# Implementazione di Finding Triangles con Hadoop MapReduce

## Sistemi di elaborazione di grandi quantità di dati 2013

Nicola Febbrari
Università degli Studi di Verona
Facoltà MM.FF.NN.
nicola.febbrari@studenti.univr.it

13 gennaio 2014

## 1 Introduzione

Lo scopo del progetto è quello di implementare un algoritmo per calcolare il numero di triangoli presenti in un grafo, 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 esponenziale del numero di utenti che interagistcono con questi sistemi ha avuto come diretta conseguenza un incremento della quantità di dati che devono essere registrati, gestiti ed ovviamente elaborati. Il businnes ha subito capito quanto fossero imprtanti tutte queste informazioni e per questo motivo negli ultimi c'è stato un grosso hype su come sia possibile recuperare, elaborare e dedurre nuove informazioni a partire da una base di informazioni enorme.
Un social network può essere rappresentato matematicamente da un grafo e una caratteristica molto interessante di questo grafo e' il numero di triangoli[1] contenuti in sesso. Il numero di questi triangoli rapportato al totale dei nodi che costituiscono il grafo è un indice di quanto sia *SOCIAL* il social network rappresentato dal grafo analizzato.
La conoscenza di questa informazione potrebbe essere utilizzata su larga scala per valutare particolari aspetti sociologigi o più semplicemente epr scopi commerciali.

[1] 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.

# 3 Strumenti e Framework

**Sistema** Apache Hadoop è un framework opensource utilizzato per l' elaborazione di grandi quantità di dati. Si basa su MapReduce, un paradigma di programmazione parallela con il quale è possibile realizzare algoritmi applicabili a sitemi distributi e con alto grado di scalabilità.

**Sviluppo** L'implementazione dell 'algoritmo di Finding Triangles è stata realizzata scrivedo un programma Java che utilizza ed estende le Hadoop API di Base.

**Testing** Come suite di testing è stato utilizzato MRUnit una libreria Java che consente di fare Unit test sui jobs di MapReduce.

**UI** Come tool di supporto è stato utilizzato Hue, un' inerfaccia web-based per la gestione ed il monitoraggio del file sistem di Hadoop (HDFS) e dei jobs MapReduce.

**Versioning** Come sistema di versioning è stato utilizzato GitHub, un servizio web che utilizza la piattaforma GIT.

**Sorgente dati** Come sistema di versioning è stato utilizzato GitHub, un servizio web che utilizza la piattaforma GIT.

# 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.
Esempio date le relazioni A - K, K - B e A - B (in cui A¡K¡B ) il Job 1 elabora le relazioni in cui K è nodo in un caso maggiore o minore unendo queste relazioni trova al coppia A - B che verrà cercate nel Job2. Se questa relazione è presente allora esiste sicuramente il atriangolo A K B

**Job1 Mapper** Nella parte di mapper del primo Job come prima cosa escludo tutti gli archi che portano nello stesso nodo e poi definisco le relazioni solamente come da nodo minore a nodo maggiore in modo da evitare duplicazioni. La definizione di triangolo in un grafo non prende in considerazione la direzione delle relazioni fra i nodi e nel mio caso i nodi sono rappresentati da dei valori Long, quindi la relazione di ordinamento è immediata.
In questa fase per ogni relazione mando in output 2 oggetti ¡¡VALUE1,BOOL¿,VALUE2¿. Come KEY (¡VALUE1,BOOL¿) del VALUE (VALUE2) che passo al reducer uso un dato strutturato contenente il nodo e un valore booleano che identifica se il nodo VALUE è

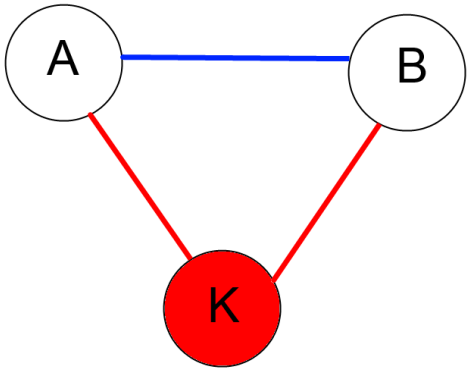


**Figura 4.1:** *Il Job 1 elabora le relazioni rosse in cui K in un caso è nodo maggiore e nell altro caso minore. Il Job2 cerca la relazione blu A B.*

un nodo minore o maggiore di quello della KEY secondo la relazione di ordinamento definita.

Listing 1: Implementazione del Mapper1

```
@Override
  public void map(LongWritable key, Text value, Context
     context)
      throws IOException, InterruptedException {
    String line = value.toString();
    String[] sp = line.split("\s+");// splits on TAB
    Long lp0=Long.parseLong(sp[0]);
    Long lp1=Long.parseLong(sp[1]);
    if (lp0!= lp1) {  //exclude self relation
      if (lp0 < lp1) {
        textP.set(lp1, false);
        text.set(lp0);
        context.write(textP, text);// "0" link_from
        textP.set(lp0, true);
        text.set(lp1);
        context.write(textP, text);// "1" link_to
      } else {
        textP.set(lp0, false);
        text.set(lp1);
        context.write(textP, text);// "0" link_from

        textP.set(lp1, true);
        text.set(lp0);
```

```
            context.write(textP, text);// "1" link_to
          }
        }
    }
```

**Job1 Reducer** Per suddividere correttamente tutti gli elementi prodotti dal Mapper sono stati ridefiniti il SortComparator e il GroupingComparator. Il nuovo GroupingComparator ragruppa le chiavi valutando solamente il valore del nodo chiave K e garantendo che tutte le relazioni che contengono K come nodo maggiore o come nodo minore vengano elaborate dallo stesso reducer. Il SortComparator ridefinito invece ordina gli elementi in base al valore del nodo chiave K e in caso di uguale nodo chiave mette prima gli elementi con valore booleano false nella chiave.
Il reducer garzie all'ordinamento esamina prima tutte le relazioni in cui il nodo chiave è nodo maggiore, queste coppie (nodo valore - nodo chiave) salvati in una lista L. Una volta che tutte le relazioni in cui il nodo chiave è nodo maggiore sono finite inizieranno le relazioni in cui il nodo chiave è nodo minore. Per ognuna di queste nuove relazioni (es. K-B) il reducer cicla la lista L mandando in output per ogni elemento (A-K) in L una relazione (A-B, K). Infatti la presenza della relazione A-B nel grafo chiuderebbe il triangolo A-K-B

Il redoucer, grazie alla redifinizione del GroupinComparator1 (ragruppa le chiavi valutando solamente il valore del nodo della chiave) e del Comparator1 (ordina le chiavi mettendo prima quelle con valore false ) tutti gli output del task di Map. Prima salva in una lista l'elenco dei nodi padre del nodo chiave, una volta che tutti i nodi sono stati ciclati, per ogni nodo figlio, scrive in output la terna di valori con la coppia nodo padre - nodo figlio che completerebbe il triangolo.

Listing 2: Implementazione del Reducer1

```
@Override
protected void reduce(LongBit key, Iterable<LongWritable>
   values, Context context)
    throws IOException, InterruptedException {
  partialJoin.clear();
  LongWritable k = key.getFirst();

  for (LongWritable valText : values) {
    Long val = valText.get();
    if (!key.getSecond().get())// link from
    {
      if (!partialJoin.contains(val))
        partialJoin.add(val);
    } else // link to
    {
```

```
            WriteContext(k, context, val);
          }
        }
      }
```

**Job2 Mapper** Nella secondo Job l'output del primo Job viene unito nuovamente alla sorgente dati iniziale. Il secondo mapper in questo scorre l'unione dei 2 input, nel caso la relazione è una relazione presente nel grafo iniziale scrive in output un elemento ((A,B,false),0) se invece è generata dal Job 1 scrive in output ((A,B,true),K) dove A,B sono nodi della relazione e K è nodo intermedio fra A e B calcolato da Job1.

**Job2 Reducer** Il secondo reducer ridefinendo in modo analogo al Reducer1 una seconda versione del SortComparator e del GroupingComparator, scorre l'input prima cerca le relazioni AB che chiuderebbero un triangolo e poi per ogni relazione generata dal Job1 con output AB manda in output AKB.

## 4.2 Algoritmo single Job

L'algoritmo a Job singolo è molto più complesso del precedente. L'idea è quella di costruire una suddivisione delle relazioni in input nei vari reducer definendo delle chiavi strutturate e abbinate ai vari task di Reduce. Questa suddivisione viene realizzata definendo una funzione Hash sul valore del nodo stesso. Ogni trinagolo è formato da 3 nodi (x,y,z) i quali sono legati dalle 3 relazioni A=(x,y), B=(x,z) e C=(y,z). In base a questo ogni relazione presente nel grafo può essre una delle 3 che compongono il triangolo, quindi dato R=(u,v), R può essere una relazione di tipo A e quindi il triangolo sarebbe (u,v,?) oppure di tipo B con (u,?,v) o C con (?,u,v).

**SingleJob Mapper** Il task di Map divide le relazioni in input in tante parti quante le possibili casistiche in cui la relazione può essere utilizzata per completare un triangolo. Per implementare questa suddivisione, come prima cosa, viene definito un paramentro bk che indica quanti sono i possibili valori in output della funzione hash. In base a questo parametro ogni relazione di input R(x,y) viene distribuita nei rispettivi 3bk reducers utilizzando la funzione di hash h e seguindo questo schema: (h(x),h(y),1¡=i¡=b) impotizzando che R sia di tipo A , (h(x),1¡=i¡=b,h(y)) se R fosse di tipo B e (1¡=i¡=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.

Listing 3: Implementazione del Mapper single Job

```
@Override
public void map(LongWritable key, Text value, Context
    context)
     throws IOException, InterruptedException {
```

```
    Configuration conf = context.getConfiguration();
    this.buckets = conf.getInt("b",3);

    String line = value.toString();
    line = line.replaceAll("^\\s+", "");
    String[] sp = line.split("\\s+");// splits on TAB
    Long lp0 = Long.parseLong(sp[0]);
    Long lp1 = Long.parseLong(sp[1]);
    if (lp0 != lp1) {
      if (lp0 < lp1) {
        SetContext(context, lp0, lp1);
      } else {
        SetContext(context, lp1, lp0);
      }
    }
  }

  private void SetContext(Context context, Long lp0, Long
     lp1)
      throws IOException, InterruptedException {
    to.set(lp1);
    for (long j = 0; j < buckets; j++) {
      context.write(new LongLongLongLong("A",lp0 % buckets,
         lp1 % buckets, j, lp0), to);// "1"
      context.write(new LongLongLongLong("B",lp0 % buckets,
         j, lp1 % buckets, lp0), to);// "1"
      context.write(new LongLongLongLong("C",j, lp0 %
         buckets, lp1 % buckets, lp0), to);// "1"
    }
  }
```

**SingleJob Reducer** Il reducer, doppo una opportuna ridefinizione delle classi di Gruouping e Ordinamento, si limita semplicemente a scorrere tutti gli elementi in input. L'ordinamente e definito in modo da analizzare, per ogni nodo, prima le possili relazioni A e B. A questo punto viene utilizzata HashTable in memoria che memorizza il possibile trinagolo composto dalle relazioni A e B e per ogni trinagolo si definisce una chiave formata dai nodi della relazione mancante per il completamento del triangolo (relazione di tipo C). Quando il reducer trova una relazione di tipo C in input controlla se esiste una chiave con nella Hashtable con quella relazione e se ne trova una elimina l'elemento dalla HashTable e mette in output il triangolo.

Listing 4: Implementazione del Reducer single Job

```
@Override
protected void reduce(LongLongLongLong key,
    Iterable<LongWritable> values,
     Context context) throws IOException,
        InterruptedException {
  //partialJoin.clear();
  List<Long> tmpList = new LinkedList<Long>();

  for (LongWritable valText : values) {
    Long from = key.getfourth().get();
    Long to = valText.get();
    Pair<Long, Long> p = CreatePair(to, from);
    if (partialJoin.containsKey(p)  &&
       key.getRel().toString().equals("C")) {
      WriteContext(partialJoin.get(p), context);
      partialJoin.remove(p);
    }
    for (Long tmpTo : tmpList) {
      Pair<Long, Long> l = CreatePair(to, tmpTo);
      if (!to.equals(tmpTo) &&
         key.getRel().toString().equals("B")) {
        Triplet<Long, Long, Long> t  = new Triplet<Long,
           Long, Long>(from, l.getValue0(),l.getValue1());
        if(partialJoin.containsKey(l)){
          if(!partialJoin.get(l).contains(t))
            partialJoin.get(l).add(t);
        }
        else{
          List<Triplet<Long, Long, Long>> lt= new
             LinkedList<Triplet<Long, Long, Long>>();
          lt.add(new Triplet<Long, Long, Long>(from,
             l.getValue0(),l.getValue1()));
          partialJoin.put(l,lt);
        }

      }
    }
    if (!tmpList.contains(to)&&
       key.getRel().toString().equals("A")) {
      tmpList.add(to);
    }
  }
}
```

**Ottimizzazioni**

# 5 Complessità

**Algoritmo 2 Jobs**

**Algoritmo Single Jobs**

# 6 Osservazioni