Co-purchase Analysis

Report del progetto del corso *Scalable e Cloud programming*

Luca Polese Matricola: 0001136728

Dipartimento di Ingegneria e Scienze Informatiche, Università di Bologna

1. Introduzione

Nell'era dei big data e dell'e-commerce, l'analisi delle abitudini di consumo degli utenti riveste un ruolo fondamentale per migliorare l'esperienza d'acquisto e ottimizzare le strategie di vendita. In questo contesto, il presente progetto si propone di implementare un'analisi di co-acquisto di prodotti sfruttando la potenza del calcolo distribuito per elaborare grandi quantità di dati in modo efficiente.

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.

Si vuole fornire pertanto un'analisi dettagliata della correlazione tra gli acquisti.

1.1. Descrizione del problema

Per lo sviluppo e la validazione dell'analisi, si utilizza un dataset derivato da una versione semplificata del InstaCart Online Grocery Basket Analysis Dataset, disponibile su Kaggle. Questo dataset raccoglie informazioni sugli ordini effettuati tramite un'app di delivery della spesa al supermercato. Ogni ordine effettuato da un utente viene registrato con un identificativo univoco e associato ai prodotti acquistati, permettendo di analizzare i comportamenti d'acquisto e le correlazioni tra gli articoli.

Il dataset fornito è in formato `CSV` e contiene coppie di valori `(o, p)`, dove `o` rappresenta l'identificativo univoco dell'ordine e `p` l'identificativo del prodotto acquistato.

Se un ordine include più prodotti, vengono registrate più righe nel dataset, ciascuna corrispondente a un prodotto acquistato. L'output finale dell'analisi sarà un file `CSV` contenente coppie di prodotti con il numero di volte in cui sono stati acquistati insieme, nel formato: `(prodotto1, prodotto2, frequenza)`.

1.2. Obiettivi

Attraverso l'implementazione di questo progetto, si intende dimostrare la possibilità di effettuare un elaborazione scalabile dei dati attraverso l'adozione di Apache Spark.

L'implementazione del progetto avviene utilizzando il paradigma MapReduce e la programmazione funzionale in Scala, sfruttando Apache Spark per l'elaborazione parallela del dataset. L'intero processo viene eseguito su Google Cloud Dataproc, un servizio cloud che consente di gestire cluster Spark in modo scalabile e ottimizzato per il calcolo distribuito.

2. Implementazione

L'implementazione dell'analisi di co-acquisto è stata sviluppata utilizzando Scala e Apache Spark, sfruttando le capacità di calcolo distribuito per elaborare grandi volumi di dati in modo efficiente. Il progetto è stato configurato con SBT (Simple Build Tool), utilizzando la versione 2.12.18 di Scala e Apache Spark 3.5.0. Queste scelte tecnologiche permettono di beneficiare delle più recenti ottimizzazioni di Spark per la gestione dei dati distribuiti, garantendo compatibilità e prestazioni elevate.

Il file di configurazione SBT definisce le dipendenze essenziali per il progetto, includendo `spark-core` e `spark-sql`, necessari per la gestione e l'elaborazione dei dati. `spark-core` fornisce le API di base per l'elaborazione distribuita, mentre `spark-sql` consente di sfruttare un'interfaccia simile a SQL per manipolare i dati strutturati con DataFrame e Dataset. Questa combinazione di strumenti consente di implementare un'analisi scalabile ed efficiente dei dati di co-acquisto.

L'implementazione è stata progettata per eseguire il processo di analisi in più fasi: caricamento e preprocessing del dataset, aggregazione delle coppie di prodotti co-acquistati e scrittura dei risultati in un file CSV finale. Grazie all'uso del paradigma MapReduce e delle trasformazioni funzionali di Spark, il codice è strutturato in modo modulare, migliorando la leggibilità e la manutenibilità. Inoltre, l'esecuzione del progetto su Google Cloud

Dataproc assicura una gestione ottimale delle risorse computazionali, adattandosi dinamicamente al volume di dati da analizzare.

2.1. Inizializzazione dell'Ambiente

L'applicazione avvia una Spark Session con configurazioni ottimizzate per il calcolo distribuito:

```
val spark = SparkSession.builder()
  .appName("Co-Purchase Analysis")
  .config("spark.executor.cores", "3")
  .config("spark.speculation", "true")
  .config("spark.serializer", "org.apache.
      spark.serializer.KryoSerializer")
  .getOrCreate()
```

- `spark.executor.cores`: Imposta 3 core per ogni esecutore, garantendo una distribuzione equa del carico e riservando sempre un core a Yarn. Questa configurazione è stata scelta a seguito di una serie di test che hanno evidenziato un significativo miglioramento delle prestazioni, nonostante la limitazione dei core disponibili per l'esecuzione.
- `spark.speculation`: Abilita la gestione dei task più lenti per evitare rallentamenti globali.
- `spark.serializer`: Utilizza KryoSerializer, che riduce l'overhead della serializzazione.

L'input e l'output sono letti e scritti su Google Cloud Storage, specificando il bucket come parametro.

2.2. Lettura del Dataset

Il dataset viene letto dal bucket GCP e partizionato per migliorare la parallelizzazione:

```
val rawData: RDD[String] = spark.
    sparkContext.textFile(inputPath,
    numPartitions)
```

- `textFile`: Carica il file CSV contenente gli ordini e i prodotti acquistati.
- `numPartitions`: Definisce il numero di partizioni da ottenere a partire dal dataset per garantire un'elaborazione distribuita efficace.

Per garantire un'elaborazione distribuita efficiente, il numero di partizioni del dataset viene determinato dinamicamente sulla base delle risorse disponibili nel cluster Spark. Secondo la documentazione ufficiale di Spark (Sezione Tuning Spark), il livello di parallelismo deve essere sufficientemente elevato affinché tutte le risorse del cluster siano utilizzate in modo ottimale. In particolare, Spark consente di controllare il numero di partizioni nei file letti tramite `textFile`, specificando un parametro opzionale che influisce sul parallelismo.

Il numero di partizioni è calcolato come segue:

```
val cores = spark.conf.get("spark.executor
    .cores").toInt
val instances = spark.conf.get("spark.
    executor.instances").toInt
val numPartitions = cores * instances * 3
```

Questa formula si basa sulla best practice suggerita da Spark, secondo cui è consigliabile assegnare tra 2 e 3 task per ogni core del cluster. Come indicato nella documentazione, il livello di parallelismo nelle operazioni distribuite, come `groupByKey` e `reduceByKey`, viene determinato in base al numero di partizioni dell'`RDD` principale. Qui moltiplichiamo il numero totale di core disponibili `(cores * instances)` per 3 per garantire un adeguato livello di parallelismo.

2.3. Parsing e Preparazione dei Dati

Dopo la lettura, i dati vengono trasformati in un formato strutturato:

```
val orderProductPairs: RDD[(Int, Int)] =
    rawData
  .map(line => {
    val cols = line.split(",")
    (cols(0).toInt, cols(1).toInt)
  })
  .partitionBy(new HashPartitioner(
      numPartitions))
  .cache()
```

- `map`: Trasforma ogni riga del CSV in una coppia `(order_id, product_id)`.
- `partitionBy(new HashPartitioner(numPartitions))`: Distribuisce i dati uniformemente tra i nodi del cluster.
- `cache()`: Memorizza i dati in memoria per evitare ricalcoli inutili.

2.4. Creazione delle Coppie di Co-Acquisto

Si raggruppano i prodotti in base all'ordine di appartenenza:

```
val orderToProducts: RDD[(Int, Iterable[
    Int])] = orderProductPairs.groupByKey
    ()
```

- `groupByKey()`: Aggrega tutti i prodotti acquistati in ogni ordine.

Successivamente, si generano coppie di prodotti co-acquistati:

```
val coPurchasePairs: RDD[((Int, Int), Int)] = orderToProducts.flatMap { case (_, products) =>
  val productList = products.toList
  for {
    i <- productList
    j <- productList if i < j
  } yield ((i, j), 1)
}
```

- `flatMap`: Per ogni ordine, genera tutte le possibili coppie di prodotti.
- `if i < j`: Evita duplicati generando solo coppie `(p1, p2)` con `p1 < p2`.
- Ogni coppia riceve il valore `1`, indicante un'istanza di co-acquisto.

2.5. Aggregazione dei Risultati

Per ottenere il numero totale di co-acquisti, si utilizza `reduceByKey`:

```
val coPurchaseCounts: RDD[((Int, Int), Int)] = coPurchasePairs.reduceByKey(_ + _, numPartitions)
```

- `reduceByKey(_ + _)`: Somma i valori associati a ciascuna coppia di prodotti.
- `numPartitions`: Mantiene il calcolo distribuito efficiente.

2.6. Scrittura dei Risultati

I risultati vengono salvati su Google Cloud Storage:

```
coPurchaseCounts.map{
  case ((p1, p2), count) => s"$p1,$p2,$count"
}.repartition(1).saveAsTextFile(outputPath)
```

- `map`: Converte ogni coppia `(p1, p2, count)` in una stringa.
- `repartition(1)`: Garantisce che i dati vengano salvati in un unico file.

3. Test e Risultati ottenuti

Per valutare le prestazioni dell'implementazione, sono stati eseguiti molteplici test su Google Cloud Platform (GCP) utilizzando Dataproc. Le esecuzioni sono state effettuate su cluster con un numero variabile di nodi (1, 2, 3 e 4) e con diverse configurazioni per ottimizzare le prestazioni. L'obiettivo di questi test era determinare l'impatto delle configurazioni hardware e delle ottimizzazioni software sul tempo di esecuzione e sull'efficienza computazionale.

3.1. Configurazione della Memoria in Spark

La gestione della memoria in Spark è stata ottimizzata tramite configurazioni dinamiche basate sullo **scenario di esecuzione** (single-node vs multi-node). Nei file di configurazione (`cgp_cluster_scaling.sh`), vengono modificati i parametri `spark.driver.memory` e `spark.executor.memory` prima dell'esecuzione del job.

3.1.1. Allocazione della memoria

- Single-node mode
  - `spark.driver.memory = 6g`
  - `spark.executor.memory = 4g`
- Multi-node mode
  - `spark.driver.memory = 4g`
  - `spark.executor.memory = 4g`

Questa configurazione viene aggiornata direttamente nel codice Scala (`CoPurchaseAnalysis.scala`) tramite il seguente comando:

```
sed -i "s/\.config(\"spark\.driver\.memory\",␣\"[0-9]\+g\")/\.config(\"spark\.driver\.memory\",␣\"6g\")/" src/main/scala/CoPurchaseAnalysis.scala
sed -i "s/\.config(\"spark\.executor\.memory\",␣\"[0-9]\+g\")/\.config(\"spark\.executor\.memory\",␣\"4g\")/" src/main/scala/CoPurchaseAnalysis.scala
```

Ciò garantisce che il job venga eseguito con le impostazioni di memoria corrette in base alla tipologia di esecuzione.

3.2. Implicazioni sulle Prestazioni

1. **Single-node execution**: viene assegnata più memoria al driver per gestire meglio il carico computazionale senza frammentare gli RDD.
2. **Multi-node execution**: la memoria è bilanciata tra il driver e gli executor per garantire una distribuzione efficiente del carico tra i nodi del cluster.

Inoltre, i parametri di memoria vengono passati esplicitamente al comando di esecuzione di Spark su Dataproc:

```
gcloud dataproc jobs submit spark \
    --cluster=$CLUSTER_NAME \
    --region=$REGION \
    --class=CoPurchaseAnalysis \
    --jars="gs://$GCP_BUCKET/$jar_file" \
```

```
    --properties="spark.executor.instances
        =$executor_instances,spark.driver.
        memory=$driver_memory,spark.
        executor.memory=$executor_memory"
        \
    -- $GCP_BUCKET
```

Questo approccio assicura flessibilità nella configurazione e consente di adattare le risorse in base alle esigenze del carico di lavoro.

3.3. Scelta delle Macchine: n1-standard vs. n2-standard

Sono state utilizzate istanze della serie **N2** al posto di istanze della serie **N1** per i seguenti motivi:

- **Prestazioni**: Le istanze **N2** utilizzano processori Intel Cascade Lake, offrendo prestazioni superiori rispetto alle **N1** basate su Skylake
- **Costo**: Per questo caso d'uso, le istanze `n2-standard-2` e `n2-standard-4` offrono un miglior rapporto prezzo/prestazioni

Maggiori dettagli sulle istanze utilizzate sono disponibili nella documentazione ufficiale di GCP: N1, N2.

3.4. Test implementativi effettuati

Durante lo sviluppo del progetto, sono stati eseguiti test di performance per confrontare le prestazioni del sistema con e senza l'adozione di ottimizzazioni specifiche di Apache Spark. La versione iniziale del codice non utilizzava tecniche avanzate per la gestione dei dati, risultando in tempi di esecuzione più lunghi, in particolare durante le operazioni di trasformazione sul dataset. Con il miglioramento della versione finale, sono stati implementati diversi meccanismi di ottimizzazione che hanno avuto un impatto significativo, come è possibile notare alla sezione 3.6

Tutti i test sono stati eseguiti utilizzando lo stesso dataset, il che ha permesso di confrontare l'efficacia delle ottimizzazioni implementate, garantendo che i tempi di esecuzione potessero essere attribuiti esclusivamente alle modifiche nel codice e non a variazioni nei dati.

Le ottimizzazioni principali implementate nel sistema sono le seguenti:

- **Speculative Execution**: `.config("spark.speculation", "true")`
  Questo meccanismo permette di ridurre l'impatto degli stragglers, ovvero task che impiegano più tempo rispetto agli altri
- **Kryo Serializer**: `.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
  KryoSerializer migliora l'efficienza della serializzazione rispetto al serializer di default di Java
- **Caching**: `.cache()`
  Il caching viene utilizzato per ridurre il ricalcolo dei dati condivisi tra diverse operazioni di trasformazione.

Per ulteriori dettagli sulle ottimizzazioni, si faccia riferimento alla documentazione ufficiale di Spark [**spark˙tuning**].

3.5. Configurazione dei sistemi adottati

È possibile vedere il dettaglio delle configurazioni adottate nella Tabella 1

3.6. Risultati delle Esecuzioni

È possibile vedere il dettaglio delle esecuzioni adottate nella Figura 1(a) e nella Figura 1(b)

3.7. Analisi dei Risultati

L'analisi comparativa tra le due serie di test evidenzia che:

- L'abilitazione della **Speculative Execution** ha ridotto i tempi di esecuzione nei job con stragglers, migliorando la distribuzione del carico sui nodi
- L'uso di **KryoSerializer** ha permesso una gestione più efficiente della memoria, evitando overhead di serializzazione/deserializzazione con il formato Java predefinito
- Il **caching** ha mostrato miglioramenti nei job ripetitivi, riducendo il numero di operazioni di rilettura dei dati.

Nel complesso, l'adozione di queste ottimizzazioni ha portato a una riduzione del tempo di esecuzione a eccezione del job single node, che passano da 10 min 10 sec (senza ottimizzazioni) a 10 min 25 sec con caching e speculative execution attivati. Per job su multi node, c'è un guadagno in prestazioni, sebbene non sia particolarmente evidente.

4. Scaling Efficiency

Per analizzare l'efficienza dello scaling del sistema, è stata calcolata la **strong scaling efficiency**. La strong scaling efficiency misura come il tempo di esecuzione cambia al variare del numero di nodi nel cluster, mantenendo invariata la dimensione del

Nodi	CPU	Memoria	Regione	Disco
1	n2-standard-4	6 GB	europe-west1	240 GB
2	n2-standard-2 (master) n2-standard-4 (executor)	4 GB (driver) 4 GB (executor)	europe-west1	100 GB 100 GB
3	n2-standard-2 (master) n2-standard-4 (executor)	4 GB (driver) 4 GB (executor)	europe-west1	100 GB 100 GB
4	n2-standard-2 (master) n2-standard-4 (executor)	4 GB (driver) 4 GB (executor)	europe-west1	100 GB 100 GB

Tabella 1: Configurazioni testate

d38b00244e614358b6a68d4c61315eee	Completato	europe-west1	Spark	cluster-scp	25 mar 2025, 18:37:55	3 min 13 sec	None
1def702ddf48477db1a4e1d53e3cf433	Completato	europe-west1	Spark	cluster-scp	25 mar 2025, 18:32:57	3 min 36 sec	None
a190bcd820eb48e6848850ba3b858ba1	Completato	europe-west1	Spark	cluster-scp	25 mar 2025, 18:26:24	5 min 9 sec	None
fb4d12ee69b14b87b67e3480eaf32443	Completato	europe-west1	Spark	cluster-scp	25 mar 2025, 18:11:22	10 min 10 sec	None

(a) Tempi ottenuti dell'esecuzione di codice senza ottimizzazioni

9db7a8fdb3864190816e900e61c491a6	Completato	europe-west1	Spark	cluster-scp	26 mar 2025, 10:58:29	3 min 10 sec	None
6ad220a9c8c64ef087b67ccae8f97ddb	Completato	europe-west1	Spark	cluster-scp	26 mar 2025, 10:53:07	3 min 32 sec	None
f97b577ebc364c0d875a186895d6e2ab	Completato	europe-west1	Spark	cluster-scp	26 mar 2025, 10:46:23	4 min 58 sec	None
402f2c9ffa6c4f25a8e8de66f30e5893	Completato	europe-west1	Spark	cluster-scp	26 mar 2025, 10:30:26	10 min 25 sec	None

(b) Tempi ottenuti dell'esecuzione di codice con ottimizzazioni

problema. La formula utilizzata per il calcolo della strong scaling efficiency è la seguente:

$$E(n) = \frac{T(1)}{n \times T(n)}$$

Dove:

- $T(1)$ è il tempo di esecuzione con un singolo nodo
- $T(n)$ è il tempo di esecuzione con n nodi
- n è il numero di nodi nel cluster.

n	$T(n)$	$E(n)$
1	610s	
2	309s	0.987
3	216s	0.941
4	193s	0.790

Tabella 2: Risultati per la versione non ottimizzata

n	$T(n)$	$E(n)$
1	625	
2	298	1.048
3	212	0.983
4	190	0.822

Tabella 3: Risultati per la versione ottimizzata

5. Analisi dei risultati

L'analisi dei risultati relativi alla **strong scaling efficiency** evidenzia come il sistema risponde all'aumento dei nodi nel cluster, con e senza le ottimizzazioni implementate. All'aumentare del numero di nodi, il comportamento del sistema segue la legge di Amdahl. La causa principale della diminuzione di performance è data dai costi di sincronizzazione e comunicazione tra i nodi che aumentano al crescere del numero di nodi.

Nella versione **non ottimizzata**, i risultati mostrano che con due e tre nodi l'efficiency è alta (0.9), ma scende drasticamente a 0.79 con l'aggiunta del quarto nodo. L'introduzione di più nodi riduce il tempo di esecuzione, ma l'aumento dei costi di sincronizzazione non è compensato dalla parallellizzazione, portando a un miglioramento non lineare delle prestazioni.

Nel caso della **versione ottimizzata**, si osserva un miglioramento significativo nei tempi di esecuzione. L'efficienza per due nodi supera il valore di 1, suggerendo che le ottimizzazioni, hanno migliorato le prestazioni in modo tale da ridurre i tempi di esecuzione rispetto alla versione non ottimizzata. Tuttavia, anche in questa versione, l'efficienza decresce man mano che vengono aggiunti ulteriori nodi, con efficienza 0.82 con il quarto nodo.

Il codice sorgente è disponibile nella repository SCP-Co-Purchase-Analysis