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
https://github.com/nickxbs/LabBigData

21 giugno 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 Haddop.

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[1] 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 considerazioni grafi non diretti.

3 Strumenti e Framework

Infrastruttura Per semplicità e rapidità di configurazione ho deciso di utilizzare Claudera 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 e come IDE di sviluppo ho utilizzato IntelliJ.

[1] Dati 3 nodi (A,B,C) in un grafo, se un nodo A è collegato con un arco sia con B che con C, nel grafo viene a formarsi un triangolo se esiste anche l'arco che lega B con C.

Versioning Come sistema di versioning ho utilizzato Git e GitHub come spazio di hosting dei sorgenti. [2]

4 Implementazioni

4.1 Algoritmo 2 Jobs

L'algoritmo a 2 Jobs prevede un 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.[3]

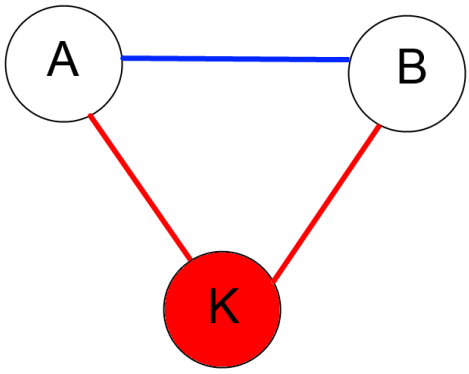

Figura 4.1: *Il Job 1 elabora le relazioni rosse in cui K è il nodo minore del triangolo. Il Job2 cerca la relazione blu A B.*

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

[2] https://github.com/nickxbs/LabBigData
[3] 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

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 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)-V generata dal Job 1, scrive in output un elemento con chiave (A,B,true) e valore V.

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.
Le funzioni di Grouping e il Partitioner mi garantiscono che tutti i valori di uno stesso nodo vengano 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 di nodi *Heavy Hitter* ed 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 triangolo sia formato da 3 nodi (X,Y,Z) i quali sono legati dalle 3 relazioni *tipo A* (X,Y) con X$\prec$Y, *tipo B* (X,Z) con X$\prec$Z e *tipo C* (Y,Z) con Y$\prec$Z.
Ogni relazione presente nel grafo può essere una delle 3 che compongono il triangolo, quindi se (U,V) è una relazione che appartiene ad un triangolo, potrebbe essere 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 Mapper, 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 b che indica quanti sono i possibili valori in output della funzione. In base a questo parametro ogni relazione di input (X,Y) viene distribuita in 3bk Reducers utilizzando la funzione H e seguendo questo schema: (h(X),h(Y),1≤i≤b) ipotizzando che (X,Y) sia di *tipo A*; (h(X),1≤i≤b,h(Y)) se (X,Y) fosse di *tipo B* e (1≤i≤b,h(X),h(Y)) se (X,Y) fosse di *tipo C*. Ogni relazione 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 Grouping e Ordinamento, scorre tutti gli elementi in input. L'ordinamento è definito in modo che, per ogni nodo K, vengano prima analizzate le possibili relazioni di *tipo A* e per ognuna di esse venga inserito il nodo destinazione A in una Lista *listK*. Completate le relazioni di *tipo A* vengono analizzate le relazioni di K di *tipo B* con B come nodo di destinazione, per ognuna di esse viene creata una *mapPair* in cui la chiave è la coppia formata da ogni elemento di *listK* e B, mentre il valore è il nodo K. Come ultimo passo vengono analizzate le relazioni di *tipo C* fra K e C, se nella *mapPair* esiste una chiave formata dalla coppia K,C con valore V e allora nel grafo esisterà un triangolo V,K,C.
L'ordinamento con cui viene eseguita questa analisi consente di ottimizzare lo spazio di memoria utilizzato e ci garantisce che ogni triangolo venga rilevato solo una volta.

Job 4 Il Job 4 è molto simile al Job3, l'unica differenza è che in questo caso si vogliono escludere tutti i triangoli di tipo *Heavy Hitter* già calcolati in precedenza.
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 $\prec$ definita precedentemente dato il triangolo T(X,Y,?) in cui X,Y è di *tipo A*, qualsiasi ? e Y sarebbero maggiori di X e quindi T 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 di 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 Sviluppo

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. 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 file dei gradi dei nodi su ogni singolo Mapper. Inoltre ho aggiunto delle classi di Helper per agevolarmi nel testing e nel debugging dei singoli Job.

Tabella 6.1: *Facebook*

Nodi	4039
Archi	88234
Triangoli	1612010

Tabella 6.2: *Amazon*

Nodi	334863
Archi	925872
Triangoli	667129

6 Risultati

6.1 Social circles: Facebook[4]

Riferimenti:
Job1
Job2
Job3
Job4
Job5

6.2 Amazon product co-purchasing network and ground-truth communities[5]

Riferimenti:
Job1
Job2
Job3
Job4
Job5

6.3 Youtube social network[6]

Riferimenti:
Job1
Job2
Job3
Job4

[4] http://snap.stanford.edu/data/egonets-Facebook.html
[5] http://snap.stanford.edu/data/com-Amazon.html
[6] http://snap.stanford.edu/data/com-Youtube.html

Tabella 6.3: *Amazon job1*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	3
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	6,497,550	6,481,122	12,978,672
HDFS BYTES READ	12,585,884	0	12,585,884
FILE BYTES WRITTEN	12,962,378	6,481,122	19,443,500
HDFS BYTES WRITTEN	0	15	15
Reduce input groups	0	1	1
Combine output records	0	0	0
Map input records	925,876	0	925,876
Reduce shuffle bytes	0	6	6
Reduce output records	0	1	1
Spilled Records	1,851,744	925,872	2,777,616
Map output bytes	4,629,360	0	4,629,360
Map output records	925,872	0	925,872
Combine input records	0	0	0
Reduce input records	0	925,872	925,872

Tabella 6.4: *Amazon job2*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	3
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	12,986,534	12,962,226	25,948,760
HDFS BYTES READ	12,585,884	0	12,585,884
FILE BYTES WRITTEN	25,924,658	12,962,226	38,886,884
HDFS BYTES WRITTEN	0	5,996,924	5,996,924
Reduce input groups	0	334,863	334,863
Combine output records	0	0	0
Map input records	925,876	0	925,876
Reduce shuffle bytes	0	8,655,981	8,655,981
Reduce output records	0	334,863	334,863
Spilled Records	3,703,488	1,851,744	5,555,232
Map output bytes	9,258,720	0	9,258,720
Map output records	1,851,744	0	1,851,744
Combine input records	0	0	0
Reduce input records	0	1,851,744	1,851,744

Tabella 6.5: *Amazon job3*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	27
Rack-local map tasks	0	0	1
Launched map tasks	0	0	2
Data-local map tasks	0	0	1
FILE BYTES READ	0	162	162
HDFS BYTES READ	25,171,768	0	25,171,768
FILE BYTES WRITTEN	1,636	162	1,798
Reduce input groups	0	0	0
Combine output records	0	0	0
Map input records	1,851,752	0	1,851,752
Reduce shuffle bytes	0	294 294	
Reduce output records	0	0	0
Spilled Records	0	0	0
Map output bytes	0	0	0
Map output records	0	0	0
Combine input records	0	0	0
Reduce input records	0	0	0

Tabella 6.6: *Amazon job4*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	32
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	261,215,370	161,101,890	422,317,260
HDFS BYTES READ	12,585,884	0	12,585,884
FILE BYTES WRITTEN	421,971,124	161,101,890	583,073,014
HDFS BYTES WRITTEN	0	64,947,486	64,947,486
Reduce input groups	0	3,106,206	3,106,206
Combine output records	0	0	0
Map input records	925,876	0	925,876
Reduce shuffle bytes	0	120,704,093	120,704,093
Reduce output records	0	3,077,513	3,077,513
Spilled Records	14,550,574	5,555,232	20,105,806
Map output bytes	149,991,264	0	149,991,264
Map output records	5,555,232	0	5,555,232
Combine input records	0	0	0
Reduce input records	0	5,555,232	5,555,232

Tabella 6.7: *Amazon job5*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	31
Launched map tasks	0	0	28
Data-local map tasks	0	0	8
FILE BYTES READ	67,962,550	67,705,448	135,667,998
HDFS BYTES READ	64,947,486	0	64,947,486
FILE BYTES WRITTEN	135,435,250	67,705,448	203,140,698
HDFS BYTES WRITTEN	0	14,934,482	14,934,482
Reduce input groups	0	2,297,998	2,297,998
Combine output records	0	0	0
Map input records	3,077,513	0	3,077,513
Reduce shuffle bytes	0	67,709,642	67,709,642
Reduce output records	0	667,129	667,129
Spilled Records	6,155,026	3,077,513	9,232,539
Map output bytes	61,550,260	0	61,550,260
Map output records	3,077,513	0	3,077,513
Combine input records	0	0	0
Reduce input records	0	3,077,513	3,077,513

Tabella 6.8: *Youtube*

Nodi	1134890
Archi	2987624
Triangoli	3056386

Tabella 6.9: *Youtube Job1*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	2
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	26,467,608	20,913,380	47,380,988
HDFS BYTES READ	38,720,822	0	38,720,822
FILE BYTES WRITTEN	47,331,964	20,913,380	68,245,344
HDFS BYTES WRITTEN	0	16	16
Reduce input groups	0	1	1
Combine output records	0	0	0
Map input records	2,987,628	0	2,987,628
Reduce shuffle bytes	0	6	6
Reduce output records	0	1	1
Spilled Records	6,761,678	2,987,624	9,749,302
Map output bytes	14,938,120	0	14,938,120
Map output records	2,987,624	0	2,987,624
Combine input records	0	0	0
Reduce input records	0	2,987,624	2,987,624

Tabella 6.10: *Youtube jonb2*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	3
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	68,766,330	41,826,748	110,593,078
HDFS BYTES READ	38,720,822	0	38,720,822
FILE BYTES WRITTEN	110,546,965	41,826,748	152,373,713
HDFS BYTES WRITTEN	0	20,572,080	20,572,080
Reduce input groups	0	1,134,890	1,134,890
Combine output records	0	0	0
Map input records	2,987,628	0	2,987,628
Reduce shuffle bytes	0	41,826,748	41,826,748
Reduce output records	0	1,134,890	1,134,890
Spilled Records	15,792,371	5,975,248	21,767,619
Map output bytes	29,876,240	0	29,876,240
Map output records	5,975,248	0	5,975,248
Combine input records	0	0	0
Reduce input records	0	5,975,248	5,975,248

Tabella 6.11: *Youtube jonb3*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	8
Rack-local map tasks	0	0	1
Launched map tasks	0	0	2
Data-local map tasks	0	0	1
FILE BYTES READ	0	286,452	286,452
HDFS BYTES READ	77,441,644	0	77,441,644
FILE BYTES WRITTEN	286,900	286,452	573,352
HDFS BYTES WRITTEN	0	38,598	38,598
Reduce input groups	0	1,104	1,104
Combine output records	0	0	0
Map input records	5,975,256	0	5,975,256
Reduce shuffle bytes	0	246,584	246,584
Reduce output records	0	2,396	2,396
Spilled Records	9,876	9,876	19,752
Map output bytes	266,652	0	266,652
Map output records	9,876	0	9,876
Combine input records	0	0	0
Reduce input records	0	9,876	9,876

Tabella 6.12: *Youtube jonb4*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	8
Launched map tasks	0	0	1
Data-local map tasks	0	0	1
FILE BYTES READ	1,015,288,761	519,751,156	1,535,039,917
HDFS BYTES READ	38,720,822	0	38,720,822
FILE BYTES WRITTEN	1,534,045,113	519,751,156	2,053,796,269
HDFS BYTES WRITTEN	0	444,873,166	444,873,166
Reduce input groups	0	8,638,874	8,638,874
Combine output records	0	0	0
Map input records	2,987,628	0	2,987,628
Reduce shuffle bytes	0	453,774,961	453,774,961
Reduce output records	0	21,716,418	21,716,418
Spilled Records	52,897,973	17,922,452	70,820,425
Map output bytes	483,906,204	0	483,906,204
Map output records	17,922,452	0	17,922,452
Combine input records	0	0	0
Reduce input records	0	17,922,452	17,922,452

Tabella 6.13: *Youtube job5*

Counter	Map	Reduce	Total
Launched reduce tasks	0	0	10
Launched map tasks	0	0	8
Data-local map tasks	0	0	8
FILE BYTES READ	548,195,058	477,761,292	1,025,956,350
HDFS BYTES READ	444,873,166	0	444,873,166
FILE BYTES WRITTEN	1,024,734,424	477,761,292	1,502,495,716
HDFS BYTES WRITTEN	0	59,461,190	59,461,190
Reduce input groups	0	12,602,543	12,602,543
Combine output records	0	0	1
Map input records	21,716,418	0	21,716,418
Reduce shuffle bytes	0	477,761,568	477,761,568
Reduce output records	0	3,053,990	3,053,990
Spilled Records	46,578,558	21,716,418	68,294,976
Map output bytes	434,328,360	0	434,328,360
Map output records	21,716,418	0	21,716,418
Combine input records	0	0	0
Reduce input records	0	21,716,418	21,716,418

Job5

7 Sviluppi futuri

Mi sarebbe piaciuto completare il progetto generando come artefatto del processo un immagine Docker pronta all'uso [7]. 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.

[7] Una prima versione è disponibile su https://hub.docker.com/r/nickxbs/labbigdata