



CloudiPi

Cloud Computing

Gianvito Alcamo
Andrea Gennari
Giampietro Nardone

Contents

1	Dataset	2
2	Hadoop MapReduce Pseudo Codice	2
3	Risultati Sperimentali	3
3.1	Test su Hadoop e Spark	3
3.2	Test su Hadoop con In-Mapper Combiner e con Hadoop Combiner	4
3.3	Test su Programma Sequenziale	5
4	Spark	5
4.1	Stage 0 - Preprocessing e tokenizzazione	5
4.2	Stage 1 - Aggregazione globale per parola	6
4.3	Stage 2 - Scrittura del risultato	6

1 Dataset

Abbiamo utilizzato il sito gutenberg.org per scaricare numerosi file di testo, in particolare libri, dizionari, encyclopedie, con dimensioni variabili da pochi kilobyte (KB) fino a qualche decina di megabyte (MB), da impiegare nei test. A partire da questi file, abbiamo sviluppato un programma Python, `merging.py`, che consente di generare set di dati con dimensioni totali prefissate e dimensioni dei singoli file controllate, in base alle necessità del test. Il set iniziale ottenuto ha una dimensione complessiva di circa 1GB. Tali configurazioni sono state impiegate per eseguire un'analisi comparativa delle diverse implementazioni dell'inverted index. I dettagli delle combinazioni utilizzate sono riportati nella Sezione 3, dedicata ai risultati sperimentali.

2 Hadoop MapReduce Pseudo Codice

Algorithm 1: Hadoop Inverted Index con Combiner

```
1 InvertedIndexCombiner
  Input : Insieme di file di testo come input
  Output: Indice invertito con frequenza delle parole
            per file

2 Mapper:
3 foreach linea di input do
4   Ottieni il nome del file corrente file
5   Suddividi la linea in parole: words  $\leftarrow$  split(line)
6   foreach word  $\in$  words do
7     if word non vuota then
8       lowerWord  $\leftarrow$  word convertito in
                     minuscolo
9       Emetti (lowerWord, (file, 1))
10    end
11   end
12 end

13 Combiner (locale su ogni nodo Map):
14 foreach word con valori associati {(filei, counti)} do
15   Inizializza HashMap fileCounts
16   foreach (filei, counti) do
17     fileCounts[filei]  $\leftarrow$ 
                  fileCounts[filei] + counti
18   end
19   foreach (file, count)  $\in$  fileCounts do
20     Emetti (word, (file, count))
21   end
22 end

23 Reducer:
24 foreach word con valori associati {(filei, counti)} do
25   Inizializza mappa freqMap
26   foreach (filei, counti) do
27     freqMap[filei]  $\leftarrow$  freqMap[filei] + counti
28   end
29   Costruisci stringa di output concatenando le
      coppie (file, count)
30   Emetti (word, outputValue)
31 end
```

Algorithm 2: Hadoop Inverted Index con In-Mapper Combiner

```
1 InvertedIndex
  Input : Insieme di file di testo come input
  Output: Indice invertito con frequenza delle parole
            per file

2 Mapper:
3 Inizializza HashMap freqMap
4 Metodo Map:
5 foreach linea di input do
6   Ottieni il nome del file corrente file
7   Suddividi la linea in parole: words  $\leftarrow$  split(line)
8   foreach word  $\in$  words do
9     if word non vuota then
10      lowerWord  $\leftarrow$  word convertito in
                     minuscolo
11      Inizializza Word_File wf(lowerWord,
                     file)
12      Inserisci wf in freqMap con valore
                     incrementato di 1 o 1
13    end
14  end
15 end
16 Metodo cleanup:
17 foreach elemento entry di freqMap do
18   ottieni chiave wf da entry
19   ottieni valore val da entry
20   Emetti (wf.word, (wf.file, val))
21 end

22 Reducer:
23 Metodo reduce:
24 foreach word con valori associati {(filei, counti)} do
25   Inizializza mappa freqMap
26   foreach (filei, counti) do
27     freqMap[filei]  $\leftarrow$  freqMap[filei] + counti
28   end
29   Costruisci stringa di output concatenando le
      coppie (file, count)
30   Emetti (word, outputValue)
31 end
```

3 Risultati Sperimentalni

3.1 Test su Hadoop e Spark

Prima di partire con i test sui due programmi Hadoop, quello con Hadoop Combiner e quello con In-Mapper Combiner, abbiamo voluto valutare il numero ottimale di reducer da usare, usando come parametro il tempo di esecuzione. Abbiamo dunque testato un dataset di dimensione 512MB con file di 64MB, variando i Reducer da 1 a 12. Scelto il numero di Reducer abbiamo potuto effettuare gli altri test.

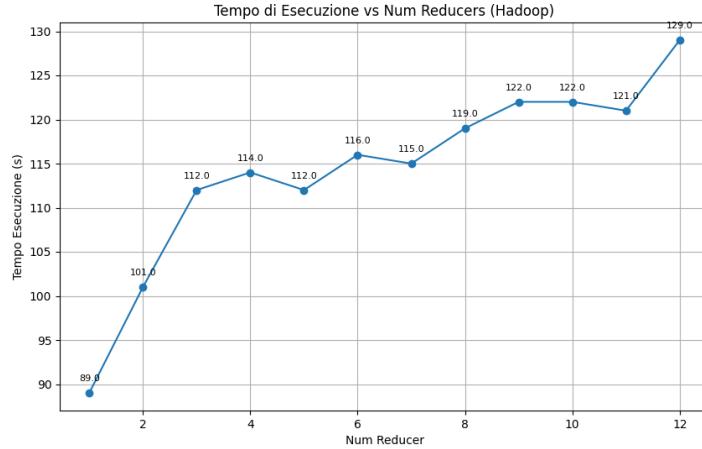


Figure 1: Tempi di esecuzione di Hadoop al variare dei reducer

Come si evince dal grafico, l'utilizzo di un solo Reducer è sufficiente per avere buoni risultati in termini di tempo per il caso in questione, quindi abbiamo usato un solo reducer. Presentiamo dunque i dati su memoria e tempo di esecuzione relativi ad Hadoop con In-Mapper Combiner e Spark. Abbiamo fissato la dimensione complessiva dei dati in 1GB, facendo variare la grandezza dei file, da 1MB a 1GB.

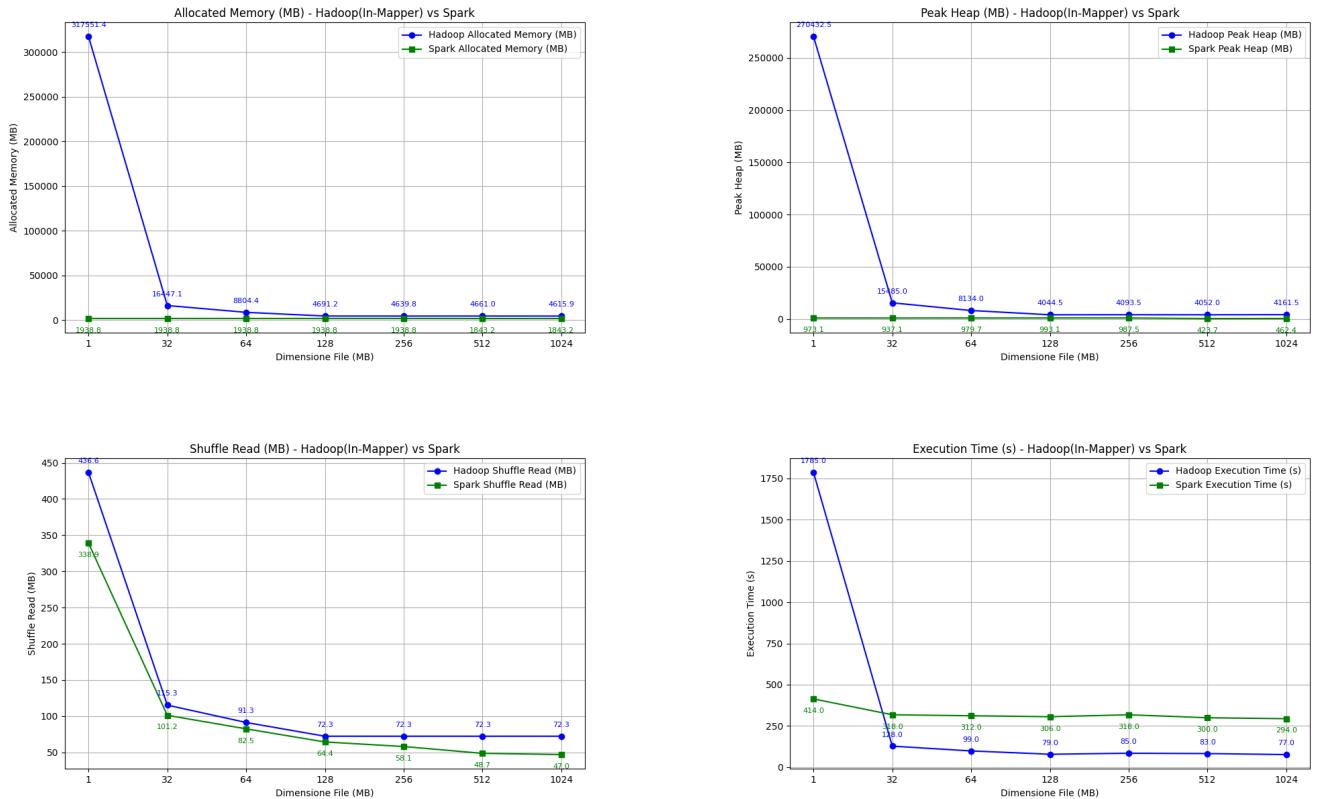


Figure 2: Confronto Hadoop Vs Spark su 1GB di dati totali

Come ulteriore confronto tra Hadoop (In-Mapper Combiner) e Spark, prendiamo un dataset di dimensione complessiva di 2GB composto da 2 file di testo grandi 1GB ciascuno.

Confronto Hadoop vs Spark su dataset da 2 GB (2 file da 1 GB)



Figure 3: Confronto Hadoop Vs Spark su 2 file da 1GB ciascuno

3.2 Test su Hadoop con In-Mapper Combiner e con Hadoop Combiner

Effettuiamo dei test anche per confrontare le due implementazioni in Hadoop, una con In-Mapper Combiner e l'altra con Hadoop Combiner. Anche in questo caso il dataset ha dimensione fissata a 1GB, variando le dimensioni dei file.

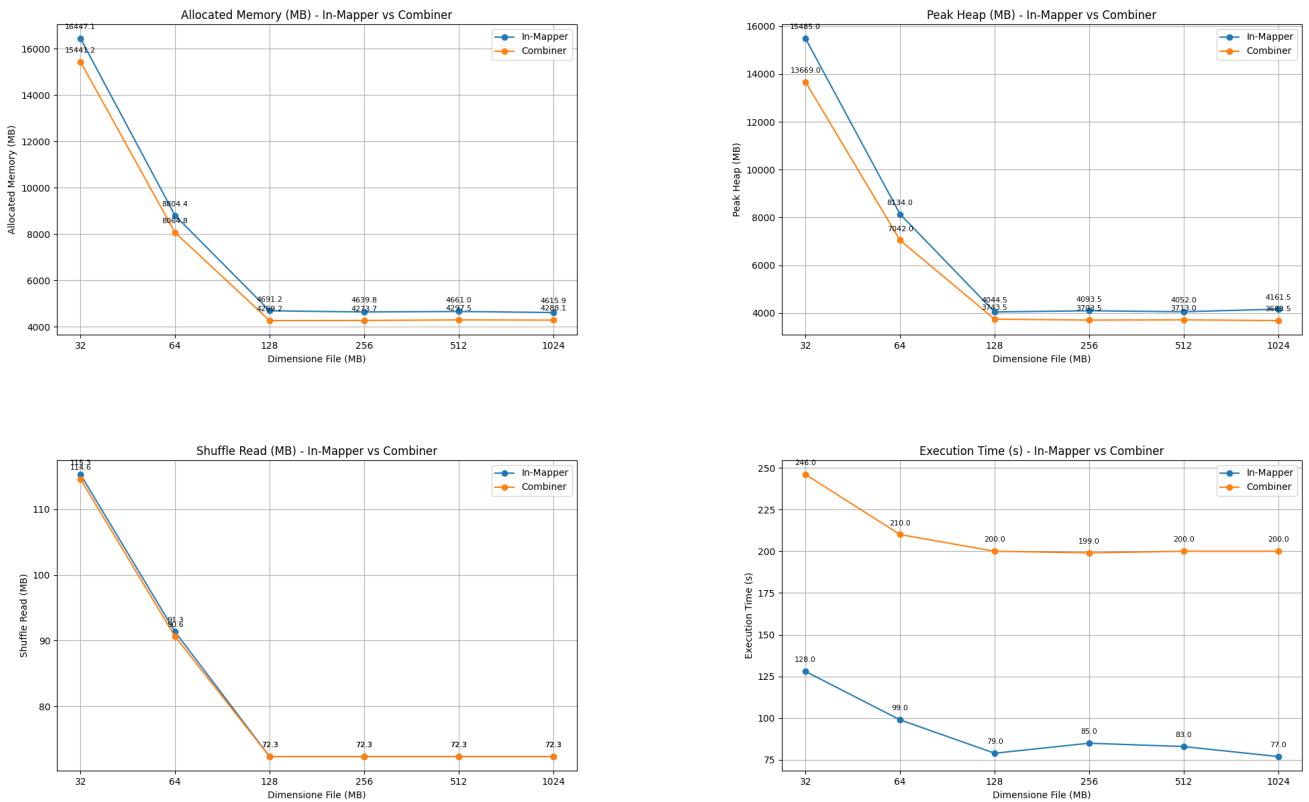


Figure 4: Confronto Hadoop In-Mapper Combiner Vs Hadoop Combiner

3.3 Test su Programma Sequenziale

Il programma python sequenziale è stato testato per dare un'idea del comportamento senza metodi specifici come Hadoop e Spark. Anche in questo caso abbiamo utilizzato un dataset di dimensione fissata 1GB e abbiamo variato la dimensione dei file.

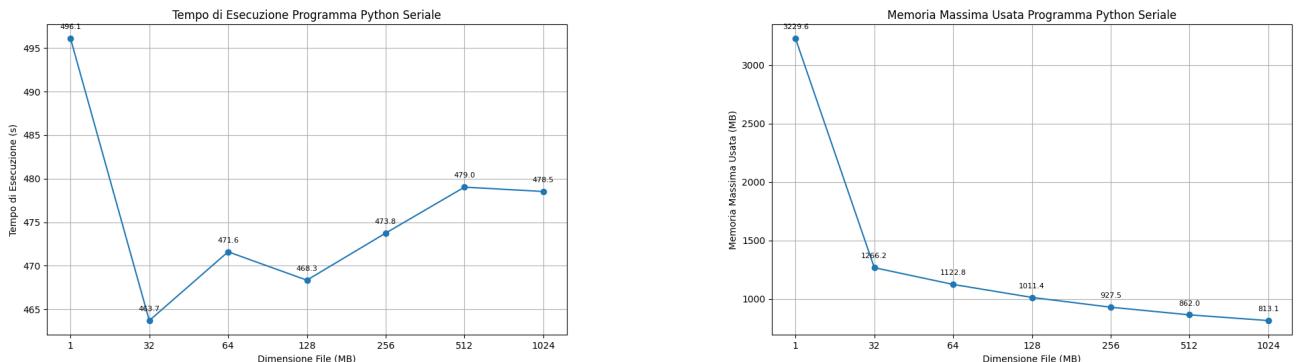


Figure 5: Test su Programma Python Sequenziale

4 Spark

L'applicazione che implementa l'Inverted Index in Spark inizia caricando i file all'interno di un DataFrame. Questo approccio consente di associare ad ogni riga di testo il nome del file di provenienza, mediante l'aggiunta di una nuova colonna filename ottenuta attraverso la funzione `input_file_name()`. Il DataFrame viene poi convertito in un RDD, così da consentire un controllo più efficiente della memoria durante le successive fasi di elaborazione. Rispetto ad alternative come `wholeTextFiles()`, questa strategia evita il caricamento in memoria dell'intero contenuto dei file insieme ai rispettivi nomi, risultando più scalabile.

4.1 Stage 0 - Preprocessing e tokenizzazione

Nello Stage 0 viene eseguita la fase di caricamento e preprocessamento iniziale dei dati. I file vengono letti riga per riga tramite `read.text()`, e ad ogni riga viene associato il relativo nome di file grazie alla funzione `input_file_name()`, producendo un DataFrame con due colonne: il testo (`value`) e il nome del file (`filename`).

Successivamente, il DataFrame viene convertito in un RDD. Ogni riga viene trasformata in una coppia (`filename`, `riga`), dalla quale si esegue la tokenizzazione del testo tramite un'espressione regolare. Viene quindi utilizzato il costruttore `Counter` per contare localmente quante volte ciascuna parola compare all'interno della riga. In output si ottengono coppie del tipo `((parola, filename), conteggio locale)`. Infine, tramite `reduceByKey` viene eseguita un'aggregazione delle occorrenze di ciascuna parola. Tale operazione rappresenta la **wide transformation** principale di questo stage, in quanto richiede uno shuffle per raggruppare le chiavi `(parola, filename)`.

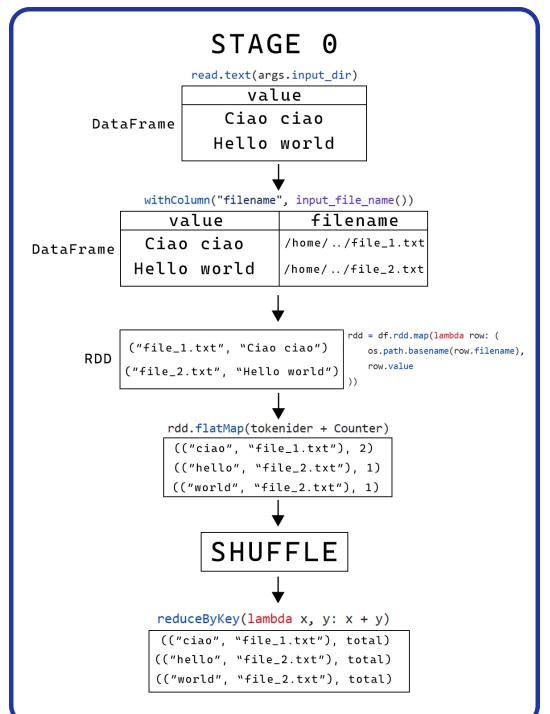


Figure 6: Schema dello Stage 0

4.2 Stage 1 - Aggregazione globale per parola

In questo stage si procede alla costruzione dell'indice invertito vero e proprio. I dati vengono rimappati per trasformare la chiave da **(parola, filename)** a **parola**, mentre il valore diventa una lista contenente il nome del file e il conteggio: **(parola, [(filename, conteggio)])**.

Successivamente, viene eseguito un secondo **reduceByKey** per aggregare tutte le liste associate alla stessa parola, concatenando i risultati: **(parola, [(file1, count1), (file2, count2), ...])**.

Anche questa fase comporta una **wide transformation**, dovuta al secondo **reduceByKey** sulla chiave **parola**, che attiva un nuovo shuffle.

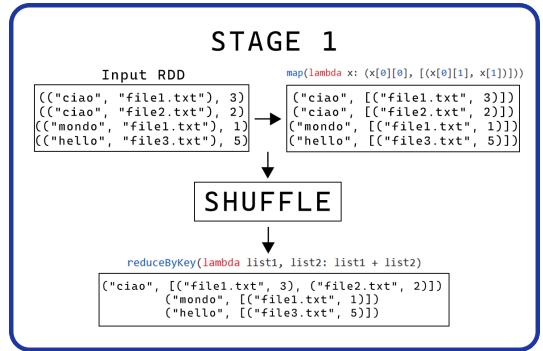


Figure 7: Schema dello Stage 1

4.3 Stage 2 - Scrittura del risultato

Nello Stage 2 viene eseguita l'azione finale **saveAsTextFile()**, che avvia l'esecuzione completa del job e produce i file di output contenenti l'indice invertito.

Prima della scrittura, Spark può effettuare un **repartition** implicito per distribuire i dati di output.

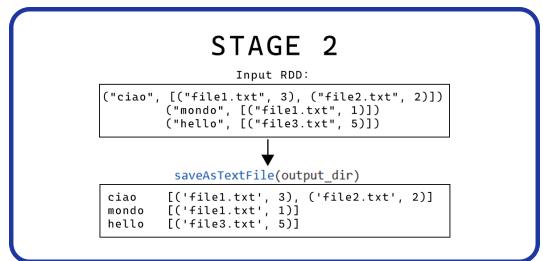


Figure 8: Schema dello Stage 2