Project Exam

Marco Miozza

Repo: https://github.com/marcomiozza/Scalable.git

1. Problema e specifiche

Caso di studio

L'applicazione analizza un insieme di eventi sismici per individuare co-occorrenze tra località geografiche differenti nella stessa giornata. Per rendere l'analisi più robusta, le coordinate di latitudine e longitudine vengono arrotondate alla prima cifra decimale, così da considerare aree geografiche approssimate. Analogamente, l'informazione temporale viene ridotta alla sola data, adottando una finestra temporale di un giorno. A partire da queste trasformazioni, l'obiettivo è determinare la coppia di località distinte che co-occorre più frequentemente nello stesso giorno. L'output è costituito dalla coppia individuata e dall'elenco delle date, in ordine crescente, in cui la co-occorrenza si verifica, eliminando eventuali duplicati dovuti all'arrotondamento.

Il problema viene utilizzato come caso di studio per analizzare la strong scalability del sistema distribuito. In particolare, si osserva come varia il tempo di esecuzione al crescere del numero di worker, mantenendo invariati dataset e algoritmo.

Per ogni configurazione viene misurato il tempo totale di esecuzione del job; tali valori vengono poi utilizzati per valutare la strong scalability del sistema, calcolando lo speedup rispetto a una configurazione di riferimento e analizzando l'efficienza al crescere del numero di worker.

2. Dataset

Tutti i dataset utilizzati condividono la stessa struttura logica: ogni record rappresenta un evento sismico ed è descritto da coordinate geografiche e informazione temporale. il formato è CSV con header e lo schema dei campi è identico in tutte le versioni disponibili. Le differenze riguardano esclusivamente la dimensione, intesa come numero di record e volume complessivo dei dati. Questa uniformità permette di utilizzare la stessa pipeline di elaborazione sia per i test locali su dataset ridotti sia per le analisi di scalabilità sul dataset completo eseguite su Dataproc. Le prove di strong scaling sono state effettuate utilizzando il dataset di dimensione maggiore, le cui caratteristiche sono:

- Numero di record: **3.445.752**
- Dimensione su disco: **174,63 MB**

In tutte le esecuzioni sperimentali il dataset è rimasto invariato, così da mantenere costante il carico computazionale. Le differenze nei tempi di esecuzione osservate tra le varie configurazioni sono quindi attribuibili unicamente alla variazione del numero di worker del cluster e alla conseguente distribuzione del lavoro.

3. Approccio e implementazione

L'algoritmo è organizzato come una sequenza di **trasformazioni su RDD** che vengono materializzate solo quando viene invocata un'**azione**. L'obiettivo è ridurre progressivamente il volume dei dati e limitare le operazioni di shuffle alle sole fasi necessarie per il calcolo delle co-occorrenze.

La prima fase riguarda il parsing e la normalizzazione dei dati di input. L'RDD viene ottenuto dalla lettura del CSV e successivamente ripartizionato tramite `repartition(numPartitions)` per controllare il livello di parallelismo. Ogni record viene trasformato leggendo latitudine, longitudine e data; le coordinate vengono arrotondate alla prima cifra decimale e l'informazione temporale viene ridotta alla sola data. Questa trasformazione è implementata tramite una `map`:

```
.map { row =>
  val lat = Math.round(row.getAs[String]("latitude").toDouble * 10.0) / 10.0
  val lon = Math.round(row.getAs[String]("longitude").toDouble * 10.0) / 10.0
  val date = LocalDate.parse(row.getAs[String]("date").substring(0, 10))
  ((lat, lon, date), 1)
}
```

Poiché l'arrotondamento può produrre più eventi nella stessa cella spazio-temporale, la fase successiva è una deduplicazione basata sulla chiave `(lat, lon, date)`. Questa operazione è realizzata tramite `reduceByKey`, che è una trasformazione wide e comporta uno shuffle tra partizioni:

```
.reduceByKey(_ + _)
.map { case ((lat, lon, date), _) => (date, Location(lat, lon)) }
```

In questo modo ogni località contribuisce al massimo una volta per ciascun giorno. Successivamente, gli eventi vengono aggregati per data utilizzando `aggregateByKey`, costruendo per ogni giorno l'insieme delle località distinte in cui si sono verificati terremoti:

```
.aggregateByKey(Set.empty[Location])(
  (set, loc) => set + loc,
  (set1, set2) => set1 ++ set2
)
.filter(_._2.size > 1)
```

Anche questa è una trasformazione wide, poiché richiede uno shuffle per raggruppare le chiavi uguali. Il filtro elimina le date con una sola località, che non possono generare co-occorrenze. Per ciascuna data rimanente vengono quindi generate tutte le coppie distinte di località tramite una `flatMap` (trasformazione narrow):

```
.flatMap { case (_, locations) =>
  val locList = locations.toList
  for {
    i <- locList.indices
    j <- i + 1 until locList.size
  } yield (LocationPair(locList(i), locList(j)).normalized, 1)
}
```

Le coppie prodotte vengono poi aggregate globalmente con un'ulteriore `reduceByKey`, che calcola il numero totale di co-occorrenze per ciascuna coppia:

```
.reduceByKey(_ + _)
```

A questo punto viene individuata la coppia con il numero massimo di co-occorrenze tramite un'azione `reduce`, che materializza il risultato sul driver:

```
val (winningPair, maxCount) = pairCounts.reduce { (a, b) =>
  if (a._2 > b._2) a else b
}
```

Infine, una volta determinata la coppia vincente, vengono recuperate le date in cui tale coppia co-occorre. L'ordinamento è effettuato tramite `sortBy` (trasformazione wide), mentre `collect()` rappresenta l'azione che riporta i risultati sul driver:

```
.keys
.sortBy(identity[LocalDate])
.collect()
```

Le strutture intermedie `events` ed `eventsByDate` vengono mantenute in memoria tramite `cache()`, poiché riutilizzate in più fasi dell'elaborazione. Questa scelta evita ricalcoli lungo il DAG delle trasformazioni e contribuisce alla riduzione del tempo totale di esecuzione.

4. Setup sperimentale

Le configurazioni sperimentali sono state ottenute creando cluster con un numero variabile di worker, mantenendo invariati il tipo di macchina, la regione e la dimensione dei dischi.

```
gcloud dataproc clusters create eq-wX \
  --region=us-central1 \
  --num-workers=X \
  --master-machine-type=n2-standard-4 \
  --worker-machine-type=n2-standard-4 \
  --master-boot-disk-size=240 \
  --worker-boot-disk-size=240
```

Le configurazioni considerate corrispondono a valori di X compresi tra 2 e 5, identificati come w2, w3, w4 e w5. Per ciascuna configurazione il cluster è stato creato ex novo, così da garantire condizioni di esecuzione omogenee. L'applicazione è stata eseguita tramite sottomissione di un job Spark al cluster corrispondente, utilizzando lo stesso comando in tutte le prove, variando unicamente i parametri dipendenti dalla configurazione.

```
gcloud dataproc jobs submit spark \
  --cluster=eq-wX \
  --region=us-central1 \
  --jar=gs://<bucket>/earthquake.jar \
  -- \
  <input-path> \
  <output-path> \
  <num-partitions>
```

Partizionamento e metodologia di misura

Il numero di partizioni è fornito all'applicazione come parametro di esecuzione ed è utilizzato per ripartizionare l'RDD di input nella fase iniziale dell'elaborazione. Il valore adottato cresce in modo proporzionale al numero di worker del cluster, mantenendo un rapporto costante tra risorse disponibili e grado di parallelismo. Nello specifico, sono state utilizzate le seguenti configurazioni:

- 2 worker, 24 partizioni
- 3 worker, 36 partizioni
- 4 worker, 48 partizioni
- 5 worker, 60 partizioni

Questa scelta consente di migliorare il bilanciamento del carico tra i nodi del cluster, in particolare durante le fasi di aggregazione e shuffle, riducendo il rischio che singole partizioni diventino colli di bottiglia. Il tempo di esecuzione è misurato all'interno dell'applicazione come differenza tra l'istante di avvio e il termine dell'elaborazione, utilizzando `System.currentTimeMillis()`.

5. Risultati e analisi di strong scalability

L'analisi di strong scalability è stata condotta mantenendo fisso il dataset e variando il numero di worker del cluster da 2 a 5.
Tempi misurati:

- 2 worker: 523343 ms (≈ 523 s)
- 3 worker: 346364 ms (≈ 346 s)
- 4 worker: 264816 ms (≈ 265 s)
- 5 worker: 227427 ms (≈ 227 s)

La riduzione percentuale del tempo di esecuzione passando da 2 a 5 worker è pari a:

(523343 − 227427) / 523343 = 0.565 ≈ **56%**

Speedup

Lo speedup è definito come: S(n) = T(2) / T(n)

dove T(n) è il tempo di esecuzione con n worker e T(2) è il tempo misurato con 2 worker, utilizzato come baseline. Sostituendo i valori sperimentali si ottiene:

- S(2) = 523343 / 523343 = **1.00**
- S(3) = 523343 / 346364 ≈ **1.51**
- S(4) = 523343 / 264816 ≈ **1.98**
- S(5) = 523343 / 227427 ≈ **2.30**

I risultati mostrano un aumento monotono dello speedup al crescere del numero di worker. Tuttavia, il comportamento risulta sub-lineare rispetto alla scalabilità ideale: l'incremento di prestazioni non è proporzionale all'aumento delle risorse, a causa dell'overhead di comunicazione e della presenza di una componente non parallelizzabile dell'applicazione.

Strong Scaling Efficiency

Non essendo possibile eseguire il job con un solo worker su Dataproc, la configurazione a 2 worker viene utilizzata come baseline.

L'efficienza di strong scaling è definita come:

E(n) = T(2) / ((n/2) · T(n)) = (2/n) · S(n)

Sostituendo i valori sperimentali si ottiene:

E(2) = (2/2) · 1.00 = **1.00**

E(3) = (2/3) · 1.51 = 0.6667 · 1.51 ≈ **1.01**

E(4) = (2/4) · 1.98 = 0.5 · 1.98 ≈ **0.99**

E(5) = (2/5) · 2.30 = 0.4 · 2.30 ≈ **0.92**

Un valore di efficienza pari a 1 indica una scalabilità perfettamente lineare rispetto alla baseline. I risultati mostrano che fino a 4 worker le risorse aggiuntive vengono sfruttate quasi interamente, mentre con 5 worker si osserva una lieve perdita di efficienza dovuta all'overhead di comunicazione e alla frazione non parallelizzabile del programma.