

# Scalable and Cloud Programming

## Co-purchase Analysis

Luca Orlandello, ID 0001136759

13 maggio 2025

## 1 Descrizione del problema e lavoro svolto

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. Tutto il codice del progetto è disponibile al seguente repository GitHub <https://github.com/11111luca/copurchase-scalable>.

In sezione 2 è descritto l'implementazione del codice dell'analisi di co-acquisto del progetto mentre nelle sezioni 3 e 4 sono presentati e discussi i risultati dell'analisi di scalabilità.

## 2 Approccio utilizzato

Sono state implementate due versioni dello stesso codice: `CoPurchaseAnalysisNoPartitioning` e `CoPurchaseAnalysis`. La prima versione non utilizza nessun partitioner mentre la seconda versione è esattamente lo stesso codice della prima ma con l'aggiunta di partitioner. Di seguito sono spiegate più in dettaglio le due implementazioni.

In `CoPurchaseAnalysisNoPartitioning` come prima cosa viene fatto parsing del dataset in formato csv in input ottenendo un RDD di coppie (*IdOrdine*, *IdProdotto*), chiamiamo `orderIdProductId` questo RDD. Successivamente viene eseguito questo snippet di codice Scala-Spark.

```
val prod1IdProd2IdCounter = orderIdProductId.groupByKey()  
  .flatMap(p => for (x <- p._2; y <- p._2; if x < y) yield ((x, y), 1))
```

La `groupByKey` trasforma `orderIdProductId` in un nuovo RDD di coppie (*IdOrdine*, *l*) dove *l* è una collection iterabile di tutti e soli gli identificati di prodotti che appartengono allo stesso ordine. La successiva `flatMap` produce un RDD di coppie (*IdProdotto1*, *IdProdotto2*, 1) dove la prima componente della coppia è a sua volta una coppia di identificativi di prodotti che appartengono allo stesso ordine, mentre la seconda componente della coppia è un contatore inizializzato a 1. La condizione `x < y` nella for expression evita la creazione di coppie (*IdProdotto1*, *IdProdotto2*, 1) con *IdProdotto1* uguale a *IdProdotto2* e poiché l'ordine degli identificativi dei due prodotti non è rilevante, evita anche la creazione di entrambe le coppie (*IdProdotto1*, *IdProdotto2*, 1) e (*IdProdotto2*, *IdProdotto1*, 1). Infine è stata applicata una `reduceByKey(_ + _)` ottenendo così un RDD di coppie (*IdProdotto1*, *IdProdotto2*, *Freq*) dove *Freq* è la frequenza di quante volte *IdProdotto1* e *IdProdotto2* sono stati acquistati nello stesso ordine di acquisto.

`CoPurchaseAnalysis` è esattamente uguale a `CoPurchaseAnalysisNoPartitioning` tranne per fatto che prima della `groupByKey` viene applicato al RDD un partizionamento tramite un `HashPartitioner`. Il precedente snippet di codice viene modificato come segue.

```
val prod1IdProd2IdCounter = orderIdProductId
  .partitionBy(new HashPartitioner(partitionsNumber))
  .groupByKey()
  .flatMap(p => for (x <- p._2; y <- p._2; if x < y) yield ((x, y), 1))
```

Il numero di partizioni `partitionsNumber` è impostato a 2x il numero totale di core nel cluster. Impostare il numero di partizioni come il numero totale di core nel cluster per una costante moltiplicativa è suggerito nel articolo al seguente link<sup>1</sup>. Nell'articolo è suggerito di impostare il numero di partizioni a 3x in numero totale di core nel cluster ma provando per tentativi, in questo specifico caso, moltiplicare per 2 ha prodotto tempi di esecuzione leggermente più bassi e per questo è stato scelto. È stato utilizzato un `HashPartitioner` per fare in modo che coppie (*IdOrdine*, *IdProdotto*) con lo stesso *IdOrdine* siano inserite nella stessa partizione e quindi memorizzate sullo stesso nodo worker, in modo da ridurre lo shuffling della successiva `groupByKey`.

### 3 Analisi di scalabilità e risultati sperimentali

Per entrambe le implementazioni `CoPurchaseAnalysisNoPartitioning` e `CoPurchaseAnalysis` descritte in sezione 2, è stata valutata la scalabilità tramite lo stesso esperimento. Entrambe le implementazioni sono state eseguite 4 volte, rispettivamente con un cluster Spark di 1, 2, 3, 4 nodi worker, misurando il tempo di esecuzione in secondi e calcolando *speed up* e *strong scaling efficiency*, i risultati sono riportati nelle tabelle 1 e 2.

workers	tempo di esecuzione	speed up	strong scaling efficiency
1	14m 59s (899s)	-	-
2	7m 44s (464s)	1.94	0.97
3	7m 46s (466s)	1.93	0.64
4	7m 23s (443s)	2.03	0.51

Tabella 1: Analisi di scalabilità di `CoPurchaseAnalysisNoPartitioning`

workers	tempo di esecuzione	speed up	strong scaling efficiency
1	11m 31s (691s)	-	-
2	7m 11s (431s)	1.60	0.80
3	5m 44s (344s)	2.00	0.67
4	4m 47s (287s)	2.41	0.60

Tabella 2: Analisi di scalabilità di `CoPurchaseAnalysis`

Per permettere la riproducibilità dei risultati tutte le macchine di ogni cluster su cui è stata effettuata una misurazione del tempo di esecuzione sono state scelte della tipologia **n1-standard-4** e con installata l'immagine **2.2.53-debian12**. Inoltre ogni cluster è stato creato nella regione **europa-west2** e zona **europa-west2-c**. Le macchine di tipologia **n1-standard-4** sono della famiglia N serie 1 hanno 16G di memoria e 4 vCPU, maggiori dettagli si possono trovare nella documentazione di Google Cloud<sup>2</sup>. L'immagine **2.2.53-debian12** ha preinstallato Apache Spark 3.5.3 e Scala 2.12.18, maggiori dettagli si possono trovare nella documentazione di Google Cloud<sup>3</sup>. Inoltre l'esatta procedura su come ripetere le misurazioni è documentata nel file `README.md` nel repository GitHub del progetto.

<sup>1</sup><https://engineering.salesforce.com/how-to-optimize-your-apache-spark-application-with-partitions-257f2c1bb414/>

<sup>2</sup><https://cloud.google.com/compute/docs/machine-resource>

<sup>3</sup><https://cloud.google.com/dataproc/docs/concepts/versioning/dataproc-release-2.2>

## 4 Discussione e conclusioni

Dalla tabella 1 si osserva che `CoPurchaseAnalysisNoPartitioning` non ha nessun beneficio nell'utilizzare più di due worker, poiché il tempo di esecuzione con 2, 3, 4 workers e anche lo speed up rimane circa lo stesso. Quindi `CoPurchaseAnalysisNoPartitioning` è una implementazione che non scala. Invece `CoPurchaseAnalysis` in tabella 1, trae beneficio dall'aumentare il numero di worker del cluster perché lo speed up aumenta e il tempo di esecuzione diminuisce drasticamente passando da 1 a 4 workers. Ciò che permette a `CoPurchaseAnalysis` di scalare è proprio l'utilizzo del `HashPartitioner` perché riduce lo shuffling causato dalla `groupByKey` come spiegato in sezione 2.