Progetto di Scalable and Cloud Programming

Matteo Galiazzo
Dipartimento di Informatica - Scienza e Ingegneria
Università di Bologna
matteo.galiazzo@studio.unibo.it

Anno Accademico 2025-26

Abstract

Il presente report descrive l'implementazione in Scala e Spark di un sistema per l'analisi di co-occorrenza di terremoti, eseguito su piattaforma Google Cloud DataProc. Viene illustrato l'approccio algoritmico Map-Reduce utilizzato per identificare le coppie di località con la maggiore frequenza sismica simultanea e viene proposta un'analisi preliminare di scalabilità e prestazioni al variare delle risorse del cluster e del partizionamento dei dati.

Contents

1 Lavoro svolto

L'obiettivo del progetto è stato realizzare un'applicazione distribuita per l'analisi di un dataset di eventi sismici, al fine di individuare la coppia di località caratterizzata dal massimo numero di co-occorrenze e di elencare le date in cui tali eventi si sono verificati.

Come richiesto dalle specifiche, il progetto è stato sviluppato utilizzando Scala e Apache Spark, adottando il paradigma Map-Reduce per l'elaborazione distribuita. Il codice sorgente completo e le istruzioni per l'esecuzione sono disponibili nella repository GitHub pubblica al seguente indirizzo:

`https://github.com/gekoxyz/unibo-scalable-and-cloud`

1.1 Pre-elaborazione dei dati

Basandosi sulle specifiche del progetto, al dataset sono state applicate le seguenti trasformazioni:

- Le coordinate geografiche (latitudine e longitudine) sono state approssimate alla prima cifra decimale con arrotondamento all'intero più vicino, raggruppando così eventi distinti in macro-aree.
- La co-occorrenza è stata definita sulla base di una finestra temporale giornaliera. Gli orari specifici sono stati ignorati, considerando solo la data.
- Sono stati rimossi gli eventi duplicati, per evitare che più eventi registrati nella stessa area e nello stesso giorno venissero conteggiati erroneamente come co-occorrenze multiple.

2 Approccio utilizzato

L'implementazione si basa su un'architettura a due passaggi ottimizzata per minimizzare l'overhead dello shuffle e l'uso della memoria.

2.1 Gestione dei Dati e Strutture

Sono state definite due *case class* principali per modellare i dati: `Position`, che gestisce la logica di ordinamento delle coordinate per garantire che la coppia (A, B) sia equivalente a (B, A), ed `EventKey`, che associa una posizione a una data specifica.

Il dataset viene caricato dal bucket Google Cloud Storage e immediatamente ripartizionato tramite il metodo `repartition` per bilanciare il carico tra i nodi worker fin dalle prime fasi.

2.2 Algoritmo di Analisi

L'analisi segue questi step logici:

1. **Raggruppamento:** Gli eventi unici vengono raggruppati per data. L'RDD risultante viene messo in cache (`.cache()`) poiché sarà necessario accedervi due volte.
2. **Conteggio delle Coppie (Passo 1):** Per ogni data, vengono generate tutte le combinazioni possibili di coppie di località. Utilizzando `reduceByKey` con un numero parametrico di partizioni, il sistema calcola la frequenza di ogni coppia. Questo approccio è efficiente perché trasmette attraverso la rete solo i contatori e le chiavi, non le liste complete delle date.
3. **Selezione del Massimo:** Tramite una semplice azione di `reduce`, viene identificata la coppia con il conteggio massimo.
4. **Recupero delle Date (Passo 2):** Una volta nota la "coppia vincente", il sistema scansiona nuovamente i dati raggruppati (già in cache) per filtrare e raccogliere solo le date in cui quella specifica coppia è apparsa. Questo evita di dover trascinare l'intera lista di date durante la fase costosa di shuffle del passo 1.

2.3 Automazione e Cloud

Per gestire l'esecuzione su Google Cloud DataProc, è stato sviluppato uno script Python (`run.py`) che automatizza l'intero ciclo di vita del benchmarking:

- Creazione del cluster con specifiche hardware definite (macchine `n2-standard-4`, dischi da 100GB).
- Sottomissione dei job Spark con diverse configurazioni di partizionamento.
- Download dei risultati e distruzione automatica del cluster per preservare i crediti.

3 Analisi di scalabilità e prestazioni

In questa sezione vengono analizzate le prestazioni dell'applicazione al variare delle risorse computazionali (scalabilità forte) e della configurazione interna di Spark (tuning delle partizioni).

3.1 Setup Sperimentale

Tutti i test sono stati eseguiti sulla **versione completa** del dataset. I cluster DataProc sono stati configurati come segue:

- **Master:** 1 nodo `n2-standard-4`
- **Worker:** Variabile (2, 3, 4 nodi `n2-standard-4`)
- **Storage:** 100GB per nodo

Per ogni configurazione del cluster, sono stati testati diversi livelli di parallelismo impostando il numero di partizioni a 4, 8, 12 e 16.

3.2 Risultati dei Benchmark

Nota: I test sono attualmente in corso. Di seguito vengono presentate le tabelle e i grafici predisposti per la raccolta dati.

La Tabella 1 mostra i tempi di esecuzione (in secondi) rilevati per le diverse combinazioni di Worker e Partizioni.

Partizioni	2 Worker (8 core)	3 Worker (12 core)	4 Worker (16 core)
4	[INSERIRE TEMPO]	[INSERIRE TEMPO]	[INSERIRE TEMPO]
8	[INSERIRE TEMPO]	[INSERIRE TEMPO]	[INSERIRE TEMPO]
12	[INSERIRE TEMPO]	[INSERIRE TEMPO]	[INSERIRE TEMPO]
16	[INSERIRE TEMPO]	[INSERIRE TEMPO]	[INSERIRE TEMPO]

Table 1: Tempi di esecuzione al variare delle risorse e del partizionamento.

3.3 Analisi della Scalabilità

[DA COMPLETARE UNA VOLTA DISPONIBILI I DATI]
Placeholder per l'analisi: Si osserva che aumentando il numero di nodi worker da 2 a 4, il tempo di esecuzione diminuisce, evidenziando la capacità di Spark di scalare orizzontalmente. In particolare:

- Con un numero basso di partizioni (es. 4), l'aumento dei worker potrebbe non portare benefici lineari a causa del sottoutilizzo delle CPU disponibili.
- La configurazione ottimale sembra essere quella con [X] partizioni su [Y] worker, dove il bilanciamento tra overhead di gestione dei task e parallelismo effettivo è massimizzato.

3.4 Impatto del Partizionamento

[DA COMPLETARE UNA VOLTA DISPONIBILI I DATI]
Placeholder per l'analisi: L'utilizzo esplicito di `repartition` e del parametro di partizionamento in `reduceByKey` ha mostrato un impatto significativo. Un numero di partizioni troppo basso su cluster grandi (es. 4 partizioni su 4 worker) crea colli di bottiglia, lasciando core inattivi. Viceversa, un numero eccessivo di partizioni (es. 16 su 2 worker) introduce un overhead di scheduling non necessario.

4 Conclusioni

Il progetto ha permesso di verificare l'efficacia di Apache Spark nell'analisi di dataset sismici di grandi dimensioni. L'approccio a due passaggi ha garantito un utilizzo efficiente della memoria, mentre i test su DataProc hanno evidenziato l'importanza di un corretto tuning del numero di partizioni in relazione alle risorse fisiche disponibili nel cluster.