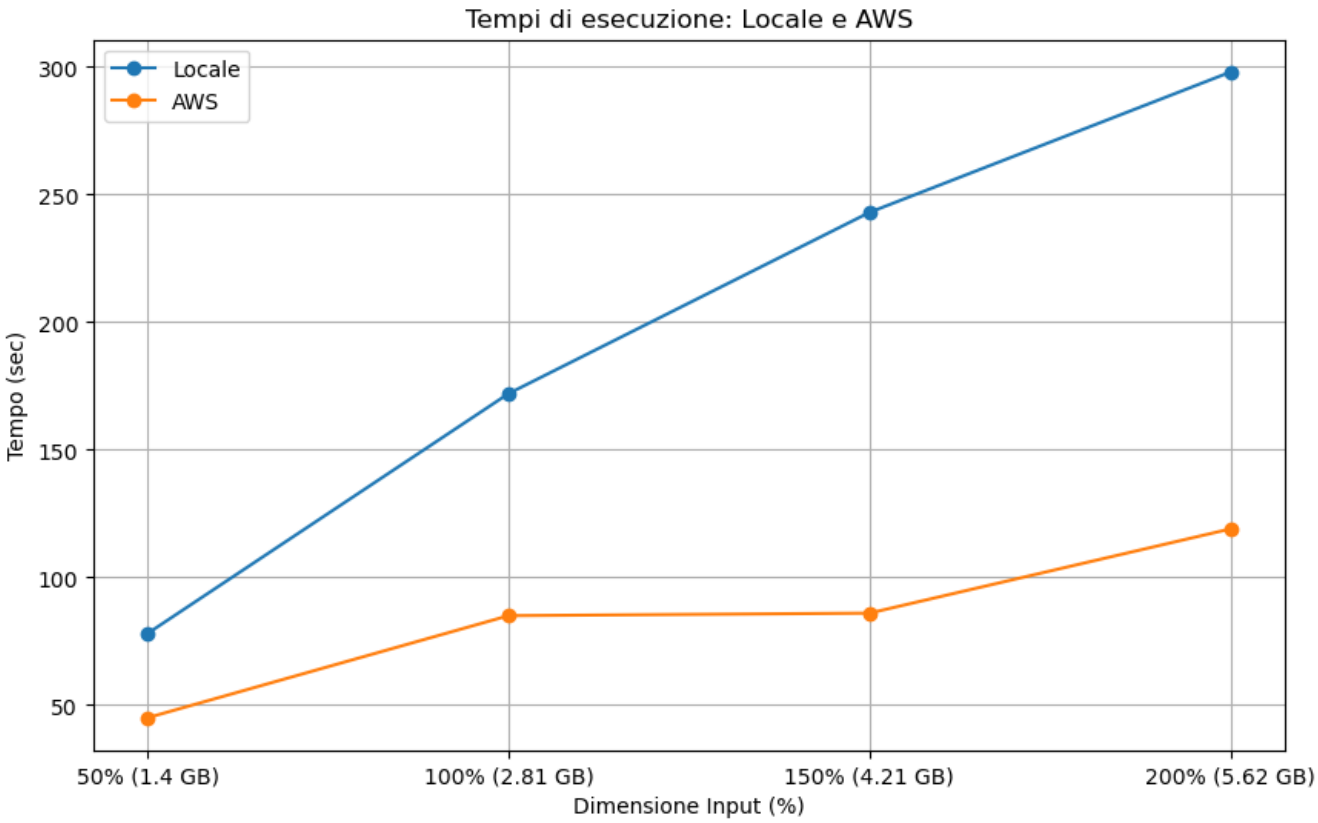
```
# Select tickers with the same 3-year trend patterns
spark.sql("
CREATE OR REPLACE TEMP VIEW output AS
SELECT
    COLLECT_SET(s1.`ticker`) AS `tickers`,
    CONCAT('[', s1.`change1`, ', ', s2.`change2`, ', ', s3.`change3`,
   ']') AS `changes`,
    CONCAT('[', s1.`year1`, ', ', s2.`year2`, ', ', s3.`year3`, ']') AS `
   years`
FROM
    stock_trends s1
JOIN
    stock_trends s2 ON s1.`ticker` = s2.`ticker` AND s1.`year1` + 1 = s2
   .`year2`
JOIN
    stock_trends s3 ON s1.`ticker` = s3.`ticker` AND s1.`year1` + 2 = s3
   .`year3`
GROUP BY
    s1.`year1`, s2.`year2`, s3.`year3`, s1.`change1`, s2.`change2`, s3.`
   change3`
HAVING SIZE(COLLECT_SET(s1.`ticker`)) > 1
")

# Output
output = spark.sql("SELECT * FROM output")
```

6 Risultati

Tutti i job, realizzati con le diverse tecnologie, sono stati eseguiti più volte sia in locale che su AWS con i dataset in input di varie dimensioni. Eseguire il codice in **locale** significa farlo girare sul proprio computer con le risorse limitate alla disponibilità della propria macchina. Eseguire il codice su un cluster con AWS significa utilizzare le risorse computazionali fornite da AWS. Questo può essere fatto attraverso diversi servizi, come EC2 (Elastic Compute Cloud) per le macchine virtuali, EMR (Elastic MapReduce) per l'elaborazione dei dati a grande scala, e altri.
I risultati, a parte alcune eccezioni, sono in linea con le aspettative: a parità

di tecnologia, l'esecuzione è più efficiente su AWS rispetto all'ambiente locale, e all'aumentare della dimensione dell'input aumentano anche i tempi di esecuzione.

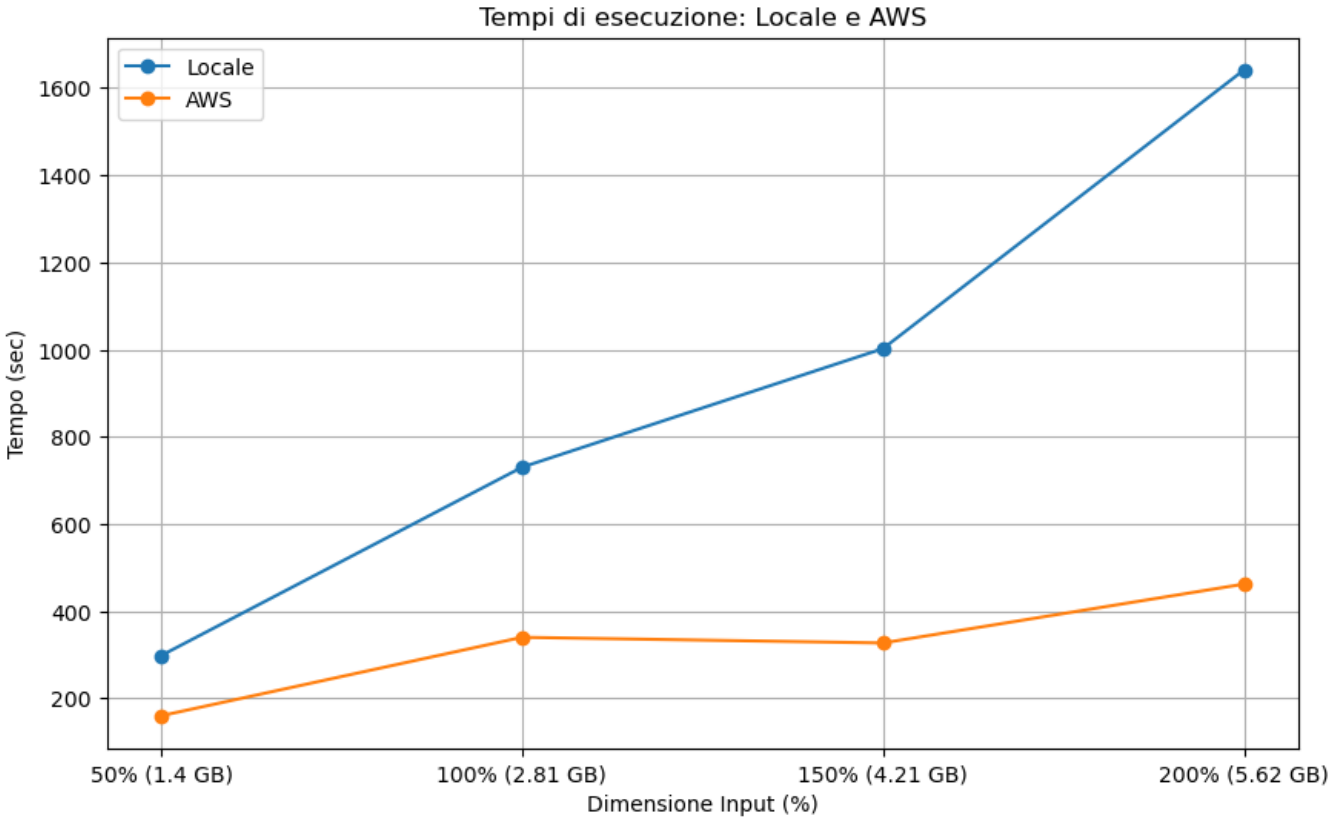
6.1 Job 1 - Hadoop



	50%	100%	150%	200%
LOCALE	78 sec	172 sec	243 sec	298 sec
AWS	45 sec	85 sec	86 sec	119 sec

Table 4: Tempi di esecuzione

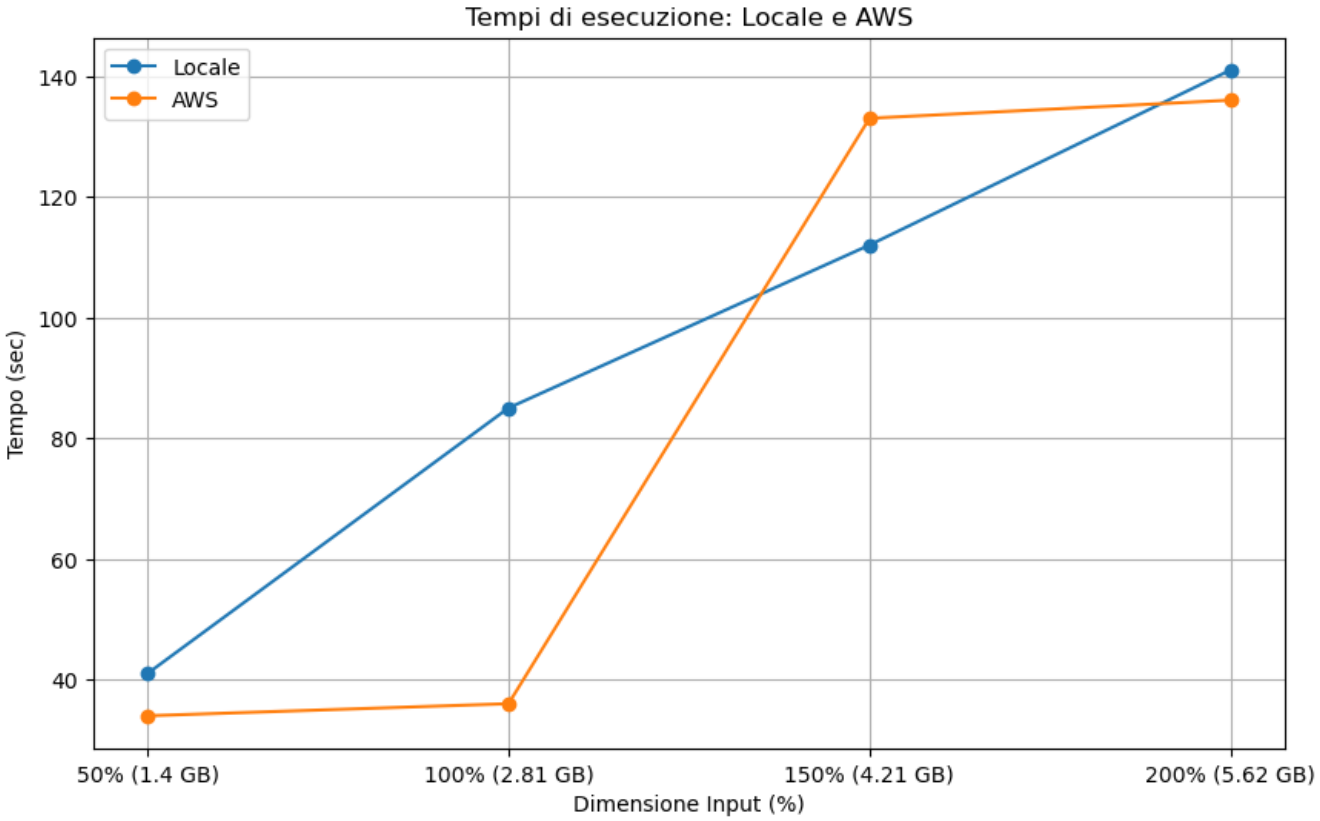
6.2 Job 1 - Hive



	50%	100%	150%	200%
LOCALE	298 sec	730 sec	1002 sec	1641 sec
AWS	160 sec	340 sec	327 sec	462 sec

Table 5: Tempi di esecuzione

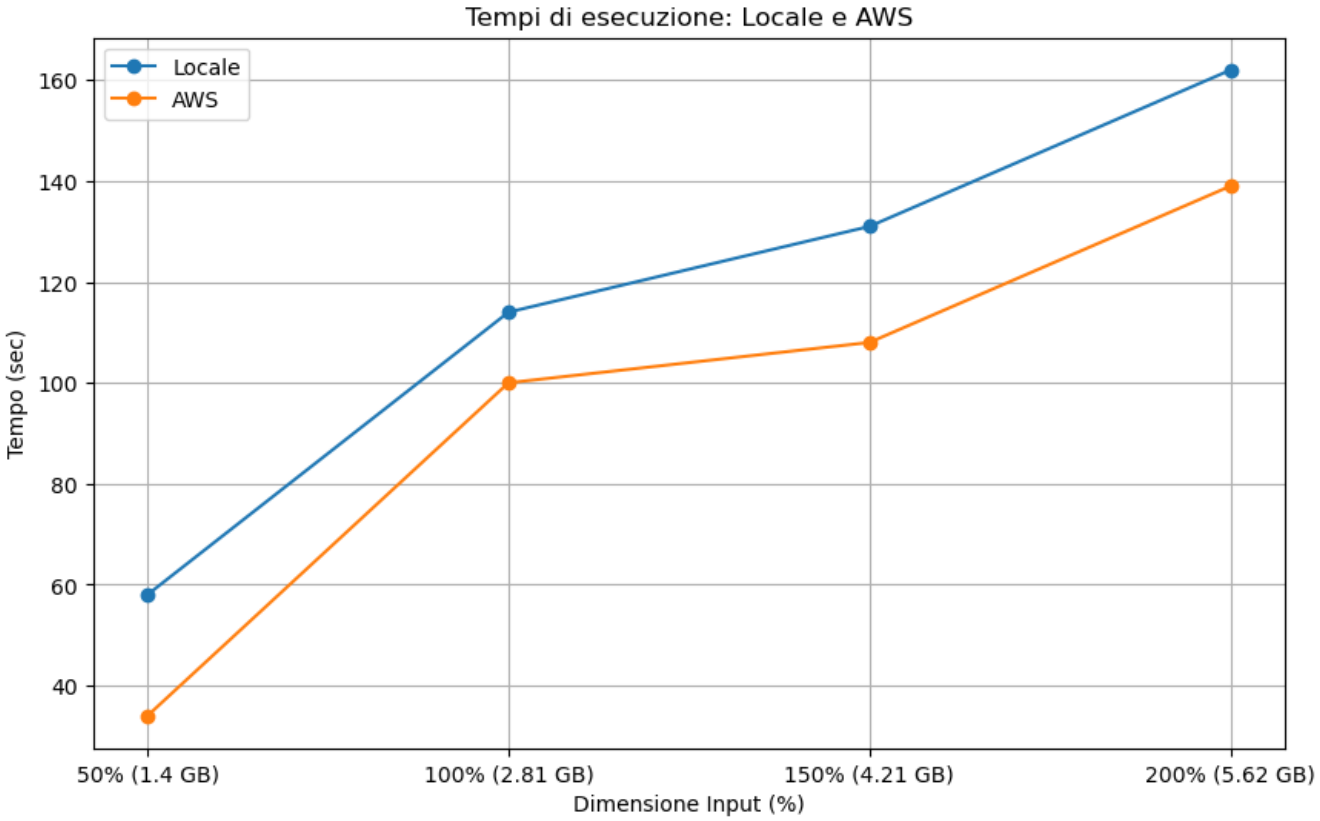
6.3 Job 1 - Spark Core



	50%	100%	150%	200%
LOCALE	41 sec	85 sec	112 sec	141 sec
AWS	34 sec	36 sec	133 sec	136 sec

Table 6: Tempi di esecuzione

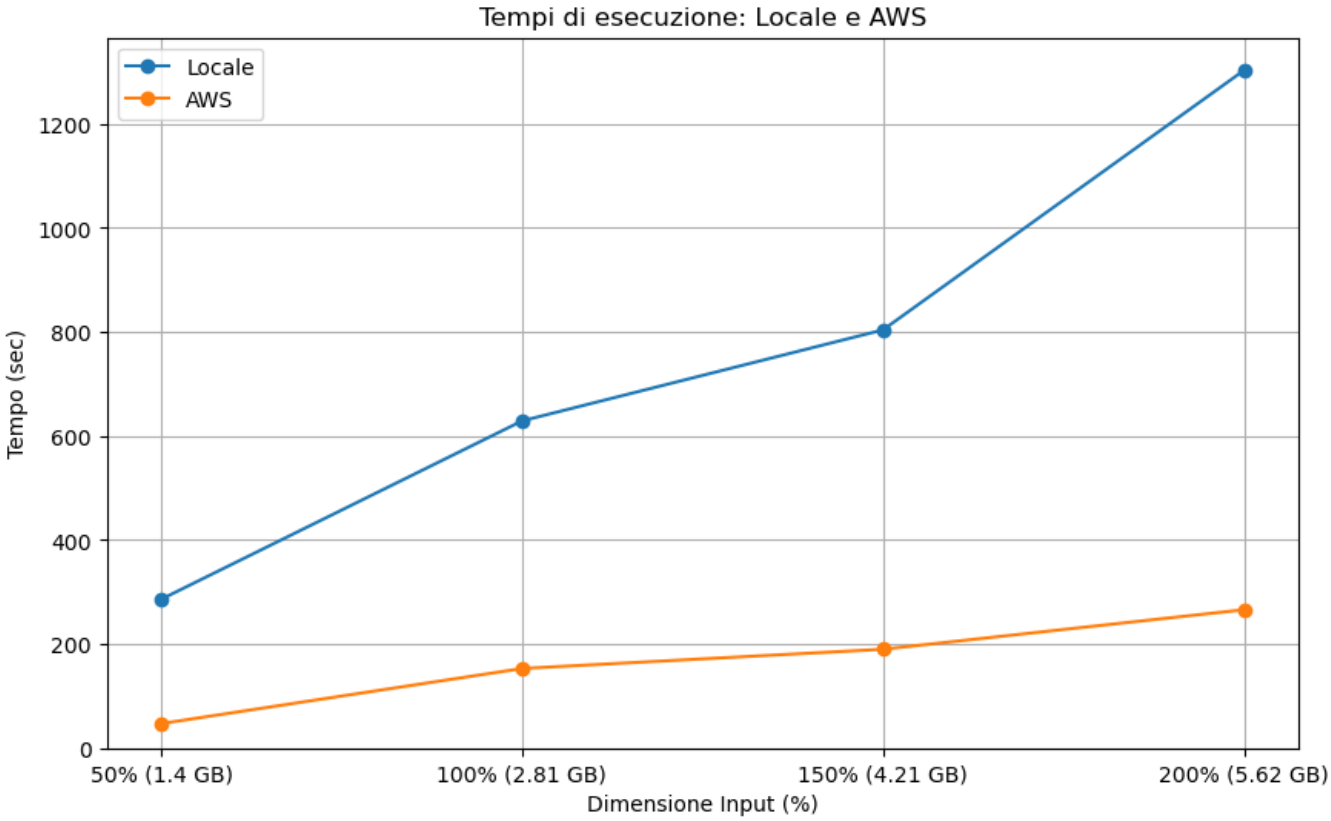
6.4 Job 1 - Spark SQL



	50%	100%	150%	200%
LOCALE	58 sec	114 sec	131 sec	162 sec
AWS	34 sec	100 sec	108 sec	139 sec

Table 7: Tempi di esecuzione

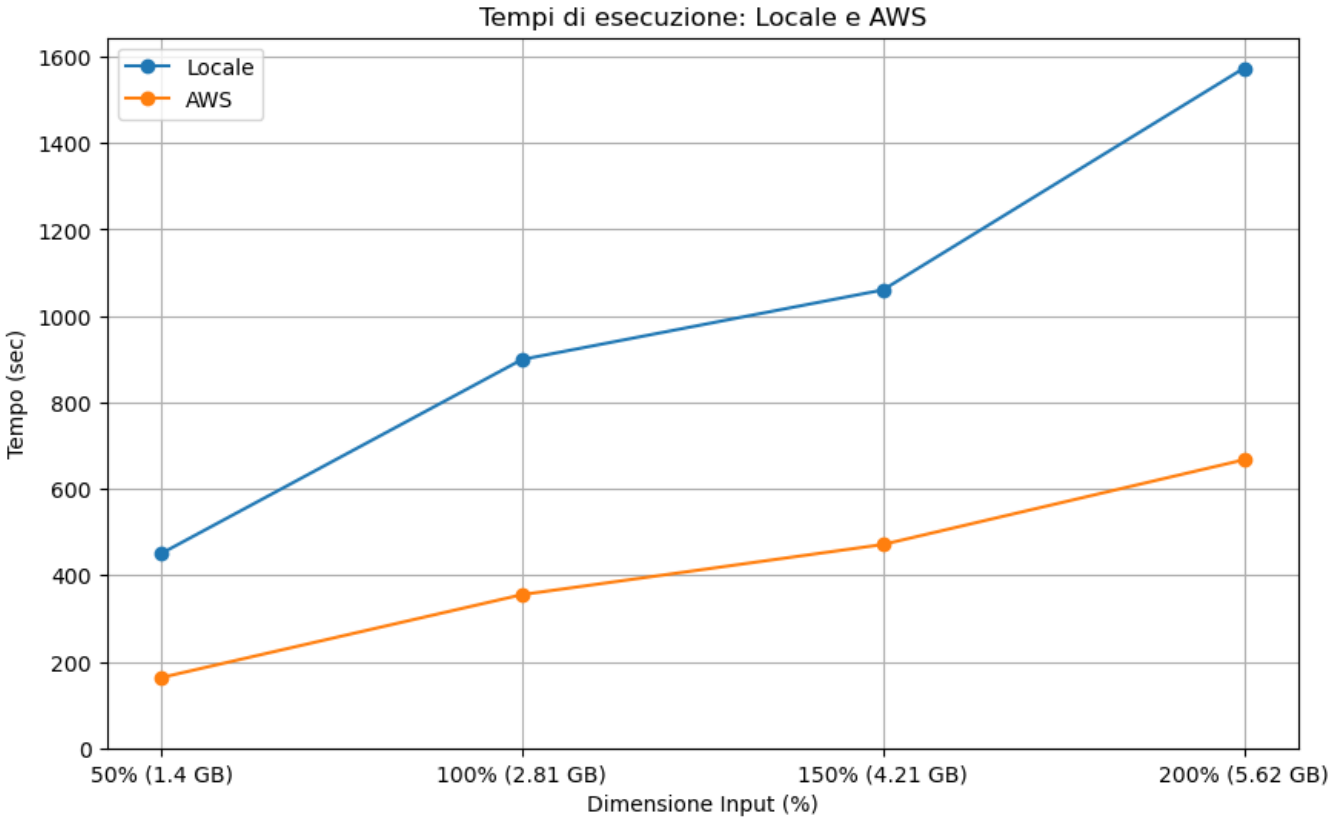
6.5 Job 2 - Hadoop



	50%	100%	150%	200%
LOCALE	286 sec	629 sec	804 sec	1303 sec
AWS	47 sec	153 sec	190 sec	266 sec

Table 8: Tempi di esecuzione

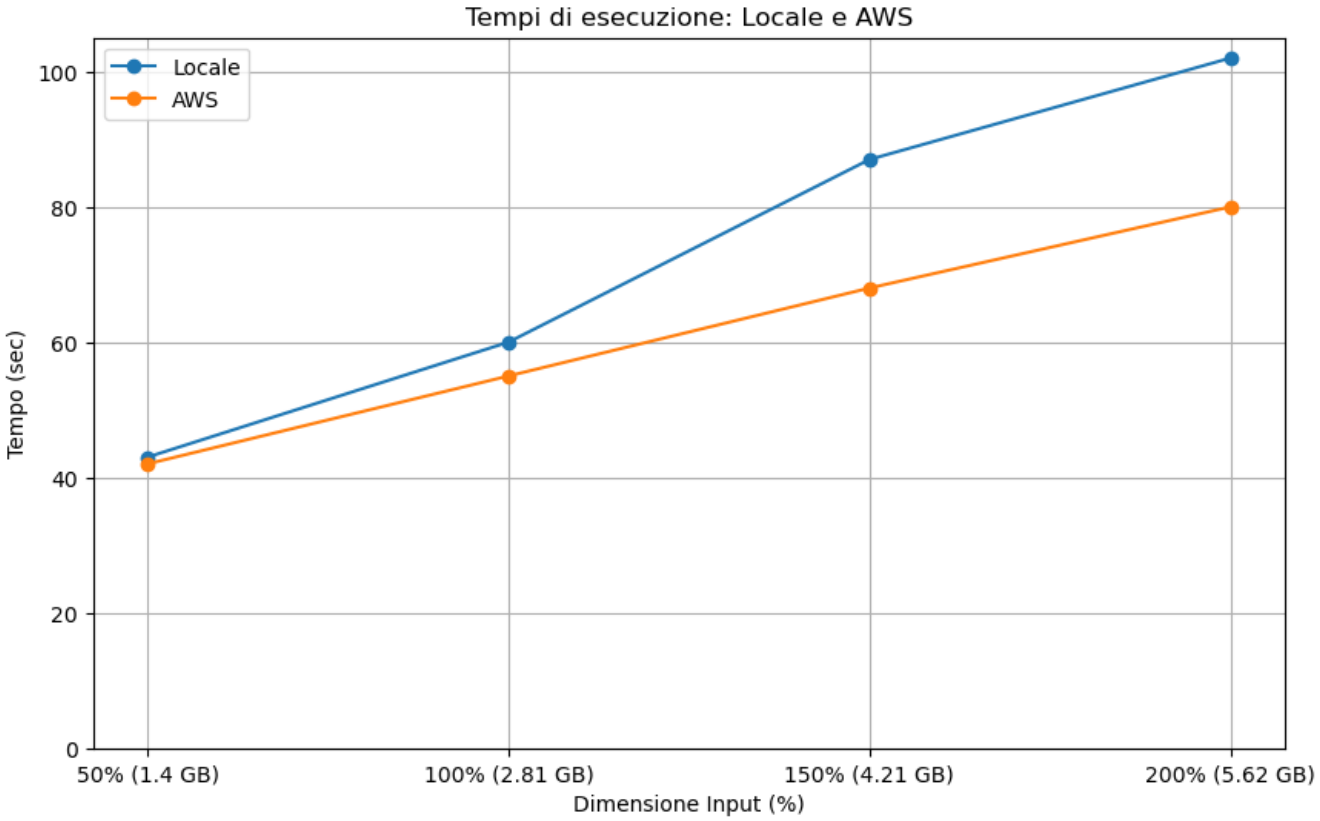
6.6 Job 2 - Hive



	50%	100%	150%	200%
LOCALE	450 sec	899 sec	1060 sec	1573 sec
AWS	163 sec	355 sec	471 sec	667 sec

Table 9: Tempi di esecuzione

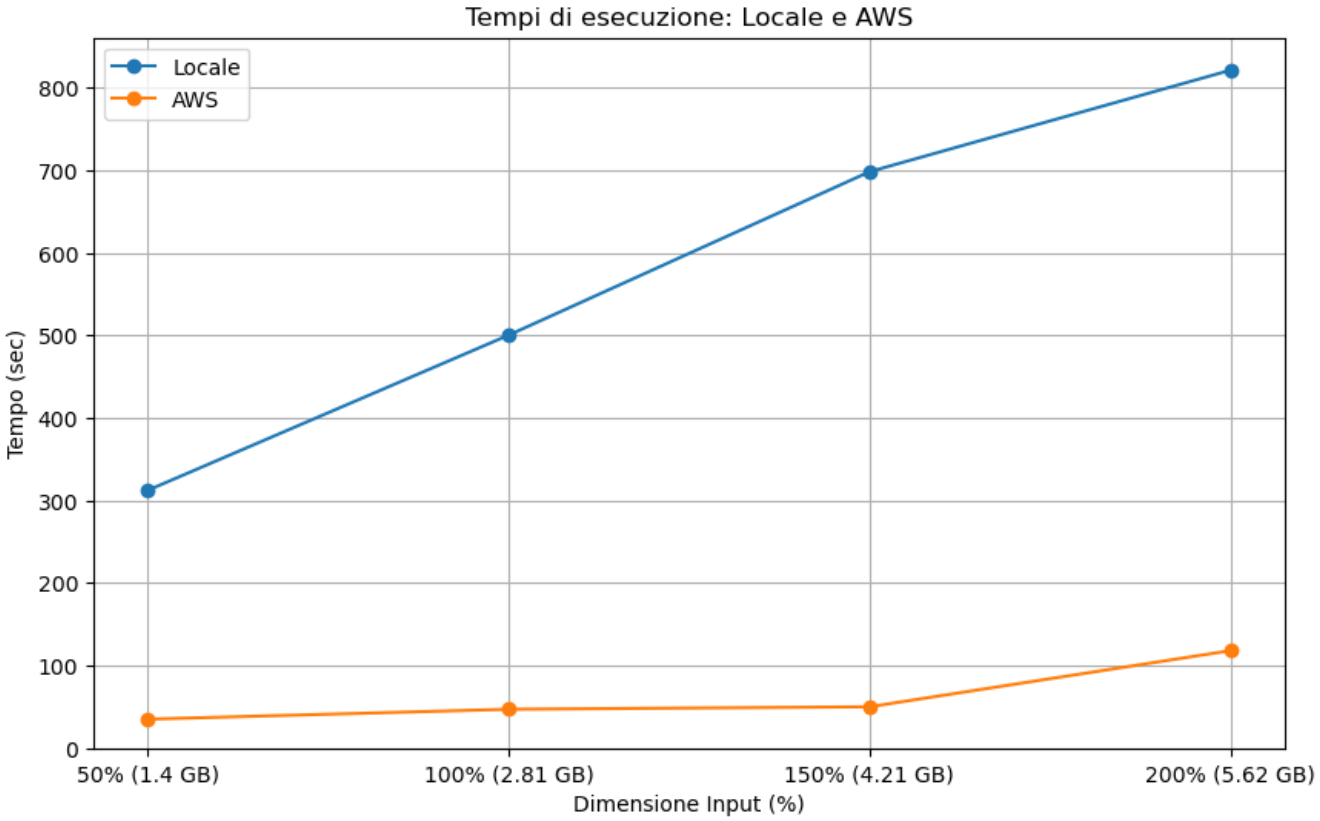
6.7 Job 2 - Spark Core



	50%	100%	150%	200%
LOCALE	43 sec	60 sec	87 sec	102 sec
AWS	42 sec	55 sec	68 sec	80 sec

Table 10: Tempi di esecuzione

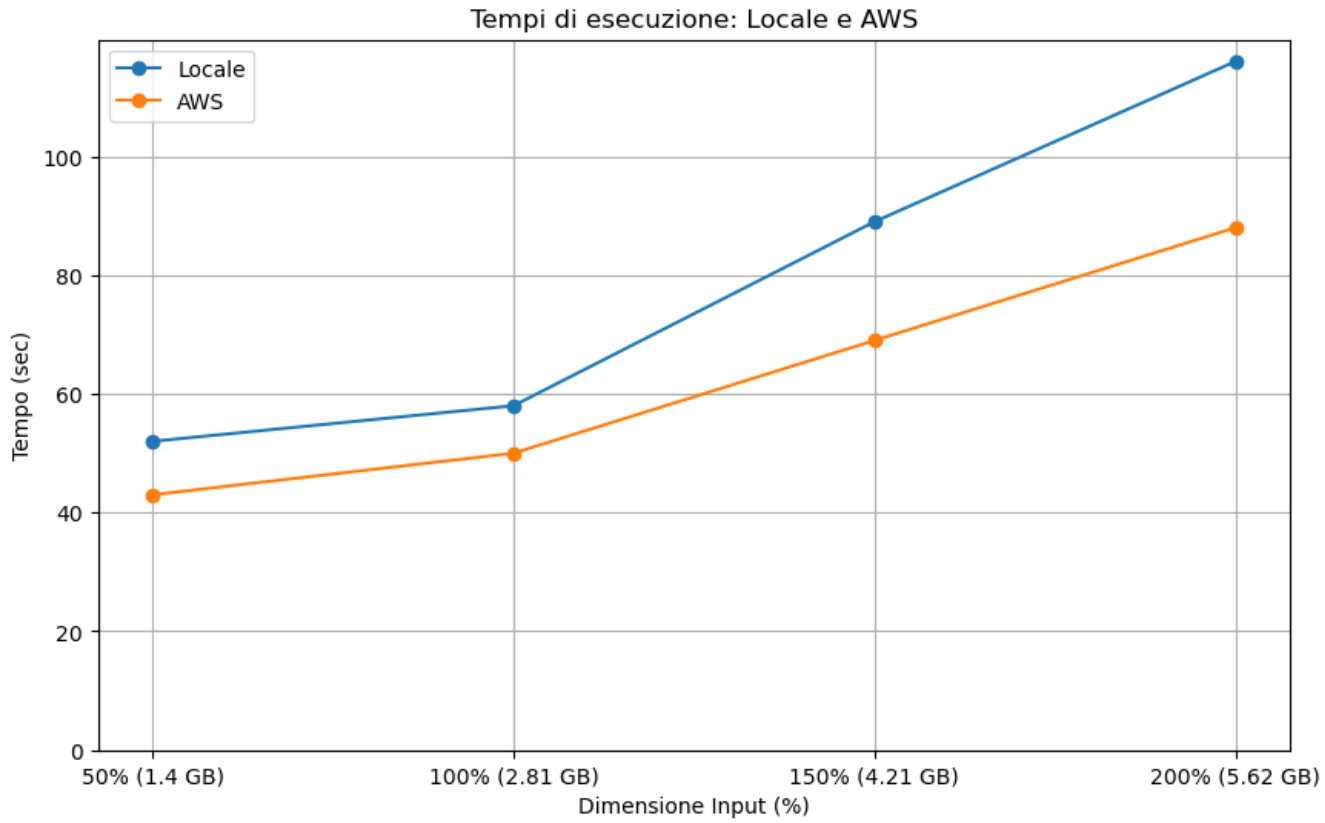
6.8 Job 2 - Spark SQL



	50%	100%	150%	200%
LOCALE	312 sec	500 sec	698 sec	821 sec
AWS	35 sec	47 sec	50 sec	118 sec

Table 11: Tempi di esecuzione

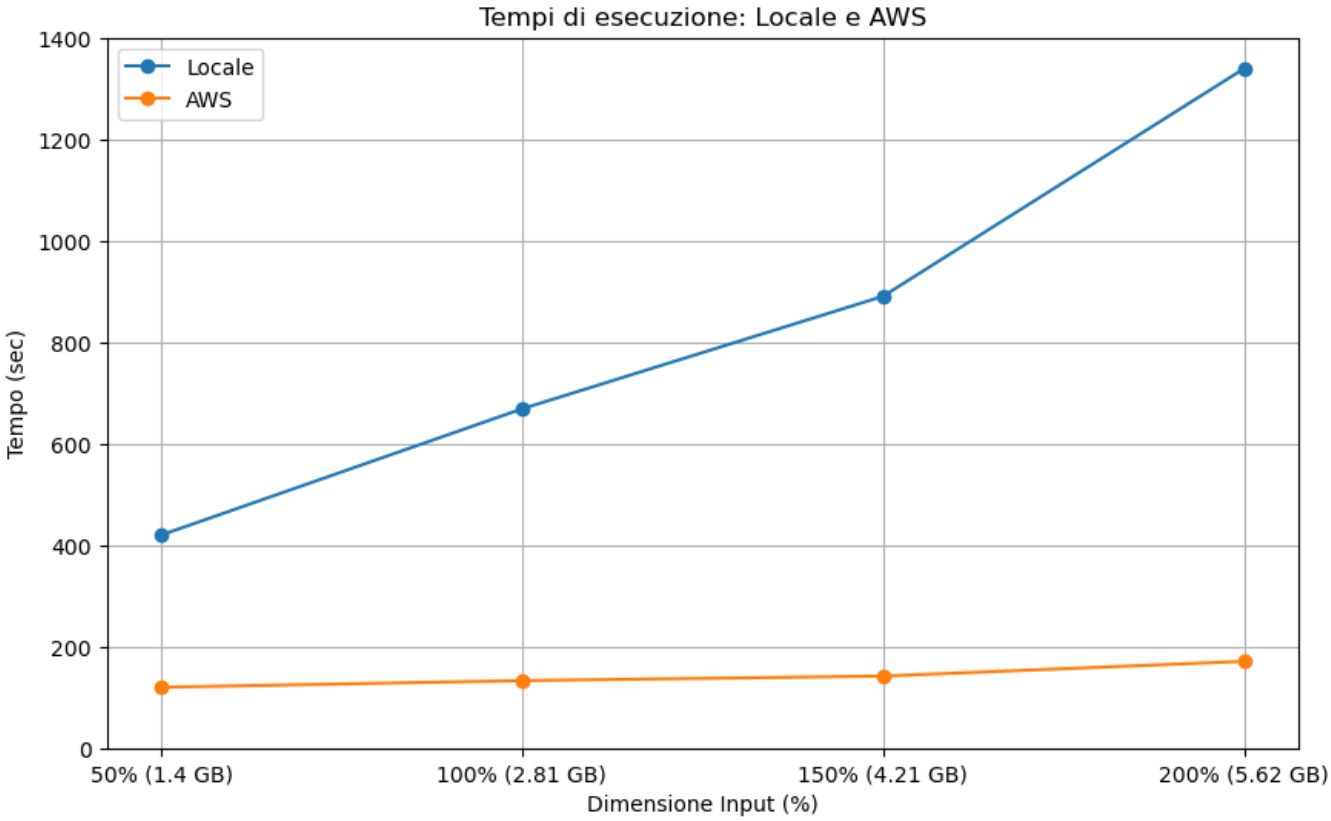
6.9 Job 3 - Hadoop



	50%	100%	150%	200%
LOCALE	52 sec	58 sec	89 sec	116 sec
AWS	43 sec	50 sec	69 sec	88 sec

Table 12: Tempi di esecuzione

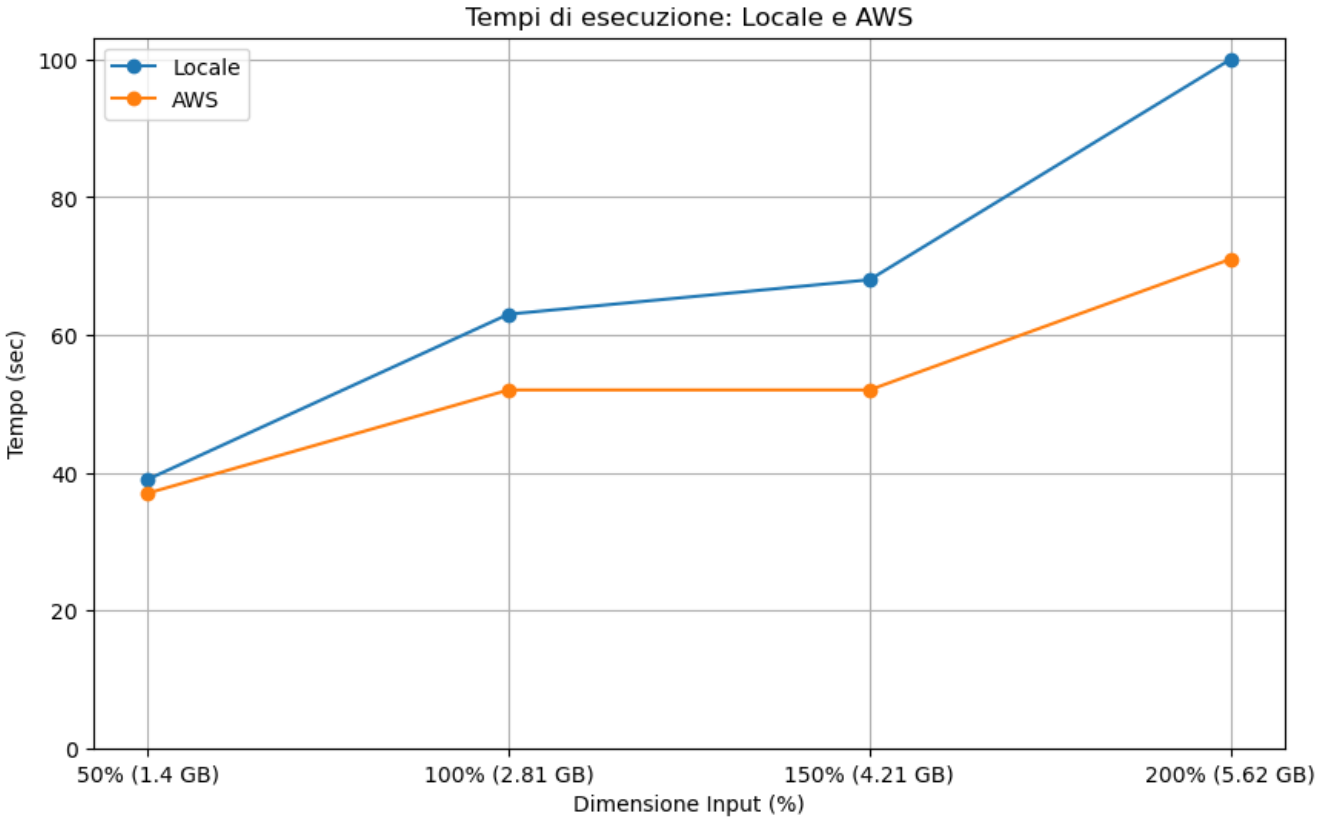
6.10 Job 3 - Hive



	50%	100%	150%	200%
LOCALE	420 sec	668 sec	891 sec	1340 sec
AWS	120 sec	133 sec	142 sec	171 sec

Table 13: Tempi di esecuzione

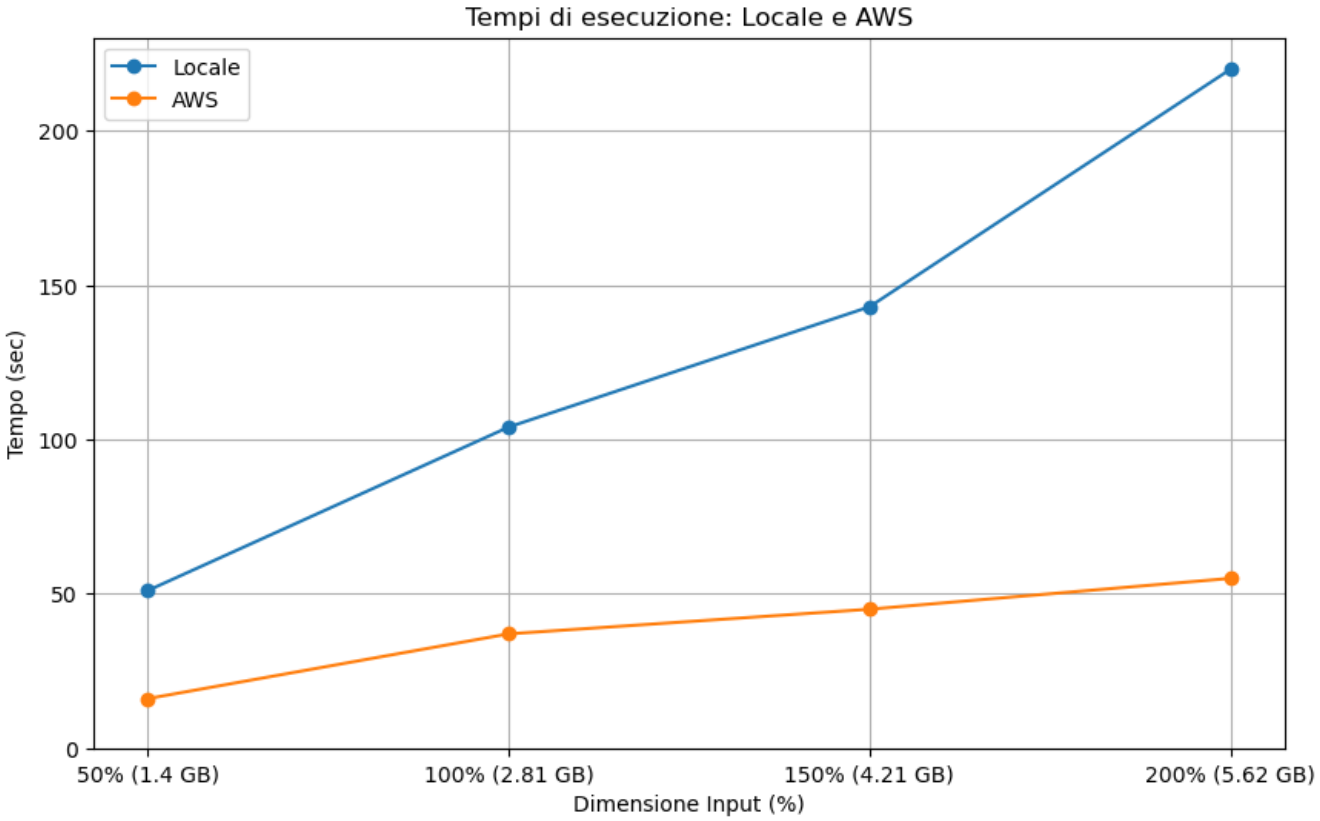
6.11 Job 3 - Spark Core



	50%	100%	150%	200%
LOCALE	39 sec	63 sec	68 sec	100 sec
AWS	37 sec	52 sec	52 sec	71 sec

Table 14: Tempi di esecuzione

6.12 Job 3 - Spark SQL



	50%	100%	150%	200%
LOCALE	51 sec	104 sec	143 sec	220 sec
AWS	16 sec	37 sec	45 sec	55 sec

Table 15: Tempi di esecuzione