

Progetto di Big Data

US Used cars

Giulia Gaglione (559057) e Sara Marrapesa (565910)

A.A. 2024/2025

Indice

1	Il progetto	6
2	Il dataset	7
3	Le tecnologie	11
3.1	Map-Reduce	11
3.2	Spark Core	11
3.3	Spark SQL	11
4	Preparazione del dataset	12
5	Job 1	16
5.1	Benchmarks	16
5.2	Map-Reduce	17
5.2.1	Benchmark del 10%	18
5.2.2	Benchmark del 30%	19
5.2.3	Benchmark del 50%	19
5.2.4	Benchmark del 70%	20
5.2.5	Benchmark del 100%	20
5.3	Spark Core	20
5.3.1	Benchmark del 10%	22
5.3.2	Benchmark del 30%	22
5.3.3	Benchmark del 50%	22

5.3.4	Benchmark del 70%	23
5.3.5	Benchmark del 100%	23
5.4	Spark SQL	24
5.4.1	Benchmark del 10%	25
5.4.2	Benchmark del 30%	26
5.4.3	Benchmark del 50%	26
5.4.4	Benchmark del 70%	27
5.4.5	Benchmark del 100%	27
5.5	Risultati	28
6	Job 2	30
6.1	Benchmarks	30
6.2	Map-Reduce	31
6.2.1	Benchmark del 10%	34
6.2.2	Benchmark del 30%	35
6.2.3	Benchmark del 50%	35
6.2.4	Benchmark del 70%	36
6.2.5	Benchmark del 100%	36
6.3	Spark Core	37
6.3.1	Benchmark del 10%	39
6.3.2	Benchmark del 30%	39
6.3.3	Benchmark del 50%	40
6.3.4	Benchmark del 70%	40
6.3.5	Benchmark del 100%	41
6.4	Spark SQL	42
6.4.1	Benchmark del 10%	44
6.4.2	Benchmark del 30%	44
6.4.3	Benchmark del 50%	45
6.4.4	Benchmark del 70%	46
6.4.5	Benchmark del 100%	46
6.5	Risultati	47
7	AWS	49
7.1	Job 1 - Map-Reduce	49
7.1.1	Benchmark del 10%	49
7.1.2	Benchmark del 30%	49
7.1.3	Benchmark del 50%	50

7.1.4	Benchmark del 70%	50
7.1.5	Benchmark del 100%	51
7.2	Job 1 - Spark Core	51
7.2.1	Benchmark del 10%	51
7.2.2	Benchmark del 30%	52
7.2.3	Benchmark del 50%	52
7.2.4	Benchmark del 70%	53
7.2.5	Benchmark del 100%	53
7.3	Job 1 - Spark SQL	54
7.3.1	Benchmark del 10%	54
7.3.2	Benchmark del 30%	54
7.3.3	Benchmark del 50%	55
7.3.4	Benchmark del 70%	55
7.3.5	Benchmark del 100%	56
7.3.6	Risultati	56
7.4	Job 2 - Map-Reduce	59
7.4.1	Benchmark del 10%	59
7.4.2	Benchmark del 30%	59
7.4.3	Benchmark del 50%	60
7.4.4	Benchmark del 70%	60
7.4.5	Benchmark del 100%	61
7.5	Job 2 -Spark Core	61
7.5.1	Benchmark del 10%	61
7.5.2	Benchmark del 30%	62
7.5.3	Benchmark del 50%	62
7.5.4	Benchmark del 70%	63
7.5.5	Benchmark del 100%	63
7.6	Job 2 - Spark SQL	64
7.6.1	Benchmark del 10%	64
7.6.2	Benchmark del 30%	65
7.6.3	Benchmark del 50%	65
7.6.4	Benchmark del 70%	66
7.6.5	Benchmark del 100%	67
7.7	Job 2 - Risultati	67

8	Confronto	70
8.1	Job 1	70
8.1.1	Map-Reduce	70
8.1.2	Spark Core	71
8.1.3	Spark SQL	72
8.2	Job 2	73
8.2.1	Map-Reduce	73
8.2.2	Spark Core	74
8.2.3	Spark SQL	75
9	Considerazioni finali	76

Elenco degli algoritmi

1	Pulizia dataset usato - Pandas	14
2	Estrazione dataset per Job 1 e Job 2	15
3	Generazione di benchmark da <code>job1_dataset.csv</code>	17
4	Mapper Job1: Estrae chiave e valori da input CSV	17
5	Reducer Job1: calcolo statistiche aggregate per chiave	18
6	SparkCore Job1: aggregazione statistiche per <code>make_name</code> e <code>model_name</code>	21
7	SparkSQL Job1: aggregazione statistiche su <code>make_name</code> e <code>model_name</code>	25
8	Job2: Creazione benchmark dataset da CSV	31
9	Mapper Job2: Classificazione per città, anno e fascia di prezzo	32
10	Reducer Job2: Statistiche per città, anno e fascia di prezzo	34
11	Spark Core Job2: Statistiche per città, anno e fascia di prezzo	38
12	Spark SQL Job2: Statistiche per città, anno e fascia di prezzo	43

Link del git del progetto:

https://github.com/giug2/big_data_project.git

1 Il progetto

La relazione riguarda il secondo progetto del corso di Big Data dell'A.A. 2024/2025. Il progetto ha come obiettivo l'analisi di un ampio dataset di auto usate proveniente da Kaggle, contenente circa 3 milioni di record e oltre 60 attributi per veicolo.

La mole di dati ha reso necessario un accurato processo di preparazione, che ha previsto la pulizia, la normalizzazione e la trasformazione delle informazioni raccolte, al fine di garantire la qualità delle analisi successive.

Successivamente, sono stati implementati due job, ciascuno sviluppato sfruttando tre tecnologie Big Data di grande rilevanza: Map-Reduce, Spark Core e Spark SQL. Questi strumenti sono stati scelti per la loro capacità di gestire e processare grandi volumi di dati in maniera scalabile e performante, permettendo di estrarre insight significativi dal dataset a disposizione.

Il lavoro svolto ha quindi permesso di confrontare diverse metodologie di elaborazione dati distribuita, evidenziando punti di forza e limiti di ciascuna, oltre a fornire risultati concreti utili per ulteriori approfondimenti.

2 Il dataset

Link per il dataset:

<https://www.kaggle.com/datasets/ananyamital/us-used-cars-dataset>

Il dataset utilizzato è il **US Used cars dataset**, disponibile su Kaggle, offre un'ampia raccolta di dati relativi a veicoli usati venduti negli Stati Uniti.

Il dataset contiene circa 3 milioni di record riguardanti auto usate in vendita fino al 2020. Ogni record ha 66 colonne, di cui si dà un elenco di seguito:

1. **vin**: Numero di identificazione del veicolo, una stringa univoca codificata per ogni veicolo.
2. **back_legroom**: Spazio per le gambe nel sedile posteriore.
3. **bed**: Categoria della dimensione del cassone (area di carico aperta) nei pickup. Un valore nullo indica solitamente che il veicolo non è un pickup.
4. **bed_height**: Altezza del cassone in pollici.
5. **bed_length**: Lunghezza del cassone in pollici.
6. **body_type**: Tipo di carrozzeria del veicolo.
7. **cabin**: Categoria della cabina nei pickup.
8. **city**: Città in cui è elencata l'auto.
9. **city_fuel_economy**: Consumo di carburante in città in km per litro.
10. **combine_fuel_economy**: Consumo di carburante combinato (media ponderata tra città e autostrada) in km per litro.
11. **daysonmarket**: Numero di giorni da quando il veicolo è stato elencato per la prima volta sul sito.
12. **dealer_zip**: Codice ZIP del concessionario.
13. **description**: Descrizione del veicolo nella pagina dell'annuncio.
14. **engine_cylinders**: Configurazione del motore (es. I4, V6).
15. **engine_displacement**: Cilindrata del motore, misura del volume totale dei cilindri esclusa la camera di combustione.

16. **engine_type**: Tipo di motore (es. I4, V6).
17. **exterior_color**: Colore esterno del veicolo, solitamente come indicato nella brochure.
18. **fleet**: Indica se il veicolo faceva parte di una flotta.
19. **frame_damaged**: Indica se il telaio del veicolo è danneggiato.
20. **franchise_dealer**: Indica se il concessionario è affiliato a un franchising.
21. **franchise_make**: Marca associata al franchising.
22. **front_legroom**: Spazio per le gambe nel sedile anteriore (in pollici).
23. **fuel_tank_volume**: Capacità del serbatoio del carburante in galloni.
24. **fuel_type**: Tipo di carburante utilizzato dal veicolo.
25. **has_accidents**: Indica se il veicolo ha registrato incidenti.
26. **height**: Altezza del veicolo in pollici.
27. **highway_fuel_economy**: Consumo di carburante in autostrada in km per litro.
28. **horsepower**: Potenza del motore espressa in cavalli.
29. **interior_color**: Colore degli interni del veicolo, come indicato nella brochure.
30. **isCab**: Indica se il veicolo è stato precedentemente utilizzato come taxi.
31. **is_certified**: Indica se il veicolo è certificato, coperto da garanzia.
32. **is_cpo**: Indica se il veicolo è un usato certificato dal concessionario.
33. **is_new**: Indica se il veicolo è stato lanciato da meno di 2 anni.
34. **is_oemcpo**: Indica se il veicolo è un usato certificato dal produttore.
35. **latitude**: Latitudine della posizione del concessionario.
36. **length**: Lunghezza del veicolo in pollici.
37. **listed_date**: Data in cui il veicolo è stato elencato sul sito.

- 38. **listing_color**: Colore dominante dell'esterno del veicolo.
- 39. **listing_id**: ID univoco dell'annuncio sul sito.
- 40. **longitude**: Longitudine della posizione del concessionario.
- 41. **main_picture_url**: URL dell'immagine principale dell'annuncio.
- 42. **major_options**: Principali optional o caratteristiche del veicolo.
- 43. **make_name**: Marca del veicolo.
- 44. **maximum_seating**: Numero massimo di posti a sedere del veicolo.
- 45. **mileage**: Chilometraggio del veicolo al momento dell'annuncio.
- 46. **model_name**: Modello del veicolo.
- 47. **owner_count**: Numero di precedenti proprietari del veicolo.
- 48. **power**: Potenza del motore.
- 49. **price**: Prezzo di vendita del veicolo.
- 50. **salvage**: Indica se il veicolo ha un titolo di recupero.
- 51. **savings_amount**: Importo risparmiato rispetto al prezzo originale o di mercato.
- 52. **seller_rating**: Valutazione del venditore.
- 53. **sp_id**: ID del venditore o del punto vendita.
- 54. **sp_name**: Nome del venditore o del punto vendita.
- 55. **theft_title**: Indica se il veicolo ha un titolo associato a furto.
- 56. **torque**: Coppia del motore.
- 57. **transmission**: Tipo di trasmissione del veicolo.
- 58. **transmission_display**: Descrizione visualizzata della trasmissione.
- 59. **trimId**: ID della versione o allestimento specifico del veicolo.
- 60. **trim_name**: Nome della versione o allestimento specifico del veicolo.
- 61. **vehicle_damage_category**: Categoria di danno del veicolo.

- 62. **wheel_system**: Sistema di trazione del veicolo.
- 63. **wheel_system_display**: Descrizione visualizzata del sistema di trazione.
- 64. **wheelbase**: Passo del veicolo in pollici.
- 65. **width**: Larghezza del veicolo in pollici.
- 66. **year**: Anno di produzione del veicolo.

La dimensione complessiva del dataset è di 9 GB.

3 Le tecnologie

Nell'ambito del progetto sono state utilizzate diverse tecnologie appartenenti all'ecosistema Big Data, con l'obiettivo di fare analisi distribuite su un dataset di grandi dimensioni.

In particolare, sono stati impiegati Map-Reduce, Spark Core e Spark SQL.

3.1 Map-Reduce

Map-Reduce è il modello di programmazione introdotto da Google e implementato in Hadoop per l'elaborazione distribuita di dati su larga scala.

Il paradigma si basa su due fasi principali:

- **Map**: trasforma i dati di input in coppie chiave-valore;
- **Reduce**: aggrega e combina i risultati basati sulle chiavi comuni.

3.2 Spark Core

Apache Spark è un motore di elaborazione distribuita più recente rispetto a Hadoop, noto per le sue elevate prestazioni grazie all'elaborazione in-memory.

Il modulo **Spark Core** fornisce le funzionalità di base per l'elaborazione distribuita, come il supporto per la gestione delle risorse, la distribuzione dei task e le primitive di manipolazione dei dati, come RDD, (Resilient Distributed Dataset).

3.3 Spark SQL

Spark SQL è un modulo di Apache Spark che permette di interrogare i dati utilizzando un linguaggio simile a SQL, con il supporto per l'integrazione diretta con dataframe e sorgenti dati strutturate.

Offre ottimizzazioni avanzate, permettendo di eseguire query complesse con efficienza superiore rispetto ad alternative come Hive.

4 Preparazione del dataset

Tramite l'utilizzo della libreria *pandas* si è eseguita una pulizia preliminare sul dataset messo in analisi, contenuto nel file CSV `used_cars_data.csv`.

Inizialmente, vengono lette solo le colonne rilevanti per le analisi dei due job scelti, ossia: `make_name`, `model_name`, `price`, `year`, `city`, `daysonmarket` e `description`.

Dopo aver caricato i dati, lo script esegue diverse operazioni di pulizia.

- **Valori mancanti:** stampa il numero di valori nulli per ciascuna colonna e il numero di righe contenenti almeno un valore nullo, dopodiché rimuove tutte queste righe.
- **Stringhe vuote:** verifica se i campi `make_name` o `model_name` contengono stringhe vuote (o solo spazi) e rimuove le righe corrispondenti.
- **Prezzi non validi:** elimina le righe in cui il prezzo è minore o uguale a zero.
- **Anni non realistici:** filtra le righe per mantenere solo quelle con un anno compreso tra il 1950 e il 2025, considerando questo intervallo come plausibile per un'auto usata.
- **Outlier sui prezzi:** calcola gli outlier nel campo `price` secondo la regola dell'intervallo interquartile (IQR), ma senza rimuoverli; si limita a segnalarli.

Una volta pulito, il dataset viene salvato in un nuovo file chiamato `dataset_pulito.csv`.

```
Valori nulli per colonna:
city          0
daysonmarket  0
description   77901
make_name     0
model_name    0
price         0
year          0
dtype: int64
Righe con almeno un valore nullo: 77901 (2.60%)
Righe con campi 'make_name' o 'model_name' vuoti: 0 (0.00%)
Righe con prezzo <= 0: 0 (0.00%)
Righe con anni non realistici: 304 (0.01%)
Outlier di prezzo rilevati (ma non rimossi): 84555 (2.89%)

Pulizia completata. Righe finali: 2921835 (da 3000040)
```

Figura 1: Panoramica dell'impatto della pulizia effettuata.

Algorithm 1 Pulizia dataset usato - Pandas

```
1: function LOAD_AND_CLEAN_DATASET(file_path)
2:   Definire le colonne da estrarre: make_name, model_name, price, year,
   city, daysonmarket, description
3:   Leggere il file CSV usando solo le colonne specificate
4:   Salvare il numero totale di righe originali
5:   Calcolare i valori nulli per ciascuna colonna
6:   Contare quante righe contengono almeno un valore nullo
7:   Stampare le informazioni sui valori nulli
8:   Rimuovere le righe con almeno un valore nullo
9:   Identificare le righe con make_name o model_name vuoti (dopo strip)
10:  Contare e stampare queste righe
11:  Rimuovere queste righe
12:  Identificare le righe con price  $\leq 0$ 
13:  Contare e stampare queste righe
14:  Rimuovere queste righe
15:  Identificare le righe con year  $< 1950$  o  $> 2025$ 
16:  Contare e stampare queste righe
17:  Rimuovere queste righe
18:  Calcolare Q1 e Q3 del prezzo
19:  Calcolare  $IQR = Q3 - Q1$ 
20:  Calcolare i limiti inferiore e superiore per outlier
21:  Contare e stampare le righe con prezzo fuori dai limiti (senza rimuoverle)
22:  Resettare l'indice del DataFrame
23:  Salvare il nuovo DataFrame su un file CSV
24:  Stampare il numero finale di righe
25: end function
```

Dopo aver pulito il dataset, sono stati creati due dataset, ciascuno riferito a un job, in cui si tiene conto solo delle colonne di interesse per le analisi statistiche relative al job stesso.

Algorithm 2 Estrazione dataset per Job 1 e Job 2

```
1: Definire il percorso del dataset: dataset_pulito.csv
2: Job 1
3: Definire le colonne da estrarre: make_name, model_name, price, year
4: Leggere il dataset a blocchi (chunksize=10000) usando solo le colonne
   selezionate
5: Inizializzare una lista vuota per accumulare i blocchi
6: for ogni blocco nel dataset do
7:     Aggiungere il blocco alla lista
8: end for
9: Concatenare tutti i blocchi in un unico DataFrame
10: Scrivere il DataFrame in job1_dataset.csv
11: Job 2
12: Definire le colonne da estrarre: city, daysonmarket, description, price,
    year
13: Leggere il dataset a blocchi (chunksize=10000) usando solo le colonne
    selezionate
14: Inizializzare una lista vuota per accumulare i blocchi
15: for ogni blocco nel dataset do
16:     Aggiungere il blocco alla lista
17: end for
18: Concatenare tutti i blocchi in un unico DataFrame
19: Scrivere il DataFrame in job2_dataset.csv
```

5 Job 1

”Un job che sia in grado di generare le statistiche di ciascuna marca di automobile (make_name) presente nel dataset indicando, per ogni marca: (A) il nome della marca e (B) una lista di modelli (model_name) per quella marca indicando, per ciascun modello: (I) il numero di auto presenti nel dataset, (II) il prezzo (price) minimo, massimo e medio di auto di quel modello nel dataset e (III) l’elenco degli anni in cui il modello è presente nel dataset.”

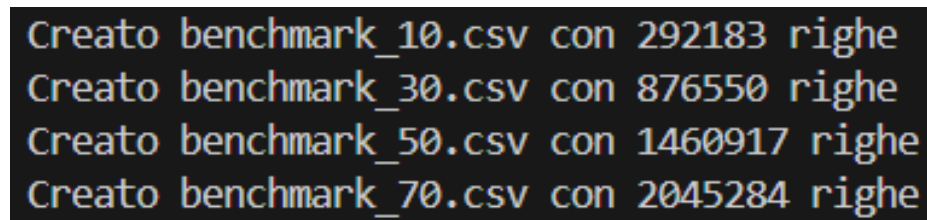
5.1 Benchmarks

Dopo aver pulito il dataset nella fase di preprocessing, si sono creati dei benchmark del dataset relativo al primo job.

Tramite l’utilizzo della libreria *pandas* si è caricato il dataset CSV, si è poi effettuato uno shuffle casuale dei dati e, in seguito, si sono creati dei sottoinsiemi di dati di dimensioni crescenti, salvandoli in file separati.

Il dataset è stato dunque suddiviso in sottoinsiemi secondo quattro percentuali predefinite 10%, 30%, 50% e 70%.

Il dataset completo inerente al job 1 contiene 2921835 record.



```
Creato benchmark_10.csv con 292183 righe  
Creato benchmark_30.csv con 876550 righe  
Creato benchmark_50.csv con 1460917 righe  
Creato benchmark_70.csv con 2045284 righe
```

Figura 2: Divisione del dataset in benchmark.

Algorithm 3 Generazione di benchmark da `job1_dataset.csv`

```
1: Definire il percorso del dataset: Job 1/job1_dataset.csv
2: Caricare il dataset nel DataFrame df
3: Eseguire lo shuffle del dataset con random_state = 42
4: Definire le percentuali di benchmark: {10%, 30%, 50%, 70%}
5: for ogni percentuale  $p$  in {0.1, 0.3, 0.5, 0.7} do
6:   Calcolare il numero di righe:  $n = \lfloor p \cdot \text{len}(\text{df}) \rfloor$ 
7:   Estrarre il sottoinsieme: benchmark = df[0 : n]
8:   Salvare il sottoinsieme come benchmark.p.csv, dove  $p$  è la percentuale
   in formato intero
9: end for
```

5.2 Map-Reduce

Il job viene girato tramite due script differenti: il primo script è il *mapper*, mentre il secondo è il *reducer*.

Il mapper legge per ogni riga i quattro campi dei dati del file CSV: la marca (`make_name`), il modello (`model_name`), il prezzo e l'anno di produzione.

Successivamente, il mapper costruisce una chiave univoca per ogni combinazione marca-modello, concatenandole con il simbolo `#` come separatore.

Algorithm 4 Mapper Job1: Estrae chiave e valori da input CSV

```
1: for ogni riga in input standard do
2:   Leggere la riga e rimuovere spazi bianchi
3:   Dividere la riga sul carattere ' in: make_name, model_name, price, year
4:   Costruire la chiave: key = make_name + "#" + model_name
5:   Stampare su output standard: key, tabulazione, "1", price, year
   separati da tab
6: end for
```

Il reducer si occupa invece di aggregare i dati emessi dal mapper, calcolando per ciascuna chiave (marca#modello) le statistiche desiderate: numero totale di veicoli, prezzo minimo, prezzo massimo, prezzo medio e l'elenco completo degli anni di produzione.

Il reducer scorre ogni riga dell'output prodotto dal mapper ed estrae la chiave e il valore. In particolare, il valore viene parsato per ricavarne il prezzo e l'anno di produzione. Se la chiave corrente cambia rispetto alla precedente, significa che il blocco di dati relativo a quella chiave è terminato, quindi il reducer calcola le statistiche per quel gruppo ed emette l'output aggregato.

L'output finale per ogni marca-modello è una riga che include tutte queste

informazioni in formato leggibile, con gli anni elencati in ordine alfabetico e separati da virgole.

Algorithm 5 Reducer Job1: calcolo statistiche aggregate per chiave

```
1: Inizializzare:
   current_key = None
   count = 0
   total_price = 0.0
   min_price  $\leftarrow +\infty$ 
   max_price  $\leftarrow -\infty$ 
   years  $\leftarrow \{\}$ 
2: function PRINT_STATS
3:   if current_key esiste then
4:     Calcolare avg_price = total_price / count
5:     Ordinare years e convertirla in stringa separata da virgola
6:     Stampare su output:
       current_key: Numero totale auto: count, Prezzo minimo: min_price,
       Prezzo massimo: max_price, Prezzo medio: avg_price, Anni: [years-list]
7:   end if
8: end function
9: for ogni linea in input standard do
10:   Provare a dividere la linea in key e value usando il separatore "\t1"
11:   Dividere value in price_str e year con "\t"
12:   Convertire price_str in float price
13:   if key diversa da current_key e current_key non è None then
14:     Chiamare print_stats()
15:     Reimpostare      count, total_price, min_price, max_price,
       years
16:   end if
17:   Aggiornare current_key = key
18:   Incrementare count += 1
19:   Aggiornare total_price += price
20:   Aggiornare min_price = min(min_price, price)
21:   Aggiornare max_price = max(max_price, price)
22:   Aggiungere year all'insieme years
23:   if errore di parsing then
24:     Saltare la linea
25:   end if
26: end for
27: Chiamare print_stats() per l'ultimo gruppo
```

5.2.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Map-Reduce sono i seguenti:

```

AM General#Hummer      Numero totale auto: 1, Prezzo minimo: 71995.0, Prezzo massimo: 71995.0, Prezzo medio: 71995.00, Anni: [2000]
Acura#CL               Numero totale auto: 2, Prezzo minimo: 2290.0, Prezzo massimo: 3200.0, Prezzo medio: 2745.00, Anni: [2001,2002]
Acura#CLX              Numero totale auto: 261, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23886.67, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#CLX Hybrid       Numero totale auto: 2, Prezzo minimo: 8995.0, Prezzo massimo: 14000.0, Prezzo medio: 11547.00, Anni: [2011]
Acura#Integra          Numero totale auto: 2, Prezzo minimo: 3995.0, Prezzo massimo: 5999.0, Prezzo medio: 4997.00, Anni: [2000,2001]
Acura#MDX              Numero totale auto: 1219, Prezzo minimo: 1700.0, Prezzo massimo: 63745.0, Prezzo medio: 40014.58, Anni: [2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
Acura#MDX Hybrid Sport Numero totale auto: 23, Prezzo minimo: 32285.0, Prezzo massimo: 61175.0, Prezzo medio: 54399.17, Anni: [2017,2018,2020]
Acura#NSX              Numero totale auto: 4, Prezzo minimo: 130000.0, Prezzo massimo: 198795.0, Prezzo medio: 151445.00, Anni: [2003,2018,2019,2020]
Acura#RX               Numero totale auto: 818, Prezzo minimo: 6550.0, Prezzo massimo: 50020.0, Prezzo medio: 34677.24, Anni: [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021]
Acura#RL               Numero totale auto: 12, Prezzo minimo: 3995.0, Prezzo massimo: 15454.0, Prezzo medio: 7908.67, Anni: [2002,2005,2006,2007,2008,2010,2011]

```

Figura 3: Risultato del job 1 in Map-Reduce con benchmark del 10%.

Il job ha impiegato 3.88 secondi.

5.2.2 Benchmark del 30%

I risultati ottenuti dal running del job 1 tramite Map-Reduce sono i seguenti:

```

AM General#Hummer      Numero totale auto: 1, Prezzo minimo: 71995.0, Prezzo massimo: 71995.0, Prezzo medio: 71995.00, Anni: [2000]
Acura#CL               Numero totale auto: 5, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4177.00, Anni: [2001,2002,2003]
Acura#CLX              Numero totale auto: 1362, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 24896.59, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#CLX Hybrid       Numero totale auto: 6, Prezzo minimo: 8995.0, Prezzo massimo: 17999.0, Prezzo medio: 12456.00, Anni: [2011]
Acura#Integra          Numero totale auto: 3, Prezzo minimo: 3995.0, Prezzo massimo: 47900.0, Prezzo medio: 19298.00, Anni: [2000,2001]
Acura#MDX              Numero totale auto: 3568, Prezzo minimo: 1600.0, Prezzo massimo: 63745.0, Prezzo medio: 39941.21, Anni: [2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
Acura#MDX Hybrid Sport Numero totale auto: 65, Prezzo minimo: 28941.0, Prezzo massimo: 61175.0, Prezzo medio: 53995.28, Anni: [2017,2018,2019,2020]
Acura#NSX              Numero totale auto: 12, Prezzo minimo: 118990.0, Prezzo massimo: 198795.0, Prezzo medio: 144456.67, Anni: [2003,2004,2017,2018,2019,2020]
Acura#RX               Numero totale auto: 2434, Prezzo minimo: 4995.0, Prezzo massimo: 50020.0, Prezzo medio: 34523.35, Anni: [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021]
Acura#RL               Numero totale auto: 42, Prezzo minimo: 1999.0, Prezzo massimo: 15454.0, Prezzo medio: 6772.71, Anni: [2000,2002,2004,2005,2006,2007,2008,2009,2010,2011]

```

Figura 4: Risultato del job 1 in Map-Reduce con benchmark del 30%.

Il job ha impiegato 5.86 secondi.

5.2.3 Benchmark del 50%

I risultati ottenuti dal running del job 1 tramite Map-Reduce sono i seguenti:

```

AM General#Hummer      Numero totale auto: 2, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67997.00, Anni: [2000]
Acura#CL               Numero totale auto: 7, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4283.57, Anni: [2001,2002,2003]
Acura#CLX              Numero totale auto: 1362, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 24824.29, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#CLX Hybrid       Numero totale auto: 11, Prezzo minimo: 8995.0, Prezzo massimo: 17999.0, Prezzo medio: 12294.73, Anni: [2013,2014]
Acura#Integra          Numero totale auto: 5, Prezzo minimo: 2695.0, Prezzo massimo: 47900.0, Prezzo medio: 13516.80, Anni: [2000,2001]
Acura#MDX              Numero totale auto: 5829, Prezzo minimo: 1600.0, Prezzo massimo: 63745.0, Prezzo medio: 39775.10, Anni: [2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
Acura#MDX Hybrid Sport Numero totale auto: 108, Prezzo minimo: 28941.0, Prezzo massimo: 61175.0, Prezzo medio: 53897.68, Anni: [2017,2018,2019,2020]
Acura#NSX              Numero totale auto: 20, Prezzo minimo: 94950.0, Prezzo massimo: 198795.0, Prezzo medio: 141162.05, Anni: [2003,2004,2017,2018,2019,2020]
Acura#RX               Numero totale auto: 4108, Prezzo minimo: 4995.0, Prezzo massimo: 50020.0, Prezzo medio: 34295.94, Anni: [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021]
Acura#RL               Numero totale auto: 64, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 6930.31, Anni: [2000,2002,2004,2005,2006,2007,2008,2009,2010,2011]

```

Figura 5: Risultato del job 1 in Map-Reduce con benchmark del 50%.

Il job ha impiegato 6.89 secondi.

5.2.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Map-Reduce sono i seguenti:

```
RM General#Numero Numero totale auto: 2, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67997.00, Anni: [2000]
Acura#CL Numero totale auto: 11, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4070.36, Anni: [2001,2002,2003]
Acura#ILX Numero totale auto: 1877, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23990.73, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#ILX Hybrid Numero totale auto: 15, Prezzo minimo: 8398.0, Prezzo massimo: 17999.0, Prezzo medio: 12031.93, Anni: [2013,2014]
Acura#Integra Numero totale auto: 6, Prezzo minimo: 2695.0, Prezzo massimo: 47900.0, Prezzo medio: 11797.17, Anni: [2000,2001]
Acura#MDX Numero totale auto: 8154, Prezzo minimo: 1600.0, Prezzo massimo: 63745.0, Prezzo medio: 39700.53, Anni: [2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
Acura#MDX Hybrid Sport Numero totale auto: 146, Prezzo minimo: 20841.0, Prezzo massimo: 62170.0, Prezzo medio: 53685.92, Anni: [2017,2018,2019,2020]
Acura#NX Numero totale auto: 29, Prezzo minimo: 94950.0, Prezzo massimo: 198795.0, Prezzo medio: 139103.79, Anni: [2002,2003,2004,2005,2017,2018,2019,2020]
Acura#RDX Numero totale auto: 5707, Prezzo minimo: 4495.0, Prezzo massimo: 50020.0, Prezzo medio: 34351.32, Anni: [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021]
Acura#RL Numero totale auto: 93, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 7080.39, Anni: [2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011]
```

Figura 6: Risultato del job 1 in Map-Reduce con benchmark del 70%.

Il job ha impiegato 9.85 secondi.

5.2.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Map-Reduce sono i seguenti:

```
RM General#Numero Numero totale auto: 3, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67264.67, Anni: [2000]
Acura#CL Numero totale auto: 18, Prezzo minimo: 1000.0, Prezzo massimo: 6000.0, Prezzo medio: 3789.39, Anni: [2001,2002,2003]
Acura#ILX Numero totale auto: 2636, Prezzo minimo: 6995.0, Prezzo massimo: 39305.0, Prezzo medio: 24009.32, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#ILX Hybrid Numero totale auto: 20, Prezzo minimo: 8398.0, Prezzo massimo: 17999.0, Prezzo medio: 12319.09, Anni: [2013,2014]
Acura#Integra Numero totale auto: 6, Prezzo minimo: 2695.0, Prezzo massimo: 47900.0, Prezzo medio: 11797.17, Anni: [2000,2001]
Acura#MDX Numero totale auto: 11660, Prezzo minimo: 1600.0, Prezzo massimo: 63745.0, Prezzo medio: 39763.15, Anni: [2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020]
Acura#MDX Hybrid Sport Numero totale auto: 219, Prezzo minimo: 20841.0, Prezzo massimo: 62170.0, Prezzo medio: 53551.93, Anni: [2017,2018,2019,2020]
Acura#NX Numero totale auto: 43, Prezzo minimo: 94950.0, Prezzo massimo: 198795.0, Prezzo medio: 141731.09, Anni: [2002,2003,2004,2005,2017,2018,2019,2020]
Acura#RDX Numero totale auto: 8173, Prezzo minimo: 4495.0, Prezzo massimo: 50520.0, Prezzo medio: 34383.18, Anni: [2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021]
Acura#RL Numero totale auto: 126, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 6994.63, Anni: [2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011]
```

Figura 7: Risultato del job 1 in Map-Reduce con benchmark del 100%.

Il job ha impiegato 11.87 secondi.

5.3 Spark Core

Tramite l'interfaccia *PySpark*, si è elaborare in modo distribuito il file CSV contenente i dati inerenti al primo job, andando a raggruppare i record per marca e modello dell'auto e calcolando statistiche sul prezzo e gli anni di produzione.

Dopo aver importato le librerie necessarie, il programma accetta un parametro dalla riga di comando, ossia il percorso del file CSV da analizzare. Successivamente, avvia una *sessione Spark*.

Una volta aperta la sessione Spark, il programma carica il file CSV come un insieme di righe, chiamato RDD. La prima riga del file viene identificata come intestazione e viene quindi rimossa dai dati da elaborare.

Nella fase di elaborazione, per ogni riga valida, viene costruita una coppia: da

un lato la chiave composta da marca e modello dell'auto, dall'altro la serie di valori utili per le statistiche, ovvero un contatore di uno per indicare un singolo veicolo, il prezzo della vettura, e infine un set contenente l'anno di produzione. Le righe vengono poi aggregate per chiave, ossia per combinazione di marca e modello. Durante questa fase, Spark somma i contatori per calcolare il numero totale di auto per ogni modello, somma i prezzi per calcolare la media, confronta i prezzi per determinare il minimo e il massimo, e unisce tutti i set di anni per ottenere l'elenco completo degli anni in cui quel modello è stato venduto.

Infine, per ogni modello viene costruito un dizionario che riporta: la marca e il modello, il numero di veicoli, il prezzo minimo, massimo e medio, e la lista ordinata degli anni di produzione.

Il programma stampa a schermo solo i primi dieci risultati ottenuti.

Infine, termina chiudendo la sessione Spark.

Algorithm 6 SparkCore Job1: aggregazione statistiche per `make_name` e `model_name`

1: **Parsing argomenti:**

Ricevere `-input` come percorso file CSV

riga `-output` commentata, serve solo se salvataggio su file locale

2: Avviare sessione Spark con nome "SparkCore_Job1"

3: Caricare file CSV come RDD di righe testuali

4: Estrarre la prima riga come intestazione (header)

5: Filtrare via l'intestazione dall'RDD

6: Trasformare e filtrare i dati:

map dividere ogni riga in lista di colonne usando la virgola come separatore

filter mantenere solo righe con 4 colonne e colonne price e year non vuote

map creare coppia chiave-valore:

chiave = (`make_name`, `model_name`)

valore = (1, prezzo, prezzo, prezzo, insieme_anni)

reduceByKey sommare count e prezzi, calcola min e max prezzi, unire insieme anni

map calcolare statistiche finali:

numero auto,

prezzo minimo,

prezzo massimo,

prezzo medio (arrotondato),

lista ordinata degli anni

7: Stampare a video i primi 10 risultati calcolati

8: Terminare la sessione Spark

5.3.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Spark Core sono i seguenti:

```
[make_name] Audi, model_name: 'A6', num_cars: 326, min_price: 2995.0, max_price: 77210.0, avg_price: 30700.0, years: [1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000]
[make_name] Chevrolet, model_name: 'Camaro', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Honda, model_name: 'Civic', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Lexus, model_name: 'Lexus', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Nissan, model_name: 'Nissan', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] GMC, model_name: 'GMC', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Ford, model_name: 'Ford', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Fiat, model_name: 'Fiat', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
```

Figura 8: Risultato del job 1 in Spark Core con benchmark del 10%.

Il job ha impiegato 1.48 secondi.

```
25/06/07 10:08:33 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 10:08:33 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 1.481494 s
[make_name] Audi, model_name: 'A6', num_cars: 326, min_price: 2995.0, max_price: 77210.0,
```

Figura 9: Tempo impiegato per il job 1 in Spark Core con benchmark del 10%.

5.3.2 Benchmark del 30%

I risultati ottenuti dal running del job 1 tramite Spark Core sono i seguenti:

```
[make_name] Audi, model_name: 'A6', num_cars: 326, min_price: 1995.0, max_price: 77210.0, avg_price: 30700.0, years: [1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000]
[make_name] Chevrolet, model_name: 'Camaro', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Honda, model_name: 'Civic', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Lexus, model_name: 'Lexus', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Nissan, model_name: 'Nissan', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] GMC, model_name: 'GMC', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Ford, model_name: 'Ford', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
[make_name] Fiat, model_name: 'Fiat', num_cars: 1000, min_price: 1995.0, max_price: 19950.0, avg_price: 2000.0, years: [1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980]
```

Figura 10: Risultato del job 1 in Spark Core con benchmark del 30%.

Il job ha impiegato 2.58 secondi.

```
25/06/07 10:14:05 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 10:14:05 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 2.584302 s
25/06/07 10:14:05 INFO SparkContext: SparkContext is stopping with exitCode 0.
[make_name] Audi, model_name: 'A6', num_cars: 326, min_price: 1995.0, max_price: 77210.0,
```

Figura 11: Tempo impiegato per il job 1 in Spark Core con benchmark del 30%.

5.3.3 Benchmark del 50%

I risultati ottenuti dal running del job 1 tramite Spark Core sono i seguenti:

```

{"make_name": "Audi", "model_name": "A8", "num_cars": 1500, "min_price": 100000, "max_price": 197700, "avg_price": 152000, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Chevrolet", "model_name": "Trax", "num_cars": 1500, "min_price": 15000, "max_price": 107700, "avg_price": 36000, "years": [2010, 2016, 2017, 2018, 2019, 2020, 2021]},
{"make_name": "Hyundai", "model_name": "Santa Fe Sport", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 37200, "years": [2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "GMC", "model_name": "Sierra 2500", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 37100, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Jeep", "model_name": "Patriot", "num_cars": 1500, "min_price": 15000, "max_price": 80000, "avg_price": 31000, "years": [2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Mitsubishi", "model_name": "Frontier", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 30000, "years": [1999, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Chevrolet", "model_name": "Silverado 1500", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 31000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F150", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F250", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F350", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},

```

Figura 12: Risultato del job 1 in Spark Core con benchmark del 50%.

Il job ha impiegato 3.84 secondi.

```

25/06/07 10:20:11 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 10:20:11 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 3.841262 s
{"make_name": "Audi", "model_name": "A8", "num_cars": 1500, "min_price": 100000, "max_price": 197700, "avg_price": 152000, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}

```

Figura 13: Tempo impiegato per il job 1 in Spark Core con benchmark del 50%.

5.3.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Spark Core sono i seguenti:

```

{"make_name": "Audi", "model_name": "A8", "num_cars": 1500, "min_price": 100000, "max_price": 197700, "avg_price": 152000, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Chevrolet", "model_name": "Trax", "num_cars": 1500, "min_price": 15000, "max_price": 107700, "avg_price": 36000, "years": [2010, 2016, 2017, 2018, 2019, 2020, 2021]},
{"make_name": "Hyundai", "model_name": "Santa Fe Sport", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 37200, "years": [2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "GMC", "model_name": "Sierra 2500", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 37100, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Jeep", "model_name": "Patriot", "num_cars": 1500, "min_price": 15000, "max_price": 80000, "avg_price": 31000, "years": [2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Mitsubishi", "model_name": "Frontier", "num_cars": 1500, "min_price": 10000, "max_price": 100000, "avg_price": 30000, "years": [1999, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Chevrolet", "model_name": "Silverado 1500", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 31000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F150", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F250", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F350", "num_cars": 1500, "min_price": 10000, "max_price": 115000, "avg_price": 30000, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}

```

Figura 14: Risultato del job 1 in Spark Core con benchmark del 70%.

Il job ha impiegato 5.08 secondi.

```

25/06/07 10:22:10 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 10:22:10 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 5.086048 s
{"make_name": "Audi", "model_name": "A8", "num_cars": 1500, "min_price": 100000, "max_price": 197700, "avg_price": 152000, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}

```

Figura 15: Tempo impiegato per il job 1 in Spark Core con benchmark del 70%.

5.3.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Spark Core sono i seguenti:

```
[{"make_name": "Land Rover", "model_name": "Discovery Sport", "num_cars": 1560, "min_price": 15995.0, "max_price": 70000.0, "avg_price": 41865.75, "years": [2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Land Rover", "model_name": "Range Rover Evoque", "num_cars": 2297, "min_price": 12995.0, "max_price": 49395.0, "avg_price": 30542.19, "years": [2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F Focus", "num_cars": 17556, "min_price": 1100.0, "max_price": 17000.0, "avg_price": 3623.64, "years": [1999, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Ford", "model_name": "F Fiesta", "num_cars": 2009, "min_price": 2000.0, "max_price": 11000.0, "avg_price": 3093.22, "years": [2010, 2017, 2018, 2019, 2020, 2021]},
{"make_name": "Ford", "model_name": "F C-Max", "num_cars": 1356, "min_price": 1100.0, "max_price": 3000.0, "avg_price": 1886.0, "years": [2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]},
{"make_name": "Ford", "model_name": "F C-Max", "num_cars": 2395, "min_price": 5000.0, "max_price": 42000.0, "avg_price": 27642.50, "years": [2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]},
{"make_name": "Hyundai", "model_name": "i30", "num_cars": 2489, "min_price": 800.0, "max_price": 11100.0, "avg_price": 1691.55, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Hyundai", "model_name": "i30", "num_cars": 253, "min_price": 19900.0, "max_price": 54815.0, "avg_price": 33197.76, "years": [2015, 2016, 2017, 2018]},
{"make_name": "Hyundai", "model_name": "i30", "num_cars": 6049, "min_price": 800.0, "max_price": 11300.0, "avg_price": 16654.7, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Hyundai", "model_name": "i30", "num_cars": 6845, "min_price": 1100.0, "max_price": 30000.0, "avg_price": 18732.69, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]},
{"make_name": "Citroen", "model_name": "C3", "num_cars": 10519, "min_price": 800.0, "max_price": 13270.0, "avg_price": 22497.74, "years": [2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}]
```

Figura 16: Risultato del job 1 in Spark Core con benchmark del 100%.

Il job ha impiegato 7.59 secondi.

```
25/06/07 10:24:16 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 10:24:16 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 7.589418 s
{"make_name": "Land Rover", "model_name": "Discovery Sport", "num_cars": 1560, "min_price": 15995.0,
```

Figura 17: Tempo impiegato per il job 1 in Spark Core con benchmark del 100%.

5.4 Spark SQL

Tramite l'interfaccia PySpark, si elabora in modo distribuito il file CSV contenente i dati inerenti al primo job, andando a raggruppare i record per marca e modello dell'auto e calcolando statistiche sul prezzo e gli anni di produzione. Dopo aver importato le librerie necessarie, il programma inizia con la gestione dell'argomento preso da riga di comando che indica il percorso del file CSV da elaborare.

Segue nel codice l'inizializzazione della sessione Spark, configurata per essere eseguita localmente.

Operazione cruciale è la definizione dello schema del dataset che aiuta Spark a interpretare correttamente i dati e a evitare errori di inferenza. Viene infatti esplicitamente indicato che il file CSV contiene quattro colonne: il nome della marca (make_name) e del modello (model_name) dell'auto, il prezzo (price) e l'anno di produzione (year). A ciascuna colonna viene associato un tipo di dato specifico: testo per i nomi, double per il prezzo e intero per l'anno.

Il programma procede così con la lettura del file CSV utilizzando Spark SQL. I dati vengono caricati in un DataFrame e immediatamente registrati come una vista temporanea chiamata "job1.dataset". Questo consente di eseguire query SQL direttamente sul dataset, come se fosse una tabella relazionale.


```
SELECT
    make_name ,
    model_name ,
    COUNT(*) AS numero_auto ,
    MIN(price) AS prezzo_minimo ,
    MAX(price) AS prezzo_massimo ,
    ROUND(AVG(price), 2) AS prezzo_medio ,
    COLLECT_SET(year) AS anni_presenti
FROM job1_dataset
GROUP BY make_name , model_name
ORDER BY make_name , model_name;
```

Dopo la query e dopo aver trasformato l'elenco degli anni, il programma visualizza le prime dieci righe del risultato in console. Dopodiché, chiude la sessione Spark, concludendo l'elaborazione.

Algorithm 7 SparkSQL Job1: aggregazione statistiche su `make_name` e `model_name`

- 1: **Parsing argomenti:**
 - Ricevere `-input` con path al file CSV
 - # riga `-output` commentata, serve solo se salvataggio su file locale
 - 2: Avviare sessione Spark con configurazione `driver.host=localhost` e nome `"Job1_sparksql"`
 - # riga `driver.host=localhost` serve solo il programma viene fatto girare in locale
 - 3: Definire schema esplicito del dataset con tipi:
 - `make_name` (string), `model_name` (string), `price` (double), `year` (int)
 - 4: Leggere il file CSV con lo schema, selezionando le colonne d'interesse
 - 5: Creare una vista temporanea `"job1_dataset"` per query SQL
 - 6: Definire la query riportata sopra
 - 7: Eseguire la query e creare vista temporanea `"model_statistics"`
 - 8: Convertire la colonna degli anni in stringa separata da virgola usando `concat_ws`
 - 9: Mostrare le prime 10 righe con `show()`
 - 10: Terminare la sessione Spark
-

5.4.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

Big Data

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti	anni_presenti_str
AM General	Hummer	5	40995.0	71995.0	50177.8	[1998, 1999, 1993, 2000]	1998, 1999, 1993, 2000
AMC	AMX	3	32500.0	49990.0	39130.0	[1968, 1969, 1970]	1968, 1969, 1970
AMC	Javelin	1	35450.0	35450.0	35450.0	[1974]	1974
AMC	Rambler American	3	9999.0	23900.0	14721.33	[1965, 1964]	1965, 1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	[1961]	1961
Acura	CL	12	1995.0	6800.0	3939.92	[2001, 1998, 2002, 2003, 1999, 1997]	2001, 1998, 2002, 2003, 1999, 1997
Acura	ILX	1362	6995.0	35920.0	24024.28	[2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015]	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	11	8995.0	17999.0	12294.73	[2013, 2014]	2013, 2014
Acura	Integra	6	2695.0	49700.0	111762.5	[2001, 1993, 2000]	2001, 1993, 2000
Acura	Legend	4	4995.0	15999.0	8972.25	[1998, 1989, 1993, 1994]	1998, 1989, 1993, 1994

only showing top 10 rows

Figura 22: Risultato del job 1 in Spark SQL con benchmark del 50%.

Il job ha impiegato 0.29 secondi.

```
25/06/07 09:30:45 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
25/06/07 09:30:45 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 09:30:45 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 0.294098 s
25/06/07 09:30:45 INFO CodeGenerator: Code generated in 23.069474 ms
25/06/07 09:30:45 INFO BlockManagerInfo: Removed broadcast_2_piece0 on localhost:33123 in memory (size: 18.0 KiB, free: 434.4 MiB)
25/06/07 09:30:45 INFO CodeGenerator: Code generated in 27.647562 ms
```

Figura 23: Tempo impiegato per il job 1 in Spark SQL con benchmark del 50%.

5.4.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti	anni_presenti_str
AM General	Hummer	10	40995.0	124095.0	71032.4	[1998, 1995, 1999, 1993, 2000, 1997]	1998, 1995, 1999, 1993, 2000, 1997
AM General	M151A2	1	13995.0	13995.0	13995.0	[1968]	1968
AMC	AMX	3	32500.0	49990.0	39130.0	[1968, 1969, 1970]	1968, 1969, 1970
AMC	Ambassador	1	25900.0	25900.0	25900.0	[1967]	1967
AMC	Javelin	2	25000.0	35450.0	30225.0	[1974]	1974
AMC	Rambler American	5	9999.0	23900.0	14617.0	[1965, 1964]	1965, 1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	[1961]	1961
Acura	CL	18	1943.0	6800.0	3775.61	[2001, 1998, 2002, 2003, 1999, 1997]	2001, 1998, 2002, 2003, 1999, 1997
Acura	ILX	1377	6995.0	35920.0	23999.72	[2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015]	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	15	8396.0	17999.0	12031.93	[2013, 2014]	2013, 2014

only showing top 10 rows

Figura 24: Risultato del job 1 in Spark SQL con benchmark del 70%.

Il job ha impiegato 0.32 secondi.

```
25/06/07 09:32:58 INFO DAGScheduler: Job 1 is finished. Cancelling potential speculative or zombie tasks for this job
25/06/07 09:32:58 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 09:32:58 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 0.325025 s
25/06/07 09:32:58 INFO CodeGenerator: Code generated in 20.653416 ms
25/06/07 09:32:58 INFO CodeGenerator: Code generated in 10.538733 ms
```

Figura 25: Tempo impiegato per il job 1 in Spark SQL con benchmark del 70%.

5.4.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti	anni_presenti_str
AM General	Hummer	17	42933.0	139900.0	74723.76	[1990, 1995, 1999, 1993, 2000, 1997]	1990,1995,1999,1993,2000,1997
AM General	Humvee	1	31990.0	31990.0	31990.0	[1991]	1991
AM General	M151A2	1	13995.0	13995.0	13995.0	[1968]	1968
AMC	AMX	4	28500.0	49990.0	36472.5	[1968, 1969, 1970]	1968,1969,1970
AMC	Ambassador	1	25900.0	25900.0	25900.0	[1967]	1967
AMC	Concord	1	13999.0	13999.0	13999.0	[1980]	1980
AMC	Javelin	3	25000.0	37995.0	32815.0	[1969, 1974]	1969,1974
AMC	Rambler American	7	9999.0	25999.0	17141.14	[1965, 1966, 1964]	1965,1966,1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	[1961]	1961
Acura	CL	28	1000.0	6000.0	3620.89	[2001, 1998, 2002, 2003, 1999, 1997]	2001,1998,2002,2003,1999,1997

only showing top 10 rows

Figura 26: Risultato del job 1 in Spark SQL con benchmark del 100%.

Il job ha impiegato 0.39 secondi.

```

25/06/07 09:37:40 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
25/06/07 09:37:40 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 09:37:40 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 0.392794 s
25/06/07 09:37:40 INFO CodeGenerator: Code generated in 13.424125 ms
25/06/07 09:37:40 INFO CodeGenerator: Code generated in 9.432041 ms

```

Figura 27: Tempo impiegato per il job 1 in Spark SQL con benchmark del 100%.

5.5 Risultati

Dall'analisi dei tempi di esecuzione del primo job, è evidente che *Spark SQL* è la tecnologia più efficiente tra le tre utilizzate durante il progetto.

La crescita dei tempi con l'aumentare del benchmark è contenuta, confermando la capacità di Spark SQL di gestire query in modo altamente ottimizzato, evitando il più possibile operazioni su disco.

Spark Core ha tempi intermedi tra le due tecnologie, sensibilmente superiori rispetto a Spark SQL, ma comunque inferiori rispetto a MapReduce.

Il motivo principale di questa differenza risiede nel modello di esecuzione di Spark, che sfrutta il mantenimento dei dati in memoria tra le trasformazioni, evitando i costi legati alla scrittura e lettura da disco, tipici invece dell'approccio di MapReduce.

Nonostante ciò, Spark Core non beneficia delle ottimizzazioni specifiche del motore SQL e delle strategie di esecuzione fisica che invece caratterizzano Spark SQL, motivo per cui risulta meno performante in operazioni che possono essere espresse come query.

Infine, *MapReduce* risulta essere la tecnologia meno efficiente. L'incremento, più ripido rispetto alle altre tecnologie, riflette la natura batch-oriented di MapReduce, che prevede la scrittura su disco tra ogni fase del job e non sfrutta l'elaborazione in memoria. Tale approccio, pur offrendo vantaggi in termini di

tolleranza ai guasti, si traduce in tempi di esecuzione più elevati, rendendolo meno adatto a scenari dove la velocità di analisi è una priorità.

Benchmarks	Map-Reduce (s)	Spark Core (s)	Spark SQL (s)
10%	3.88	1.48	0.22
30%	5.86	2.58	0.38
50%	6.89	3.84	0.29
70%	9.85	5.08	0.32
100%	11.87	7.59	0.39

Tabella 1: Tempi di esecuzione per il primo job in base al diverso benchmark e alla diversa tecnologia.

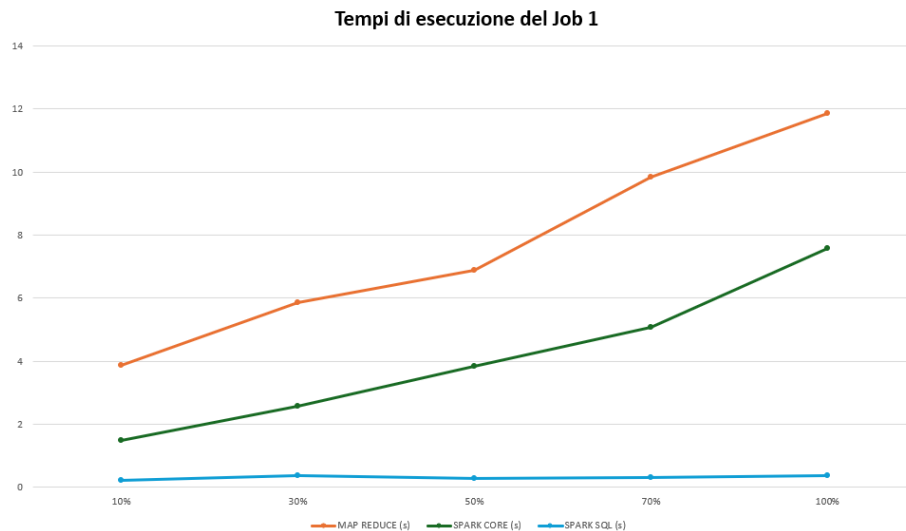


Figura 28: Tempi di esecuzione del job 1 a confronto.

6 Job 2

”Un job che sia in grado di generare un report contenente, per ciascuna città (city) e per ciascun anno (year): il numero di modelli di auto in vendita quell’anno appartenenti a tre fasce di prezzo (alto: sopra i 50K, medio: tra 20K e 50K, basso: inferiore a 20K) indicando, per ciascuna fascia, oltre al numero di auto in quella fascia, la media dei giorni di presenza delle auto sul mercato (daysonmarket) e le tre parole più frequenti che appaiono nella descrizione delle auto (description).”

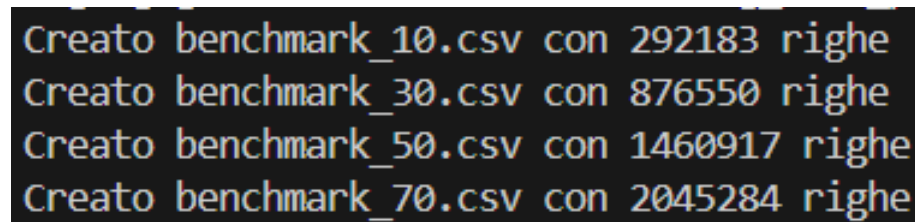
6.1 Benchmarks

Dopo aver pulito il dataset nella fase di preprocessing, si sono creati dei benchmark del dataset relativo al secondo job.

Tramite l’utilizzo della libreria *pandas* si è caricato il dataset CSV, si è poi effettuato uno shuffle casuale dei dati e, in seguito, si sono creati dei sottoinsiemi di dati di dimensioni crescenti, salvandoli in file separati.

Il dataset è stato dunque suddiviso in sottoinsiemi secondo quattro percentuali predefinite 10%, 30%, 50% e 70%.

Il dataset completo inerente al job 2 contiene 2921835 record.



```
Creato benchmark_10.csv con 292183 righe
Creato benchmark_30.csv con 876550 righe
Creato benchmark_50.csv con 1460917 righe
Creato benchmark_70.csv con 2045284 righe
```

Figura 29: Divisione del dataset in benchmark.

Algorithm 8 Job2: Creazione benchmark dataset da CSV

```
1: Input: file job2_dataset.csv
2: Caricare il dataset CSV in un DataFrame
3: Eseguire shuffle dei dati con seme fisso per garantire riproducibilità
4: Definire percentuali per i benchmark: 10%, 30%, 50%, 70%
5: for ogni percentuale  $p$  in  $[0.1, 0.3, 0.5, 0.7]$  do
6:   Calcolare il numero di righe da includere:  $N = \text{int}(\text{len}(df) * p)$ 
7:   Estrarre le prime  $N$  righe dal DataFrame
8:   Salvare il sottoinsieme in un file CSV con nome benchmark_Xp.csv dove
    $X$  è la percentuale
9:   Stampare a schermo conferma della creazione
10: end for
```

6.2 Map-Reduce

Il job viene girato tramite due script differenti: il primo script è il *mapper*, mentre il secondo è il *reducer*.

Il primo script rappresenta la fase mapper e legge riga per riga il file CSV passato da input standard.

All'inizio del programma è definita una funzione che, dato un prezzo, lo classifica in una fascia qualitativa: "basso", "medio" o "alto". Se il prezzo non è valido, la funzione restituisce semplicemente None, così da ignorare eventuali righe problematiche.

Dopo aver definito questa funzione, il programma entra in un ciclo in cui processa ogni riga letta.

Per ogni riga, vengono eliminati eventuali spazi superflui e verificate le condizioni minime per poter elaborare correttamente i dati, in particolare il fatto che ci siano almeno cinque campi. A questo punto, si tenta di convertire i giorni sul mercato in un numero intero e si prova a classificare il prezzo usando la funzione precedentemente definita.

Se entrambe queste operazioni vanno a buon fine, si procede con la pulizia della descrizione: il testo viene convertito tutto in minuscolo e vengono estratte solamente le parole costituite da lettere dell'alfabeto, escludendo numeri, simboli e punteggiatura. Queste parole vengono poi unite tra loro da virgole, così da creare una stringa.

Infine, per ogni riga, il programma stampa una stringa nel formato chiave-valore. La chiave è costruita unendo la città, l'anno e la fascia di prezzo, separati da doppio due punti ::; mentre il valore è composto da: il numero 1, che rappresenta un singolo annuncio, il numero di giorni per cui l'annuncio è rimasto online, e la

descrizione ripulita.

Questo formato è pensato per essere passato al reducer, che si occupa poi di aggregare le informazioni per ciascuna chiave.

Algorithm 9 Mapper Job2: Classificazione per città, anno e fascia di prezzo

```
1: function FASCIA_PREZZO(price)
2:   Try converti price in float
3:   if price  $\geq$  50000 then
4:     return “alto”
5:   else if 20000  $\leq$  price  $\leq$  50000 then
6:     return “medio”
7:   else
8:     return “basso”
9:   end if
10:  Except: return None
11: end function
12: for all line in stdin do
13:   Rimuovere spazi iniziali/finali da line
14:   if line è vuota then continue
15:   Dividere line in fields separati da virgola
16:   if meno di 5 campi then continue
17:   Estrarre city, daysonmarket, description, price, year
18:   Try convertire daysonmarket in intero
19:   Calcolare price_tag = FASCIA_PREZZO(price)
20:   if price_tag è None then continue
21:   Convertire description in minuscolo
22:   Estrarre parole alfabetiche da description con regex
23:   Unire parole con virgola
24:   Print                                city::year::fascia_prezzo \t
      1::daysonmarket::descrizione_pulita
25:   Except: continue
26:
```

Il secondo script costituisce la fase reducer della stessa procedura e legge i dati prodotti dal mapper. Ogni riga contiene dunque una chiave e un valore con le informazioni sull’annuncio, ovvero il contatore, i giorni sul mercato e la lista delle parole nella descrizione.

Il reducer mantiene in memoria la chiave attuale, un contatore degli annunci, un accumulatore per i giorni sul mercato e un dizionario per contare la frequenza delle parole nella descrizione. Finché la chiave rimane la stessa, il reducer aggiorna questi contatori, sommando i giorni e incrementando le occorrenze delle parole.

Quando rileva un cambiamento di chiave, significa che ha terminato di elaborare un gruppo e quindi calcola la media dei giorni sul mercato per quel gruppo. Poi ordina le parole per frequenza e seleziona le tre più comuni, che verranno usate per rappresentare le parole chiave più usate nelle descrizioni.

Dopo aver stampato i risultati, il reducer azzerà gli accumulatori e comincia a lavorare sulla nuova chiave. Alla fine del file, elabora anche l'ultimo gruppo rimasto in memoria.

L'output consiste quindi in righe che mostrano, per ciascuna combinazione di città, anno e fascia di prezzo, il numero totale di annunci, la media dei giorni di permanenza sul mercato, e le tre parole più usate nelle descrizioni.

Algorithm 10 Reducer Job2: Statistiche per città, anno e fascia di prezzo

```
1: function RISULTATO(key, num_auto, tot_giorni, conta_parole)
2:   if num_auto = 0 then
3:     avg_giorni  $\leftarrow$  0
4:   else
5:     avg_giorni  $\leftarrow$  round(tot_giorni / num_auto, 2)
6:   end if
7:   top3  $\leftarrow$  prime 3 parole più frequenti in conta_parole
8:   key_tab  $\leftarrow$  key con “::” sostituiti da tab
9:   Print key_tab, num_auto, avg_giorni, top3-words
10: end function
11: Inizializzare: conta_parole  $\leftarrow$  {}, current_key  $\leftarrow$  None, num_auto  $\leftarrow$  0,
    tot_giorni  $\leftarrow$  0
12: for ogni line da stdin do
13:   Pulire line
14:   if line è vuota then continue
15:   Try separare key e value
16:   Estrarre counter, giorni_mercato, descrizione da value
17:   Except: continue
18:   if current_key  $\neq$  None e key  $\neq$  current_key then
19:     RISULTATO(current_key, num_auto, tot_giorni, conta_parole)
20:     Reset variabili di aggregazione
21:     num_auto  $\leftarrow$  num_auto + counter
22:     tot_giorni  $\leftarrow$  tot_giorni + giorni_mercato
23:     Tokenizzare descrizione su virgole
24:     for ogni parola non vuota do
25:       Incrementare conta_parole[word]
26:     end for
27:     current_key  $\leftarrow$  key
28:
29:     if current_key  $\neq$  None then
30:       RISULTATO(current_key, num_auto, tot_giorni, conta_parole)
31:     end if
```

6.2.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Map-Reduce sono i seguenti:

Abilene	2010	basso	1	19.0	power,car,for
Abilene	2020	medio	1	3.0	zoom,mpg,how
Acton	2019	medio	1	619.0	front,rear,audio
Acworth	2003	basso	1	25.0	w,rear,inc
Acworth	2010	basso	1	51.0	airbags,available,on
Acworth	2013	basso	1	101.0	available,on,and
Acworth	2017	basso	1	8.0	airbags,available,on
Addison	2010	basso	1	491.0	clean,and,to
Adel	2020	alto	1	21.0	the,package,ford
Adel	2020	medio	3	31.0	the,and,equipment

Figura 30: Risultato del job 2 in Map-Reduce con benchmark del 10%.

Il job ha impiegato 9.53 secondi.

6.2.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Map-Reduce sono i seguenti:

Abilene	2010	basso	1	19.0	power,car,for
Abilene	2020	medio	1	3.0	zoom,mpg,how
Acton	2019	medio	1	619.0	front,rear,audio
Acworth	2005	basso	1	4.0	available,on,and
Acworth	2008	basso	1	4.0	airbags,available,on
Acworth	2010	basso	2	56.0	airbags,available,on
Acworth	2011	basso	1	6.0	power,am,fm
Acworth	2013	basso	3	72.0	rear,front,pwr
Acworth	2014	basso	1	101.0	power,airbags,available
Akron	2008	basso	1	26.0	power,wheel,additional

Figura 31: Risultato del job 2 in Map-Reduce con benchmark del 30%.

Il job ha impiegato 21.04 secondi.

6.2.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Map-Reduce sono i seguenti:

Acton	2019	medio	2	323.0	front,rear,audio
Acworth	2002	basso	1	39.0	available,on,and
Acworth	2003	basso	1	25.0	w,rear,inc
Acworth	2005	basso	1	4.0	available,on,and
Acworth	2008	basso	2	6.0	available,on,power
Acworth	2010	basso	3	46.33	available,on,power
Acworth	2011	basso	2	5.0	power,airbags,am
Acworth	2013	basso	4	82.0	rear,power,wheel
Acworth	2014	basso	2	54.0	power,available,on
Acworth	2015	medio	2	64.5	and,front,rear

Figura 32: Risultato del job 2 in Map-Reduce con benchmark del 50%.

Il job ha impiegato 35.73 secondi.

6.2.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Map-Reduce sono i seguenti:

Abilene	2019	basso	1	26.0	this,and,you
Abilene	2020	medio	1	3.0	zoom,mpg,how
Acton	2009	basso	1	35.0	front,rear,audio
Acton	2019	basso	1	583.0	front,rear,audio
Acton	2019	medio	2	323.0	front,rear,audio
Acworth	2002	basso	1	39.0	available,on,and
Acworth	2003	basso	1	25.0	w,rear,inc
Acworth	2005	basso	1	4.0	available,on,and
Acworth	2008	basso	2	6.0	available,on,power
Acworth	2010	basso	3	46.33	available,on,power

Figura 33: Risultato del job 2 in Map-Reduce con benchmark del 70%.

Il job ha impiegato 43.83 secondi.

6.2.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Map-Reduce sono i seguenti:

Abilene	2020	medio	1	3.0	zoom,mpg,how
Acton	2009	basso	1	35.0	front,rear,audio
Acton	2019	basso	3	592.0	front,rear,audio
Acton	2019	medio	2	323.0	front,rear,audio
Acworth	2002	basso	1	39.0	available,on,and
Addison	2007	basso	3	84.0	power,wheel,airbags
Addison	2008	basso	3	16.33	power,air,wheel
Acworth	2005	basso	1	4.0	available,on,and
Acworth	2007	basso	1	37.0	available,on,power
Acworth	2010	basso	4	40.25	airbags,available,power

Figura 34: Risultato del job 2 in Map-Reduce con benchmark del 100%.

Il job ha impiegato 71.77 secondi.

6.3 Spark Core

Dopo aver importato le librerie necessarie, il programma accetta un parametro dalla riga di comando, ossia il percorso del file CSV da analizzare. Successivamente, avvia una sessione Spark.

Una volta aperta la sessione Spark, il programma carica il file CSV come un insieme di righe, chiamato RDD. La prima riga del file viene identificata come intestazione e viene quindi rimossa dai dati da elaborare.

Vengono definite due funzioni ausiliarie, fondamentali per l'analisi: la prima per il calcolo della fascia di prezzo, la seconda l'analisi del campo description e, dopo aver escluso le stopwords della lingua inglese, la restituzione delle tre parole più frequenti.

Dopo aver strutturato i dati in chiavi e valori, il programma procede ad aggregarli per ciascuna combinazione unica di città, anno e fascia di prezzo. Si sommando dunque i conteggi, ottenendo così il numero totale di auto in quella fascia per quella città e anno; si sommano i giorni totali sul mercato, per calcolare successivamente la media; e si concatenano le descrizioni testuali in un'unica grande stringa, che servirà per l'estrazione delle parole più frequenti.

Una volta completata l'aggregazione, ogni gruppo viene ulteriormente elaborato per ottenere i risultati finali, calcolando la media dei giorni sul mercato e le tre parole più frequenti nella descrizione.

Il programma stampa in console le prime dieci righe del risultato. Infine, chiude la sessione Spark.

Algorithm 11 Spark Core Job2: Statistiche per città, anno e fascia di prezzo

```
1: function CATEGORIA_PREZZO(prezzo)
2:   if prezzo > 50000 then
3:     return "alto"
4:   else if prezzo > 20000 then
5:     return "medio"
6:   else
7:     return "basso"
8:   end if
9: end function
10: function TOP3(descrizione)
11:   stopwords ← insieme di parole comuni da ignorare
12:   word_counts ← dizionario vuoto
13:   for ogni parola in descrizione do
14:     Convertire in minuscolo
15:     if alfabetica, più lunga di una lettera, non in stopwords then
16:       Incrementare conteggio in word_counts
17:     end if
18:   end for
19:   Ordinare word_counts per frequenza decrescente
20:   return prime 3 parole
21: end function
22: Avviare sessione Spark
23: Leggere file CSV in RDD
24: Rimuovere intestazione dal dataset
25: Eseguire trasformazioni su RDD:
    • Split su virgola
    • Filtro su righe valide (5 campi, valori numerici dove richiesto)
    • Estrarre: city, daysonmarket, description,  

       fascia_prezzo(prezzo), year
    • Creare chiave: (city, year, fascia_prezzo)
    • Creare valore: (1, daysonmarket, descrizione)
    • Applicare reduceByKey per aggregare:
      – somma numero auto
      – somma giorni sul mercato
      – concatena descrizioni
    • Calcolare media giorni sul mercato
    • Applicare top3() alla descrizione aggregata
26: Stampare le prime 10 righe
27: Fermare la sessione Spark
    =0
```

6.3.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Spark Core sono i seguenti:

```
('Olathe', 2020, 'basso', 5, 178.6, ['air', 'wheel', 'system'])
('Spofford', 2016, 'basso', 1, 75.0, ['power', 'tilt', 'sentry'])
('Hubbard', 2020, 'alto', 2, 22.5, ['front', 'seat', 'rear'])
('Marshall', 2020, 'medio', 1, 232.0, ['seat', 'door', 'chevrolet'])
('Tucson', 2019, 'medio', 8, 78.12, ['door', 'front', 'steering'])
('Topeka', 2014, 'medio', 1, 14.0, ['please', 'call', 'us'])
('Laurel', 2010, 'basso', 2, 202.0, ['dual', 'it', 'is'])
('Surprise', 2020, 'medio', 14, 126.71, ['dual', 'bags', 'head'])
('West', 2020, 'medio', 2, 191.0, ['front', 'rear', 'seat'])
('Sheldon', 2001, 'basso', 1, 1412.0, ['black', 'plow', 'truck'])
```

Figura 35: Risultato del job 2 in Spark Core con benchmark del 10%.

Il job ha impiegato 10.75 secondi.

```
25/06/07 11:55:09 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:55:09 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 10.745783 s
25/06/07 11:55:09 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 36: Tempo impiegato per il job 2 in Spark Core con benchmark del 10%.

6.3.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Spark Core sono i seguenti:

```
('Clearwater', 2020, 'medio', 50, 95.38, ['group'])
('Gaithersburg', 2020, 'medio', 47, 119.91, [])
('Oakhurst', 2020, 'alto', 11, 23.36, ['front', 'seat', 'rear'])
('Greensburg', 2019, 'medio', 8, 171.25, ['front', 'seat', 'rear'])
('Parkersburg', 2019, 'medio', 5, 132.0, ['red', 'tag', 'minimum'])
('Keyport', 2015, 'basso', 1, 35.0, ['power', 'head', 'car'])
('Irwin', 2020, 'medio', 20, 190.15, ['door', 'seat', 'front'])
('Millerton', 2019, 'medio', 2, 420.5, ['front', 'seat', 'manual'])
('Broken Arrow', 1951, 'medio', 1, 46.0, ['at', 'to', 'visit'])
('Butte', 2020, 'medio', 2, 146.5, ['on', 'is', 'vehicle'])
```

Figura 37: Risultato del job 2 in Spark Core con benchmark del 30%.

Il job ha impiegato 17.86 secondi.

```
25/06/07 11:57:27 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:57:27 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 17.860155 s
('Clearwater', 2020, 'medio', 50, 95.38, ['group'])
```

Figura 38: Tempo impiegato per il job 2 in Spark Core con benchmark del 30%.

6.3.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Spark Core sono i seguenti:

```
('Modesto', 2020, 'alto', 34, 47.88, ['power', 'seat', 'front'])
('Altavista', 2017, 'basso', 2, 32.0, ['front', 'rear', 'seat'])
('Richburg', 2015, 'basso', 2, 159.5, ['power', 'dual', 'head'])
('Pasadena', 1969, 'alto', 1, 705.0, ['state', 'our', 'vehicles'])
('Englewood', 2020, 'medio', 18, 143.0, ['seat', 'door', 'chevrolet'])
('Rose City', 2021, 'alto', 1, 29.0, ['audio', 'front', 'air'])
('Paducah', 2020, 'medio', 6, 45.17, ['backup', 'camera', 'bluetooth'])
('Van Nuys', 2020, 'alto', 8, 191.25, ['seat', 'front', 'rear'])
('Green Bay', 2019, 'basso', 2, 21.0, ['state', 'emissions'])
('Weatherford', 2020, 'alto', 43, 146.09, ['door', 'front', 'steering'])
```

Figura 39: Risultato del job 2 in Spark Core con benchmark del 50%.

Il job ha impiegato 39.19 secondi.

```
25/06/07 11:59:19 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:59:19 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 39.196938 s
('Modesto', 2020, 'alto', 34, 47.88, ['power', 'seat', 'front'])
```

Figura 40: Tempo impiegato per il job 2 in Spark Core con benchmark del 50%.

6.3.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Spark Core sono i seguenti:


```
(('Saco', 2014, 'medio', 2, 23.0, ['power', 'sentry', 'dual']))
('Gaithersburg', 2021, 'alto', 1, 6.0, [])
('Millerton', 2019, 'medio', 2, 420.5, ['front', 'seat', 'manual'])
('Columbus', 2019, 'medio', 4, 41.5, ['test', 'drive', 'car'])
('Ash Flat', 2020, 'medio', 20, 161.85, ['front', 'seat', 'rear'])
('El Paso', 2021, 'basso', 6, 26.33, ['chevrolet', 'front', 'seat'])
('Thornton', 2020, 'alto', 16, 136.06, ['power', 'remote', 'steering'])
('Sherwood', 2020, 'medio', 4, 0.0, ['front', 'door', 'rear'])
('Henderson', 2016, 'basso', 7, 61.43, ['front', 'rear', 'of'])
('Clearwater', 2005, 'basso', 3, 81.0, ['power', 'tilt', 'dual'])
```

Figura 41: Risultato del job 2 in Spark Core con benchmark del 70%.

Il job ha impiegato 57.23 secondi.

```
25/06/07 12:01:23 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 12:01:23 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 57.233440 s
('Saco', 2014, 'medio', 2, 23.0, ['power', 'sentry', 'dual'])
```

Figura 42: Tempo impiegato per il job 2 in Spark Core con benchmark del 70%.

6.3.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Spark Core sono i seguenti:

```
(('Fenton', 2017, 'alto', 1, 20.0, ['steering', 'wood', 'air']))
('Southampton', 2002, 'alto', 1, 82.0, ['is', 'body', 'manual'])
('New Bedford', 2013, 'medio', 1, 1229.0, ['power', 'head', 'tilt'])
('Littleton', 2019, 'medio', 3, 4.33, ['to', 'of', 'call'])
('Haverhill', 2011, 'basso', 1, 69.0, ['front', 'seat', 'rear'])
('East Providence', 2017, 'medio', 1, 168.0, ['front', 'rear', 'manual'])
('Madison', 2010, 'basso', 3, 15.33, ['power', 'air', 'seat'])
('Monroe', 2020, 'alto', 36, 29.25, ['power', 'seat', 'front'])
('Portsmouth', 2019, 'basso', 1, 157.0, ['front', 'seat', 'manual'])
('Dearborn', 2019, 'alto', 1, 19.0, ['front', 'wheel', 'rear'])
```

Figura 43: Risultato del job 2 in Spark Core con benchmark del 100%.

Il job ha impiegato 50.12 secondi.

```
25/06/07 12:03:26 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 12:03:26 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:179, took 50.121649 s
('Fenton', 2017, 'alto', 1, 20.0, ['steering', 'wood', 'air'])
```

Figura 44: Tempo impiegato per il job 2 in Spark Core con benchmark del 100%.

6.4 Spark SQL

Tramite l'interfaccia PySpark, si elabora in modo distribuito il file CSV contenente i dati inerenti al secondo job, andando a raggruppare i record per marca e modello dell'auto e calcolando statistiche sul prezzo e gli anni di produzione.

Dopo aver importato le librerie necessarie, il programma inizia con la gestione dell'argomento preso da riga di comando che indica il percorso del file CSV da elaborare.

Segue nel codice l'inizializzazione della sessione Spark, configurata per essere eseguita localmente.

Operazione cruciale è la definizione dello schema del dataset che aiuta Spark a interpretare correttamente i dati e a evitare errori di inferenza. Viene infatti esplicitamente indicato che il file CSV contiene cinque colonne: il nome della marca (make name), il prezzo (price), l'anno di produzione (year), i giorni sul mercato (daysonmarket) e una descrizione testuale dell'annuncio (description). A ciascuna colonna viene associato un tipo di dato specifico: testo per il nome e la descrizione, double per il prezzo e intero per l'anno e per i giorni.

Il programma procede così con la lettura del file CSV utilizzando Spark SQL. I dati vengono caricati in un DataFrame e immediatamente registrati come una vista temporanea chiamata "job2_dataset". Questo consente di eseguire query SQL direttamente sul dataset, come se fosse una tabella relazionale.

```
SELECT
    city,
    year,
    CASE
        WHEN price < 20000 THEN 'basso'
        WHEN price BETWEEN 20000 AND 50000 THEN 'medio'
        ELSE 'alto'
    END AS fascia,
    COUNT(*) AS numero_macchine,
    AVG(daysonmarket) AS avg_daysonmarket,
    COLLECT_LIST(description) AS descriptions_list
FROM job2_dataset
GROUP BY city, year,
    CASE
        WHEN price < 20000 THEN 'basso'
        WHEN price BETWEEN 20000 AND 50000 THEN 'medio'
        ELSE 'alto'
```

END

Dopo la query e dopo aver trasformato l'elenco degli anni, il programma visualizza le prime dieci righe del risultato in console. Dopodiché, chiude la sessione Spark, concludendo l'elaborazione.

Algorithm 12 Spark SQL Job2: Statistiche per città, anno e fascia di prezzo

```
1: function PROCESS_ROW(riga)
2:   Estrarre city, year, fascia, numero_macchine, avg_daysonmarket,
   descriptions_list
3:   Inizializzare dizionario word_counts
4:   for ogni parola in descriptions_list do
5:     Convertire in minuscolo
6:     if alfabetica, lunghezza > 1, non presente nelle stopwords then
7:       Incrementare conteggio in word_counts
8:     end if
9:   end for
10:  Ordinare le parole per frequenza
11:  Restituire (city, year, fascia, numero_macchine,
   avg_daysonmarket, top3_parole)
12: end function
13: Avviare SparkSession con configurazione driver.host=localhost e nome "Job2
   sparksql"
14: # riga driver.host=localhost serve solo il programma viene fatto girare in
   locale
15: Definire schema del dataset
16: Leggere CSV come DataFrame
17: Filtrare righe con daysonmarket e year non nulli
18: Registrare come vista temporanea job2_dataset
19: Definire la query riportata sopra
20: Post-elaborazione del DataFrame:
    • Applicare ROUND alla media di daysonmarket
    • Concatenare la lista delle descrizioni in una stringa
    • Suddividere la stringa in singole parole
21: Convertire in RDD
22: Applicare map(process_row) per calcolare top-3 parole frequenti
23: Costruire un nuovo DataFrame con schema:
    • città, anno, fascia
    • numero auto, media giorni, top-3 parole
24: Mostrare le prime 10 righe del risultato
25: Fermare la sessione Spark
```

6.4.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Spark SQL sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Abbeville	2018	medio	1	35.0	rear, air, power
Abbeville	2019	medio	3	340.0	rear, air, to
Abbeville	2020	alto	2	24.0	rear, front, power
Abbeville	2020	medio	3	150.0	rear, front, passenger
Aberdeen	2007	basso	3	208.0	power, at, rear
Aberdeen	2012	basso	4	65.5	power, rear, air
Aberdeen	2014	basso	1	1.0	air, passenger, door
Aberdeen	2015	basso	6	30.0	door, air, wheel
Aberdeen	2019	medio	5	129.4	air, front, door
Aberdeen	2021	medio	2	30.5	air, door, vanity

only showing top 10 rows

Figura 45: Risultato del job 2 in Spark SQL con benchmark del 10%.

Il job ha impiegato 3.52 secondi.

```
25/06/07 11:37:32 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:37:32 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 3.523334 s
25/06/07 11:37:32 INFO CodeGenerator: Code generated in 6.525658 ms
```

Figura 46: Tempo impiegato per il job 2 in Spark SQL con benchmark del 10%.

6.4.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Spark SQL sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Abbeville	2015	basso	2	167.5	rear, front, type
Aberdeen	2005	basso	2	96.0	power, front, includes
Aberdeen	2014	basso	9	109.0	power, front, rear
Abilene	1995	medio	1	37.0	great, red, gmc
Abilene	2006	basso	3	43.0	rear, front, power
Abilene	2016	medio	5	40.4	door, front, rear
Abilene	2021	alto	1	7.0	nl, air, actuated
Abingdon	2014	basso	2	16.5	front, of, rear
Abingdon	2020	medio	26	90.08	front, rear, driver
Abington	2007	basso	7	264.86	front, rear, type

only showing top 10 rows

Figura 47: Risultato del job 2 in Spark SQL con benchmark del 30%.

Il job ha impiegato 7.49 secondi.

```
25/06/07 11:40:03 INFO DAGSchedulerImpl: Killing all running tasks in stage 2: stage finished
25/06/07 11:40:03 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 7.494869 s
25/06/07 11:40:03 INFO CodeGenerator: Code generated in 6.604208 ms
```

Figura 48: Tempo impiegato per il job 2 in Spark SQL con benchmark del 30%.

6.4.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Spark SQL sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Aberdeen	2014	basso	13	107.62	power, rear, front
Abilene	2006	basso	3	43.0	rear, front, power
Abilene	2016	medio	13	26.77	front, door, rear
Abilene	2021	alto	4	13.5	nl, air, communicat...
Abingdon	2014	basso	3	11.67	front, rear, of
Abingdon	2020	medio	46	85.07	front, rear, driver
Abington	2010	basso	11	214.64	front, rear, type
Acton	2005	basso	1	5.0	front, wheel, power
Acton	2016	basso	4	22.5	nl, air, front
Acworth	2020	basso	4	34.5	bag, wheel, power

only showing top 10 rows

Figura 49: Risultato del job 2 in Spark SQL con benchmark del 50%.

Il job ha impiegato 9.24 secondi.

```

25/06/07 11:43:12 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:43:12 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 9.239630 s
25/06/07 11:43:12 INFO CodeGenerator: Code generated in 18.079141 ms

```

Figura 50: Tempo impiegato per il job 2 in Spark SQL con benchmark del 50%.

6.4.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Spark SQL sono i seguenti:

```

+-----+-----+-----+-----+-----+-----+
| city|year|fascia|num_macchine|avg_daysonmarket|top_3_words|
+-----+-----+-----+-----+-----+-----+
|Aberdeen|2014|basso|22|103.09|power,rear,front|
|Abilene|2021|alto|4|13.5|nl,air,communicat...|
|Abingdon|2014|basso|3|11.67|front,rear,of|
|Abingdon|2020|medio|62|86.63|front,rear,driver|
|Abington|2010|basso|12|264.75|front,rear,type|
|Acton|2005|basso|2|27.5|front,seat,wheel|
|Acworth|2020|basso|9|36.0|wheel,bag,front|
|Ada|2015|medio|2|34.0|front,rear,impact|
|Adams|2019|medio|1|220.0|nl,driver,infotai...|
|Addison|2011|alto|1|91.0|brabus,to,is|
+-----+-----+-----+-----+-----+-----+
only showing top 10 rows

```

Figura 51: Risultato del job 2 in Spark SQL con benchmark del 70%.

Il job ha impiegato 13.16 secondi.

```

25/06/07 11:45:50 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:45:50 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 13.159170 s
25/06/07 11:45:50 INFO CodeGenerator: Code generated in 168.170101 ms

```

Figura 52: Tempo impiegato per il job 2 in Spark SQL con benchmark del 70%.

6.4.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Spark SQL sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Abbeville	2016	medio	1	37.0	air, door, in
Aberdeen	2014	basso	31	83.39	power, front, rear
Abilene	2006	basso	5	48.6	front, rear, nl
Abilene	2021	alto	4	13.5	nl, air, communicat...
Abingdon	2014	basso	4	22.0	front, rear, of
Abingdon	2020	medio	93	90.13	front, rear, driver
Abingdon	2021	alto	1	13.0	front, driver, rear
Abington	2010	basso	18	284.11	front, rear, type
Acton	2005	basso	3	24.33	front, power, wheel
Acworth	2020	basso	15	35.4	bag, wheel, power

only showing top 10 rows

Figura 53: Risultato del job 2 in Spark SQL con benchmark del 100%.

Il job ha impiegato 23.23 secondi.

```
25/06/07 11:50:21 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
25/06/07 11:50:21 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 23.234550 s
25/06/07 11:50:21 INFO CodeGenerator: Code generated in /0.769707.ms
```

Figura 54: Tempo impiegato per il job 2 in Spark SQL con benchmark del 100%.

6.5 Risultati

Anche nell'esecuzione del secondo job, *Spark SQL* si conferma la soluzione più rapida nei benchmark con dataset di dimensioni contenute, ma il vantaggio si riduce man mano che cresce la quantità di dati elaborati.

Un elemento interessante emerge nel benchmark al 100%, dove *Spark Core*, nonostante il picco registrato al 70%, riesce a completare l'esecuzione in 50.12 secondi, migliorando sensibilmente rispetto al valore precedente. Questo comportamento anomalo suggerisce che Spark Core potrebbe essere stato influenzato da fattori non strutturali, che hanno causato un rallentamento improvviso.

MapReduce, pur mantenendo una crescita più regolare e prevedibile nei tempi di esecuzione, si conferma la tecnologia meno performante nel complesso, soprattutto nel benchmark finale al 100%. Nonostante in alcuni casi sia riuscita a ottenere risultati migliori di Spark Core, la sua architettura continua a mostrare i limiti legati alla scrittura e lettura su disco tra ogni fase dell'elaborazione.

Benchmarks	Map-Reduce (s)	Spark Core (s)	Spark SQL (s)
10%	9.53	10.75	3.52
30%	21.04	17.86	7.49
50%	35.73	39.19	9.24
70%	43.83	57.23	13.16
100%	71.77	50.12	23.23

Tabella 2: Tempi di esecuzione per il secondo job in base al diverso benchmark e alla diversa tecnologia.

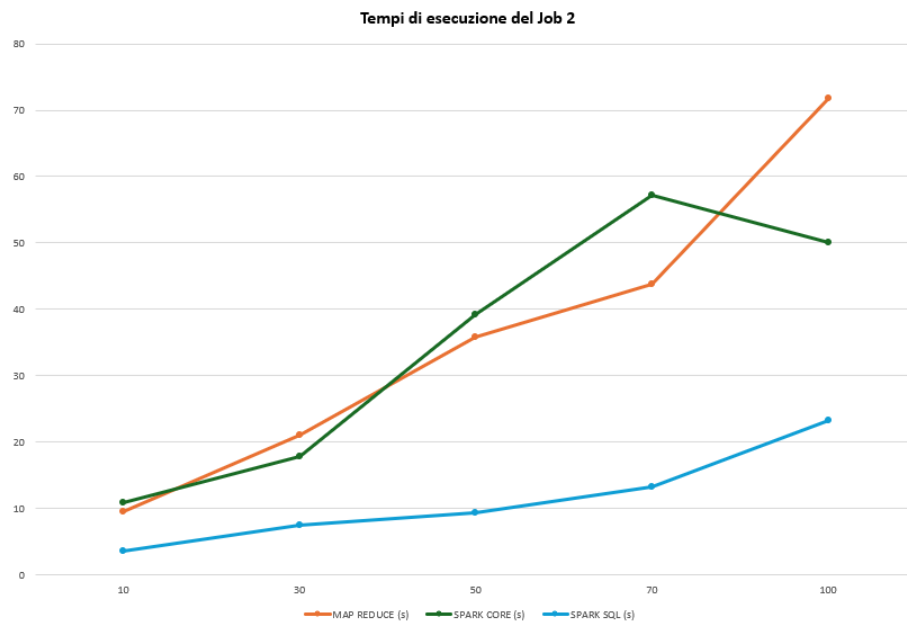


Figura 55: Tempi di esecuzione del job 2 a confronto.

7 AWS

7.1 Job 1 - Map-Reduce

7.1.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Map-Reduce su AWS sono i seguenti:

```
AM General#Hummer      Numero totale auto: 1, Prezzo minimo: 71995.0, Prezzo massimo: 71995.0, Prezzo medio: 71995.00, Anni: [2000]
Acura#CL               Numero totale auto: 2, Prezzo minimo: 2290.0, Prezzo massimo: 3200.0, Prezzo medio: 2745.00, Anni: [2001,2002]
Acura#ILX              Numero totale auto: 261, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23886.67, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#RL               Numero totale auto: 12, Prezzo minimo: 3995.0, Prezzo massimo: 15454.0, Prezzo medio: 7908.67, Anni: [2002,2005,2006,2007,2008,2010,2011]
Alfa Romeo#Stelvio     Numero totale auto: 191, Prezzo minimo: 24500.0, Prezzo massimo: 88490.0, Prezzo medio: 40543.23, Anni: [2018,2019,2020]
Aston Martin#DB9       Numero totale auto: 4, Prezzo minimo: 41498.0, Prezzo massimo: 98500.0, Prezzo medio: 58219.50, Anni: [2006,2007,2008,2014]
Aston Martin#V8 Vantage Numero totale auto: 10, Prezzo minimo: 36000.0, Prezzo massimo: 109888.0, Prezzo medio: 63701.70, Anni: [2007,2008,2011,2013,2015,2016]
Aston Martin#Vantage   Numero totale auto: 15, Prezzo minimo: 127950.0, Prezzo massimo: 214809.0, Prezzo medio: 183750.73, Anni: [2019,2020]
Audi#A4 Allroad        Numero totale auto: 34, Prezzo minimo: 14900.0, Prezzo massimo: 58340.0, Prezzo medio: 33042.41, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Audi#A5 Sportback      Numero totale auto: 103, Prezzo minimo: 27990.0, Prezzo massimo: 55940.0, Prezzo medio: 41920.35, Anni: [2018,2019,2020]
```

Figura 56: Risultato del job 1 in Map-Reduce con benchmark del 10%.

Il job ha impiegato 27.29 secondi.

7.1.2 Benchmark del 30%

I risultati ottenuti dal running del job 1 tramite Map-Reduce su AWS sono i seguenti:

```
AM General#Hummer      Numero totale auto: 1, Prezzo minimo: 71995.0, Prezzo massimo: 71995.0, Prezzo medio: 71995.00, Anni: [2000]
Acura#CL               Numero totale auto: 5, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4177.00, Anni: [2001,2002,2003]
Acura#ILX              Numero totale auto: 795, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23996.59, Anni: [2013,2014,2015,2016,2017,2018,2019,2020]
Acura#RL               Numero totale auto: 42, Prezzo minimo: 1999.0, Prezzo massimo: 15454.0, Prezzo medio: 6772.71, Anni: [2000,2002,2004,2005,2006,2007,2008,2009,2010,2011]
Alfa Romeo#8C Competizione Numero totale auto: 2, Prezzo minimo: 249995.0, Prezzo massimo: 353900.0, Prezzo medio: 301947.50, Anni: [2008,2009]
Alfa Romeo#Stelvio     Numero totale auto: 615, Prezzo minimo: 21997.0, Prezzo massimo: 89240.0, Prezzo medio: 41114.07, Anni: [2018,2019,2020]
Aston Martin#DB9       Numero totale auto: 15, Prezzo minimo: 28900.0, Prezzo massimo: 130007.0, Prezzo medio: 73475.07, Anni: [2006,2007,2008,2014,2015,2016]
Aston Martin#V8 Vantage Numero totale auto: 21, Prezzo minimo: 29991.0, Prezzo massimo: 109888.0, Prezzo medio: 60174.14, Anni: [2007,2008,2010,2011,2013,2014,2015,2016]
Aston Martin#Vantage   Numero totale auto: 42, Prezzo minimo: 127950.0, Prezzo massimo: 228752.0, Prezzo medio: 182810.52, Anni: [2019,2020]
Aston Martin#Virage    Numero totale auto: 1, Prezzo minimo: 79990.0, Prezzo massimo: 79990.0, Prezzo medio: 79990.00, Anni: [2012]
```

Figura 57: Risultato del job 1 in Map-Reduce con benchmark del 30%.

Il job ha impiegato 27.57 secondi.

7.1.3 Benchmark del 50%

I risultati ottenuti dal running del job 1 tramite Map-Reduce su AWS sono i seguenti:

```
AM General#Hummer      Numero totale auto: 2, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67997.00, Anni:
[2000]
Acura#CL               Numero totale auto: 7, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4283.57, Anni:
[2001,2002,2003]
Acura#ILX              Numero totale auto: 1362, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 24024.28, Anni:
[2013,2014,2015,2016,2017,2018,2019,2020]
Acura#RL               Numero totale auto: 64, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 6930.31, Anni:
[2000,2002,2004,2005,2006,2007,2008,2009,2010,2011]
Alfa Romeo#8C Competizione Numero totale auto: 2, Prezzo minimo: 249995.0, Prezzo massimo: 353900.0, Prezzo medio:
301947.50, Anni: [2008,2009]
Alfa Romeo#Stelvio     Numero totale auto: 1035, Prezzo minimo: 21997.0, Prezzo massimo: 89240.0, Prezzo medio: 40873.37, Anni:
[2018,2019,2020]
Aston Martin#DB9       Numero totale auto: 23, Prezzo minimo: 28900.0, Prezzo massimo: 134900.0, Prezzo medio: 72060.48, Anni:
[2005,2006,2007,2008,2011,2012,2014,2015,2016]
Aston Martin#V8 Vantage Numero totale auto: 33, Prezzo minimo: 29900.0, Prezzo massimo: 109888.0, Prezzo medio: 58323.76, Anni:
[2006,2007,2008,2010,2011,2012,2013,2014,2015,2016]
Aston Martin#Vantage   Numero totale auto: 70, Prezzo minimo: 121306.0, Prezzo massimo: 228752.0, Prezzo medio: 180187.54,
Anni: [2019,2020]
Aston Martin#Virage    Numero totale auto: 1, Prezzo minimo: 79990.0, Prezzo massimo: 79990.0, Prezzo medio: 79990.00, Anni:
[2012]
```

Figura 58: Risultato del job 1 in Map-Reduce con benchmark del 50%.

Il job ha impiegato 28.91 secondi.

7.1.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Map-Reduce su AWS sono i seguenti:

```
AM General#Hummer      Numero totale auto: 2, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67997.00, Anni:
[2000]
Acura#CL               Numero totale auto: 11, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4070.36, Anni:
[2001,2002,2003]
Acura#ILX              Numero totale auto: 1877, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23990.73, Anni:
[2013,2014,2015,2016,2017,2018,2019,2020]
Acura#RL               Numero totale auto: 93, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 7080.39, Anni:
[2000,2001,2002,2004,2005,2006,2007,2008,2009,2010,2011]
Alfa Romeo#8C Competizione Numero totale auto: 2, Prezzo minimo: 249995.0, Prezzo massimo: 353900.0, Prezzo medio:
301947.50, Anni: [2008,2009]
Alfa Romeo#Stelvio     Numero totale auto: 1436, Prezzo minimo: 21997.0, Prezzo massimo: 105735.0, Prezzo medio: 41244.59,
Anni: [2018,2019,2020]
Aston Martin#DB9       Numero totale auto: 30, Prezzo minimo: 28900.0, Prezzo massimo: 134900.0, Prezzo medio: 70379.30, Anni:
[2005,2006,2007,2008,2010,2011,2012,2014,2015,2016]
Aston Martin#V8 Vantage Numero totale auto: 42, Prezzo minimo: 29900.0, Prezzo massimo: 109888.0, Prezzo medio: 55887.98, Anni:
[2006,2007,2008,2010,2011,2012,2013,2014,2015,2016]
Aston Martin#Vantage   Numero totale auto: 104, Prezzo minimo: 109995.0, Prezzo massimo: 228752.0, Prezzo medio: 177086.07,
Anni: [2019,2020]
Aston Martin#Virage    Numero totale auto: 1, Prezzo minimo: 79990.0, Prezzo massimo: 79990.0, Prezzo medio: 79990.00, Anni:
[2012]
```

Figura 59: Risultato del job 1 in Map-Reduce con benchmark del 70%.

Il job ha impiegato 30.57 secondi.

7.1.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Map-Reduce su AWS sono i seguenti:

```
AM General#Hummer      Numero totale auto: 3, Prezzo minimo: 63999.0, Prezzo massimo: 71995.0, Prezzo medio: 67264.67, Anni:
[2000]
Acura#CL               Numero totale auto: 18, Prezzo minimo: 1000.0, Prezzo massimo: 6000.0, Prezzo medio: 3789.39, Anni:
[2001,2002,2003]
Acura#ILX              Numero totale auto: 2636, Prezzo minimo: 6995.0, Prezzo massimo: 39305.0, Prezzo medio: 24009.32, Anni:
[2013,2014,2015,2016,2017,2018,2019,2020]
Acura#RL               Numero totale auto: 126, Prezzo minimo: 250.0, Prezzo massimo: 15454.0, Prezzo medio: 6994.63, Anni:
[2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011]
Alfa Romeo#8C Competizione Numero totale auto: 2, Prezzo minimo: 249995.0, Prezzo massimo: 353900.0, Prezzo medio:
301947.50, Anni: [2008,2009]
Alfa Romeo#Stelvio     Numero totale auto: 2109, Prezzo minimo: 21997.0, Prezzo massimo: 105735.0, Prezzo medio: 41322.08,
Anni: [2018,2019,2020]
Aston Martin#DB9       Numero totale auto: 42, Prezzo minimo: 28900.0, Prezzo massimo: 134900.0, Prezzo medio: 68163.74, Anni:
[2005,2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016]
Aston Martin#Vantage Numero totale auto: 58, Prezzo minimo: 29900.0, Prezzo massimo: 109888.0, Prezzo medio: 55445.31, Anni:
[2006,2007,2008,2009,2010,2011,2012,2013,2014,2015,2016]
Aston Martin#Vantage   Numero totale auto: 151, Prezzo minimo: 109995.0, Prezzo massimo: 228752.0, Prezzo medio: 177158.93,
Anni: [2019,2020]
Aston Martin#Virage    Numero totale auto: 2, Prezzo minimo: 74900.0, Prezzo massimo: 79990.0, Prezzo medio: 77445.00, Anni:
[2012]
```

Figura 60: Risultato del job 1 in Map-Reduce con benchmark del 100%.

Il job ha impiegato 29.59 secondi.

7.2 Job 1 - Spark Core

7.2.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Spark Core su AWS sono i seguenti:

```
{ "make_name": "Ford", "model_name": "Mustang", "num_cars": 1732, "min_price": 1399.0, "max_price": 240900.0, "avg_price": 20357.48, "years": [1965, 1966, 1968, 1969, 1970, 1971, 1973, 1983,
1990, 1993, 1994, 1995, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "Audi", "model_name": "A8", "num_cars": 326, "min_price": 2995.0, "max_price": 77210.0, "avg_price": 36266.44, "years": [2003, 2004, 2006, 2007, 2008, 2009, 2010, 2011, 2012,
2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "GMC", "model_name": "Sierra 2500HD", "num_cars": 407, "min_price": 6800.0, "max_price": 84900.0, "avg_price": 53147.3, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 20
08, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "Honda", "model_name": "Accord", "num_cars": 3780, "min_price": 999.0, "max_price": 89995.0, "avg_price": 22093.78, "years": [1996, 1997, 1999, 2000, 2001, 2002, 2003, 2004,
2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "RAM", "model_name": "ProMaster City", "num_cars": 263, "min_price": 6950.0, "max_price": 30100.0, "avg_price": 22280.61, "years": [2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "Cadillac", "model_name": "Escalade", "num_cars": 448, "min_price": 2000.0, "max_price": 102300.0, "avg_price": 58149.02, "years": [2002, 2003, 2004, 2005, 2006, 2007, 2008,
2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "Ford", "model_name": "Ecosport", "num_cars": 1500, "min_price": 12000.0, "max_price": 31500.0, "avg_price": 22067.51, "years": [2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] },
{ "make_name": "Cadillac", "model_name": "VTS", "num_cars": 1030, "min_price": 17599.0, "max_price": 71385.0, "avg_price": 41516.99, "years": [2017, 2018, 2019, 2020, 2021] },
{ "make_name": "Jeep", "model_name": "Wrangler Unlimited", "num_cars": 2550, "min_price": 16990.0, "max_price": 92900.0, "avg_price": 40545.5, "years": [2007, 2008, 2009, 2010, 2011, 2012,
2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021] },
{ "make_name": "Mazda", "model_name": "CX-30", "num_cars": 348, "min_price": 20989.0, "max_price": 32244.0, "avg_price": 27149.93, "years": [2020, 2021] }
25/06/10 14:02:28 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 1.408998 s
```

Figura 61: Risultato del job 1 in Spark Core con benchmark del 10%.

Il job ha impiegato 1.41 secondi.

```
25/06/10 14:02:28 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 1.408998 s
```

Figura 62: Tempo impiegato per il job 1 in Spark Core con benchmark del 10%.

7.2.2 Benchmark del 30%

I risultati ottenuti dal running del job 1 tramite Spark Core su AWS sono i seguenti:

```
[{"name": "Ford", "model": "Mustang", "num_cars": 5180, "min_price": 484.0, "max_price": 299980.0, "avg_price": 29662.9, "years": [1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1983, 1985, 1986, 1987, 1988, 1996, 1997, 1999, 1994, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Audi", "model": "A5", "num_cars": 944, "min_price": 1998.0, "max_price": 77210.0, "avg_price": 38782.68, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Civic", "model": "Sierra 2000HD", "num_cars": 1550, "min_price": 4500.0, "max_price": 96435.0, "avg_price": 52944.57, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Honda", "model": "Accord", "num_cars": 11150, "min_price": 999.0, "max_price": 89995.0, "avg_price": 21923.64, "years": [1981, 1990, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "RAM", "model": "ProMaster City", "num_cars": 776, "min_price": 6950.0, "max_price": 38875.0, "avg_price": 22332.72, "years": [2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Cadillac", "model": "Escalade", "num_cars": 1079, "min_price": 2880.0, "max_price": 180825.0, "avg_price": 57905.03, "years": [1999, 2000, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Ford", "model": "Focusport", "num_cars": 4033, "min_price": 11000.0, "max_price": 31500.0, "avg_price": 22062.49, "years": [2010, 2019, 2020, 2021]}, {"name": "Cadillac", "model": "XT5", "num_cars": 3114, "min_price": 17599.0, "max_price": 73085.0, "avg_price": 41690.36, "years": [2017, 2018, 2019, 2020, 2021]}, {"name": "Jeep", "model": "Wrangler Unlimited", "num_cars": 7831, "min_price": 10500.0, "max_price": 109872.0, "avg_price": 40366.86, "years": [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Mazda", "model": "CX-30", "num_cars": 1006, "min_price": 19991.0, "max_price": 36480.0, "avg_price": 27011.26, "years": [2020, 2021]}]
```

Figura 63: Risultato del job 1 in Spark Core con benchmark del 30%.

Il job ha impiegato 3.41 secondi.

```
25/06/10 14:05:34 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 3.413648 s
```

Figura 64: Tempo impiegato per il job 1 in Spark Core con benchmark del 30%.

7.2.3 Benchmark del 50%

I risultati ottenuti dal running del job 1 tramite Spark Core su AWS sono i seguenti:

```
[{"name": "Ford", "model": "Mustang", "num_cars": 8537, "min_price": 484.0, "max_price": 299980.0, "avg_price": 29509.11, "years": [1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1983, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Audi", "model": "A5", "num_cars": 1550, "min_price": 1998.0, "max_price": 79775.0, "avg_price": 39289.69, "years": [1999, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Civic", "model": "Sierra 2000HD", "num_cars": 2000, "min_price": 4500.0, "max_price": 98000.0, "avg_price": 53339.94, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Honda", "model": "Accord", "num_cars": 10200, "min_price": 484.0, "max_price": 89995.0, "avg_price": 21804.51, "years": [1981, 1989, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "RAM", "model": "ProMaster City", "num_cars": 1397, "min_price": 6950.0, "max_price": 34975.0, "avg_price": 22515.26, "years": [2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Cadillac", "model": "Escalade", "num_cars": 2602, "min_price": 2880.0, "max_price": 180825.0, "avg_price": 58425.49, "years": [1999, 2000, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Ford", "model": "Focusport", "num_cars": 4033, "min_price": 11000.0, "max_price": 31500.0, "avg_price": 21808.97, "years": [2010, 2019, 2020, 2021]}, {"name": "Cadillac", "model": "XT5", "num_cars": 3279, "min_price": 16400.0, "max_price": 73085.0, "avg_price": 41519.1, "years": [2017, 2018, 2019, 2020, 2021]}, {"name": "Jeep", "model": "Wrangler Unlimited", "num_cars": 13289, "min_price": 10500.0, "max_price": 109872.0, "avg_price": 40281.1, "years": [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Mazda", "model": "CX-30", "num_cars": 1784, "min_price": 19990.0, "max_price": 36480.0, "avg_price": 27075.48, "years": [2020, 2021]}]
```

Figura 65: Risultato del job 1 in Spark Core con benchmark del 50%.

Il job ha impiegato 4.13 secondi.

```
25/06/10 14:08:24 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 4.135197 s
```

Figura 66: Tempo impiegato per il job 1 in Spark Core con benchmark del 50%.

7.2.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Spark Core su AWS sono i seguenti:

```
[{"name": "Ford", "model_name": "Mustang", "num_cars": 12807, "min_price": 449.0, "max_price": 29998.0, "avg_price": 28526.41, "years": [1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Audi", "model_name": "A5", "num_cars": 2168, "min_price": 495.0, "max_price": 79775.0, "avg_price": 39076.85, "years": [1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "GMC", "model_name": "Sierra 2500HD", "num_cars": 3841, "min_price": 4099.0, "max_price": 92880.0, "avg_price": 53298.29, "years": [2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Honda", "model_name": "Accord", "num_cars": 26995, "min_price": 484.0, "max_price": 89995.0, "avg_price": 21884.91, "years": [1981, 1989, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "RAM", "model_name": "ProMaster City", "num_cars": 1827, "min_price": 6958.0, "max_price": 38980.0, "avg_price": 22617.37, "years": [2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Cadillac", "model_name": "Escalade", "num_cars": 3402, "min_price": 2080.0, "max_price": 118500.0, "avg_price": 52250.5, "years": [1999, 2000, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Ford", "model_name": "F250Super", "num_cars": 11121, "min_price": 18995.0, "max_price": 33185.0, "avg_price": 21940.16, "years": [2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Cadillac", "model_name": "XT5", "num_cars": 7369, "min_price": 15999.0, "max_price": 73085.0, "avg_price": 41561.55, "years": [2017, 2018, 2019, 2020, 2021]}, {"name": "Jeep", "model_name": "Wrangler Unlimited", "num_cars": 18574, "min_price": 9090.0, "max_price": 189872.0, "avg_price": 40183.08, "years": [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Mazda", "model_name": "CX-30", "num_cars": 2310, "min_price": 19900.0, "max_price": 300450.0, "avg_price": 27150.31, "years": [2020, 2021]}]
```

Figura 67: Risultato del job 1 in Spark Core con benchmark del 70%.

Il job ha impiegato 8.08 secondi.

```
25/06/10 14:10:04 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 8.086726 s
```

Figura 68: Tempo impiegato per il job 1 in Spark Core con benchmark del 70%.

7.2.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Spark Core su AWS sono i seguenti:

```
[{"name": "Land Rover", "model_name": "Discovery Sport", "num_cars": 3364, "min_price": 15995.0, "max_price": 75899.0, "avg_price": 41445.33, "years": [2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Mazda", "model_name": "MAZDA3", "num_cars": 8979, "min_price": 347.0, "max_price": 35200.0, "avg_price": 17888.15, "years": [2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Land Rover", "model_name": "Range Rover Evoque", "num_cars": 2297, "min_price": 12995.0, "max_price": 84399.0, "avg_price": 40542.19, "years": [2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Alfa Romeo", "model_name": "4C", "num_cars": 25, "min_price": 41995.0, "max_price": 97979.0, "avg_price": 59048.6, "years": [2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Mazda", "model_name": "CX-3", "num_cars": 1984, "min_price": 11190.0, "max_price": 30045.0, "avg_price": 19763.22, "years": [2016, 2017, 2018, 2019, 2020, 2021]}, {"name": "Hyundai", "model_name": "Elantra", "num_cars": 2580, "min_price": 808.0, "max_price": 111979.0, "avg_price": 16001.86, "years": [1997, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Lexus", "model_name": "LC 500", "num_cars": 232, "min_price": 19900.0, "max_price": 54815.0, "avg_price": 33347.76, "years": [2015, 2016, 2017, 2018]}, {"name": "Chevrolet", "model_name": "Traverse", "num_cars": 25459, "min_price": 1900.0, "max_price": 59015.0, "avg_price": 38697.79, "years": [2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]}, {"name": "Mazda", "model_name": "XT4", "num_cars": 4884, "min_price": 27787.0, "max_price": 57815.0, "avg_price": 43755.11, "years": [2019, 2020]}, {"name": "Cadillac", "model_name": "XT4", "num_cars": 3579, "min_price": 2695.0, "max_price": 109995.0, "avg_price": 18625.1, "years": [2011, 2012, 2013, 2014, 2015, 2016, 2017]}]
```

Figura 69: Risultato del job 1 in Spark Core con benchmark del 100%.

Il job ha impiegato 9.45 secondi.

```
25/06/10 14:15:15 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 9.448149 s
```

Figura 70: Tempo impiegato per il job 1 in Spark Core con benchmark del 100%.

7.3 Job 1 - Spark SQL

7.3.1 Benchmark del 10%

I risultati ottenuti dal running del job 1 tramite Spark SQL su AWS sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti
JAM General	Hummer	3	49995.0	71995.0	58996.67	1998, 1999, 2000
JMC	AMX	1	34980.0	34980.0	34980.0	1969
JMC	Rambler American	1	23980.0	23980.0	23980.0	1965
Acura	CL	3	1995.0	3280.0	2455.0	2001, 1999, 2002
Acura	ILX	261	6995.0	39920.0	23866.67	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	2	8995.0	12899.0	10547.0	2013
Acura	Integra	2	3995.0	5999.0	4997.0	2001, 2000
Acura	Legend	2	8995.0	15999.0	12466.0	1990, 1989
Acura	MDX	1219	1788.0	63745.0	48814.58	2004, 2019, 2005, 2020, 2006, 2013, 2007, 2014, 2015, 2001, 2016, 2008, 2002, 2017, 2009, 2003, 2010, 2011, 2018, 2012
Acura	MDX Hybrid Sport	23	32285.0	61175.0	54399.17	2020, 2017, 2018

only showing top 10 rows

Figura 71: Risultato del job 1 in Spark SQL con benchmark del 10%.

Il job ha impiegato 1.83 secondi.

```
25/06/11 09:24:31 INFO YarnScheduler: Kitting all running tasks in Stage 2: Stage finished
25/06/11 09:24:31 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 1.834395 s
25/06/11 09:24:31 INFO CodeGenerator: Code generated in 15.745966 ms
```

Figura 72: Tempo impiegato per il job 1 in Spark SQL con benchmark del 10%.

7.3.2 Benchmark del 30%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti
JAM General	Hummer	4	49995.0	71995.0	59222.5	1998, 1999, 1993, 2000
JMC	AMX	1	34980.0	34980.0	34980.0	1969
JMC	Javelin	1	35450.0	35450.0	35450.0	1974
JMC	Rambler American	2	10295.0	23980.0	17097.5	1965, 1964
Acura	CL	8	1995.0	6880.0	3591.25	2001, 1998, 2002, 2003, 1999
Acura	ILX	795	6995.0	39920.0	23966.59	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	6	8995.0	17999.0	12466.0	2013
Acura	Integra	3	3995.0	49980.0	19289.0	2001, 2000
Acura	Legend	3	4995.0	15999.0	9663.0	1990, 1989, 1994
Acura	MDX	3568	1688.0	63745.0	39941.21	2004, 2019, 2005, 2020, 2006, 2013, 2007, 2014, 2015, 2001, 2016, 2008, 2002, 2017, 2009, 2003, 2010, 2011, 2018, 2012

only showing top 10 rows

Figura 73: Risultato del job 1 in Spark SQL con benchmark del 30%.

Il job ha impiegato 1.79 secondi.

```
25/06/11 09:27:16 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 1.790307 s
25/06/11 09:27:16 INFO CodeGenerator: Code generated in 36.656956 ms
```

Figura 74: Tempo impiegato per il job 1 in Spark SQL con benchmark del 30%.

7.3.3 Benchmark del 50%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti
AM General	Hummer	5	49995.0	71995.0	60177.8	1998, 1999, 1993, 2000
AMC	AMX	3	32500.0	49990.0	39130.0	1968, 1969, 1970
AMC	Javelin	1	35450.0	35450.0	35450.0	1974
AMC	Rambler American	3	9999.0	23900.0	14731.33	1965, 1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	1961
Acura	CL	12	1995.0	6000.0	3939.92	2001, 1998, 2002, 2003, 1999, 1997
Acura	ILX	1362	6995.0	35920.0	24024.28	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	11	8995.0	17999.0	12294.73	2013, 2014
Acura	Integra	6	2695.0	47900.0	11762.5	2001, 1993, 2000
Acura	Legend	4	4995.0	15999.0	8972.25	1990, 1989, 1993, 1994

only showing top 10 rows

Figura 75: Risultato del job 1 in Spark SQL con benchmark del 50%.

Il job ha impiegato 0.44 secondi.

```
25/06/11 09:29:18 INFO YarnScheduler: Killing all running tasks in stage 2: Stage finished
25/06/11 09:29:18 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 0.440571 s
25/06/11 09:29:18 INFO CodeGenerator: Code generated in 35.810824 ms
```

Figura 76: Tempo impiegato per il job 1 in Spark SQL con benchmark del 50%.

7.3.4 Benchmark del 70%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti
AM General	Hummer	10	49995.0	124995.0	71832.4	1998, 1995, 1999, 1993, 2000, 1997
AM General	M151A2	1	13995.0	13995.0	13995.0	1968
AMC	AMX	3	32500.0	49990.0	39130.0	1968, 1969, 1970
AMC	Ambassador	1	25900.0	25900.0	25900.0	1967
AMC	Javelin	2	25800.0	35450.0	30225.0	1974
AMC	Rambler American	5	9999.0	23900.0	14617.8	1965, 1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	1961
Acura	CL	18	1943.0	6000.0	3775.61	2001, 1998, 2002, 2003, 1999, 1997
Acura	ILX	1877	6995.0	35920.0	23990.73	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	15	8398.0	17999.0	12831.93	2013, 2014

only showing top 10 rows

Figura 77: Risultato del job 1 in Spark SQL con benchmark del 70%.

Il job ha impiegato 1.73 secondi.

```
25/06/11 09:35:23 INFO TaskScheduler: Killing all running tasks in stage 2. Stage finished
25/06/11 09:35:23 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 1.734121 s
25/06/11 09:35:23 INFO CodeGenerator: Code generated in 41.984818 ms
```

Figura 78: Tempo impiegato per il job 1 in Spark SQL con benchmark del 70%.

7.3.5 Benchmark del 100%

I risultati ottenuti dal running del job 1 tramite Spark SQL sono i seguenti:

make_name	model_name	numero_auto	prezzo_minimo	prezzo_massimo	prezzo_medio	anni_presenti
AM General	Hummer	10	49995.0	124995.0	71032.4	1998, 1995, 1999, 1993, 2000, 1997
AM General	M151A2	1	13995.0	13995.0	13995.0	1968
AMC	AMX	3	32500.0	49990.0	39130.0	1968, 1969, 1970
AMC	Ambassador	1	25900.0	25900.0	25900.0	1967
AMC	Javelin	2	25000.0	35450.0	30225.0	1974
AMC	Rambler American	5	9999.0	23900.0	14617.8	1965, 1964
AMC	Rambler Classic	1	4000.0	4000.0	4000.0	1961
Acura	CL	18	1943.0	6000.0	3775.61	2001, 1998, 2002, 2003, 1999, 1997
Acura	ILX	1877	6995.0	35920.0	23990.73	2019, 2016, 2020, 2017, 2013, 2014, 2018, 2015
Acura	ILX Hybrid	15	8398.0	17999.0	12031.93	2013, 2014

only showing top 10 rows

Figura 79: Risultato del job 1 in Spark SQL con benchmark del 100%.

Il job ha impiegato 1.73 secondi.

```
25/06/11 09:35:23 INFO TaskScheduler: Killing all running tasks in stage 2. Stage finished
25/06/11 09:35:23 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 1.734121 s
25/06/11 09:35:23 INFO CodeGenerator: Code generated in 41.984818 ms
```

Figura 80: Tempo impiegato per il job 1 in Spark SQL con benchmark del 100%.

7.3.6 Risultati

Analizzando i dati, si nota subito un cambiamento significativo nei tempi di esecuzione di *MapReduce*, che rimangono sorprendentemente stabili, oscillando tra circa 27 e 30 secondi. Questo comportamento suggerisce che l'infrastruttura AWS ha assorbito bene l'aumento di carico, distribuendo efficacemente il lavoro tra i nodi. Tuttavia, MapReduce resta comunque la tecnologia più lenta in termini assoluti, come già emerso nell'ambiente locale.

Per quanto riguarda *Spark Core*, i tempi di esecuzione sono molto più contenuti rispetto a MapReduce, soprattutto nei benchmark inferiori. La crescita dei tempi è più evidente rispetto all'esecuzione locale, ma rimane in linea con le aspettative. Spark Core beneficia chiaramente dell'esecuzione distribuita in AWS, mantenendo buone prestazioni anche con dataset più ampi.

Spark SQL, invece, conferma la propria efficienza anche su AWS, pur con una certa variabilità nei risultati. Questo andamento leggermente irregolare può essere dovuto al caching, all'ottimizzazione del piano di esecuzione o a fenomeni legati alla gestione delle risorse su AWS. Tuttavia, in tutti i benchmark, Spark SQL resta molto competitivo.

Nel complesso, il passaggio a un'infrastruttura cloud come AWS ha reso MapReduce più scalabile, con tempi costanti, ma senza migliorarne l'efficienza assoluta. Spark Core e Spark SQL, invece, traggono maggiore vantaggio dal parallelismo offerto da AWS, riuscendo a completare il job in tempi estremamente brevi. Questo rafforza ulteriormente l'idea che Spark sia meglio progettato per ambienti in-memory distribuiti, dove la comunicazione tra i nodi e l'elaborazione parallela possono essere sfruttate al massimo.

Benchmarks	Map-Reduce (s)	Spark Core (s)	Spark SQL (s)
10%	27.29	1.41	1.83
30%	27.57	3.41	1.79
50%	28.91	4.13	0.44
70%	30.57	8.08	1.73
100%	29.59	9.45	1.73

Tabella 3: Tempi di esecuzione su AWS per il primo job in base al diverso benchmark e alla diversa tecnologia.

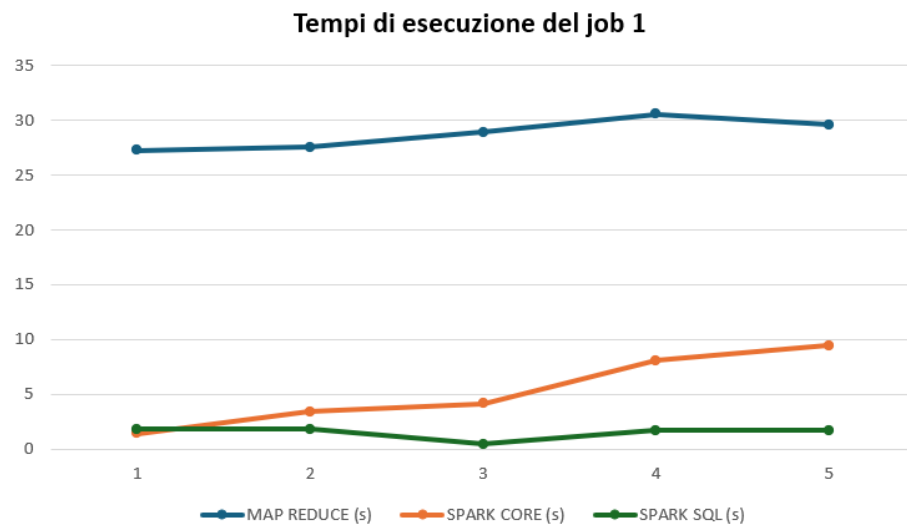


Figura 81: Tempi di esecuzione su AWS del job 1 a confronto.

7.4 Job 2 - Map-Reduce

7.4.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Map-Reduce su AWS sono i seguenti:

Atlanta	2019	basso	2	28.0	your, test, drive
Atmore	2020	basso	2	273.0	front, system, rear
Atmore	2020	medio	5	115.4	front, rear, seat
Atoka	2012	basso	1	412.0	air, wheel, am
Atoka	2018	medio	1	699.0	power, wheel, am
Audubon	2020	alto	4	80.5	awd, ltz, avier
Aurora	2016	medio	1	28.0	airbags, wheel, am
Austin	2009	basso	3	57.0	and, this, power
Austin	2012	basso	5	59.4	airbags, this, power
Austin	2013	basso	1	29.0	to, and, irene

Figura 82: Risultato del job 2 in Map-Reduce con benchmark del 10%.

Il job ha impiegato 30.07 secondi.

7.4.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Map-Reduce su AWS sono i seguenti:

AM General#Hummer	Numero totale auto: 1, Prezzo minimo: 71995.0, Prezzo massimo: 71995.0, Prezzo medio: 71995.00, Anni: [2000]
Acura#CL	Numero totale auto: 5, Prezzo minimo: 2290.0, Prezzo massimo: 6000.0, Prezzo medio: 4177.00, Anni: [2001, 2002, 2003]
Acura#ILX	Numero totale auto: 795, Prezzo minimo: 6995.0, Prezzo massimo: 35920.0, Prezzo medio: 23996.59, Anni: [2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
Acura#RL	Numero totale auto: 42, Prezzo minimo: 1999.0, Prezzo massimo: 15454.0, Prezzo medio: 6772.71, Anni: [2000, 2002, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011]
Alfa Romeo#8C Competizione	Numero totale auto: 2, Prezzo minimo: 249995.0, Prezzo massimo: 353900.0, Prezzo medio: 301947.50, Anni: [2008, 2009]
Alfa Romeo#Stelvio	Numero totale auto: 615, Prezzo minimo: 21997.0, Prezzo massimo: 89240.0, Prezzo medio: 41114.07, Anni: [2018, 2019, 2020]
Aston Martin#DB9	Numero totale auto: 15, Prezzo minimo: 28900.0, Prezzo massimo: 130007.0, Prezzo medio: 73475.07, Anni: [2006, 2007, 2008, 2014, 2015, 2016]
Aston Martin#V8 Vantage	Numero totale auto: 21, Prezzo minimo: 29991.0, Prezzo massimo: 109888.0, Prezzo medio: 60174.14, Anni: [2007, 2008, 2010, 2011, 2013, 2014, 2015, 2016]
Aston Martin#Vantage	Numero totale auto: 42, Prezzo minimo: 127950.0, Prezzo massimo: 228752.0, Prezzo medio: 182810.52, Anni: [2019, 2020]
Aston Martin#Virage	Numero totale auto: 1, Prezzo minimo: 79990.0, Prezzo massimo: 79990.0, Prezzo medio: 79990.00, Anni: [2012]

Figura 83: Risultato del job 2 in Map-Reduce con benchmark del 30%.

Il job ha impiegato 42.72 secondi.

7.4.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Map-Reduce su AWS sono i seguenti:

<u>Adel</u>	2020	medio	20	79.2	<u>the, and, equipment</u>
<u>Advance</u>	2000	basso	1	98.0	<u>front, rear, w</u>
<u>Advance</u>	2015	basso	5	51.4	<u>and, w, front</u>
<u>Affton</u>	1979	basso	1	28.0	<u>for, to, this</u>
<u>Afton</u>	1975	basso	1	231.0	<u>has, a, and</u>
<u>Afton</u>	2019	medio	4	23.5	<u>a, all, with</u>
<u>Aitkin</u>	2020	alto	1	5.0	<u>rear, front, power</u>
<u>Akron</u>	2009	basso	4	42.5	<u>airbags, power, and</u>
<u>Akron</u>	2012	basso	4	37.25	<u>airbags, power, head</u>
<u>Akron</u>	2018	basso	2	26.5	<u>power, airbags, wheel</u>

Figura 84: Risultato del job 2 in Map-Reduce con benchmark del 50%.

Il job ha impiegato 62.84 secondi.

7.4.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Map-Reduce su AWS sono i seguenti:

<u>Addison</u>	2003	basso	1	8.0	<u>power, wheel, seat</u>
<u>Adel</u>	2020	medio	22	75.14	<u>the, and, equipment</u>
<u>Advance</u>	2000	basso	1	98.0	<u>front, rear, w</u>
<u>Advance</u>	2015	basso	6	47.33	<u>and, w, front</u>
<u>Affton</u>	1979	basso	1	28.0	<u>for, to, this</u>
<u>Afton</u>	1975	basso	1	231.0	<u>has, a, and</u>
<u>Afton</u>	2019	medio	5	30.2	<u>all, a, offers</u>
<u>Aitkin</u>	2020	alto	3	4.33	<u>rear, front, power</u>
<u>Akron</u>	2009	basso	4	42.5	<u>airbags, power, and</u>
<u>Akron</u>	2012	basso	6	44.17	<u>power, airbags, wheel</u>

Figura 85: Risultato del job 2 in Map-Reduce con benchmark del 70%.

Il job ha impiegato 86.88 secondi.

7.4.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Map-Reduce su AWS sono i seguenti:

<u>Adel</u>	2020	medio	36	100.08	<u>the, and, equipment</u>
Advance	2000	basso	1	98.0	<u>front, rear, w</u>
Advance	2015	basso	7	48.43	<u>and, front, w</u>
<u>Affton</u>	1979	basso	1	28.0	<u>for, to, this</u>
<u>Afton</u>	1975	basso	1	231.0	<u>has, a, and</u>
<u>Afton</u>	2013	basso	1	21.0	<u>extremely, spacious, and</u>
<u>Afton</u>	2019	medio	6	27.33	<u>a, all, for</u>
<u>Aitkin</u>	2020	alto	3	4.33	<u>rear, front, power</u>
Akron	2009	basso	6	34.17	<u>airbags, power, and</u>
Akron	2012	basso	11	39.18	<u>power, airbags, wheel</u>

Figura 86: Risultato del job 2 in Map-Reduce con benchmark del 100%.

Il job ha impiegato 105.81 secondi.

7.5 Job 2 -Spark Core

7.5.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Spark Core su AWS sono i seguenti:

```
('Lufkin', 2011, 'basso', 1, 7.0, ['front', 'rear', 'seat'])
('Saint Charles', 2020, 'medio', 8, 142.88, ['front', 'manual', 'seat'])
('Corsicana', 2021, 'basso', 1, 20.0, ['chevrolet', 'front', 'seat'])
('Lawrenceville', 2020, 'alto', 10, 218.2, ['seat', 'front', 'power'])
('Guthrie', 2020, 'alto', 1, 258.0, ['ford'])
('Hayward', 2010, 'basso', 1, 21.0, ['power', 'head', 'tilt'])
('East Rochester', 2020, 'medio', 40, 85.62, ['door', 'front', 'seat'])
('Columbus', 2013, 'medio', 1, 18.0, ['local', 'nice', 'clean'])
('Stillwater', 2021, 'alto', 1, 49.0, ['front', 'door', 'rear'])
('Saint Peters', 2020, 'medio', 5, 140.6, ['in', 'to', 'of'])
25/06/10 15:50:24 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 87: Risultato del job 2 in Spark Core con benchmark del 10%.

Il job ha impiegato 5.15 secondi.

```
25/06/10 15:58:24 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 5.155962 s
```

Figura 88: Tempo impiegato per il job 2 in Spark Core con benchmark del 10%.

7.5.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Spark Core su AWS sono i seguenti:

```
('Benton', 2019, 'medio', 7, 27.0, ['front', 'seat', 'rear'])
('Minot', 2020, 'alto', 8, 97.38, ['door', 'front', 'rear'])
('Kingsport', 2020, 'medio', 76, 108.95, ['front', 'rear', 'seat'])
('Melbourne', 2020, 'medio', 62, 110.71, ['front', 'rear', 'door'])
('Lakewood', 2010, 'basso', 3, 71.0, ['power', 'front', 'rear'])
('Tranquillity', 2020, 'medio', 3, 84.33, ['traverse', 'fwd', 'cloth'])
('Spofford', 2014, 'basso', 3, 102.67, ['power', 'head', 'tilt'])
('Ashtabula', 2020, 'medio', 9, 64.44, ['door', 'seat', 'front'])
('Melbourne', 2017, 'basso', 4, 19.0, ['front', 'seat', 'manual'])
('Scottsdale', 2020, 'medio', 243, 84.17, ['includes', 'front', 'rear'])
25/06/10 15:56:03 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 89: Risultato del job 2 in Spark Core con benchmark del 30%.

Il job ha impiegato 5.46 secondi.

```
25/06/10 15:56:03 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 5.466189 s
```

Figura 90: Tempo impiegato per il job 2 in Spark Core con benchmark del 30%.

7.5.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Spark Core su AWS sono i seguenti:

```
('Southaven', 2019, 'medio', 37, 61.46, ['air', 'side', 'steering'])
('Vienna', 2020, 'medio', 1, 362.0, ['come', 'see', 'newLey'])
('Wiscasset', 2014, 'basso', 1, 246.0, ['one', 'year', 'warranty'])
('Harlingen', 2019, 'basso', 1, 85.0, ['front', 'seat', 'manual'])
('Falls Church', 2020, 'alto', 15, 168.33, ['retail', 'customer', 'ford'])
('Denver', 2019, 'basso', 52, 37.62, ['test', 'drive', 'car'])
('Topeka', 2014, 'medio', 1, 14.0, ['please', 'call', 'us'])
('Phillipsburg', 2014, 'basso', 4, 78.25, ['front', 'seat', 'rear'])
('Hackettstown', 2020, 'medio', 50, 96.92, ['front', 'rear', 'seat'])
('Alva', 2013, 'medio', 1, 254.0, ['butler', 'spike', 'bed'])
25/06/10 18:32:20 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 91: Risultato del job 2 in Spark Core con benchmark del 50%.

Il job ha impiegato 8.09 secondi.

```
25/06/10 18:32:20 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 8.093814 s
```

Figura 92: Tempo impiegato per il job 2 in Spark Core con benchmark del 50%.

7.5.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Spark Core su AWS sono i seguenti:

```
('Tucson', 2013, 'basso', 39, 68.46, ['front', 'door', 'have'])
('Greenville', 2020, 'alto', 46, 84.61, ['front', 'seat', 'power'])
('Orland Park', 2020, 'alto', 46, 88.52, ['front', 'power', 'seat'])
('North Hollywood', 2020, 'alto', 41, 164.46, ['please', 'contact', 'us'])
('Tucson', 2017, 'basso', 103, 47.98, ['front', 'door', 'steering'])
('Mcallen', 2021, 'medio', 17, 17.71, ['door', 'chevrolet', 'seat'])
('Manassas', 2021, 'medio', 22, 9.95, ['front', 'seat', 'rear'])
('Auburn', 2021, 'medio', 71, 30.63, ['enjoy', 'nice', 'weather'])
('Fowlerville', 2020, 'medio', 16, 82.81, ['of', 'it', 'metallic'])
('Garland', 2015, 'basso', 22, 391.18, ['power', 'head', 'dual'])
25/06/10 19:06:37 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 93: Risultato del job 2 in Spark Core con benchmark del 70%.

Il job ha impiegato 9.76 secondi.

```
25/06/10 19:06:37 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 9.764359 s
```

Figura 94: Tempo impiegato per il job 2 in Spark Core con benchmark del 70%.

7.5.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Spark Core su AWS sono i seguenti:

```
(('Fort Worth', 2013, 'basso', 9, 37.33, ['power', 'head', 'wheel']))
('Monticello', 2018, 'medio', 5, 24.8, ['front', 'seat', 'rear'])
('Paterson', 2010, 'basso', 2, 25.0, ['rear', 'front', 'pwr'])
('Southold', 2016, 'medio', 1, 222.0, ['digital', 'test', 'ford'])
('Lynnfield', 2019, 'medio', 5, 74.4, ['is', 'one', 'of'])
('Storrs Mansfield', 2016, 'basso', 1, 309.0, [])
('Springfield', 2018, 'medio', 7, 33.43, ['rear', 'front', 'door'])
('Roselle', 2016, 'basso', 2, 61.0, ['front', 'seat', 'rear'])
('Troy', 2016, 'basso', 1, 7.0, ['front', 'seat', 'power'])
('Eatontown', 2003, 'basso', 1, 8.0, ['front', 'brakes', 'part'])
25/06/10 19:20:39 INFO SparkContext: SparkContext is stopping with exitCode 0.
```

Figura 95: Risultato del job 2 in Spark Core con benchmark del 100%.

Il job ha impiegato 12.76 secondi.

```
25/06/10 19:20:39 INFO DAGScheduler: Job 1 finished: runJob at PythonRDD.scala:191, took 12.757617 s
```

Figura 96: Tempo impiegato per il job 2 in Spark Core con benchmark del 100%.

7.6 Job 2 - Spark SQL

7.6.1 Benchmark del 10%

I risultati ottenuti dal running del job 2 tramite Spark SQL su AWS sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Abilene	2018	alto	1	6.0	air, door, audio
Acton	2017	basso	2	42.0	nl, air, steering
Ada	2013	medio	1	23.0	front, door, rear
Addison	2009	basso	5	33.8	front, rear, type
Adrian	2016	basso	1	72.0	power, at, to
Adrian	2018	basso	1	11.0	door, front, magnetic
Airway Heights	2018	medio	3	57.33	to, air, power
Alachua	2017	basso	1	1.0	air, door, wheel
Alachua	2018	basso	1	20.0	air, rear, trunk
Alameda	2018	medio	1	30.0	is, of, our

only showing top 10 rows

Figura 97: Risultato del job 2 in Spark SQL con benchmark del 10%.

Il job ha impiegato 2.92 secondi.


```
25/06/11 09:25:34 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 2.917542 s
```

Figura 98: Tempo impiegato per il job 2 in Spark SQL con benchmark del 10%.

7.6.2 Benchmark del 30%

I risultati ottenuti dal running del job 2 tramite Spark SQL su AWS sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Abilene	2018	alto	1	6.0	air, door, audio
Acton	2017	basso	6	63.0	nl, air, rear
Ada	2013	medio	2	16.0	front, door, rear
Addison	1980	medio	1	239.0	is, leather, it
Addison	1996	basso	1	78.0	super, low, miles
Addison	2009	basso	14	41.57	front, rear, seat
Addison	1969	alto	2	18.5	in, to, is
Adel	2019	basso	1	48.0	air, ford, our
Adrian	2016	basso	2	50.0	door, steering, front
Adrian	2018	basso	1	11.0	door, front, magnetic

only showing top 10 rows

Figura 99: Risultato del job 2 in Spark SQL con benchmark del 30%.

Il job ha impiegato 4.98 secondi.

```
25/06/11 09:32:52 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 4.980906 s
```

Figura 100: Tempo impiegato per il job 2 in Spark SQL con benchmark del 30%.

7.6.3 Benchmark del 50%

I risultati ottenuti dal running del job 2 tramite Spark SQL su AWS sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Acton	2017	basso	7	79.43	nl,air,rear
Addison	1980	medio	1	239.0	is,leather,it
Addison	2007	medio	2	37.0	chrome,only,of
Addison	2009	basso	18	70.39	front,rear,seat
Addison	1969	alto	2	18.5	in,to,is
Addison	2000	basso	1	25.0	power,wheel,front
Adel	2019	basso	2	65.5	front,rear,type
Adrian	2016	basso	2	50.0	door,steering,front
Aiken	2005	basso	2	291.0	air,conditioning,...
Aiken	2015	medio	1	19.0	front,instrument,...

only showing top 10 rows

Figura 101: Risultato del job 2 in Spark SQL con benchmark del 50%.

Il job ha impiegato 6.69 secondi.

```
25/06/11 09:36:22 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 6.688090 s
```

Figura 102: Tempo impiegato per il job 2 in Spark SQL con benchmark del 50%.

7.6.4 Benchmark del 70%

I risultati ottenuti dal running del job 2 tramite Spark SQL su AWS sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Acton	2017	basso	15	75.6	nl,air,rear
Addison	1980	medio	1	239.0	is,leather,it
Addison	2007	medio	3	39.67	in,of,rear
Addison	2011	alto	1	91.0	brabus,to,is
Addison	1969	alto	2	18.5	in,to,is
Adel	2019	basso	2	65.5	front,rear,type
Adrian	2016	basso	2	50.0	door,steering,front
Aiken	2000	basso	1	197.0	door,air,wheels
Aiken	2015	medio	1	19.0	front,instrument,...
Airway Heights	2018	medio	7	52.57	to,power,air

only showing top 10 rows

Figura 103: Risultato del job 2 in Spark SQL con benchmark del 70%.

Il job ha impiegato 6.79 secondi.

```
25/06/11 09:45:35 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 6.798392 s
```

Figura 104: Tempo impiegato per il job 2 in Spark SQL con benchmark del 70%.

7.6.5 Benchmark del 100%

I risultati ottenuti dal running del job 2 tramite Spark SQL su AWS sono i seguenti:

city	year	fascia	num_macchine	avg_daysonmarket	top_3_words
Addison	1980	medio	1	239.0	is, leather, it
Addison	2011	alto	1	91.0	brabus, to, is
Addison	1969	alto	2	18.5	in, to, is
Adrian	2016	basso	3	37.0	door, steering, wheel
Ainsworth	2015	medio	2	440.5	steering, at, to
Airway Heights	2018	medio	12	52.33	to, power, air
Alachua	2017	basso	6	67.83	air, door, vanity
Alamosa	2020	alto	8	27.13	power, heated, remote
Albany	2013	basso	21	66.29	front, rear, door
Albion	2016	basso	2	214.5	front, rear, type

only showing top 10 rows

Figura 105: Risultato del job 2 in Spark SQL con benchmark del 100%.

Il job ha impiegato 6.25 secondi.

```
25/06/11 10:06:16 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 6.256098 s
```

Figura 106: Tempo impiegato per il job 2 in Spark SQL con benchmark del 100%.

7.7 Job 2 - Risultati

Per quanto riguarda l'elaborazione del secondo job su AWS, i tempi di esecuzione di *MapReduce* risultano molto più elevati rispetto alle altre tecnologie e aumentano in modo piuttosto lineare con l'aumentare del volume di dati. Nonostante la scalabilità offerta da AWS, MapReduce continua ad avere un collo di bottiglia della scrittura e lettura da disco tra le fasi, che ne limita fortemente la velocità di esecuzione anche su infrastrutture distribuite.

Spark Core, invece, beneficia in modo evidente dell'ambiente distribuito offerto

da AWS. I tempi di esecuzione rimangono bassi per i benchmark inferiori. Questo risultato rappresenta un miglioramento netto rispetto all'esecuzione locale, dove Spark Core aveva avuto un andamento più altalenante.

Spark SQL, infine, si conferma la tecnologia più efficiente anche in questo contesto. I suoi tempi sono i più bassi in assoluto per quasi tutti i benchmark. A differenza dell'esecuzione locale, dove Spark SQL mostrava una crescita più visibile con l'aumento dei dati, su AWS riesce a mantenere prestazioni estremamente costanti.

Questo suggerisce che Spark SQL sia in grado di sfruttare appieno le ottimizzazioni del piano di esecuzione e il parallelismo nativo offerto da Spark in ambienti cloud.

Questi risultati rafforzano ulteriormente la validità di Spark come soluzione di riferimento per l'elaborazione distribuita moderna, con Spark SQL come scelta privilegiata.

Benchmarks	Map-Reduce (s)	Spark Core (s)	Spark SQL (s)
10%	30.07	5.15	2.91
30%	42.72	5.46	4.98
50%	62.84	8.09	6.69
70%	86.88	9.76	6.79
100%	105.81	12.76	6.25

Tabella 4: Tempi di esecuzione su AWS per il secondo job in base al diverso benchmark e alla diversa tecnologia.

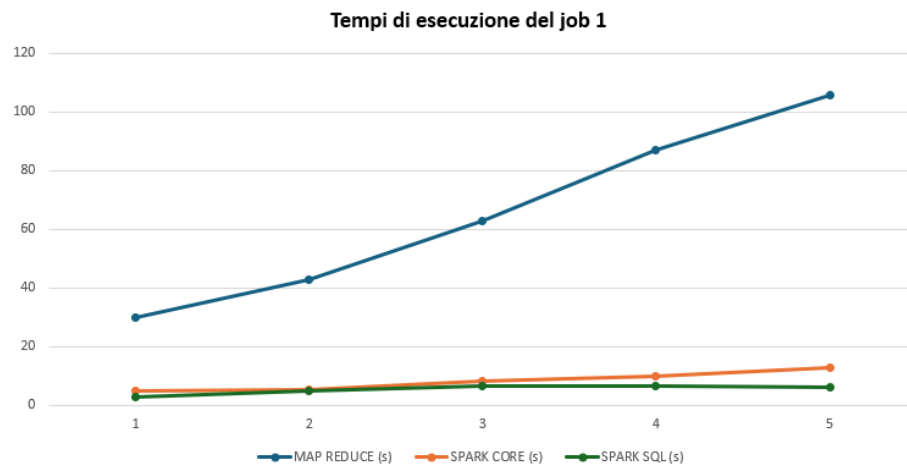


Figura 107: Tempi di esecuzione su AWS del job 2 a confronto.

8 Confronto

8.1 Job 1

8.1.1 Map-Reduce

Dal confronto tra l'esecuzione locale e su AWS del primo job con MapReduce emerge un dato sorprendente: i tempi sono nettamente migliori in locale. Mentre in locale si osserva una crescita lineare dei tempi al crescere del dataset, su AWS i tempi sono molto più alti ma quasi costanti. Questo suggerisce che l'overhead iniziale su AWS, forse dovuto all'orchestrazione del cluster o all'I/O distribuito, non viene compensato da un reale vantaggio in parallelo per job di questa scala. In sintesi, per job relativamente piccoli o medi, MapReduce su AWS risulta inefficiente rispetto all'esecuzione su macchina locale.

Benchmarks	Locale (s)	AWS (s)
10%	3.88	27.48
30%	5.86	27.57
50%	6.89	28.91
70%	9.85	30.57
100%	11.87	29.59

Tabella 5: Tempi di esecuzione per il primo job in base al diverso benchmark.

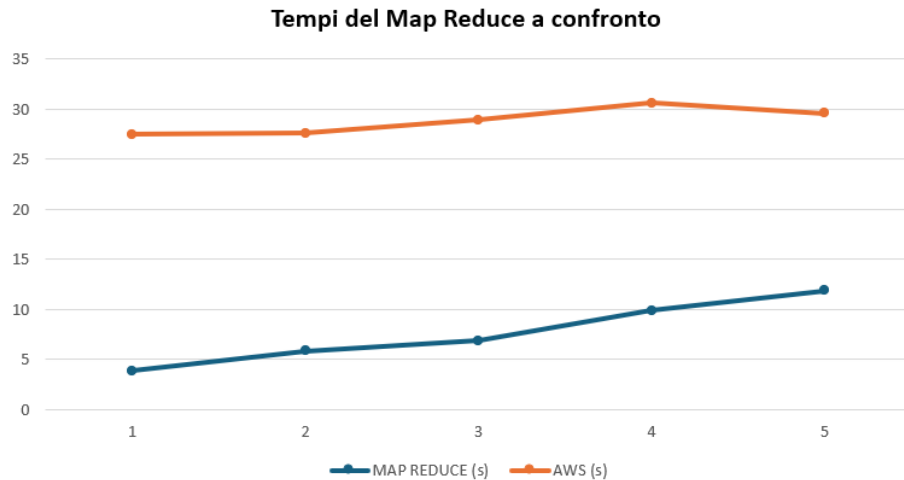


Figura 108: Tempi di esecuzione del job 1 a confronto.

8.1.2 Spark Core

Nel confronto tra l'esecuzione del primo job con Spark Core in locale e su AWS, i risultati mostrano una tendenza interessante. Per i benchmark più piccoli, i tempi sono molto simili o leggermente migliori in locale. Tuttavia, con l'aumento del volume di dati, l'ambiente locale mantiene prestazioni migliori rispetto ad AWS. Questo indica che, per questo specifico job, Spark Core non riesce a sfruttare pienamente i vantaggi del cluster AWS, forse a causa di un overhead distributivo superiore ai benefici del parallelismo.

Benchmarks	Locale (s)	AWS (s)
10%	1.48	1.41
30%	2.58	3.41
50%	3.84	4.13
70%	5.08	8.08
100%	7.59	9.45

Tabella 6: Tempi di esecuzione per il primo job in base al diverso benchmark.

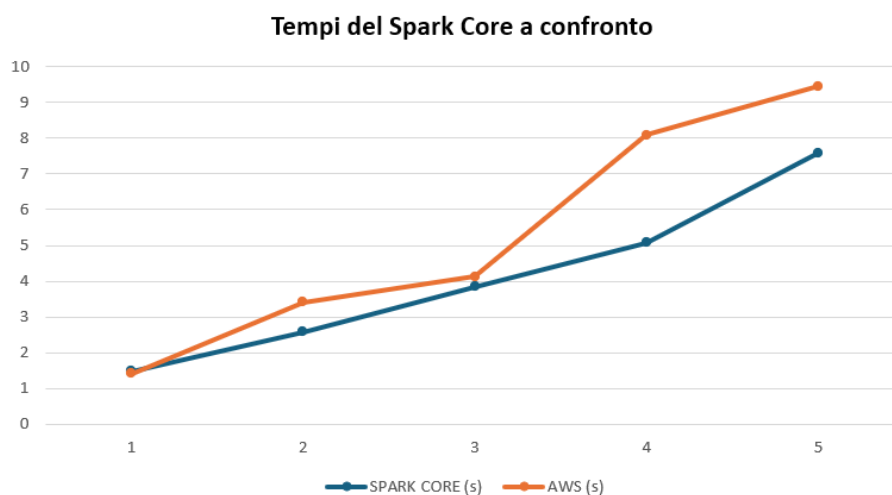


Figura 109: Tempi di esecuzione del job 1 a confronto.

8.1.3 Spark SQL

Nel confronto tra l'esecuzione del primo job con Spark SQL in locale e su AWS, emerge chiaramente che l'ambiente locale offre prestazioni nettamente superiori. Questo suggerisce che, per job molto leggeri e ottimizzati come questo, l'overhead introdotto dall'infrastruttura distribuita su AWS può superare i benefici del parallelismo. In questi casi, un'esecuzione locale ben configurata si dimostra più efficiente.

Benchmarks	Locale (s)	AWS (s)
10%	0.22	1.83
30%	0.38	1.79
50%	0.29	0.44
70%	0.32	1.73
100%	0.39	1.73

Tabella 7: Tempi di esecuzione per il primo job in base al diverso benchmark.

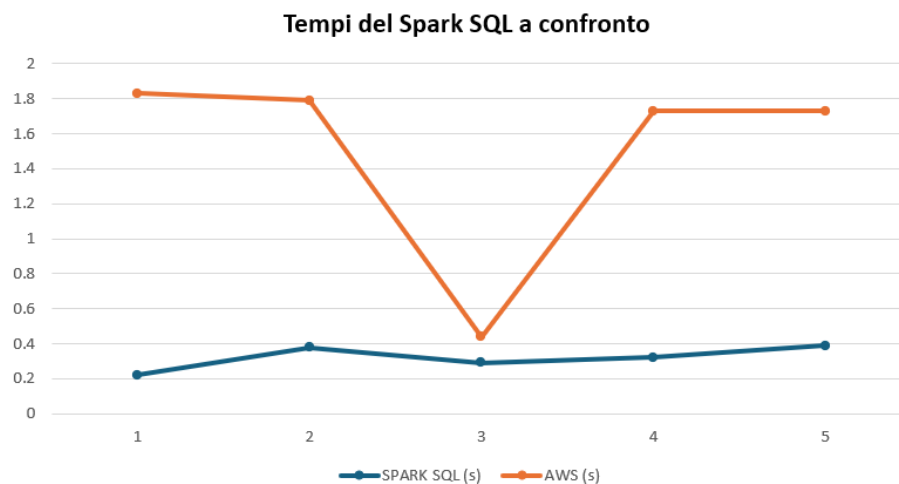


Figura 110: Tempi di esecuzione del job 1 a confronto.

8.2 Job 2

8.2.1 Map-Reduce

Nel confronto tra l'esecuzione del secondo job con MapReduce in locale e su AWS, si osserva che i tempi su AWS sono significativamente più alti in ogni benchmark rispetto all'ambiente locale. Sebbene AWS offra scalabilità e risorse distribuite, l'overhead e la latenza legati alla gestione del cluster e all'I/O su disco sembrano penalizzare fortemente le prestazioni di MapReduce. Questo indica che, per questo tipo di job, l'infrastruttura locale risulta più efficiente, soprattutto quando il carico non è sufficientemente grande da sfruttare appieno il parallelismo del cluster cloud.

Benchmarks	Locale (s)	AWS (s)
10%	9.53	30.07
30%	21.04	42.72
50%	35.73	62.84
70%	43.83	86.88
100%	71.77	105.81

Tabella 8: Tempi di esecuzione per il secondo job in base al diverso benchmark.

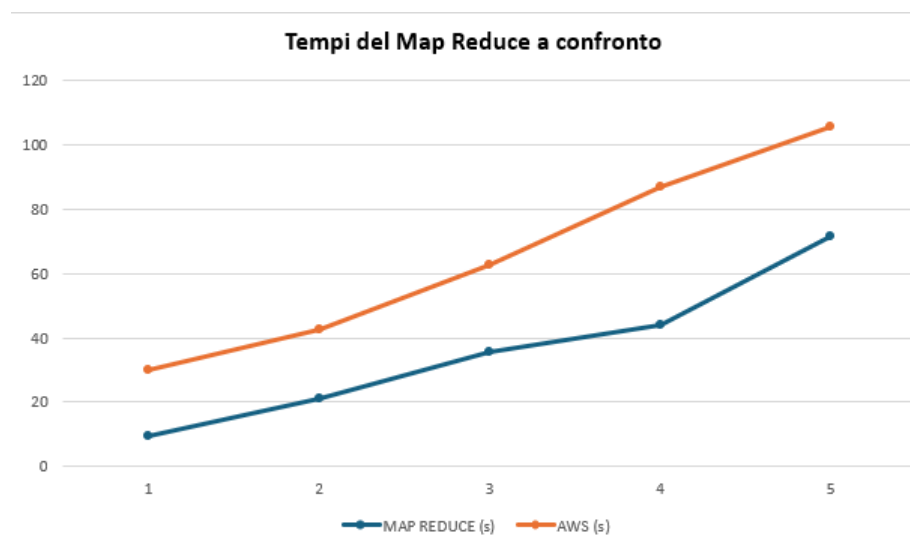


Figura 111: Tempi di esecuzione del job 2 a confronto.

8.2.2 Spark Core

Nel confronto tra l'esecuzione del secondo job con Spark Core in locale e su AWS, si nota un netto vantaggio dell'ambiente cloud. I tempi su AWS sono significativamente più bassi per tutti i benchmark, specialmente per le dimensioni di dataset più grandi, dove Spark Core su AWS dimezza o addirittura riduce ulteriormente i tempi rispetto alla versione locale. Questo indica che Spark Core sfrutta efficacemente le risorse distribuite di AWS, migliorando la scalabilità e riducendo i tempi di esecuzione per job complessi rispetto a un ambiente locale.

Benchmarks	Locale (s)	AWS (s)
10%	10.75	5.15
30%	17.86	5.46
50%	39.19	8.09
70%	57.23	9.76
100%	50.12	12.76

Tabella 9: Tempi di esecuzione per il secondo job in base al diverso benchmark.

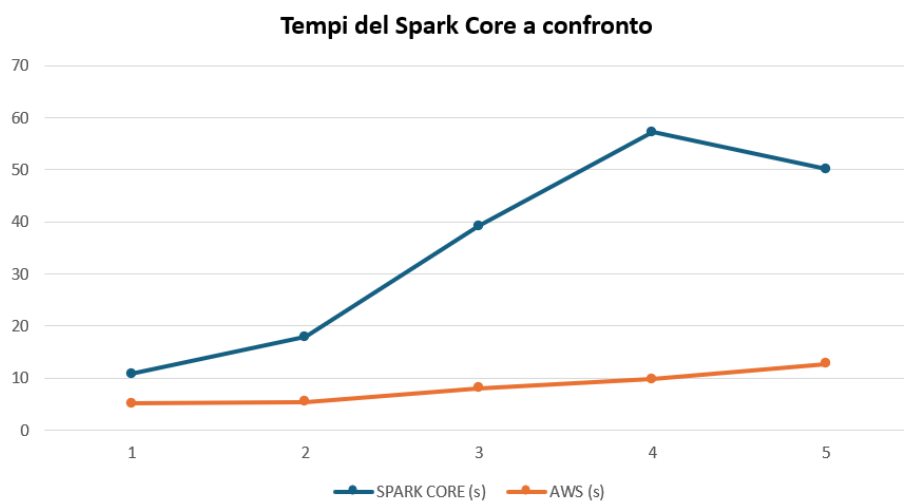


Figura 112: Tempi di esecuzione del job 2 a confronto.

8.2.3 Spark SQL

Nel confronto tra l'esecuzione del secondo job con Spark SQL in locale e su AWS emerge chiaramente il vantaggio dell'ambiente cloud. I tempi su AWS sono costantemente più bassi e, soprattutto per i dataset più grandi, si mantengono quasi stabili attorno ai 6-7 secondi, mentre in locale i tempi crescono in modo significativo fino a superare i 23 secondi al 100%. Questo dimostra che Spark SQL sfrutta al meglio le capacità di parallelismo e ottimizzazione offerte da AWS, garantendo prestazioni più rapide e scalabili per job complessi.

Benchmarks	Locale (s)	AWS (s)
10%	3.52	2.91
30%	7.49	4.98
50%	9.24	6.69
70%	13.16	6.79
100%	23.23	6.25

Tabella 10: Tempi di esecuzione per il secondo job in base al diverso benchmark.

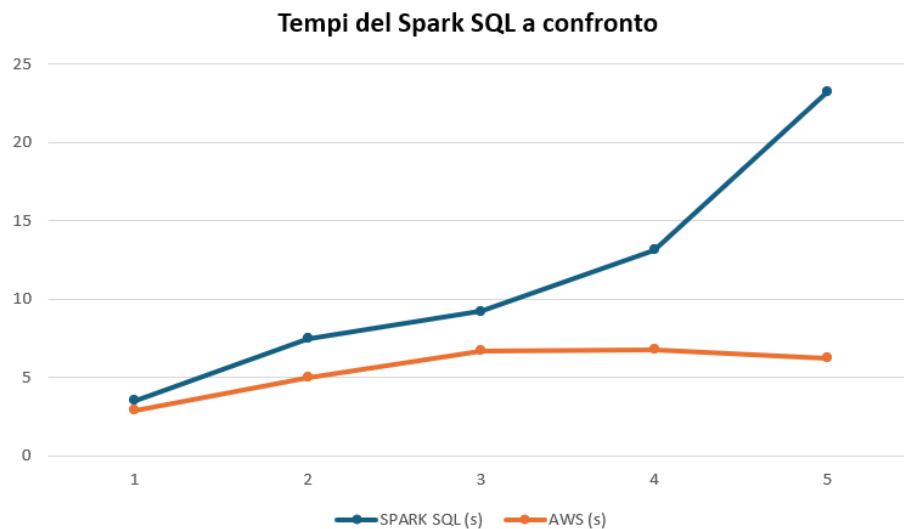


Figura 113: Tempi di esecuzione del job 2 a confronto.

9 Considerazioni finali

I risultati ottenuti evidenziano come le prestazioni dipendano da molteplici fattori, tra cui la tecnologia utilizzata, il contesto di esecuzione e la natura del job analizzato.

In generale, *MapReduce* si è dimostrato la soluzione meno performante tra le tre. La sua architettura, basata su letture e scritture ripetute su disco tra le fasi di map e reduce, rappresenta un limite strutturale, che emerge con particolare evidenza soprattutto su dataset di dimensioni più elevate o su infrastrutture distribuite come AWS, dove l'overhead di coordinamento e comunicazione incide pesantemente sulle prestazioni.

Spark Core, grazie al paradigma in-memory e alla gestione più efficiente delle operazioni distribuite, ha mostrato una scalabilità e una velocità nettamente superiori rispetto a MapReduce, soprattutto quando eseguito su AWS. La possibilità di sfruttare il parallelismo distribuito e una migliore gestione delle risorse di calcolo rende Spark Core particolarmente adatto a job più complessi.

Spark SQL si è confermato la tecnologia più efficiente in quasi tutti i casi analizzati, soprattutto per job che possono essere espressi tramite query dichiarative. Infine, l'analisi dei risultati tra esecuzione locale e cloud evidenzia come il passaggio a un'infrastruttura distribuita come AWS possa migliorare significativamente le prestazioni di Spark Core e Spark SQL, mentre per MapReduce i benefici sono meno evidenti, spesso compensati dall'overhead di gestione del cluster. Questo suggerisce che per ottenere il massimo dai sistemi distribuiti moderni, è fondamentale utilizzare tecnologie progettate per sfruttare il calcolo in-memory e il parallelismo avanzato.