

---

# Earthquake Application

## Relazione del progetto

---

Nicola Modugno

A.A. 2024-25

## 1 INTRODUZIONE E DEFINIZIONE DEL PROBLEMA

Il presente lavoro implementa in Apache Spark un sistema per l'identificazione di co-occorrenze sismiche su un dataset di 3.4 milioni di eventi registrati nel periodo compreso tra il 1990-01-01 ed il 2023-07-29

### 1.1 DEFINIZIONE DI CO-OCCORRENZA

Due località geografiche *co-occorrono* quando si verificano eventi sismici in entrambe le località nello stesso giorno ma in celle geografiche distinte. L'obiettivo è identificare la coppia di località con il massimo numero di giornate in cui entrambe registrano attività sismica.

### 1.2 PREPROCESSING DEI DATI

Il preprocessing costituisce una fase importante per la correttezza dell'analisi e si articola in tre fasi principali. La prima fase riguarda la discretizzazione spaziale: le coordinate geografiche vengono arrotondate alla prima cifra decimale con `RoundingMode.DOWN`. Questo non è un arrotondamento matematico standard, ma un troncamento delle cifre decimali, ed è essenziale implementarlo in questa modalità affinché il risultato finale sia corretto:

```
def roundCoordinate(coord: Double): Double = {  
    BigDecimal(coord).setScale(1, RoundingMode.DOWN).toDouble  
}
```

Ad esempio, la coppia di coordinate (37.547, 15.399) viene trasformata in (37.5, 15.3) e non in (37.5, 15.4) come avverrebbe con un arrotondamento matematico tradizionale.

La seconda fase riguarda la discretizzazione temporale: i timestamp vengono normalizzati estraendo solo la data nel formato YYYY-MM-DD, definendo così una finestra temporale giornaliera e rimuovendo l'informazione oraria. Infine, la terza fase consiste nella rimozione dei duplicati. Eventi multipli nella stessa cella geografica e nello stesso giorno vengono considerati come un singolo evento. Questa fase è essenziale per evitare che la coppia con massime co-occorrenze corrisponda alla stessa località.

Applicando queste trasformazioni al dataset originale di **3,445,751** eventi si ottengono **2,198,460** eventi unici, corrispondenti a celle geografiche uniche per giornata.

## 2 APPROCCI IMPLEMENTATI

Sono stati implementati tre approcci basati su diverse primitive di Apache Spark per valutare trade-off tra semplicità implementativa, utilizzo di memoria e performance.

### 2.1 APPROCCIO 1: GROUPBYKEY

La strategia più diretta utilizza groupByKey per aggregare località per data:

```
val eventsByDate = uniqueEvents
  .map { case (location, date) => (date, location) }
  .repartition(numPartitions)

val locationsByDate = eventsByDate.groupByKey()

val coOccurrences = locationsByDate.flatMap {
  case (date, locations) =>
    val locList = locations.toList.sorted
    for {
      i <- locList.indices
      j <- (i + 1) until locList.length
    } yield (LocationPair(locList(i), locList(j)), date)
}
```

Questa soluzione richiede lo shuffle completo di tutte le località per ogni data e la materializzazione in memoria delle liste di località per data può essere onerosa per giornate con molti eventi.

### 2.2 APPROCCIO 2: AGGREGATEBYKEY

Il secondo approccio utilizza aggregazione incrementale tramite Set per ridurre il volume di dati trasferiti:

```
val locationsByDate = eventsByDate
  .aggregateByKey(Set.empty[Location])(
    (set, loc) => set + loc, // combiner
    (set1, set2) => set1 ++ set2 // merger
  )
```

In questo caso l'aggregazione locale (map-side) riduce significativamente la quantità di dati nello shuffle, mentre la struttura Set elimina automaticamente eventuali duplicati rimanenti.

## 2.3 APPROCCIO 3: REDUCEBYKEY

Infine, l'approccio più ottimizzato sfrutta `reduceByKey` per la rimozione dei duplicati:

```
val uniqueEvents = normalizedEvents
  .map { case (lat, lon, date) =>
    ((Location(lat, lon), date), 1)
  }
  .reduceByKey(_ + _) // deduplicazione distribuita
  .map { case ((location, date), _) => (location, date) }
```

Questa riduzione distribuita a due livelli (map-side e reduce-side) minimizza shuffle e memoria, risultando particolarmente efficiente su dataset con alta cardinalità di chiavi.

## 2.4 CONTEGGIO CO-OCCORRENZE

Tutti gli approcci convergono nella fase finale di conteggio:

```
val pairCounts = coOccurrences
  .map { case (pair, _) => (pair, 1) }
  .reduceByKey(_ + _)

val maxPair = pairCounts.reduce((a, b) =>
  if (a._2 > b._2) a else b
)
```

Il risultato finale include la coppia vincente e l'array di date ordinate in cui co-occorre.

# 3 SETUP SPERIMENTALE

## 3.1 INFRASTRUTTURA CLOUD

Gli esperimenti sono stati condotti su Google Cloud Dataproc con configurazioni hardware omogenee:

Tabella 3.1: Configurazioni Cluster Testate

Config	Workers	vCPU Tot.	RAM Tot.	Partizioni
2W	2	12	48 GB	8, 16, 32
3W	3	16	64 GB	12, 24, 36
4W	4	20	80 GB	16, 32, 48

Tutte le macchine (master e worker) utilizzano tipo n2-standard-4 (4 vCPU, 16 GB RAM) come specificato dai requisiti di progetto. Le configurazioni Spark sono: executor memory 10GB, driver memory 6GB, con overhead di 2GB e 1GB rispettivamente.

### 3.2 PARTIZIONAMENTO

Tutti gli approcci utilizzano **Hash Partitioning** tramite il metodo `repartition(n)` per distribuire uniformemente i dati. Sono state testate configurazioni seguendo la regola empirica di  $1 - 6 \times$  il numero di vCPU disponibili per identificare la zona ottimale.

### 3.3 METRICHE RACCOLTE

Il sistema genera automaticamente metriche dettagliate per ogni esecuzione. In particolare vengono raccolti i seguenti dati: `total_events` rappresenta il numero di eventi caricati dal CSV (3,445,751); `unique_events` indica gli eventi dopo la fase di deduplicazione (2,198,460); `co_occurrences` conta le tuple (LocationPair, date) generate dal processo (225,085,862); `max_count` rappresenta il numero di giorni in cui la coppia vincente co-occorre (10,837). Vengono inoltre registrati i tempi di caricamento, analisi e totale, tutti espressi in millisecondi.

## 4 RISULTATI SPERIMENTALI

### 4.1 PERFORMANCE COMPARATIVA

La Tabella 4.1 riporta i tempi di analisi (in secondi) per diverse configurazioni:

Tabella 4.1: Tempi di Analisi per Configurazione (secondi)

Config	GroupByKey	AggregateByKey	ReduceByKey
2W-8p	882.8	837.8	992.2
2W-16p	832.3	772.6	748.8
2W-32p	714.6	724.8	747.9
3W-12p	554.5	547.2	538.5
3W-24p	455.6	492.8	488.5
3W-36p	444.5	444.0	435.3
4W-16p	403.1	432.2	442.0
4W-32p	424.8	399.6	387.3
4W-48p	<b>329.4</b>	342.5	<b>329.4</b>

Dall'analisi della tabella emerge che ReduceByKey risulta il più veloce su configurazioni ottimali (2W-16p, 4W-32p, 4W-48p), mentre GroupByKey si dimostra competitivo su cluster grandi con molte partizioni, eguagliando i tempi di esecuzione dell'approccio che implementa ReduceByKey nella configurazione 4W-48p. AggregateByKey, invece, offre performance consistenti ma generalmente inferiori rispetto agli altri due approcci.

## 4.2 IMPATTO DEL PARTIZIONAMENTO

Il numero di partizioni influenza significativamente le performance. La Figura 4.1 mostra l'effetto per l'approccio GroupByKey:

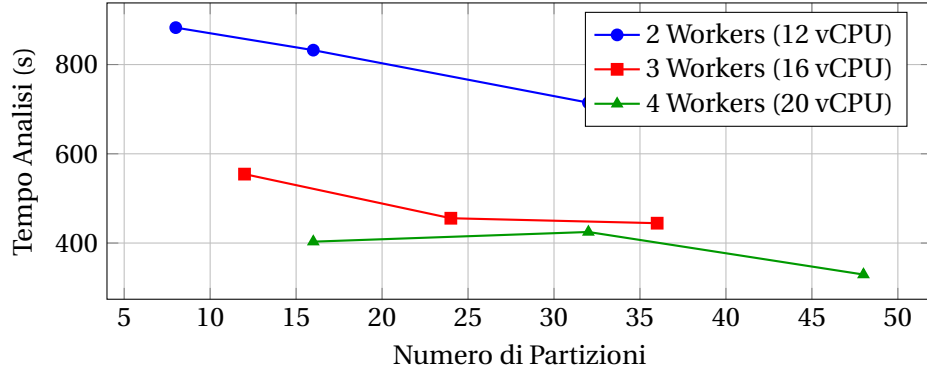


Figura 4.1: Impatto Numero Partizioni su GroupByKey

Dalla figura si evince l'esistenza di tre zone di performance distinte. Quando il numero di partizioni è inferiore a  $2 \times$  il numero di vCPU (sottopartizionamento), si verifica una sottoutilizzazione della CPU con un incremento dei tempi del 20-24%. La zona ottimale si colloca tra  $2 \times$  e  $4 \times$  il numero di vCPU, dove si raggiunge il massimo parallelismo e i tempi minimi. Infine, quando si superano  $4 \times$  vCPU (sovrappartizionamento), l'overhead di scheduling diventa apprezzabile e le performance degradano leggermente.

## 4.3 SCALABILITÀ

L'analisi di strong scaling con partizioni ottimali (circa  $2 - 3 \times$  vCPU) rivela risultati interessanti:

Tabella 4.2: Analisi Scalabilità Workers

Approccio	2W (s)	3W (s)	4W (s)	Speedup
GroupByKey	714.6	444.5	329.4	$2.17 \times$
AggregateByKey	724.8	444.0	342.5	$2.12 \times$
ReduceByKey	747.9	435.3	329.4	$2.27 \times$

Lo speedup  $S = T_{2w} / T_{4w}$  mostra eccellente scalabilità. Calcolando l'efficienza come mostrato nell'Equazione 4.1, si osserva un comportamento super-lineare:

$$\text{Efficiency} = \frac{S}{n/2} = \frac{2.17}{2} = 1.08 \text{ (108\%)} \quad (4.1)$$

Questo fenomeno può essere attribuito a tre fattori principali: il migliore utilizzo della cache dovuto a partizioni più piccole per worker, la riduzione della latency nell'aggregazione locale, e un migliore bilanciamento del carico su cluster più grandi.

## 5 DISCUSSIONE

### 5.1 INTERPRETAZIONE DEI RISULTATI

I risultati confermano che la scelta dell'approccio e del partizionamento ha impatto significativo sulle performance. ReduceByKey risulta vincente in quanto minimizza lo shuffle attraverso la combinazione locale pre-shuffle, riduce l'utilizzo di memoria evitando la materializzazione di collezioni intermedie, e scala meglio all'aumentare della cardinalità delle chiavi.

AggregateByKey offre un buon compromesso tra flessibilità e performance, mantenendo un comportamento prevedibile su diverse configurazioni ed risultando particolarmente adatto quando serve un'aggregazione personalizzata (ad esempio con strutture dati come Set o Map).

GroupByKey rimane competitivo quando il cluster dispone di RAM abbondante, il numero di località per data è moderato, e la semplicità del codice è prioritaria rispetto all'ottimizzazione estrema delle performance.

### 5.2 CONFIGURAZIONE OTTIMALE IDENTIFICATA

Per il dataset analizzato (3.4M eventi, 225M tuple intermedie), la configurazione ottimale identificata prevede l'utilizzo di un cluster con 4 workers di tipo n2-standard-4, per un totale di 20 vCPU. Il numero di partizioni ottimale è 48, corrispondente a un ratio di 2.4 volte il numero di vCPU. L'approccio raccomandato è ReduceByKey o GroupByKey, che su questa configurazione offrono performance equivalenti con un tempo di esecuzione di 329 secondi (5.5 minuti).

## 6 CONCLUSIONI

Questo lavoro ha presentato un'analisi comparativa di tre strategie per l'identificazione di co-occorrenze sismiche distribuite in Apache Spark. I risultati sperimentali hanno evidenziato tre aspetti fondamentali.

In primo luogo, la scelta dell'approccio si rivela critica: ReduceByKey offre fino al 33% di miglioramento rispetto a GroupByKey con sottopartizionamento (748s contro 992s sulla configurazione 2W-16p).

In secondo luogo, il partizionamento domina le performance complessive: una configurazione ottimale (2-4 volte il numero di vCPU) riduce il tempo fino al 63% rispetto a un sottopartizionamento (329s contro 883s sulla configurazione 4W).

Infine, la scalabilità si dimostra eccellente: è stato osservato uno speedup super-lineare di 2.27 volte tra 2 e 4 workers, indicando un'efficienza di parallelizzazione del 113%.

L'implementazione completa, inclusi gli script di deployment e il sistema di raccolta metriche, è disponibile nel repository del progetto: <https://github.com/nicola-modugno/EarthquakeApplication-SCP/>.