

# Implementazione di Finding Triangles con Hadoop MapReduce

Sistemi di elaborazione di grandi quantità di dati 2016

Nicola Febbrari

Università degli Studi di Verona

Facoltà di Scienze

`nicola.febbrari@studenti.univr.it`

3 maggio 2017

## 1 Introduzione

---

Lo scopo del progetto è quello di implementare un algoritmo per calcolare il numero di triangoli presenti in un grafo non diretto, utilizzando le tecniche di MapReduce e il Framework Hadoop.

## 2 Il problema

---

I social network negli ultimi anni hanno avuto una notevole diffusione, l'aumento del numero di utenti che interagiscono con questi sistemi ha avuto come conseguenza un incremento della quantità di dati che devono essere registrati, gestiti ed ovviamente elaborati.

Un social network può essere rappresentato matematicamente da un grafo e una caratteristica molto interessante di questo grafo è il numero di triangoli<sup>1</sup> contenuti in esso. Questo numero rapportato al totale dei triangoli che esisterebbero con una distribuzione casuale e uniforme delle relazioni può essere un indice di quanto sia social il grafo analizzato. Gli algoritmi sviluppati prendono in considerazione grafi non diretti.

## 3 Strumenti e Framework

---

**Infrastruttura** Per semplicità e rapidità di configurazione ho deciso di utilizzare Cloudera come distribuzione di Hadoop nella sua versione per Docker.

**Sviluppo** Dovendo utilizzare il Framework Hadoop il programma è stato scritto in Java utilizzando le API di Hadoop 2.6 Come IDE di sviluppo ho utilizzato IntelliJ.

---

<sup>1</sup>Dati 3 nodi (A,B,C) in un grafo, se un nodo A si relaziona sia con B che con C, nel grafo viene a formarsi un triangolo se esiste anche la relazione che lega B con C.

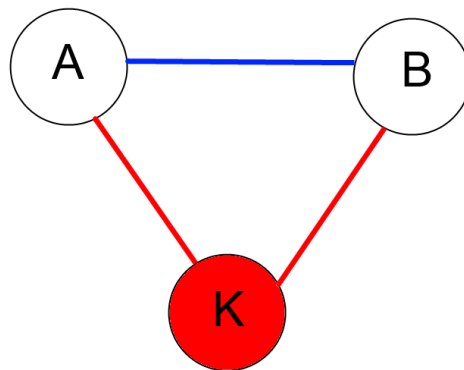
**Versioning** Come sistema di versioning ho utilizzato Git e GitHub come spazio di hosting dei sorgenti.

## 4 Implementazioni

---

### 4.1 Algoritmo 2 Jobs

L'algoritmo a 2 Jobs prevede una prima fase in cui vengono elaborate tutte le relazioni e viene creata in output la combinazione di tutti i possibili triangoli a meno dell'ultimo arco necessario per chiudere il triangolo.<sup>2</sup>



**Figura 4.1:** *Il Job 1 elabora le relazioni rosse in cui K è il nodo minore del triangolo. Il Job2 cerca la relazione blu A B.*

**Job1 Mapper** Il grafo da analizzare è rappresentato da un file di testo in cui ogni nodo è identificato da un numero intero e ogni riga rappresenta la relazione fra due nodi.

Il Mapper del primo Job carica il grafo e per ogni relazione contenuta in esso emette in output una copia *chiave-valore* in cui la chiave è il nodo minore e il valore è il nodo maggiore.

**Job1 GroupComparator** Il GroupComparator raggruppa tutte le coppie generate dal Mapper e per ogni possibile chiave crea una lista con i valori delle coppie aventi la stessa chiave

**Job1 Reducer** Il Reducer elabora l'input raggruppato dal GroupComparator e, per ogni valore in cui K è chiave, costruisce una lista *list* in cui aggiunge tutti i valori prodotti

---

<sup>2</sup>L'arco mancante è quello fra i 2 nodi maggiori del triangolo dove la relazione d'ordine del nodo è data dall'identificativo del nodo stesso

dal Mapper abbinati a  $K$ .

Quando la costruzione di *list* è terminata e tutti i valori abbinati alla chiave  $K$  sono stati inseriti esegue il processo di scrittura dell'output delle coppie *chiave-valore* in cui le chiavi sono date dalla combinazione di tutte le possibili coppie di valori presenti in *list*, mentre il valore è il nodo  $K$ .

**Job2 Mapper** Il Mapper del secondo Job unisce l'output del primo Job al grafo iniziale. Se l'elemento in INPUT A-B è una relazione di quelle presenti nel grafo iniziale, scrive in output un elemento *chiave-valore* con una chiave strutturata (A-B,false) e come valore 0, se invece l'input è una coppia *chiave-valore* (A-B)- $K$  generata dal Job 1, scrive in output un elemento con chiave (A,B,true) e valore  $K$ .

**Job2 GroupComparator** Il secondo GroupComparator raggruppa tutti gli elementi generati dal Mapper mettendo insieme quelli che hanno la stessa prima parte di chiave (A-B) indipendentemente dal valore booleano che viene utilizzato solo per l'ordinamento degli elementi da sottoporre al Reducer.

**Job2 Reducer** Il secondo Reducer per ogni iterazione analizza una chiave (A-B) che raggruppa i valori sia di (A-B,false) che di (A-B,true). Se come primo elemento trova un (A-B,false) allora tutti i valori successivi positivi sono nodi che chiudono un triangolo. Se invece il primo nodo analizzato ha chiave (A,B,true) allora l'iterazione del gruppo (A,B) può essere completamente saltata.

## 4.2 Algoritmo 5 Jobs

L'algoritmo a 5 Jobs prevede prima una fase in cui i primi 2 Jobs calcolano il totale dei degli archi e il grado di ogni singolo nodo, queste informazioni poi verranno usate nei successivi 3 Jobs, i quali andranno ad eseguire il calcolo dei triangoli.

**Job 1** Il primo Job è molto semplice e per ogni arco emette un valore 1. Il Reducer conta tutti questi valori, ne fa la somma che poi scrive come output, questo valore rappresenta la quantità di nodi presenti nel grafo.

**Job 2** Il Mapper del secondo Job per ogni relazione A-B emette 2 valori A-1 e B-1, uno per ogni nodo che compone A-B.

Le funzioni di Grouping e il Partitioner mi garantiscono che tutti i valori di uno stesso nodo vengono accumulati nello stesso Reducer e nello stesso ciclo iterativo. Completato il ciclo scrive in output, per ogni nodo, la somma di tutti i valori emessi dal Mapper. Questo valore rappresenta il grado del nodo.

**Job 3** Nella seconda fase viene eseguito l'algoritmo vero e proprio.

Obiettivo del Job 3 è di trovare tutti i triangoli che sono composti da nodi Heavy Hitter, ovvero con un grado maggiore di  $\sqrt{m}$  dove  $m$  è il numero degli archi del grafo.

Il Mapper utilizzando il lavoro dei Jobs precedenti costruisce in memoria un indice i nodi *Heavy Hitter* e esclude dall' analisi tutti i nodi non nell' indice.

Definita una relazione  $\prec$  di ordinamento dei nodi in base al grado, si può assumere che ogni trinagolo sia formato da 3 nodi  $(x,y,z)$  i quali sono legati dalle 3 relazioni typeA  $(x,y)$  con  $x \prec y$ , typeB  $(x,z)$  con  $x \prec z$  e typeC  $(y,z)$  con  $y \prec z$ .

Ogni relazione presente nel grafo può essere una delle 3 che compongono il triangolo, quindi se  $R=(u,v)$  una relazione che appartiene ad un triangolo allora sarà di *tipo A* e quindi il triangolo sarebbe  $(u,v,?)$  oppure *tipo B* con  $(u,?,v)$  o di *tipo C* con  $(?,u,v)$ .

Il task di Map per sfruttare il parallelismo dei Reducer divide l'input in tante parti quante le possibili combinazioni in cui ogni relazione può essere utilizzata per completare un triangolo.

Per implementare questa suddivisione, viene definita una funzione di hash  $H$  e un parametro *bucket* che indica quanti sono i possibili valori in output della funzione. In base a questo parametro ogni relazione di input  $R(x,y)$  viene distribuita in  $3bk$  Reducers utilizzando la funzione  $H$  e seguendo questo schema:  $(h(x),h(y),1 \leq i \leq b)$  ipotizzando che  $R$  sia di *tipo A*,  $(h(x),1 \leq i \leq b,h(y))$  se  $R$  fosse di *tipo B* e  $(1 \leq i \leq b,h(x),h(y))$  se  $R$  fosse di *tipo C*. Ogni relazione  $R$  viene inclusa in tutti i possibili Reducer in cui potrebbe essere utilizzata per il completamento di un triangolo. Il Reducer, dopo una opportuna ridefinizione delle classi di Gruouping e Ordinamento, scorrere tutti gli elementi in input. L'ordinamento è definito in modo da analizzare, per ogni nodo  $K$ , prima le possili relazioni di *tipo A*, per ognuna di esse viene inserito il nodo destinazione  $A$  in una Lista *listK*. Successivamente vengono analizzate le relazioni di  $K$  di *tipo B* con  $B$  come nodo di destinazione, per ognuna di esse viene preso creata una Map *mapPairK* in cui la chiave è la copia formata da ogni elemento di *listK* e  $B$  e il valore è il nodo  $K$ . L'analisi successiva della relazione di *tipo C* fra  $A'$  e  $B'$ , se nella *mapK* esiste una chiave formata dalla coppia  $A',B'$  con valore  $e$  allora nel grafo esisterà un triangolo  $K,A',B'$ . L'ordinamento con cui viene eseguita questa analisi consente di ottimizzare lo spazio di memoria utilizzato e ci garantisce che ogni triangolo viene rilevato solo una volta.

**Job 4** Il Job 4 è molto simile al Job3 l'unica differenze è che in questo caso si vogliono escludere tutti i triangoli di tipo *Heavy Hitter*.

Data una relazione  $x,y$  se  $x$  è *Heavy Hitter*  $x,y$  potrebbe appartenere ad un triangolo non *Heavy Hitter* se solo se  $x,y$  è di *tipo C*. Infatti per la relazione di ordinamento rel definita precedentemente dato il triangolo  $T1(x,y,?)$  in cui  $x,y$  è di *tipo A*, qualsiasi  $?$  e  $y$  sarebbero maggiori di  $x$  e quindi  $T1$  sarebbe *Heavy Hitter*, la stessa cosa se  $x,y$  fosse di *tipo B*.

Grazie a questa caratteristica il Mapper esclude le relazioni di *tipo A* e *tipo B* in cui  $X,Y$  ha *X Heavy Hitter*. Il Reducer, a differenza da quello utilizzato in precedenza non usa la Map per salvare i risultati intermedi ma li emette come output insieme a tutte le relazioni *tipo C*.

**Job 5** Completa l'algoritmo controllando se per ogni elemento calcolato dal Job4 esiste un arco di *tipo C* che chiuderebbe il triangolo.

## 5 Test

---

### 5.1 Prima iterazione

Durante la prima iterazione implementativa ho realizzato l'algoritmo a 2 Job che mi è risultato semplice dal punto di vista implementativo e mi ha aiutato a familiarizzare con il paradigma Map reduce e il framework Hadoop che non avevo mai utilizzato prima. Purtroppo l'algoritmo dal punto di vista computazionale non si completava nemmeno con grafi anche di medie dimensioni.

### 5.2 Seconda iterazione

Nella seconda iterazione ho implementato l'algoritmo presente nel libro *Mining of Massive Datasets*. Inizialmente mi sono focalizzato sull'implementazione della suddivisione nei vari Bucket ed in un secondo momento ho completato il lavoro scrivendo l'algoritmo per il calcolo dei triangoli. Purtroppo nonostante i miglioramenti nella velocità dell'algoritmo, anche questa implementazione aveva dei problemi. Dovendo mantenere Liste e Hashtable in memoria con grafi di grandi dimensioni il processo si interrompeva.

### 5.3 Terza iterazione

Nella terza iterazione ho analizzato le strutture in memoria più pesanti e ho deciso di introdurre un Job intermedio. Con questa implementazione sono riuscito ad elaborare anche grafi di grandi dimensioni.

Durante l'analisi di alcuni risultati sul cluster ho riscontrato un bug molto evidente. Il bug era causato dalle dimensioni del file contenente il grado dei nodi, questo per grafi molto grandi veniva suddiviso su più mapper e quindi in alcuni casi risultava parziale.

### 5.4 Conclusione

Nell'ultima iterazione ho corretto il bug introducendo delle DistributedCache che mi consentivano di vincolare il caricamento dell'intero dei gradi dei nodi su ogni singolo Mapper. Inoltre ho aggiunto delle classi di Helper per agevolarmi nel testing dei singoli Job.

## 6 Risultati

---

### 6.1 Social circles: Facebook<sup>3</sup>

Riferimenti:

Job1  
Job2  
Job3  
Job4

---

<sup>3</sup><http://snap.stanford.edu/data/egonets-Facebook.html>

**Tabella 6.1:** *Facebook*

Nodi	4039
Archi	88234
Triangoli	1612010

**Tabella 6.2:** *Amazon*

Nodi	334863
Archi	925872
Triangoli	667129

Job5

## 6.2 Amazon product co-purchasing network and ground-truth communities<sup>4</sup>

Riferimenti:

Job1

Job2

Job3

Job4

Job5

## 6.3 Youtube social network<sup>5</sup>

Riferimenti:

Job1

Job2

Job3

Job4

Job5

---

<sup>4</sup><http://snap.stanford.edu/data/com-Amazon.html>

<sup>5</sup><http://snap.stanford.edu/data/com-Youtube.html>

**Tabella 6.3:** *Youtube*

Nodi	1134890
Archi	2987624
Triangoli	3056386

## 7 Sviluppi futuri

---

Mi sarebbe piaciuto completare il progetto generando come artefatto del processo un'immagine Docker pronta all'uso <sup>6</sup>. Sarebbe interessante provare a fare un deploy per testare questo container in un ambiente cloud che supporta Docker.

Un secondo aspetto che mi sarebbe piaciuto approfondire è l'uso di un framework di Unit testing per Hadoop. L'unico che ho provato velocemente è Mrunit, tuttavia il progetto sembra non essere più mantenuto.

---

<sup>6</sup>Una prima versione è disponibile su <https://hub.docker.com/r/nickxbs/labbigdata>