Progetto Scalable and Cloud Programming:
Co-purchase Analysis

Andrea Morabito
0000983680

a.a. 2024/25

1. Introduzione

L'obbiettivo di questo progetto è quello di sfruttare dei benefici della potenza del calcolo distribuito su cluster cloud, ed in questo particolare caso su GCP Dataproc.

Tale obbiettivo viene raggiunto attraverso un'implementazione distribuita per l'analisi di co-acquisto di prodotti, che rappresenta un campo in forte crescita e fondamentale in moltissimi settori come ad esempio: sistemi di raccomandazione nei negozi online; ottimizzazione della disposizione dei prodotti nei supermercati; analisi delle tendenze di consumo e segmentazione della clientela; strategie di marketing.

Questa implementazione si basa sull'utilizzo delle potenzialità di Scala e Spark su un dataset di ordini costituita da una versione semplificata del "*InstaCart Online Grocery Basket Analysis Dataset*", che contiene informazioni su milioni di ordini effettuati dagli utenti attraverso un app di delivery, al cui interno sono presenti dei record nel formato *(ordine, prodotto)*, che indica che un determinato prodotto è stato acquistato in un certo ordine, ed ogni ordine può includere più prodotti. L'analisi si basa sull'identificazione delle coppie di prodotti acquistati insieme nello stesso ordine e sul calcolo della loro frequenza di co-occorrenza, ovvero calcolare quante volte due prodotti compaiono insieme nello stesso ordine
In questo modo siamo in grado di identificare le affinità tra i prodotti acquistati e quindi determinare quante volte due prodotti compaiono nello stesso ordine. Il risultato finale verrà mostrati in un file di output in formato '*.csv*' contenente le coppie di prodotti e il numero totale di ordini in cui appaiono insieme.

Le metriche utilizzate per valutare il miglior approccio includono confronti tra:
- Tempo di esecuzione.
- Utilizzo della CPU.
- Volume di dati scambiati nella fase di shuffle.
- Tempo di Garbage collector.

Tramite l'implementazione di un script, volutamente non ottimale, vedremo che i risultati ottenuti dimostrano che l'uso di tecniche di ottimizzazione in Spark per elaborare grandi volumi di dati in modo distribuito può portare a miglioramenti significativi delle prestazioni, rendendo il codice più efficiente e scalabile su cluster di dimensioni variabili. Tuttavia, bisogna fare attenzione in

determinati punti.

Nei capitoli successivi, descriveremo in dettaglio l'implementazione, analizzando le metriche di performance raccolte e discuteremo i vantaggi e gli svantaggi.

2. Approccio Implementativo

L'implementazione sfrutta l'architettura distribuita di Apache Spark e il paradigma MapReduce per processare il dataset in modo efficiente. Per lo sviluppo del codice è stato scelto un approccio Step-by-Step in modo da mantenere un approccio schematico e semplice. Di seguito sono descritti i passaggi principali:

1. **Inizializzazione e configurazione spark e GCP**
   Primo step consiste nella creazione di una sessione spark attraverso il costruttore '***.builder()***' configurando il master a '***local[*]***' per sfruttare tutte le CPU presenti nel sistema. Questa configurazione permette anche di eseguire test locali, sebbene il codice sia progettato per essere eseguito su cluster. Quindi si passa a settare i parametri per la connessione a GCP ed ai servizi associati come GCS visto che il dataset risiede all'interno del sistema di file distribuito Google Cloud Storage). La scelta di posizionare i file di input/output su cloud è legata alla necessità di gestire grandi volumi di dati in modo scalabile, sicuro e con alta disponibilità oltre a rendere il codice pronto per l'uso in ambienti cloud
2. ***Lettura del dataset***
   Il percorso del file CSV di input è passato come argomento e viene caricato in un RDD di coppie (orderId, productId). Sebbene i DataFrame sono generalmente preferibili per le ottimizzazioni automatiche, è stato scelto di lavorare con gli RDD perché i Resilient Distributed Datasets hanno una capacità di gestire grandi set di dati in modo distribuito e fault-tolerant, garantendo così maggiore controllo sull'elaborazione ed operazioni più veloci. La scelta di rendere i dati persistenti rappresenta una scelta progettuale importante per evitare costosi calcoli ripetuti nel caso di riduzioni successive perché consente di mantenere i dati in memoria durante l'elaborazione e quindi evitare riletture durante le fasi successive, scelta che richiede ovviamente una sufficiente quantità di RAM.
3. ***Generazione di coppie di prodotti e conteggio***
   In questa fase, l'RDD viene trasformato in una coppia (orderId, List(productId)) in cui ogni ordine contiene una lista di prodotti su cui viene applicata un'operazione **reduceByKey(_ ++ _)** per combinare i prodotti associati a ciascun ordine, creando una lista di prodotti distinti per ogni ordine. L'uso di *reduceByKey* consente di aggregare le co-occorrenze dei prodotti in modo distribuito ed efficiente. Infine viene applicata la **flatMap** che esplora tutte le possibili coppie distinti di prodotti per ogni ordine, evitando le coppie duplicate grazie alla condizione **if p1 < p2**. La scelta di *flatMap* assicura che tutte le combinazioni possibili siano esplorate in modo efficiente, riducendo al minimo la memoria necessaria.
4. ***Scrittura del risultato***
   Prima di andare a memorizzare i risultati ottenuti in un file csv, andiamo a definire prima uno schema per convertire il nostro RDD in DataFrame per poter sfruttare le funzioni di scrittura e quindi andiamo ad ordinare in modo decrescente i nostri risultati in base al conteggio delle co-occorrenze. Una possibile alternativa a tale approccio sarebbe stato quello di scrivere i risultati su un database o un sistema di archiviazione cloud, ma ciò avrebbe influenzato il numero di risorse da usare oltre ad far incrementare i costi

Prima di terminare, il codice raccoglie e visualizza alcune metriche di sistema per valutare le prestazioni del nostro programma. Tali metriche includo includono:

- **Conteggio dei record processati**, per verificare che tutti i dati siano stati processati correttamente e a identificare eventuali discrepanze o perdite di dati oltre a stimare la scalabilità dell'applicazione
- **Carico medio della CPU per minuto,** per identificare eventuali sovraccarichi del sistema, o colli di bottiglia, e capire se il cluster ha risorse sufficienti per il carico di lavoro. Questo valore che indica il numero medio di processi in esecuzione o in attesa di CPU negli ultimi 60 secondi e se tale valore è vicino al numero di core disponibili, il sistema è bilanciato, altrimenti il sistema risulta sovraccarico. Ad esempio su un sistema con 8 core CPU abbiamo:

CPU Load Avg	Stato del sistema
2.5	Carico leggero
7.9	Carico bilanciato, vicino alla saturazione
12.3	Sovraccarico e CPU insufficiente

- **Processori disponibili**, per avere un'indicazione della capacità di elaborazione del sistema e aiuta a ottimizzare l'allocazione delle risorse e quindi la capacità di parallelizzazione del job Spark
- **Shuffle size**, per determinare la quantità di dati spostati tra i nodi durante operazioni di shuffle e identificare inefficienze nella distribuzione dei dati e a ottimizzare le prestazioni riducendo il trasferimento di dati non necessario
- **Uptime**, che rappresenta il tempo di funzionamento continuativo del programma ed è utile per determinare l'affidabilità e la stabilità del sistema, evidenziando eventuali interruzioni o crash oltre che monitorare la stabilità e la durata dei job Spark, utile per il debugging e il confronto tra diverse esecuzioni
- **Garbage collector**, per ottimizzare la gestione della memoria e a ridurre i tempi di pausa che potrebbero influire sulle prestazioni
- **Tempo di esecuzione del programma**, per fornire una misura delle prestazioni complessive e aiuta a identificare opportunità di ottimizzazione

Come alternativa alle metriche implementate è possibile integrare strumenti di monitoraggio per tracciare l'utilizzo delle risorse e le prestazioni in tempo reale

3. Analisi delle Prestazioni e Conclusioni

3.1 Analisi

Per valutare l'efficienza e la scalabilità del sistema, è stata condotta un'analisi delle prestazioni su un dataset costituito da 32.434.489 record ed un cluster con diverse configurazioni che includono 4, 8 e 16 core. Di seguito sono riportati i risultati ottenuti relativo alle metriche raccolte che includono tempo di esecuzione, carico della CPU, dimensione dello shuffle e tempo di Garbage Collection.

Metrica	4 core	8 core	16 core
CPU load avg (1m)	2.17	2.11	2.43
Shuffle	1.96 GB	4.09 GB	8.36 GB
Uptime	22 minuti e 11 secondi	7 minuti e 56 secondi	7 minuti e 10 secondi
Garbage Collecor	22 minuti e 54 secondi	53 minuti e 44 secondi	2 ore 10 minuti e 6 secondi
Execution time	22 minuti e 2 secondi	7 minuti e 50 secondi	7 minuti e 3 secondi

Primo risultato importante riguarda il tempo di esecuzione. In questo caso è possibile osservare che l'aumento del numero di core porta ad una diminuzione del tempo di esecuzione. Passando da 4 a 8 core, il tempo si riduce di circa il 65%, mentre da 8 a 16 core la riduzione è meno marcata (circa il 5%). Questo dimostra che il sistema è in grado di sfruttare efficacemente le risorse aggiuntive per parallelizzare il carico di lavoro, ma oltre gli 8 core i guadagni in termini di tempo diventano marginali, suggerendo un possibile limite di scalabilità per questo specifico carico di lavoro.

Con lo stesso andamento del tempo di esecuzione c'è anche l'uptime che diminuisce in modo simile all'aumentare dei core. Ciò è positivo poiché dimostra che il sistema è in grado di completare le attività più rapidamente senza incorrere in crash o interruzioni, dimostrando così una maggiore stabilità e affidabilità.

Altro dato evidente è che con il dataset scelto, il carico della CPU rimane relativamente stabile tra le configurazioni, indicando che il sistema è in grado di bilanciare il carico di lavoro anche con un numero maggiore di core. Tuttavia è da notare che con 16 core, il carico della CPU aumenta leggermente, suggerendo che il sistema potrebbe essere sottoposto a un carico maggiore dovuto alla gestione di più thread o processi paralleli.

Il tempo di Garbage Collection aumenta drasticamente all'aumentare del numero di core. Questo è un risultato controintuitivo, poiché ci si aspetterebbe una gestione più efficiente della memoria con più risorse disponibili. Questo comportamento potrebbe essere dovuto a una maggiore frammentazione della memoria o a un overhead nella gestione della memoria distribuita tra più core e questo può ridurre l'efficienza complessiva del sistema.

Il volume di dati scambiati durante la fase di shuffle aumenta in modo lineare con il numero di core. Ciò è un comportamento atteso, poiché più core implicano una maggiore parallelizzazione e, di conseguenza, un maggior scambio di dati tra i nodi. In questo caso si deve fare attenzione perché lo shuffle potrebbe diventare un collo di bottiglia in configurazioni con un numero di core molto elevato, poiché richiede più risorse di rete e memoria.

3.2 Conclusioni

L'analisi delle prestazioni mostra come l'aumento del numero di core migliori significativamente l'efficienza del sistema, da cui emergono le seguenti conclusioni:

1. **Scalabilità**:
    - Il sistema dimostra un'ottima scalabilità, con un miglioramento significativo delle prestazioni all'aumentare del numero di core. Il tempo di esecuzione si riduce in modo quasi lineare, indicando che il carico di lavoro è ben distribuito e parallelizzato. Tuttavia i guadagni diventano marginali con il massimo numero di core testati e ciò rappresenta un limite di scalabilità per il dataset preso in analisi
2. **Efficienza delle risorse**:
    - L'utilizzo della CPU diminuisce all'aumentare delle risorse disponibili, suggerendo che il sistema è in grado di gestire carichi di lavoro più elevati senza sovraccaricare i processori. Inoltre la riduzione dello shuffle size e del tempo di Garbage Collector indica una gestione più efficiente dei dati e della memoria.

3. **Affidabilità**:
    - L’uptime aumenta e il numero di crash diminuisce all’aumentare del numero di core, dimostrando che il sistema diventa più stabile e affidabile con risorse aggiuntive. Anche se non si deve mettere da parte l’incremento del tempo Garbage Collector che potrebbe compromettere l'affidabilità a lungo termine.

In sintesi, l’uso di un cluster con **8 core** offre le migliori prestazioni in termini di tempo di esecuzione, efficienza delle risorse e affidabilità, per questo specifico carico di lavoro.

Interessante da discutere i risultati ottenuti con il cluster a 16 core che pur essendo più performante in termini di tempo di esecuzione, introduce inefficienze significative nella gestione della memoria e dei dati. Ci sono diversi fattori che possono spiegare il comportamento ottenuto, in particolare:

- Overhead di Parallelizzazione, associato alla gestione di più thread o processi paralleli. Ciò porta il sistema a spendere più tempo a coordinare i task tra i core (sincronizzazione, comunicazione tra thread, gestione delle dipendenze) rispetto al tempo effettivo di elaborazione
- Aumento di Shuffle e GC
    - Shuffle: il sistema tende a parallelizzare ulteriormente le operazioni e tale aumento può saturare la banda di rete o la memoria disponibile, diventando un collo di bottiglia.
    - Garbage Collector: Frammentazione della memoria (Con più thread in esecuzione, la memoria può diventare più frammentata, rendendo più difficile e lenta la raccolta dei dati inutilizzati), Overhead (dovendo gestire un numero maggiore di oggetti in memoria, aumenta il tempo necessario per liberare risorse), Pause (richiedere pause più lunghe per sincronizzare l'accesso alla memoria, riducendo l'efficienza complessiva)
- Legge di Amdahl, che afferma che il miglioramento delle prestazioni ottenibile parallelizzando un carico di lavoro è limitato dalla frazione sequenziale del programma e anche con un numero elevato di core, se una parte significativa del carico di lavoro non può essere parallelizzata, il tempo di esecuzione non può scendere al di sotto di un certo limite.

Quando si programma su cluster, è fondamentale progettare il codice pensando alla distribuzione del carico di lavoro, valutando il rapporto costo-beneficio in modo da evitare colli di bottiglia e garantire un bilanciamento ottimale delle risorse. Mentre il sistema è in grado di parallelizzare il carico di lavoro, ottimizzando le tempistiche, la gestione della memoria e dei dati diventa meno efficiente con più core, visto dall’ aumento significativo del tempo di Garbage Collection e della dimensione dello shuffle. Le risorse usate devono essere scelte con cura perché potrebbe portarci ad inefficienze con carichi di lavoro più piccoli, e configurazioni con meno core possono essere sufficienti e più economiche.