PROGETTO COMUNE MapReduce, Hive e Spark
Big Data

Davide Orienti

533107

Indice Generale

1 Introduzione

Il presente progetto nasce nell'ambito del corso di Big Data Analytics e si propone di esplorare, analizzare e confrontare l'efficienza e l'efficacia di diverse tecnologie per l'elaborazione di grandi volumi di dati. Il dataset utilizzato, intitolato "US Used Cars Dataset" e disponibile sulla piattaforma Kaggle, contiene circa 3 milioni di record relativi ad auto usate in vendita negli Stati Uniti fino al 2020, con 66 colonne che descrivono in dettaglio caratteristiche tecniche e commerciali dei veicoli.

L'obiettivo principale del progetto è duplice: da un lato, estrarre conoscenza utile e strutturata da questo vasto insieme di dati; dall'altro, valutare e documentare le differenze implementative e prestazionali tra tre tecnologie di riferimento nel panorama Big Data: MapReduce, Hive e Spark SQL.

A tal fine, sono stati progettati e sviluppati due job di analisi:

Il primo genera statistiche per marca e modello di automobile, includendo prezzi, frequenze e distribuzione temporale;

Il secondo fornisce una panoramica per città e anno, suddividendo i veicoli in fasce di prezzo, con analisi sui tempi di permanenza sul mercato e un'estrazione semantica delle parole più frequenti nelle descrizioni.

Ciascun job è stato implementato nelle tre tecnologie richieste e testato su dataset di dimensioni crescenti (100k, 1M, 3M record), al fine di effettuare un confronto oggettivo dei tempi di esecuzione e della scalabilità. La relazione descrive nel dettaglio le scelte progettuali, le trasformazioni effettuate sui dati, i risultati ottenuti e le considerazioni finali sull'efficienza delle soluzioni adottate.

2 Capitolo 1: Studio del Dataset

Il dataset analizzato è intitolato **"US Used Cars Dataset"** ed è stato scaricato dalla piattaforma Kaggle. Contiene circa **3 milioni di record** che descrivono auto usate in vendita negli Stati Uniti fino all'anno 2020. Ogni record rappresenta un annuncio di vendita e comprende informazioni tecniche, temporali, geografiche e testuali.

2.1 Struttura del Dataset

Il dataset è in formato CSV e presenta 66 colonne. Le variabili più significative per le analisi svolte sono:

- `make_name`: marca del veicolo (es. Ford, Toyota, BMW)
- `model_name`: modello del veicolo (es. Mustang, Corolla, X5)
- `price`: prezzo di vendita in dollari
- `year`: anno di produzione dell'auto
- `city`: città in cui il veicolo è messo in vendita
- `daysonmarket`: numero di giorni in cui l'auto è rimasta sul mercato
- `horsepower`: potenza del motore (in cavalli)

- `engine_displacement`: cilindrata del motore (in litri)
- `description`: descrizione testuale dell'annuncio
- `latitude`, `longitude`: coordinate geografiche
- `state`: stato USA associato alla città

2.2 Analisi Esplorativa Iniziale

- **Valori nulli e inconsistenti**: numerose colonne presentavano valori mancanti o vuoti. I record con campi fondamentali assenti (es. `make_name`, `model_name`, `price`, `year`, `city`) sono stati rimossi.
- **Distribuzione dei prezzi**: il prezzo varia da meno di 1.000 a oltre 300.000 dollari. È stata definita una classificazione in fasce:
  - *bassa*: prezzo inferiore a 20.000 dollari
  - *media*: tra 20.000 e 50.000 dollari
  - *alta*: superiore a 50.000 dollari
- **Distribuzione degli anni**: gli anni di produzione variano dai primi anni '90 al 2020, con maggiore concentrazione tra il 2005 e il 2018.
- **Marche e modelli**: il dataset comprende centinaia di marche e migliaia di modelli. Alcuni (es. Toyota Camry, Ford F-150) sono molto frequenti, altri molto rari.
- **Descrizione testuale**: il campo `description` presenta contenuti molto variabili, da una riga a interi paragrafi. È stato tokenizzato, convertito in minuscolo e ripulito dalla punteggiatura per analisi semantiche.

2.3 Trasformazioni Effettuate

- Pulizia dei record incompleti o inconsistenti
- Filtraggio delle colonne non essenziali
- Normalizzazione dei valori testuali (es. correzione di errori nei nomi delle marche)
- Suddivisione del dataset in tre file di dimensioni diverse: `100k.csv`, `1M.csv`, `3M.csv`

3 Pulizia del Dataset

La pulizia dei dati è stata effettuata tramite un notebook Python denominato `clean.ipynb`. L'obiettivo era rimuovere record inconsistenti, incompleti o non significativi, migliorando la qualità del dataset prima dell'elaborazione con le tecnologie MapReduce, Hive e Spark SQL.

3.1 Lettura e analisi iniziale

Il dataset originale è stato caricato utilizzando `pandas`:

```
import pandas as pd
df = pd.read_csv("US_Used_Cars_dataset.csv")
```

Successivamente è stata effettuata un'analisi esplorativa iniziale per individuare:

- La presenza di valori nulli con `df.isnull().sum()`
- La distribuzione dei valori nei campi chiave (`price`, `year`, `make_name`, `model_name`)
- Outlier nel campo `price` (es. auto a prezzo 0 o superiore a 300.000 dollari)

3.2 Criteri di pulizia

Sono stati applicati i seguenti filtri:

- Rimozione dei record con valori nulli nei campi essenziali:

  ```
  df.dropna(subset=["make_name", "model_name", "price", "year", "city"], inplace=Tr
  ```

- Rimozione dei record con valori anomali nel campo `price`:

  ```
  df = df[(df["price"] > 1000) & (df["price"] < 300000)]
  ```

- Conversione del campo `year` in intero, eliminando i valori non numerici:

  ```
  df = df[pd.to_numeric(df["year"], errors="coerce").notnull()]
  df["year"] = df["year"].astype(int)
  ```

- Normalizzazione dei nomi (minuscolo e stripping):

  ```
  df["make_name"] = df["make_name"].str.strip().str.lower()
  df["model_name"] = df["model_name"].str.strip().str.lower()
  ```

- Rimozione dei duplicati:

  ```
  df.drop_duplicates(inplace=True)
  ```

3.3 Esportazione del dataset pulito

Il file pulito è stato salvato in formato CSV con il nome `dataset_full_cleaned.csv`, come segue:

```
df.to_csv("dataset_full_cleaned.csv", index=False)
```

3.4 Risultati della pulizia

Al termine della pulizia:

- I record sono passati da circa 3 milioni a circa 2.4 milioni.
- Sono stati rimossi circa 600.000 record contenenti dati nulli o inconsistenti.
- Le variabili selezionate risultano coerenti, ben formattate e pronte per essere utilizzate nei job analitici.

3.5 Considerazioni

La fase di pulizia ha avuto un impatto significativo sulla qualità dei dati e sulle prestazioni successive. Eliminare dati incompleti ha evitato errori in fase di aggregazione e ridotto il tempo di elaborazione. La trasformazione dei dati testuali in formato coerente ha anche facilitato l'analisi semantica nelle descrizioni.

4 Job 1

L'obiettivo del primo job è analizzare le auto presenti nel dataset per marca (`make_name`) e modello (`model_name`), fornendo per ciascun modello le seguenti statistiche:

- Numero di auto presenti nel dataset
- Prezzo minimo, massimo e medio
- Anni di produzione (elenco di anni)

4.1 Soluzione con MapReduce

La soluzione MapReduce è stata implementata tramite due script Python: `mapper.py` e `reducer.py`, eseguiti dallo script `run.sh`.

Mapper

Il mapper legge ogni riga del dataset e, dopo aver saltato l'header, estrae i campi `make`, `model`, `price` e `year`. L'output è una coppia chiave-valore nel formato:

```
make|model \t price|year
```

Codice principale:

```
key = f"{make}|{model}"
value = f"{price}|{year}"
print(f"{key}\t{value}")
```

Reducer

Il reducer aggrega i valori ricevuti per ciascuna chiave. Calcola:

- Il numero di auto per modello (conteggio dei prezzi)
- Il prezzo minimo, massimo e medio
- L'elenco degli anni unici ordinati

Output:

```
make,model,count,min_price,max_price,avg_price,[years]
```

Esempio:

```
toyota,corolla,154,5000.0,18000.0,10250.5,[2012, 2013, 2014]
```

4.2 Soluzione con Hive

La versione Hive è stata realizzata tramite lo script `job1_hive.hql`, eseguito con `run_hive_job1.sh`. Il flusso di lavoro prevede:

- Creazione della tabella Hive `used_cars`
- Caricamento dei dati da HDFS
- Esecuzione della query `GROUP BY make_name, model_name` con funzioni di aggregazione
- Salvataggio dei risultati nella tabella `job1_stats`

Query principale:

```
SELECT make_name, model_name,
       COUNT(*) as num_cars,
       MIN(price), MAX(price), AVG(price),
       COLLECT_SET(year) as years_present
FROM used_cars
GROUP BY make_name, model_name;
```

Nota: Hive utilizza `COLLECT_SET` per ottenere gli anni distinti.

4.3 Soluzione con Spark SQL

Lo script PySpark `job1_sparksql.py` implementa il job in Spark SQL. Le fasi sono:

1. Inizializzazione della `SparkSession`
2. Lettura del file CSV da HDFS con inferenza schema
3. Pulizia dei dati: filtro su `price`, `make_name`, `model_name`
4. Creazione di una vista SQL temporanea
5. Esecuzione di una query analoga a quella Hive

Query SQL eseguita:

```
SELECT make_name, model_name,
       COUNT(*) as num_cars,
       MIN(price) as min_price,
       MAX(price) as max_price,
       ROUND(AVG(price), 2) as avg_price,
       COLLECT_SET(year) as years_present
FROM used_cars
GROUP BY make_name, model_name
ORDER BY make_name, model_name
```

Output: visualizzato con `.show(10, truncate=False)` e salvato su disco.

5 Job 2

Il secondo job ha l'obiettivo di generare un report che, per ogni combinazione di città (`city`) e anno (`year`), analizzi le auto in vendita suddividendole per fascia di prezzo. Per ciascuna fascia (*bassa*, *media*, *alta*) si desidera calcolare:

- Il numero di auto presenti nella fascia
- La media dei giorni di permanenza sul mercato (`daysonmarket`)
- Le tre parole più frequenti nella descrizione del veicolo (`description`)

5.1 Soluzione con MapReduce

Questa versione è implementata tramite `mapper.py`, `reducer.py` e uno script di esecuzione `run.sh`.

Mapper

Il mapper legge ogni riga e genera come chiave il tripletto:

`city,year,price_range`

dove la fascia di prezzo è determinata tramite:

- `low`: prezzo ¡ 20.000 dollari
- `medium`: 20.000 $\leq$ prezzo $\leq$ 50.000
- `high`: prezzo ¿ 50.000 dollari

Le descrizioni sono tokenize-izzate in parole, rimuovendo punteggiatura e normalizzando in minuscolo.

Reducer

Il reducer aggrega i valori per chiave (`city,year,range`), calcola:

- **Totale auto** = somma dei contatori
- **Media giorni** = media di `daysonmarket`
- **Top 3 parole** = parole più frequenti tramite `Counter`

5.2 Soluzione con Hive

Il file `job2_hive.hql` implementa la versione Hive, con:

- Creazione della tabella Hive
- Definizione di una colonna aggiuntiva `price_band` tramite `CASE WHEN`
- Estrazione delle tre parole più frequenti tramite `LATERAL VIEW` e `GROUP BY`
- Aggregazione su `city, year, price_band`

Esempio di assegnazione fascia di prezzo:

```
CASE
  WHEN price < 20000 THEN 'low'
  WHEN price <= 50000 THEN 'medium'
  ELSE 'high'
END AS price_band
```

5.3 Soluzione con Spark SQL

Lo script `job2_sparksql.py` è il più completo ed efficiente tra le tre soluzioni. Esso prevede:

Fasi principali

1. **Caricamento dati** con schema definito manualmente
2. **Assegnazione fascia di prezzo** con `withColumn` e `when`
3. **Creazione vista SQL**: `cars`
4. **Query statistica**: conta auto e media `daysonmarket`
5. **Tokenizzazione parole** con `explode`, `split`, `lower`
6. **Conteggio parole** e selezione delle top 3 per ciascun gruppo con `Window Function`
7. **Join** dei due risultati

Esempio di query per parole più frequenti

```
SELECT city, year, price_band, word, COUNT(*) as word_count
FROM words
GROUP BY city, year, price_band, word
```

Filtro top 3 parole: si usa `row_number()` e `collect_list()` per ogni gruppo.

5.4 Confronto tra le soluzioni

- **MapReduce**: offre controllo diretto, ma richiede molte operazioni manuali per la tokenizzazione e il conteggio.
- **Hive**: più semplice da implementare rispetto a MapReduce, ma limitato nel trattamento avanzato delle stringhe.
- **Spark SQL**: la soluzione più elegante ed efficiente, con funzioni native per gestire stringhe, aggregazioni e ranking.

5.5 Output

Tutte le tecnologie producono un output confrontabile con i seguenti campi:

- Città, anno, fascia di prezzo
- Numero di auto e media dei giorni sul mercato
- Tre parole più ricorrenti nella descrizione

Questi risultati sono stati salvati e confrontati nelle successive analisi di benchmark.

6 Automazione dei test e benchmarking comparativo

Al fine di eseguire in modo sistematico e comparabile i due job progettati con tre differenti tecnologie (MapReduce, Hive, Spark SQL) e con dataset di dimensioni crescenti (100k, 1M, 3M), sono stati sviluppati tre script Bash di orchestrazione automatica:

- `benchmarck100k.sh`
- `benchmarck1M.sh`
- `benchmarck3M.sh`

Questi script sono collocati nella root del progetto e si occupano di:

1. Selezionare il file CSV corrispondente alla dimensione del dataset desiderato (es. `100k.csv`);
2. Sovrascrivere il file di riferimento usato da tutti gli altri script, ovvero `dataset_full_cleaned.csv`;
3. Caricare il file selezionato anche in HDFS nel path corretto atteso da Hive e Spark SQL;

4. Eseguire in sequenza tutti gli script di job relativi a MapReduce, Hive e Spark SQL per Job 1 e Job 2;

5. Stampare l'output in tempo reale e salvarlo nei file log (`.log`) all'interno della directory `output/logs/`;

6. Calcolare e salvare il tempo di esecuzione per ciascun job in un file `tempo_totale_<dataset>.txt`, in formato CSV.

Meccanismo di selezione del dataset

Ogni script inizia con la selezione del dataset target, mediante le istruzioni:

```
DATASET="100k"
cp "./data/${DATASET}.csv" "./data/dataset_full_cleaned.csv"
hdfs dfs -put -f "./data/${DATASET}.csv" \
  "/user/hive/warehouse/car_data/input/dataset_full_cleaned.csv"
```

In questo modo, tutti gli script downstream (i vari `run.sh`) che lavorano su `dataset_full_cleaned.c` non devono essere modificati e continuano a funzionare con qualsiasi dimensione del dataset.

Funzione di temporizzazione

Il cuore dello script è la funzione `esegui_con_timer`, che esegue ciascun `run.sh`, stampa l'output in console, salva i log ed esegue la temporizzazione:

```
esegui_con_timer() {
  local script="$1"
  local label="$2"
  local log_path="$LOG_DIR/$(basename "$script" .sh)_${DATASET}.log"

  echo -e "\n>>> Avvio: $label"
  start=$(date +%s)

  script_dir=$(dirname "$script")
  (
    cd "$script_dir" || exit 1
    bash "$(basename "$script")"
  ) 2>&1 | tee "$log_path"

  end=$(date +%s)
  tempo=$((end - start))

  echo "$label,$tempo" >> "$RESULT_FILE"
  echo "<<< Completato: $label — Tempo: ${tempo}s"
}
```

L'uso della subshell `(cd ...)` consente di eseguire ogni job nella sua directory originale, preservando l'uso corretto dei path relativi all'interno degli script secondari.

Esecuzione dei job

Ogni script `benchmarck*.sh` esegue, in ordine, i seguenti sei job:

1. `./job1/Map_Reduce/run.sh`
2. `./job1/Hive/run_hive_job1.sh`
3. `./job1/Spark_sql/run.sh`
4. `./job2/Map_Reduce/run.sh`
5. `./job2/Hive/run_hive_job2.sh`
6. `./job2/Spark_sql/run.sh`

A ciascuno viene associata una label descrittiva (`Job1 MapReduce`, `Job2 Hive`, ecc.) per favorire la lettura dei log e dei report CSV.

Output finale

Al termine dell'esecuzione, ogni script produce:

- un file `tempo_totale_<dataset>.txt` contenente una riga CSV per ciascun job;
- sei file `.log` con output dettagliati, utili per il debugging;
- file con le prime 10 righe dei risultati salvati per ogni job, per esempio `job1_output_first10.txt`.

Benefici

Questo approccio ha consentito di:

- automatizzare l'esecuzione completa del progetto;
- confrontare i tempi di esecuzione in modo equo tra tecnologie;
- mantenere modularità nel codice senza modificare ogni singolo `run.sh`;
- facilitare la riproducibilità dei risultati.

7 Struttura e contenuto della directory `output/`

L'esecuzione automatizzata dei job, tramite gli script `benchmarck100k.sh`, `benchmarck1M.sh` e `benchmarck3M.sh`, ha prodotto una struttura organizzata nella directory `output/`. Questa cartella contiene due tipologie principali di file: (i) i risultati quantitativi (tempi di esecuzione) e (ii) i risultati qualitativi (prime 10 righe dell'output per ciascun job e tecnologia).

File `tempo_totale_<X>.txt` Ogni file `tempo_totale_X.txt` (dove `X` corrisponde a `100k`, `1M`, o `3M`) contiene un riepilogo tabellare dei tempi di esecuzione per i sei job sviluppati (Job 1 e Job 2, ciascuno implementato in MapReduce, Hive e Spark SQL). Ogni riga rappresenta un job, e il tempo è espresso in secondi.

```
Script,Tempo(s)
Job1 MapReduce,45
Job1 Hive,33
Job1 SparkSQL,22
Job2 MapReduce,56
Job2 Hive,38
Job2 SparkSQL,27
```

Questi dati sono fondamentali per:

- confrontare le performance computazionali tra tecnologie;
- valutare la scalabilità rispetto alla dimensione del dataset;
- verificare la coerenza tra runtime e complessità del job.

File `jobX_output_first10.txt` **e** `preview.txt` Per ogni combinazione `job-tecnologia`, è stato generato un file contenente le **prime 10 righe dell'output prodotto** (sia esso aggregato, trasformato o descritto). Questi file sono:

- `job1_mapreduce.txt`, `job1_hive_output_first10.txt`, `job1_sparksql_preview.txt`
- `job2_MapReduce_output_first10.txt`, `job2_hive_output_first10.txt`, `job2_sparksql_previe`

Questi output sono stati salvati automaticamente all'interno degli script `run.sh` dei rispettivi job, usando comandi `head -n 10`. L'obiettivo è duplice:

1. fornire una **verifica immediata** sulla correttezza sintattica e semantica dell'elaborazione;
2. consentire una **comparazione qualitativa tra le tecnologie**, valutando differenze nella rappresentazione, nella precisione o nella formattazione dei dati risultanti.

Ad esempio, per Job 1 i file contengono righe del tipo:

```
make_name,model_name,num_cars,min_price,max_price,avg_price,years_present
ford,f-150,412,3200,57000,24500.6,[2011,2012,2013,2014]
...
```

Mentre per Job 2, l'output è del tipo:

```
city,year,fascia_prezzo,num_annunci,media_giorni,top_3_words
chicago,2018,medio,134,23.4,clean,low,miles
...
```

Note sulla consistenza dei risultati La struttura dei file di output è stata mantenuta coerente tra le tre tecnologie. Questo ha reso possibile sia una validazione puntuale dei risultati, sia l'integrazione nei successivi report quantitativi e grafici. Inoltre, la presenza dei log completi nella sottocartella `logs/` consente un controllo granulare dell'esecuzione, utile in fase di debugging.

8 Considerazioni finali

Il progetto ha dimostrato l'efficacia dell'utilizzo di tecnologie distribuite per l'analisi di dataset di grandi dimensioni, come il caso del dataset *US Used Cars*. Attraverso l'implementazione di due job analitici distinti (uno orientato alla statistica per marca e modello, l'altro alla segmentazione per città, anno e fascia di prezzo), è stato possibile confrontare in modo sistematico le prestazioni, la semplicità implementativa e la qualità dell'output ottenuto mediante tre strumenti fondamentali nell'ecosistema Big Data: **MapReduce**, **Hive** e **Spark SQL**.

La scelta di progettare un'infrastruttura di benchmarking flessibile e riutilizzabile ha reso possibile non solo l'esecuzione controllata dei job su dataset di dimensioni diverse (100k, 1M, 3M), ma anche la raccolta automatizzata di evidenze sperimentali sui tempi di elaborazione e sulla coerenza dei risultati prodotti. L'adozione di un sistema modulare, in cui ogni job è contenuto in uno script autonomo ma integrabile, ha facilitato lo sviluppo incrementale del progetto e la riproducibilità dei test.

Tra i tre strumenti valutati, **Spark SQL** si è rivelato il più performante, grazie alla gestione in-memory dei dati e all'espressività del linguaggio. **Hive** ha offerto una sintassi semplice e leggibile, ma ha mostrato limiti in termini di efficienza. **MapReduce**, seppur meno performante, ha garantito massima flessibilità e trasparenza sull'intero flusso di elaborazione.

In conclusione, il progetto ha evidenziato l'importanza di saper bilanciare *efficienza computazionale*, *scalabilità* e *manutenibilità* del codice nell'ambito di applicazioni reali su grandi volumi di dati. Le competenze acquisite spaziano dalla manipolazione dei dati grezzi alla progettazione di workflow distribuiti, fino alla valutazione comparativa dei risultati ottenuti.