Co-Purchase-analysis

Stefano Staffolani 0001137708

Settembre 2025

1 Introduzione

L'obiettivo del progetto è quello di realizzare una implementazione in Scala + Spark di un'analisi di co-acquisto di prodotti su un dataset di acquisti. L'analisi di co-acquisto consiste nel calcolare il numero di volte in cui due prodotti fanno entrambi parte di un medesimo ordine di acquisto.
In questo modo è possibile analizzare le affinità fra prodotti: per affinità fra due prodotti si intende che se un utente acquista uno dei due prodotti, con alta probabilità acquisterà anche l'altro. Questo tipo di informazione viene spesso usata nei cosiddetti sistemi di raccomandazione.
Nella Sezione 2, sono riportate le scelte implementative passo passo, nella Sezione 3 viene descritta la configurazione del cluster Dataproc utilizzato per l'esecuzione e infine nella Sezione 4 vengono riassunti i risultati ottenuti.
Il codice del progetto è consultabile al seguente link https://github.com/stefanostaffolani/co-purchase-analysis.git[1].

2 Scelte Implementative

Per lo sviluppo del progetto è stato utilizzato il pattern Map-Reduce su Spark. Per prima cosa viene processato il dataset CSV attraverso la funzione `SparkContex.textFile` e vengono formate le coppie della forma `<idOrdine,idProdotto>` nel seguente modo:

```
val input = spark.sparkContext
  .textFile(args(0))
  .map((line) => line.split(","))
  .map(w => (w(0).toInt, w(1).toInt))
```

È stato utilizzato un Hash partitioner per suddividere il dataset attraverso la funzione `partitionBy`, come segue:

```
val partitioned = input.partitionBy(
    new HashPartitioner(partitions)
)
```

Il numero di partizioni è deciso a runtime dalla formula:

$$2 * \text{numCores} * \text{numWorkers} \tag{1}$$

dove `numCores` è fisso a 4 per via delle macchine utilizzate e `numWorkers` è un parametro che viene passato automaticamente da `submit-job.sh`, e corrisponde alla variabile di ambiente `$NUM_WORKERS`. Utlizzare come numero di partizioni un multiplo del totale dei core a disposizione nel cluster è una best practice (come riportato dal seguente articolo Medium [2]).
Per prima cosa si raggruppano tutti i prodotti per `idOrdine` attraverso la funzione groupByKey ottenendo un `org.apache.spark.rdd.RDD[(Int, Iterable[Int])]`, che associa a ciascun ordine l'insieme dei prodotti in esso contenuti. Per ogni ordine si generano tutte le coppie di prodotti rimuovendo sia duplicati sia le combinazioni invertite come segue:

```
val prodPair = partitioned
.groupByKey()
.flatMap {
  case (_, products) => {
```

```
      val prodlist = products.toSeq
      for {
        x <- prodlist
        y <- prodlist
        if (x < y)
      } yield ((x, y), 1)
    }
  }
```

ottenendo un `org.apache.spark.rdd.RDD[((Int, Int), Int)]` su cui si potrà applicare la `reduceByKey` ottenendo per ogni coppia il numero totale di ordini in cui appaiono. Il risultato viene opportunamente formattato e poi riportato ad una partizione attraverso la funzione `coalesce` o `repartition`[3]. È necessario ripartizionare il risultato per salvare su un singolo file CSV, altrimenti Spark salverà un file per ogni partizione.

```
val result = prodPair.reduceByKey(_ + _)
val prettified = result.map { case (prods, numbers) =>
  s"${prods._1},${prods._2},$numbers"
  }
val toOne = prettified.repartition(1)
```

3 Configurazione Cluster Dataproc

Per eseguire i job Spark sono state utilizzate le macchine n1-standard-4[4] di Google Cloud Platform, ognuna di queste possiede 4 vCPU e un memoria di 15 GB. I run sono stati eseguiti su cluster da 1 a 4 worker. Il bucket e il cluster sono stati creati nella stessa regione, ovvero `europe-west-1`[5]. La disk size scelta è di 240GB sia per il master che per il worker, e come immagine è stata impostata `2.0-debian10`. Per la configurazione di Spark è stata impostata la memoria del driver (`driver.memory`) a `4g`, mentre la memoria di ogni executor (`executor.memory`) è stata impostata a `6g`[6].

4 Risultati

Nelle tabelle che seguono sono riportati valori di speedup e di Strong Scaling Efficiency calcolati utilizzando le seguenti formule:

$$\text{Speedup} = \frac{T_1}{T_n} \tag{2}$$

$$\text{Strong Scaling Efficiency} = \frac{T_1}{T_n * n} = \frac{\text{Speedup}}{n} \tag{3}$$

dove T_i con $i \in \{1..4\}$ rappresenta il tempo di esecuzione con i nodi. Nella Table 1 sono riportati i valori di scalabilità utilizzando la funzione `coalesce` per ripartizionare il `RDD`. Nella Table 2 sono riportati i valori utilizzando la funzione `repartition`. Inoltre nel Figure 1 sono mostrate a grafico i valori di Speedup e Strong Scaling Efficiency dell'esecuzione con repartition.

workers	runtime	speedup	Strong Scaling Efficiency
1	546 s	1	1
2	364 s	1.5	0.75
3	357 s	1.53	0.59
4	307 s	1.78	0.44

Table 1: Scalabilità con coalesce

workers	runtime	speedup	Strong Scaling Efficiency
1	483.6 s	1	1
2	292.7 s	1.65	0.827
3	240.9 s	2.01	0.669
4	189.9 s	2.547	0.637

Table 2: Scalabilità con repartition

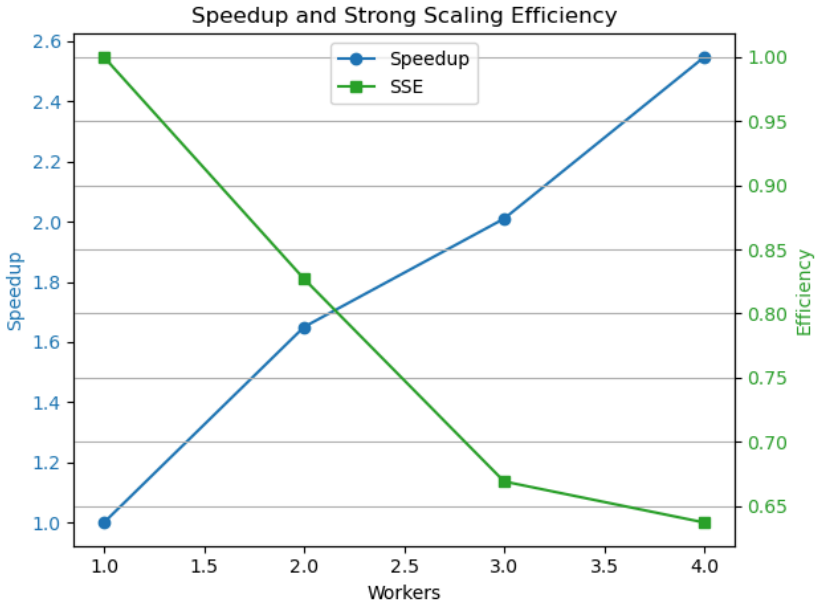

Figure 1: Scalabilità e speedup

5 Conclusione

Dai risultati ottenuti si può vedere che le migliori prestazioni si ottengono con l'utilizzo della funzione `repartition` invece di `coalesce`. Inoltre, guardando i valori di speedup in tabella Table 2 si può affermare che l'applicazione ha una buona scalabilità, infatti la Strong Scaling Efficiency scende ma rimane superiore al 60%.

Bibliography

[1] S. Staffolani, "Co-Purchase-Analysis: Project for the course of Scalable and Cloud Programming (81942) at UNIBO, academic year 2024/2025." [Online]. Available: https://github.com/stefanostaffolani/co-purchase-analysis.git

[2] R. Sounder, "Spark Default Parallelism vs Spark SQL Shuffle Parallelism Explained for Data Engineers." [Online]. Available: https://medium.com/@sounder.rahul/spark-default-parallelism-vs-spark-sql-shuffle-parallelism-explained-for-data-engineers-966795edf204

[3] Stack Overflow users, "Spark – repartition() vs coalesce()." [Online]. Available: https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce

[4] G. Cloud, "Compute Engine VM instance pricing." [Online]. Available: https://cloud.google.com/compute/vm-instance-pricing?hl=en

[5] G. Cloud, "Regions and Zones." [Online]. Available: https://cloud.google.com/compute/docs/regions-zones

[6] SparkByExamples, “Difference Between Spark Driver vs Executor.” [Online]. Available: https://sparkbyexamples.com/spark/difference-between-spark-driver-vs-executor/