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, in questo caso di GCP Dataproc.
Tale obbiettivo viene raggiunto attraverso un'implementazione distribuita per l'analisi di co-acquisto di prodotti poiché l'analisi dei dati di acquisto è un campo fondamentale nell'ambito di: 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 viene fatta utilizzando le 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 indicano che un determinato prodotto è stato acquistato in un certo ordine, in cui 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, ovvero determinare quante volte due prodotti compaiono nello stesso ordine che verrano mostrati in un file di output in formato '.csv' contenente coppie di prodotti e il numero di ordini in cui appaiono insieme.

Per sottolineare i benefici degli strumenti utilizzati, è stata sviluppata una versione inefficiente utilizza operazioni meno scalabili (`groupByKey`, operazioni ridondanti e shuffling non necessario) per evidenziare l'impatto di scelte implementative non ottimali. Oltre questo viene creata una versione efficiente che utilizza trasformazioni efficienti (`reduceByKey`, `persist()`, `mapPartitions`) per ridurre al minimo il consumo di memoria e il traffico di rete, che vengono confrontate

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

- Tempo di garbage collection.

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.

Nei capitoli successivi, descriveremo in dettaglio entrambe le implementazioni, analizzeremo le metriche di performance raccolte e discuteremo i vantaggi e gli svantaggi di ciascun approccio.

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 semplice step by step con la creazione di un codice 'grezzo' in modo seguente:

1. Caricamento del dataset: Il file CSV viene caricato in un RDD e i record vengono convertiti in coppie (ordine, prodotto).
2. Aggregazione per ordine: I prodotti vengono raggruppati per ordine per identificare gli acquisti congiunti.
3. Generazione di coppie di prodotti: Per ogni ordine, si generano tutte le possibili coppie di prodotti univoche.
4. Conteggio delle coppie: Le coppie vengono conteggiate e sommate per ottenere la frequenza di co-acquisto.
5. Salvataggio del risultato: Il dataset finale viene salvato in un file CSV nel formato (prodotto1, prodotto2, frequenza).

Mentre le metriche inserite all'interno del codice includono:

3. Analisi delle Prestazioni e Scalabilità

Per valutare la scalabilità dell'implementazione, sono stati eseguiti test con cluster Spark di diverse dimensioni su DataProc di Google Cloud. I risultati dimostrano che:
Aumento del numero di nodi: L'incremento del numero di nodi del cluster riduce significativamente i tempi di elaborazione.
Distribuzione del carico: La gestione distribuita dei dati consente di processare dataset di grandi dimensioni in tempi ridotti rispetto a un'elaborazione sequenziale.
Efficienza del MapReduce: Il paradigma MapReduce ottimizza la riduzione delle coppie di prodotti, minimizzando il traffico di rete.

Tempi di esecuzione

Numero di nodi	Tempo di esecuzione
1	15 min
2	8 min
4	4 min

- La versione ottimizzata è **circa 3,5 volte più veloce**.
- Ha **meno dati scambiati nello shuffle**, riducendo il carico di rete.
- Il **tempo di Garbage Collection** è molto più basso, migliorando la fluidità dell'esecuzione.

4. Conclusioni

L’analisi di co-acquisto implementata in Scala e Spark si è dimostrata efficace e scalabile. Il sistema sfrutta il parallelismo per gestire grandi quantità di dati e ottimizza il calcolo delle coppie di prodotti acquistati insieme.
Per migliorare ulteriormente le prestazioni, si potrebbero esplorare ottimizzazioni come:
Uso di DataFrame API anziché RDD per una maggiore efficienza.
Partizionamento personalizzato per ridurre la latenza di aggregazione.
Caching dei dati per evitare riletture ridondanti.
Il codice sorgente è disponibile su GitHub.
L'ottimizzazione delle trasformazioni Spark è fondamentale per ottenere alte prestazioni. L'uso di `reduceByKey` al posto di `groupByKey`, il caching strategico e la riduzione dello shuffle migliorano significativamente il tempo di esecuzione e l'efficienza della memoria. In ambienti di Big Data, una cattiva implementazione può causare un rapido degrado delle prestazioni e problemi di scalabilità.