UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

CORSO DI LAUREA IN
INGEGNERIA INFORMATICA

A.A. 2010-2011

Tesi di Laurea

At First Look in Map Reduce

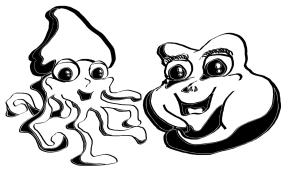
Ricerca in streaming di componenti connesse e biconnesse di grafi tramite cluster

Emanuele Paracone

Relatori: Prof. Luigi Laura
Prof. Giuseppe Italiano

19 Luglio 2011

Indice



Compendio

In questo lavoro di tesi analizziamo una soluzione al problema della ricerca esaustiva, in un grafo, delle componenti connesse, delle componenti biconnesse, degli *articulation points* e dei *bridges*, ottenuta elaborando uno stream in input in modo parallelo secondo il paradigma MapReduce.

Descriviamo l'algoritmo At First Look, il paradigma MapReduce e la sua implementazione nel framework Apache Hadoop, analizzando poi i programmi *Hitsura*, la nostra implementazione di At First Look, e *HitSoop*, l'oggetto della nostra tesi, che implementa una soluzione al problema secondo il paradigma MapReduce.

Introduzione

L'oggetto di questo lavoro di tesi è la nostra soluzione al problema della ricerca esaustiva delle componenti connesse, delle componenti biconnesse, degli *articulation points* e dei *bridges* di un grafo in input descritto da uno *stream* di dimensioni molto maggiori della disponibilità di memoria media di una singola macchina. *Autonomous Systems*, generiche reti di calcolatori, e molti altri sono gli scenari nei quali è interessante estrarre dette informazioni dai grafi che li modellano. Abbiamo implementato perciò il programma HitSoop, che raggiunge lo scopo distribuendo il lavoro complessivo su un cluster di elaboratori secondo il paradigma MapReduce. HitSoop è un programma scritto in Java per Apache Hadoop, il framework della Apache Foundation che implementa il paradigma MapReduce, eseguito su un cluster Amazon Web Services EC2.

Nell'implementazione di HitSoop ci siamo ispirati all'algoritmo At First Look, che affronta lo stesso problema secondo il modello del *classical streaming*, ed abbiamo costruito un nuovo algoritmo in grado di processare lo stream in input in modo non sequenziale, attraverso l'aggregazione delle elaborazioni intermedie effettuate su porzioni distinte dell'input complessivo.

L'interesse per l'implementazione MapReduce proviene dal sensibile vataggio in termini prestazionali che è possibile ottenere parallelizzando quanto più possibile il lavoro complessivo. Il calcolo parallelo ed il cloud, infatti, sono sempre più al centro delle attenzioni di aziende di IT di qualunque dimensione, in quanto permettono un deciso abbattimento dei costi.

Capitolo 1

At First Look

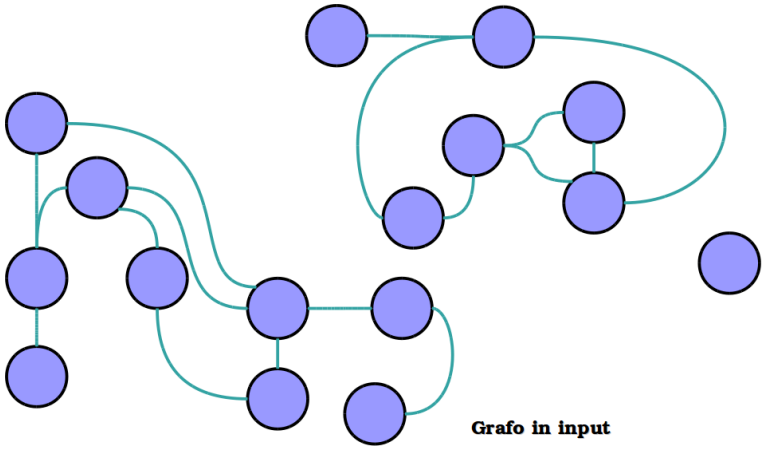
1.1 Cos'è "At First Look"

At First Look (AFL) è un algoritmo che analizza la topologia di un grafo generico non diretto e ricerca esaustivamente all'interno di questo:

- le componenti connesse (**CC**);
- le componenti bi-connesse(**BCC**);
- gli articulation points;
- i bridges.

Queste informazioni sono utili in scenari reali per l'analisi delle proprietà strutturali di grafi che modellano reti esistenti dimensionalmente complesse, come quelle degli indirizzi di un *Autonomous System* (AS) o delle relazioni astratte tipiche dei casi di studio di vari contesti (ricerca scentifica, Social Engineering, ecc.), e la loro ricerca off-line è un problema che risale agli anni settanta.

At First Look ottiene queste informazioni on-line, ovvero processando sequenzialmente il data-streaming che descrive gli archi del grafo che si vuole analizzare, attenendosi ai vincoli del *classical streaming* sul tempo di processamento delle informazioni (**PIPT** - Per Item Processing Time) e sullo spazio di memoria occupato.



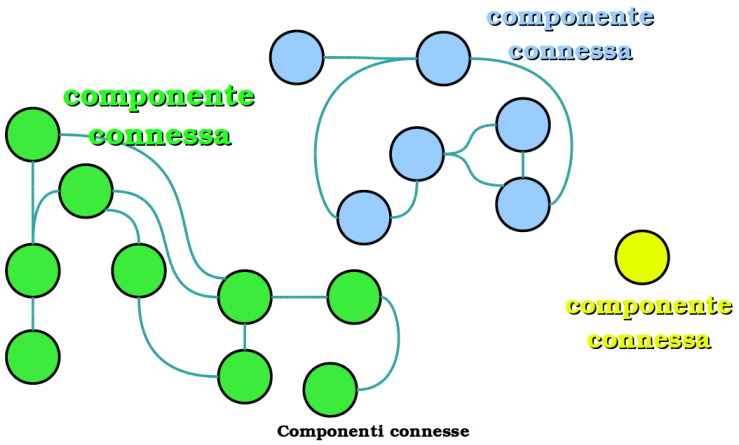
In questo capitolo tratteremo il modello del classical streaming e dei vincoli che ne derivano sulla disponibilità di memoria e sul *PIPT* rispetto alla dimensione dello stream in input, la struttura e le caratteristiche del grafo a foresta *Navigational Sketch* in cui vengono progressivamente modellate le informazioni sulla topologia del grafo in input, il funzionamento dell'algoritmo At First Look, la dimostrazione della sua correttezza durante l'esecuzione, la gestione della memoria e la struttura dati del Navigational Sketch, ed infine l'analisi delle prestazioni e del costo dell'algoritmo.

1.1.1 Componenti Connesse (CC)

Una componente connessa CC di un grafo non orientato **G**(V,E) con V nodi e E archi è un insieme di nodi $n_i \in V$ tali che

$$n_i \in CC_j \Longrightarrow \forall n_j \in CC \wedge n_j \neq n_i$$
$$\exists\ percorso(n_i, n_j) = \{arco\,(n_i, n_1)\,, arco\,(n_1, n_2)\,, ..., arco\,(n_{n-1}, n_j)\}.$$

cioè per ogni nodo appartenente ad una componente connessa CC_i esiste almeno un percorso verso ogni altro nodo appartenente alla stessa componente connessa, e nessun percorso verso alcun altro nodo.

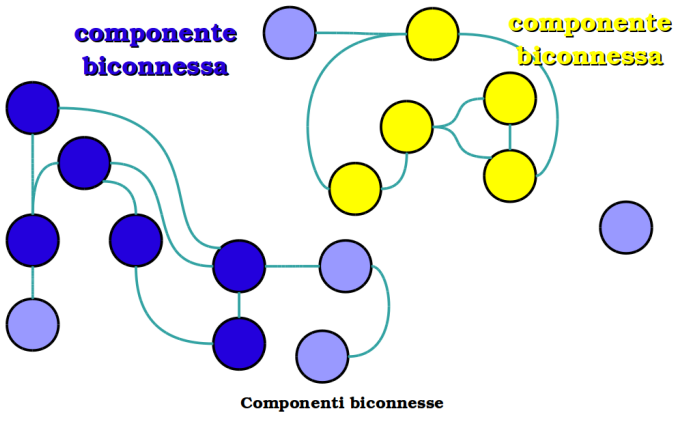


1.1.2 Componenti Bi-Connesse (BCC)

Una componente bi-connessa BCC in un grafo non orientato **G**(V,E) è un insieme di nodi $n_i \in V$ tali che

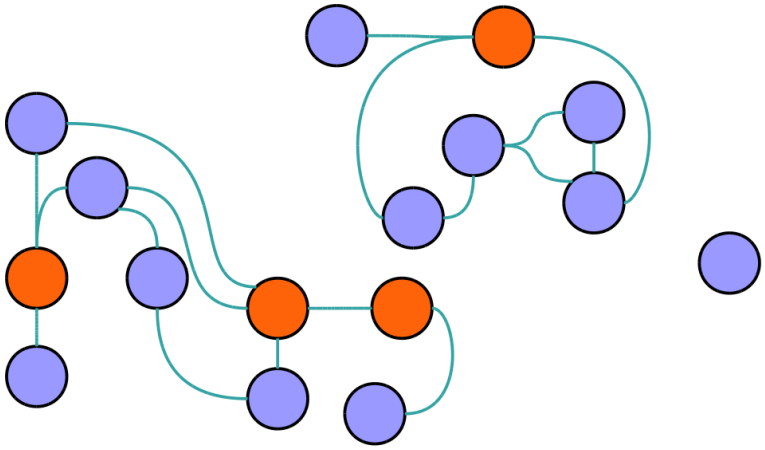
$$n_i \in BCC \Longrightarrow \forall n_j \in BCC \wedge n_j \neq n_i$$
$$\exists \, \# > 1 \; di \; percorsi(n_i, n_j).$$

ovvero presi due nodi $u, v \in BCC$ esistono almeno due percorsi distinti tra u e v in **G**.



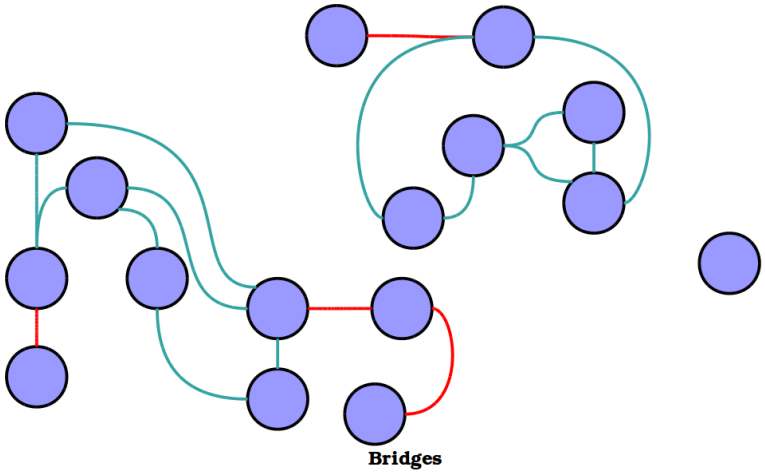
1.1.3 Articulation Points

Un *articulation point* è un vertice $v \in V$ tale che la sua rimozione da **G** aumenta il numero di componenti connesse in **G**.



1.1.4 Bridges

Un *bridge* è un arco $e \in E$ tale che la sua rimozione da **G** aumenta il numero di componenti connesse in **G**.



1.2 Il *classical streaming*

Nel *classical streaming* l'input è un data-stream che viene letto in modo sequenziale e che deve essere processato con una disponibilità di memoria molto minore rispetto alla dimensione dello stream.
Com'è facile intuire i problemi connessi riguardano:

- la velocità con cui ogni informazione prelevata dallo stream viene processata (**PIPT**) rispetto alla lunghezza di questo;

- il calcolo dell'output mantenendo una quantità di informazioni minore rispetto a quella dell'input.

1.2.1 Per Item Processing Time

Più avanti nel capitolo proveremo che il tempo T speso da At First Look per processare l'intero input di un grafo con n nodi e m archi verifica $T \in O(n \log(n) + m\alpha(m, n))$ (dove α è l'inverso della funzione di Ackermann), da cui otteniamo il tempo medio T_i di processamento per record dello stream:

$$T_i \in O\left(\log(n) + \tfrac{m}{n}\alpha(m, n)\right)$$

1.2.2 Limitazioni di memoria

Tipicamente lo stream in input ha una dimensione molto maggiore della quantità di memoria disponibile per la computazione. Siccome lo stream viene processato sequenzialmente record per record, l'output finale deve essere ottenuto modificando progressivamente una struttura dati che sintetizzi correttamente tutte e sole le informazioni strettamente necessarie.

L'informazione complessiva dell'input viene "compressa" all'interno della struttura dati, di cui viene modificato lo stato fino al raggiungimento di uno stato finale quando viene consumato l'intero streaming.

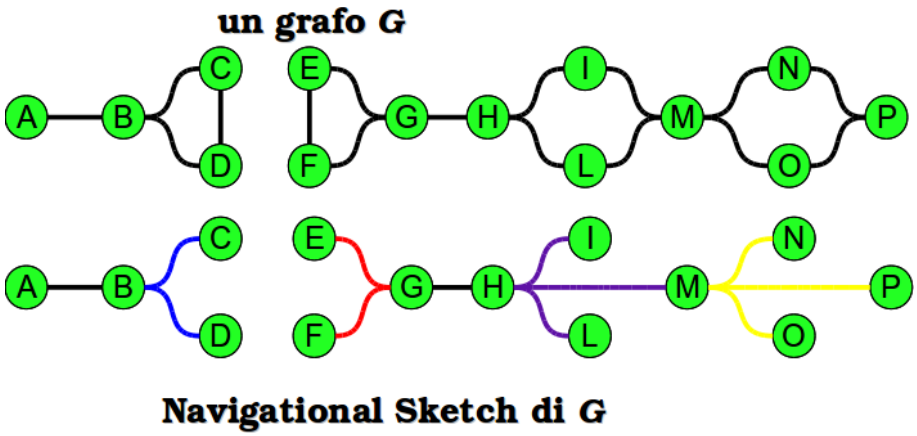
1.2.3 Algoritmi e streaming

La classe di problemi che possono essere affrontati con un modello streaming è composta da quei problemi che ammettono soluzioni che rispettino i vincoli sul PIPT e sulla richiesta di memoria appena descritti.

La ricerca di *connected components* è un problema noto appartenente alla classe appena descritta.

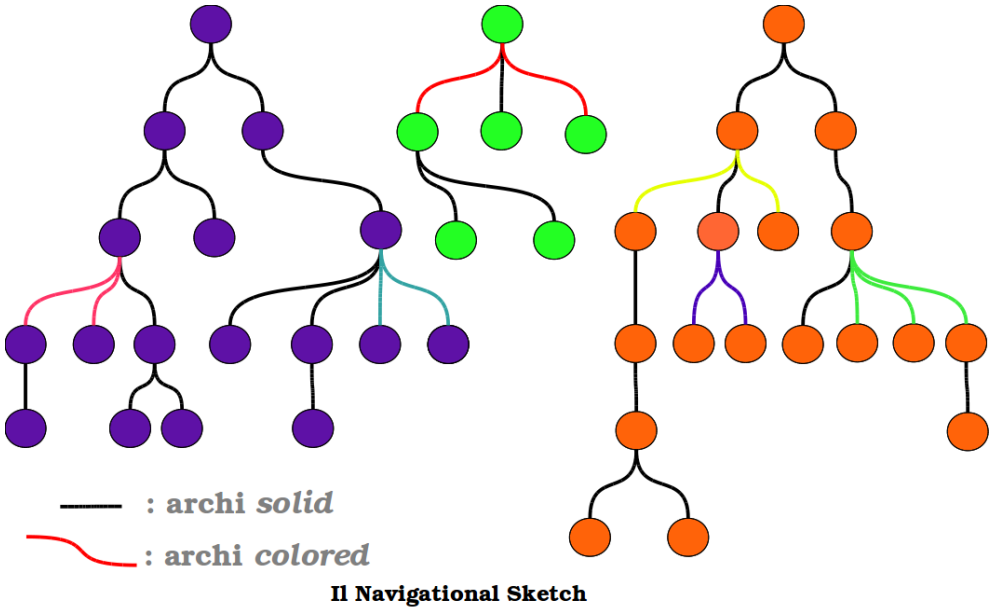
1.3 Navigational Sketch

At First Look sfrutta una struttura dati detta ***Navigational Sketch*** che modella la topologia di un grafo relativamente a componenti connesse e biconnesse, *articulation points* e *edges*.



1.3.1 La struttura di un Navigational Sketch

Il *Navigational Sketch* ***NS*** di un grafo generico $G = (V, E)$ è una foresta $NS = (V_{ns}, E_{ns})$ in cui l'insieme di nodi V_{ns} è lo stesso di G (ovvero $V_{ns} = V$), mentre l'insieme degli archi E_{ns} è composto da archi *solid* e archi *colored*.



Il Navigational Sketch

Un *Navigational Sketch NS* ha le seguenti proprietà:

1. gli alberi in *NS* modellano le componenti connesse di G;
2. gli archi *solid* in NS sono i *bridges* di G;
3. le componenti biconnesse di G sono rappresentate da sottoalberi in *NS* formati:
   - da un nodo padre e $b-1$ nodi figli (dove b è il numero di nodi appartenenti alla stessa componente biconnessa);
   - da archi tutti di uno stesso colore, unico in *NS*.

1.3.2 *Color degree* di un nodo nel Navigational Sketch

Il ***color degree*** di un nodo i di *NS* ($d_c(i)$) corrisponde alla somma del numero di archi *solid* incidenti su i più il numero di diversi colori degli archi incidenti su i.

Ad esempio un nodo i con dieci archi di uno stesso colore e due archi *solid* avrà grado

$$d_c(i) = 1 + 2 = 3,$$

mentre un nodo j con dieci archi *solid* e due *colored* di colori diversi avrà grado

$$d_c(j) = 10 + 2 = 12.$$

Lemma: Un nodo n il cui grado $d_c(n)$ sia maggiore di 1 è un *articulation point* di G.

Dimostrazione: Un nodo siffatto infatti verifica almeno una delle proprietà:

- n appartiene a più di una comopnente biconnessa;
- n appartiene ad una componente biconnessa ed è adiacente ad almeno un *bridge*;
- n è adiacente a più di un *bridge*.

1.4 L'algoritmo At First Look

1.4.1 Definizione del problema

Il problema su cui At First Look opera è il seguente:
preso un grafo generico G rappresentato come stream possibilmente non ordinato dei suoi archi, si vogliono calcolare in modo esaustivo le proprietà di connettività:

- componenti connesse;
- componenti biconnesse;
- *articulation points*;
- *bridges*.

1.4.2 Processamento dello stream

At First Look consuma lo stream in input arco per arco fino al suo esaurimento e ottiene l'output definitivo modificando nel corso dell'elaborazione lo stato di un *Navigational Sketch* ***NS*** mantenuto in memoria. Una volta terminata la lettura dell'input, il *NS* conterrà tutte le informazioni riguardanti componenti connesse e biconnesse, *articulation points* e *bridges* del grafo G descritto nello stream di input.

Le operazioni necessarie all'aggiornamento del *NS* sono:

1. verifica quando due nodi sono nello stesso albero della foresta;
2. unione di alberi;
3. verifica quando due nodi sono estremi degli stessi archi colorati;
4. trova il percorso che collega due nodi.

Queste operazioni vengono compiute in funzione dell'azione richiesta per l'ultimo arco correntemente letto dallo stream.

1.4.3 Come funziona At First Look

At First Look processa gli archi in input nel seguente modo:

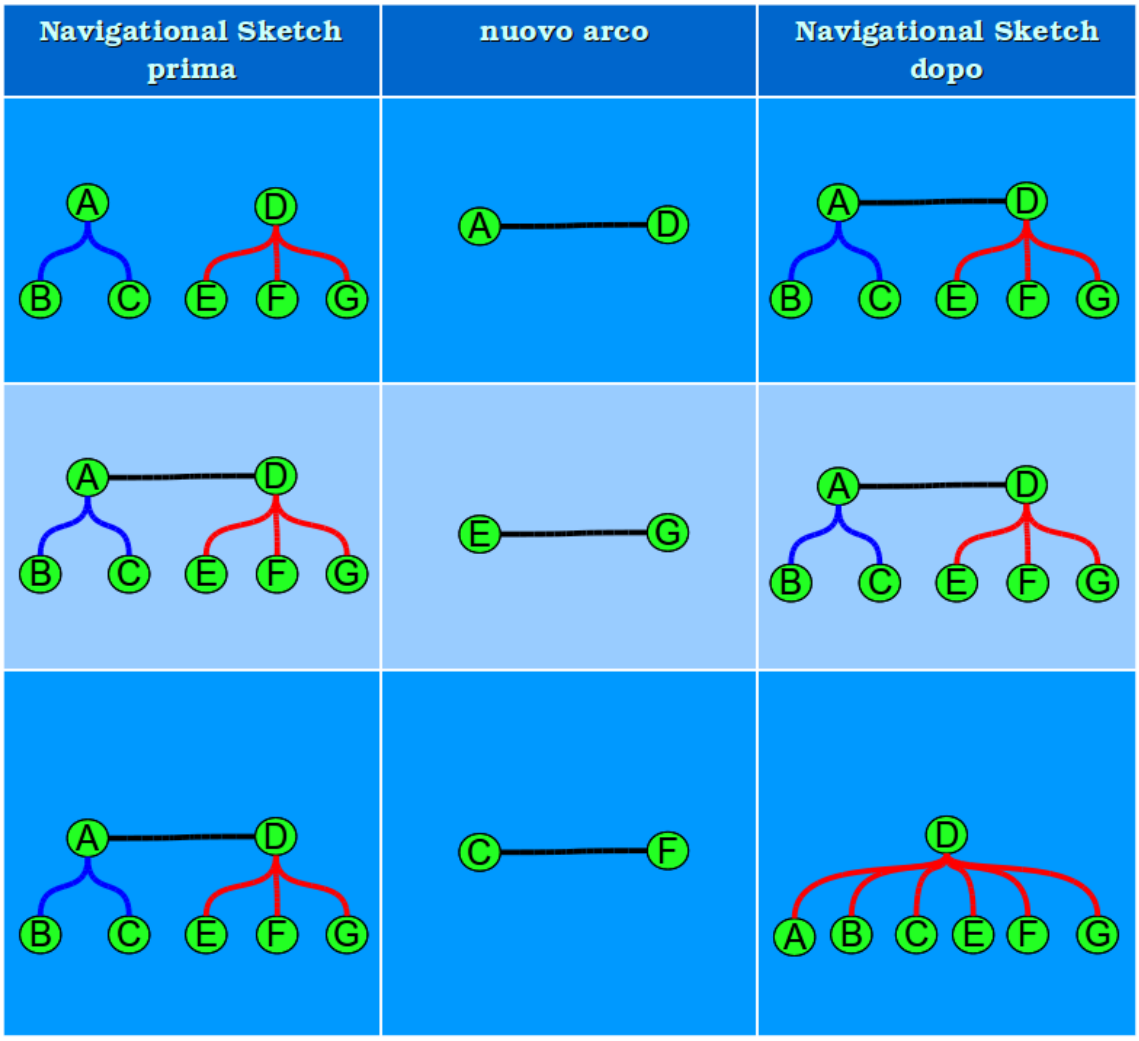
1. se l'arco corrente unisce due alberi distinti allora verrà aggiunto come arco *solid*. Infatti sarebbe l'unico arco che collega due componenti di connesse e quindi un *bridge* per definizione;
2. se l'arco corrente connette due nodi u, v all'interno della stessa componente connessa distinguiamo due casi relativamente al tipo di archi nel percorso tra u e v:
   (a) tutti gli archi nel percorso da u a v **sono dello stesso colore**: in questo caso u e v sono già nella stessa componente biconnessa;
   (b) il percorso tra u e v è composto da archi di **tipo e/o colore diverso**: in questo caso, esiste un nuovo percorso per ognuno dei nodi coinvolti e deve essere costruita un'opportuna componente biconnessa cui questi appartengono.
   Tutte le componenti biconnesse "toccate" dal percorso devono essere unite in una unica, tramite i passi:

i. costruzione del set N di tutti i nodi coinvolti, ovvero tutti i nodi per i quali passa il percorso da u a v e tutti i nodi appartenenti alle componenti biconnesse attraversate dal percorso;

ii. rimozione di tutti gli archi incidenti ad ogni nodo $n \in N$;

iii. scelta di un nodo $n_f \in N$ che diverrà il nodo "*nodo padre*" del sottoalbero rappresentante la nuova componente biconnessa;

iv. aggiunta di un arco *colored* dello stesso colore tra n_f ed ognuno dei nodi figli $n_i \in N$ con $n_i \neq n_f$.



1.5 Correttezza di At First Look

1.5.1 Invarianti

Per dimostrare la correttezza dell'algoritmo dobbiamo dimostrare che ad ogni passo dell'elaborazione, ovvero per ogni processamento di un arco letto dallo stream in input, valgano i seguenti invarianti:

1. ad ogni passo un albero nel *Navigational Sketch* è una componente connessa;
2. ad ogni passo un nodo adiacente ad un arco di colore c appartiene alla componente biconnessa rappresentata da c;
3. ad ogni passo ogni arco *solid* è un *bridge*;
4. ad ogni passo ogni nodo n tale che $d_c(n) > 1$ è un *articulation point.*

1.5.2 Alberi e componenti connesse

Teorema 1: *ad ogni passo un albero nel Navigational Sketch è una componente connessa.*

Dimostrazione: dimostriamo la correttezza del teorema per induzione sulla lunghezza dello stream di archi in input letto.

Nel *caso base* non sono ancora stati letti archi dallo stream, il *Navigational Sketch NS* è privo di archi e ogni suo nodo è un albero singleton. È facile osservare la correttezza del teorema per *NS*, in quanto, non essendoci archi tra i nodi, ogni nodo è l'unico elemento di una componente connessa.

Assumendo che sia stata dimostrata la correttezza del teorema al passo k, cioè dopo aver prelevato dallo stream k archi, vogliamo dimostrare che questa vale anche al passo $n+1$. Aggiungendo un nuovo arco e a *NS* si può verificare che:

- e collega due nodi appartenenti ad alberi differenti in *NS*. In questo caso AFL unisce i due alberi in uno unico. Il teorema vale ancora dopo l'aggiunta di e in quanto i due alberi uniti sono due componenti connesse che all'aggiunta di e vengono collegate tra loro divenendo un'unica CC.
- e collega due nodi appartenenti allo stesso albero in *NS*. In questo caso le eventuali modifiche di *NS* compiute da AFL non cambiano il set di nodi appartenenti all'albero ma solo la struttura dell'albero, come all'interno di un grafo l'aggiunta di un arco tra due nodi di una stessa CC non modifica la CC stessa.

1.5.3 Archi *colored* e componenti biconnesse

Teorema 2: *ad ogni passo un nodo adiacente ad un arco di colore c appartiene alla componente biconnessa rappresentata da c.*

Dimostrazione: dimostriamo ancora la correttezza del teorema per induzione sulla lunghezza dello stream di archi in input letto.

Nel *caso base* non sono stati aggiunti archi a *NS*, quindi non sono presenti in esso archi colorati. Il teorema risulta dunque valido in quanto non esistono componenti biconnesse in un grafo con nessun arco.

Assumendo valido il teorema al passo k, dimostriamo la sua correttezza al passo $k+1$. Come nel caso del teorema precedente, quando viene ricevuto in input il $k+1$-esimo arco e distinguiamo i casi in cui:

- e collega due alberi distinti, ovvero è un arco tra due componenti connesse distinte. In questo caso e verrà aggiunto come arco *solid* tra i due alberi senza che venga aggiunto o modificato alcun arco colorato. Poiché un arco che collega due componenti connesse non genera percorsi alternativi tra i nodi che le compongono, il teorema risulta corretto.
- e collega due nodi u e v all'interno dello stesso albero in *NS*, ovvero all'interno di una stessa CC. Può accadere che:
  - e colleghi due nodi che appartengono ad una stessa componente biconnessa. In questo caso AFL non modifica il Navigational Sketch e l'asserzione del teorema è verificata;
  - e colleghi due nodi che non appartengono alla stessa componente connessa. In questo caso AFL unisce tutti i nodi nel percorso $u \to v$ e le loro componenti biconnesse in un unico sottoalbero che modella un'unica BCC. Per la definizione di componente biconnessa, il nuovo sottoalbero così formato rappresenta correttamente un'intera BCC e il teorema è verificato.

  alta

1.5.4 Archi *solid* e *bridges*

Teorema 3: *ad ogni passo ogni arco* solid *è un* bridge.

Dimostrazione: anche in questo caso dimostriamo la correttezza del teorema per induzione sulla lunghezza dello stream di archi in input letto.

Nel *caso base* non sono stati aggiunti archi a *NS*, dunque non sono presenti archi *solid* e il teorema vale.

Assumendo valido il teorema al passo k, dimostriamo la sua correttezza al passo $k+1$. Per un nuovo arco e letto dallo stream distinguiamo ancora i casi:

- e collega due nodi appartenenti ad alberi distinti. A *NS* viene aggiunto un arco *solid* tra i due nodi unendo così due alberi in uno unico, e, poiché la sua rimozione porterebbe nuovamente la divisione dell'albero appena formatosi nei due sottoalberi originali, e è un *bridge* e il teorema vale.
- e collega due nodi dello stesso albero u e v:
  - u e v sono adiacenti ad archi dello stesso colore. In questo caso *NS* non subisce modifiche e il teorema vale.
  - u e v non sono adiacenti ad archi dello stesso colore. L'insieme, possibilmente vuoto, degli archi *solid* che appartengono al percorso $u \to v$ è rimpiazzato da archi colorati. Dato che, se viene aggiunto un arco e tra due nodi u e v della stessa componente connessa, esiste più di un percorso $P = u \to v$, eventuali archi $e_i \ : \ P = e_1, e_2, ...e_n \ \wedge \ i \in (1, n)$ che erano *bridges* prima dell'aggiunta di e non lo sono più una volta aggiunto e e il teorema è verificato.

1.5.5 Color degree $d_c(n)$ e articulation points

Teorema 4: *ad ogni passo ogni nodo n tale che* $d_c(n) > 1$ *è un* articulation point.

Dimostrazione: è facile dimostrare la correttezza del teorema a partire dai teoremi 2 e 3 e dal lemma della sezione 1.3.2.

1.6 Memoria e struttura dei dati

La struttura dati del Navigational Sketch in memoria deve mantenere le informazioni di supporto alle operazioni in 1.4.3 compiute da At First Look, ovvero su:

- gli insiemi di nodi appartenenti alla stessa CC;
- gli insiemi di nodi appartenenti alla stessa BCC;
- le connessioni tra i nodi, rappresentate come percorsi nella foresta.

Tali informazioni sono necessarie perché possano compiersi le operazioni dell'algoritmo.

1.6.1 Supporto alle operazioni

Possiamo dividere le operazioni compiute da At First Look in

- operazioni relative alle **componenti connesse**:

  A : ricerca dei nodi che appartengono allo stesso albero $\rightarrow$ $find_{CC}$;

  B : unione di alberi distinti in NS $\rightarrow$ $union_{CC}$;

- operazioni relative alle **componenti biconnesse**:

  C : ricerca dei nodi contigui ad archi dello stesso colore $\rightarrow$ $find_{BCC}$;

  D : unione di nodi connessi da archi *solid* o dello stesso colore $\rightarrow$ $union_{BCC}$

- operazioni relative alla **ricerca di percorsi** nella foresta:

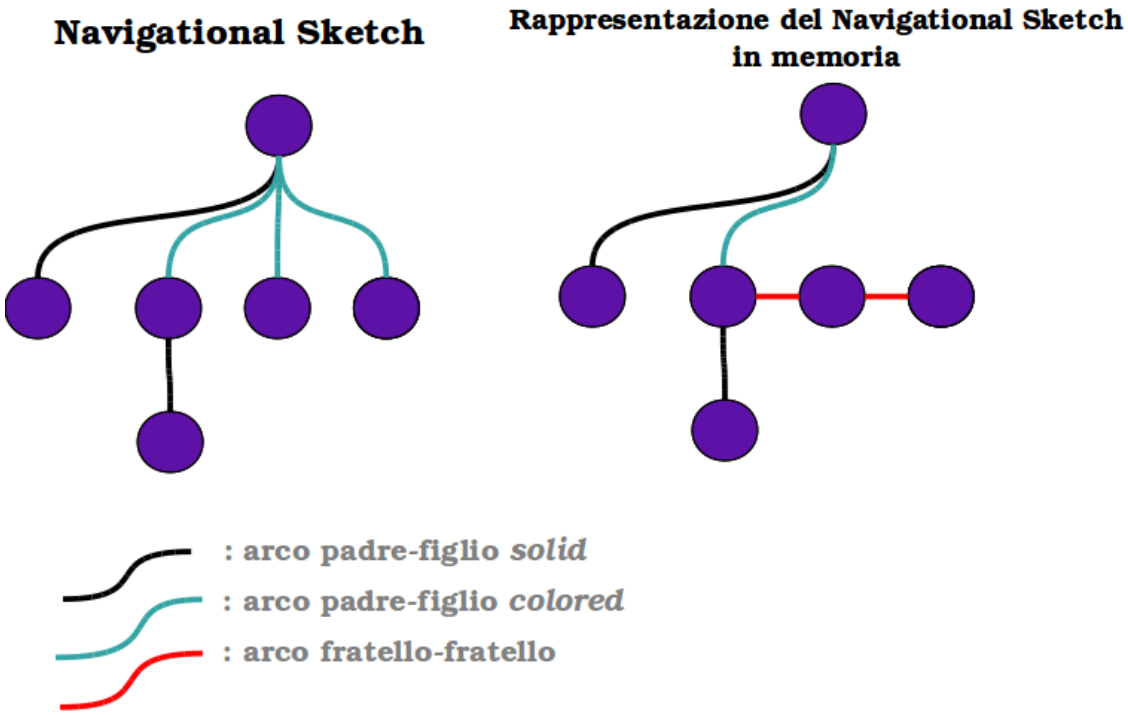
  E : ricerca del percorso tra due nodi $\rightarrow$ $LeastCommonAncestor(LCA)$

La difficoltà di implementare una struttura dati che consenta di mantenere un tempo di esecuzione ottimo delle operazioni appena descritte sorge nel momento in cui alcune di queste necessitano che i dati siano indicizzati sui nodi (cioè le funzioni di *union/find*) mentre altre sugli archi (la funzione LCA).

1.6.2 La struttura dati

La struttura dati proposta in [6] soddisfa il vincolo sull'esecuzione delle operazioni di AFL in tempo ottimo ed è una tabella che, preso in input un grafo $G = (V, E)$, ha per righe i nodi $v \in V$ e per colonne i dati su:

1. il padre di v nella foresta (se esiste) $\rightarrow$ $father$;
2. l'elemento rappresentativo della componente biconnessa di v (se esiste) $\rightarrow$ $BCC\ rep$;
3. il fratello sinistro di v nella sua componente biconnessa (se esiste) $\rightarrow$ $Left\ Brother$;
4. il fratello destro di v nella sua componente biconnessa (se esiste) $\rightarrow$ $Right\ Brother$;
5. l'elemento rappresentativo della componente connessa di v (se esiste) $\rightarrow$ $CC\ rep$;
6. la dimensione della componente biconnessa di v $\rightarrow$ $BCC\ Size$;
7. la dimensione della componente connessa di v $\rightarrow$ $CC\ Size$;



Non tutti i dati di un nodo v necessitano di essere scritti per ogni nodo in quanto è possibile dedurli attraverso le relazioni tra v e altri nodi. Ad esempio l'informazione sul padre di v può essere dedotta dalle informazioni mantenute dai fratelli di v.

Si noti inoltre che le informazioni memorizzate sono rappresentative degli archi tra i nodi del *NS*, ad eccezione delle ultime due ($BCC\ Size$ e $CC\ Size$) che invece sono di supporto all'operazione di *union-by-size*.

1.6.3 Proprietà del Navigational Sketch

La struttura dati utilizzata da At First Look riflette la struttura del Navigational Sketch, ovvero un grafo composto da una foresta in cui:

- gli alberi rappresentano le componenti connesse. La radice r di ogni albero identifica l'albero stesso, e tutti i nodi di detto albero fanno riferimento a r direttamente o indirettamente (nel caso in cui l'informazione venga ricercata presso nodi in qualche modo contigui) nel campo $CC\ rep$;
- tutti i nodi dell'albero fanno riferimento diretto o indiretto al nodo padre come identificativo della propria componente biconnessa nel campo $BCC\ rep$;

Lo spazio occupato in memoria dal Navigational Sketch per un input stream di un grafo con n nodi è di $O(n \log n)$ bit, ovvero lo spazio di memoria richiesto da At First Look è molto ridotto. Dimostreremo questa misura nella prossima sezione.

1.7 Analisi delle prestazioni

1.7.1 Numero di esecuzioni per operazione

Per poter svolgere l'analisi dell'algoritmo sull'intero stream in input occorre calcolare quante volte viene chiamata ogni *macro-operazione* del tipo LCA,*find*, *union*, ecc.

Consideriamo uno stream che descriva un grafo $G = (V, E)$ con $n = \#\{V\}$ e $m = \#\{E\}$. Al termine della sua esecuzione AFL avrà compiuto:

- $2m$ $find_{CC}$, una per ogni arco nello stream;
- $2m$ $find_{BCC}$, una per ogni arco nello stream;
- $n - 1$ $union_{CC}$, perché un nodo non può essere rimosso da una CC una volta che vi appartiene;
- $n - 1$ $union_{BCC}$, perché un nodo non può essere rimosso da una BCC una volta che vi appartiene;
- $n - 1$ LCA per trovare le BCC's che devono essere unite;

La sommma delle macro-operazioni compiute mediamente da AFL sarà dunque data da

$$2m\,(find_{CC} + find_{BCC}) + (n-1)\,(union_{CC} + union_{BCC} + LCA)$$

1.7.2 Costo delle operazioni

Il costo in termini di tempo di elaborazione dell'esecuzione di At First Look deve essere calcolato come la somma dei tempi di elaborazione di medi di ogni macro-operazione pesati in base alla loro frequenza.

Union by size: Da **RIFERIMENTO** è noto che l'implementazione dell'euristica *union by size* implica che il costo di m operazioni di *find* su n elementi e $n - 1$ operazioni di *union* sia dell'ordine di $O\,(n + m\alpha\,(m, n))$, dove α è una funzione che cresce molto lentamente corrispondente all'inverso della funzione di Ackermann. Le operazioni di *union/find* su componenti

connesse e componenti biconnesse in At First Look seguono questo andamento. Si noti che l'unione relativa alle CC's ed alle BCC's non compie le stesse operazioni, in quanto nell'unione di due CC's è necessaria una rotazione dell'albero con meno nodi.

Least Common Ancestor: Nella ricerca dell'LCA di due nodi u e v all'interno di uno stesso albero si risale contemporaneamente la genealogia di entrambi i nodi marcando i nodi visitati fino a quando si incontri un nodo già marcato. Il costo dell'operazione è $O(d)$ dove d è la profondità massima dei due nodi rispetto all'LCA. Il costo di una sequenza di $n-1$ ricerche di LCA è $O(n)$.

Dalle osservazioni appena compiute possiamo desumere il seguente teorema:

Teorema: *Il caso peggiore del tempo di processamento complessivo è*

$$O(m\alpha(m,n) + n\log n)$$

Dividendo il costo del processamento complessivo dell'intero stream per il numero m dei suoi elementi otteniamo il costo del tempo di processamento per elemento ammortizzato (amortized PIPT):

Teorema: *Il tempo di processamento ammortizzato per elemento è*

$$O\left(\tfrac{n}{m}\log n + \alpha(m,n)\right).$$

Corollario: *Se il grado medio del grafo è maggiore o uguale a* $\log n$ *il tempo di processamento ammorrtizzato per elemento è*

$$O(\alpha(m,n)).$$

Capitolo 2

MapReduce

2.1 Cos'è MapReduce

MapReduce è sia un paradigma di programmazione, che l'implementazione del paradigma in un framework. Nasce da Google nel 2004 per la computazione distribuita di grandi data-sets su cluster di elaboratori.

Lo sviluppatore di un'applicazione MapReduce definisce la funzione ***map*** (μ)

$$\mu(chiave, valore) \rightarrow (chiave_1, valore_1), (chiave_2, valore_2), ..., (chiave_n, valore_n)$$

che presa in input una coppia <*chiave, valore*> produce un set intermedio di n coppie <*chiave, valore*>, e la funzione ***reduce*** (ρ)

$$\rho(chiave, listavalori) \rightarrow (output_1), ..., (output_m)$$

che elabora tutti i valori associati ad una specifica chiave intermedia.

Da ora in avanti faremo riferimento ad una generica coppia <*chiave, valore*> semplicemente con il nome di ***coppia***.

Un programma scritto seguendo questo paradigma può essere eseguito parallelamente su un cluster costituito da un ampio parco di elaboratori.

Al programmatore sarà sufficiente aver definito correttamente le funzioni di map e reduce, mentre non dovrà curarsi dell'implementazione della parallelizzazione vera e propria dell'esecuzione su più macchine.

Spesso sviluppando un'applicazione MapReduce non è sufficiente implementare un solo ciclo composto da una sola fase di *Map* e una *Reduce*, ma c'è bisogno di definire più cicli r_i, ognuno dei quali provvisto di una funzione *map* μ_i e di una *reduce* ρ_i.

$$r_i = (\mu_i, \rho_i).$$

e in cui l'output del ciclo r_i diviene l'input per il ciclo r_{i+1}.

Da ora in avanti faremo riferiento a un ciclo $map \rightarrow shuffle \rightarrow reduce$ con il nome di ***round***.

Il sistema a run-time: dopo una fase preliminare di configurazione, si occupa di risolvere i tipici problemi della parallelizzazione, ovvero la tolleranza ai guasti (*fault tolerance*), l'intercomunicazione e lo scheduling dell'esecuzione e dei tasks tra le diverse macchine, oltre che della creazione e del mantenimento di un filesystem distribuito tra i nodi della rete.

Tipicamente si ricorre all'uso di programmi basati su MapReduce nei casi in cui si vogliano elaborare tera-byte o peta-byte di dati su cluster di centinaia o migliaia di macchine, siano esse proprietarie (es. Google) o in affitto (es. cloud).

WordCount: Per una più facile comprensione ci aiuteremo definendo secondo il paradigma *MapReduce* il programma di esempio WordCount, che opera il conteggio dell'occorrenza delle parole all'interno di un testo.

2.2 Input e Output

Come accennato, sono proprio le dimensioni dell'input e dell'output generalmente a rendere necessaria l'adozione del modello MapReduce. Il dover processare una grande quantità di dati infatti porta ad avere bisogno di una grande quantità di risorse, specialmente in termini di capacità di calcolo, e di conseguenza sempre più spesso si ricorre al calcolo parallelo su più macchine.

2.2.1 L'Input

L'input complessivo di un programma MapReduce è espresso come set di coppie <*chiave, valore*>, dove la *chiave* distingue tra input di tipo differente mentre il *valore* esprime il valore dell'input. A seconda del problema che si vuole risolvere è necessario definire dapprima la semantica e poi il tipo di tali valori.

La **struttura dell'input** è tale da permettere l'elaborazione atomica di ogni coppia ad opera della funzione *map*. Ad ogni esecuzione l'input subisce una fase di *splitting* in cui viene suddiviso in porzioni che verranno poi elaborate in parallelo.

Poiché lo *splitting* dell'input non viene generalmente definito dal programmatore, le coppie che lo costituiscono dovranno poter essere elaborate correttamente indipendentemente dall'ordine con cui vengono sottoposte al programma.

In realtà è possibile porre dei vincoli sulle possibili suddivisioni dell'input in fase preliminare. Nonostante ciò, anche una volta definita accuratamente la ripartizione dell'input in porzioni distinte, non è possibile imporre né prevedere alcuna disciplina di servizio di dette porzioni presso l'applicativo.

2.2.2 Output intermedi

Al termine della fase di mapping, gli output prodotti vengono memorizzati in un percorso temporaneo del filesystem distribuito. Da qui vengono estratti e collezionati in funzione della chiave per essere sottoposti alla funzione *reduce*.

2.2.3 L'Output

L'output di un programma MapReduce è costituito da uno o più file contenenti i risultati dell'elaborazione e memorizzati all'interno del filesystem distribuito.

2.3 La funzione *Map()*

La funzione *map* prende come input una coppia <*chiave, valore*> per volta.

```
map(String key,String value);
```

Ogni coppia viene elaborata atomicamente dalla funzione e per ognuna di esse l'output prodotto è un set di nuove coppie intermedie <*chiave, valore*> possibilmente vuoto.

Vista dallo sviluppatore la funzione sarà del tipo:

```
map (String key, String value){
  /* key: utile nel caso la funzione debba poter distinguere
   *     tipologie di input diverse all'interno dell'input
   *     complessivo sottoposto all'applicazione;
   *
   *  value: il valore dell'input
   */
```

```
  definizione della funzione...

  emissione delle coppie \mbox{$<$chiave, valore$>$} intermedie;
}
```

Nell'esempio di WordCount la funzione *map* può essere descritta dallo pseudocodice:

```
map (String key, String value){
  /* key: il nome del documento di testo da analizzare
   *
   *  value: il testo del documento
   */

per ogni parola w in value:
   produci in output la coppia <w,1>;
}
```

2.4 La funzione *Reduce()*

La funzione *reduce* prende in input una lista composta da tutte le coppie intermedie <*chiave, valore*> con lo stesso campo chiave (ovvero la lista dei valori appartenenti alle coppie con la stessa chiave),

```
reduce(String key, List <String> values)
```

e produce un set di valori di output possibilmente vuoto. Tale set di valori di output sono da considerarsi già parte del risultato definitivo dell'elaborazione dell'input. L'insieme degli output delle chiamate alla funzione *reduce* sui set di valori di ogni chiave intermedia costituisce l'output finale dell'applicativo.

Vista dallo sviluppatore la funzione *reduce()* sarà del tipo.

```
reduce (String key, List <String> values){
 /* key: chiave intermedia, serve a selezionare i valori intermedi su cui
   *     eseguire la funzione reuce();
   *
   *  values: l'insieme dei valori intermedi che condividono la stessa chiave.
   */

 definizione della funzione...
```

```
  emissione dei valori che costituiscono l'output definitivo;
}
```

Nell'esempio di WordCount la funzione *reduce* può essere descritta dallo pseudocodice:

```
 map (String key, List <String> values){
   /* key: la parola di cui si vogliono contare le occorrenze
    *
    *  values: la lista delle occorrenze
    */
  contatore=0;
  per ogni valore v in values:
    ++contatore;
  produci output (contatore);
}
```

2.5 La funzione *Combine()*

Spesso in un programma MapReduce si ha che:

- la funzione *reduce* definita dallo sviluppatore è commutativa e associativa relativamente ai propri input;
- sono presenti numerose ripetizioni nelle *coppie* intermedie che la funzione *reduce* riceve in input.

Quando si verificano queste condizioni è possibile definire una terza funzione detta ***combiner*** che associa i risultati che verrebbero elaborati insieme dalla funzione *reduce* prima di immetterli nella rete.

La funzione *combiner* è eseguita da ogni macchina che ospita le funzioni di *mapper* e sebbene generalmente condivida lo stesso codice della funzione *reduce* viene definita dallo sviluppatore come funzione indipendente.

L'introduzione di tale funzione riduce, spesso in maniera considerevole, il traffico sulla rete del cluster dovuto alla trasmissione dei risultati intermedi sopra descritti, ovvero di quelle *coppie* che verrebbero processate insieme da uno stesso *reducer*.

2.6 Struttura di MapReduce

Abbiamo visto come MapReduce si basi sull'implementazione e uso delle funzioni *map* e *reduce*. Tali funzioni verranno dunque eseguite su macchine, che a seconda del caso saranno dette

- ***mapper***;
- ***reducer***.

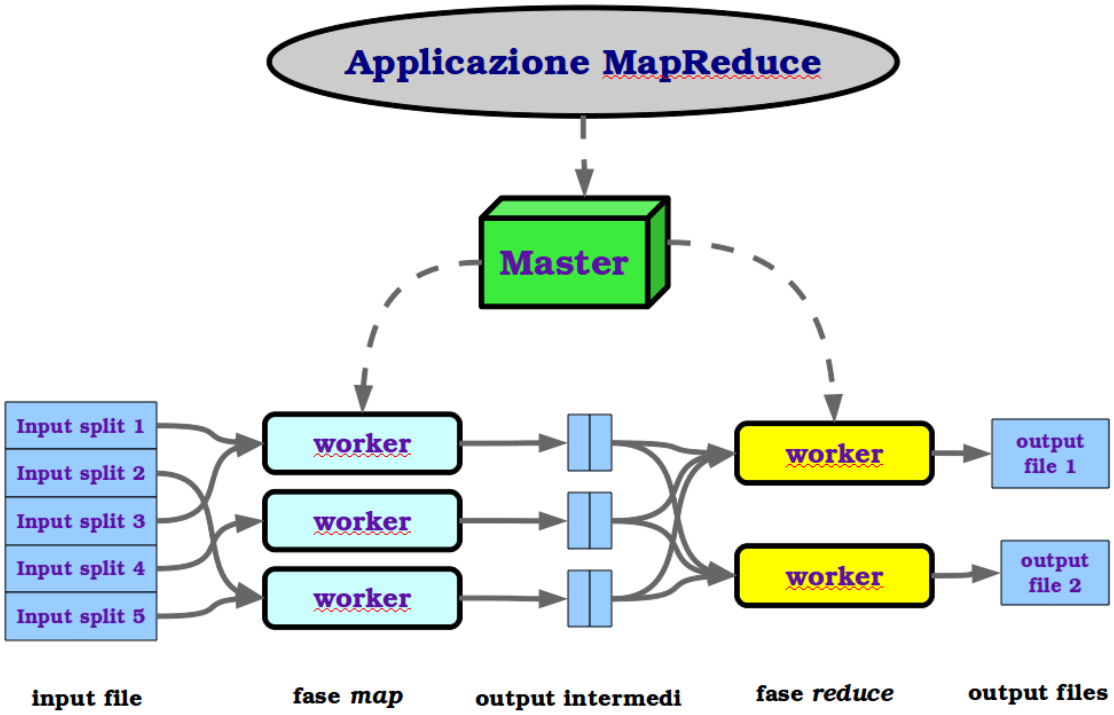
A questi va aggiunto un terzo tipo di agente, lo ***shuffler***, che si occupa di raggruppare tutti i risultati intermedi con chiavi uguali e sottoporli ad un reducer.

Negli scenari reali tipicamente i nodi che compongono il cluster svolgono sia le funzioni di *mapper* che di *reducer*.

2.6.1 L'esecuzione

Il flusso dell'esecuzione di un programma basato su MapReduce si evolve attraverso le tre fasi:

1. **Map step**
   (a) *splitting* dell'input e diffusione del programma;
   (b) Assegnazione dei *tasks* ai *mappers*;
   (c) Esecuzione dei *mappers*;
2. **Shuffle step**
   (a) Salvataggio risultati intermedi;
   (b) Ordinamento risultati intermedi;
3. **Reduce step**
   (a) Esecuzione dei *reducers*;
   (b) Ritorno del programma.



2.7 *Map* step

2.7.1 Splitting dell'input e diffusione del programma

L'input complessivo viene suddiviso (***splitting***) in **M** porzioni (*splits*), tipicamente di dimensioni che variano tra i 16 ed i 64 megabytes ciascuno.

Dopo il partizionamento dell'input viene diffuso il programma MapReduce che si vuole eseguire tra tutti i nodi che compongono il cluster MapReduce. Un nodo oltre a ricevere una copia del programma è incaricato di svolgere la funzione di ***master***. Gli altri nodi, quelli cioè addetti all'esecuzione vera e propria del programma, sono detti ***slaves***.

Il nodo *master* avvia l'esecuzione del programma su ogni nodo.

2.7.2 Assegnazione dei *tasks* ai *mappers*

Il nodo *master* assegna ad ogni nodo uno degli M *tasks*, ovvero un *t*ask per ogni *s*plit in cui è stato suddiviso l'input.

Finché non sono terminati i *tasks*, il *master* interroga lo stato di tutti i nodi *slaves* per conoscere quali tra questi sono tornati *idle* e sono quindi disponibili per eseguire i *tasks* restanti.

2.7.3 Esecuzione dei *mappers*

Ogni *mapper* a cui è stato assegnato un *task* legge lo *split* ad esso corrispondente.

Il *mapper* sottopone alla funzione *map* definita nel programma dallo sviluppatore ogni *coppia* presente nello *split* e bufferizza i risultati intermedi nella sua memoria locale.

2.8 *Shuffle* step

2.8.1 Salvataggio risultati intermedi

I risultati intermedi presenti nella memoria del mapper vengono scritti periodicamente in un percorso temporaneo del filesystem distribuito (*textbfDFS*), dove vengono partizionati in **R** regioni corrispondenti agli R *reduce tasks*. Questa operazione è gestita dalla funzione *shuffle* che non è definita dal programmatore ma nell'implementazione del sistema MapReduce.

Il *master* notificherà ai *reducers* che sono in uno stato *idle* le regioni del *DFS* dei rispettivi *tasks*.

2.8.2 Ordinamento risultati intermedi

I risultati intermedi nel *DFS* vengono ordinati dalla funzione *shuffle* in base alle *chiavi* intermedie affinchè ogni *reducer* faccia riferimento solo alle *coppie* che competono al suo *task*. Generalmente infatti più *chiavi* intermedie venono mappate su uno stesso *reduce task*.

Il *reducer* cui viene notificata dal *master* una partizione del *DFS* invoca le procedure di chiamata remota per leggere i risultati intermedi che deve sottoporre alla funzione *reduce*.

2.9 *Reduce* step

2.9.1 Esecuzione dei *reducers*

Ogni *reducer* applica la funzione *reduce* sul set di valori corrispondenti ad ogni chiave intermedia relativa al suo *task*.

L'output di ogni chiamata a *reduce* viene appeso in coda al file di output finale del programma MapReduce.

2.9.2 Termine del programma

Una volta che tutti i *mappers* e i *reducers* hanno terminato i propri *tasks* il *master* lo segnala al programma dell'utente, la chiamata MapReduce termina e ritorna al codice originale.

2.10 Valutazione di un Algoritmo MapReduce

Esistono diverse metriche per valutare l'efficienza dell'esecuzione di un algoritmo MapReduce:

- t: il numero di cicli richiesti dall'algoritmo;
- $n_{i,j}$: la somma delle dimensioni di input e output per il reducer j al *round* i-esimo:

$$n_{i,j} = size(ReduceInput_{i,j}) + size(ReduceOutput_{i,j}).$$

- M_i: la complessità dei messaggi al *round* i-esimo, ovvero la dimensione totale degli input e degli output per tutti i *reducers* al round i:

$$M_i = \sum_j n_{i,j}$$

- M: La complessità globale dei messaggi:

$$M = \sum_{i=1}^{t} M_i$$

- r_i: Il tempo interno di esecuzione per il *round* i-esimo, ovvero il maggior tempo di esecuzione richiesto da un *reducer* nel corso del round i, assumendo $r_i \geq max\{n_{i,j}\}$

- r: Il tempo interno di esecuzione globale:

$$r = \sum_{i=1}^{t} r_i$$

- B: La dimensione del buffer ai *reducers*, ovvero la quantità massima di memoria richiesta da un *reducer* nel corso dei t *rounds* per processare i propri input e output.

- L: La latenza della rete di *shuffle*, ovvero il numero di operazioni che un *mapper* o un *reducer* devono attendere prima di ricevere una prima *coppia* in input nel corso di un round.

- b: La dimensione della banda della *shuffle network* misurata in termini di numero di unità di input/output (*coppie*) che transitano nella *shuffle network* a un dato istante.

- T: Il tempo totale di esecuzione dell'algoritmo, detto anche ***tempo di esecuzione MapReduce***, che è la metrica fondamentale tramite cui valutare l'efficienza dell'algoritmo:

$$\begin{aligned} T &\in O\left(\sum_{i=1}^{t}\left(r_i + L + \frac{M_i}{b}\right)\right) \\ &= O\left(r + tL + \frac{M}{b}\right) \end{aligned}$$

  Poiché B per come è definito rappresenta il limite massimo per ogni r_i e $r = \sum_{i=1}^{t} r_i$, allora vale $r \leq tB$ e quindi

$$T = O\left(t\,(B + L) + \frac{M}{b}\right).$$

  In altri termini per dare una stima dell'efficienza dell'algoritmo può essere sufficiente catturare le informazioni relative al numero di round t ed al numero di messaggi di input/output M complessivamente transitati per i *reducers*.

Capitolo 3

Hadoop

3.1 Cos'è Hadoop

Apache Hadoop è un framework opensource prodotto e diffuso dalla *Apache Foundation* che implementa il paradigma MapReduce di Google per la computazione di grandi moli di dati in modo distribuito, scalabile e affidabile attraverso l'uso di cluster di computer potenzialmente molto estesi (fino anche a migliaia di macchine). Pensato originalmente per il progetto Nutch, un motore di ricerca opensource, ha suscitato molto interesse per le sue potenzialità general-pourpose.

Hadoop offre:

- un file system distribuito (Hadoop Distributed File Sistem - HDFS), che memorizza i dati su potenzialmente migliaia di serventi;
- la capacità di processare tali dati eseguendo il processamento in prossimità di dove sono memorizzati.

Opera attraverso una sequenza di operazioni compiute su un data set di valori $< chiave,\ valore >$ del tipo descritto in 2.1.

Ogni fase del calcolo distribuito è *fault tolerant* ovvero, nel caso in cui un nodo della rete vada in crash o subisca rallentamenti inprevisti, i task che gli competevano vengono ridistribuiti sugli altri nodi.

3.2 Il framework Hadoop

Il progetto di Hadoop è composto dai sottoprogetti:

- *Hadoop Common*: È la componente principale dell'intero progetto, contiene le librerie di supporto agli altri sottoprogetti.
- *Hadoop Distributed File System* - ***HDFS***: implementa il DFS condiviso tra i nodi del cluster, offre le funzioni di lettura e scrittura, mantiene le informazioni gestendone l'allocazione, la ridondanza e l'aggiornamento.
- *Hadoop MapReduce*: implementa il paradigma MapReduce vero e proprio sul sistema Hadoop, offre le librerie per la scrittura di programmi MapReduce.

Un programma Hadoop implementa le funzioni nelle librerie di *MapReduce*, mentre la sua esecuzione sui nodi del cluster e la gestione della corretta lettura e scrittura sull'HDFS sono gestite dalle librerie in *Hadoop Common*.

3.2.1 Hadoop Common

Offre tutte le librerie di supporto necessarie alle applicazioni client per l'esecuzione sul cluster, ovvero:

- le librerie per le comunicazioni tra nodi della rete *master* e *slave*;
- le librerie con le funzioni per la gestione dell'HDFS (lettura, scrittura, formattazione, ecc);
- le funzioni ed i file di configurazione dei demoni JobTracker, TaskTracker, NameNode, DataNode.

3.2.2 Hadoop Distributed File System - HDFS

È il file system distribuito sul cluster, costruito sul modello del Google File System. In esso sono letti e scritti i dati in input e quelli in output di ogni applicazione client. Prima di ogni esecuzione di un'applicazione Hadoop, è necessario caricare manualmente i dati in input, come anche andrà prelevato manualmente l'output, residente nell'HDFS, al termine dell'esecuzione.

L'Hadoop File System è un filesystem Unix-like in cui è presente una directory home specifica per le attività dell'utente in /user/nomeutente/ ed altre destinate ai file temporanei in /temp/*.

Sono presenti nel framework diversi programmi per la lettura e la modifica dei file dell'HDFS come *cat*, *ls*, ed altri. Nonostante quanto si potrebbe pensare, è importante notare che HDFS non è un filesystem POSIX.

3.2.3 Hadoop MapReduce

Il sottoprogetto Hadoop MapReduce colleziona una serie di librerie che consentono l'implementazione di programmi Java secondo il paradigma MapReduce. Hadoop offre anche la possibilità di implementare programmi scritti in alcuni linguaggi diversi da Java, ovvero C++ e python. Il supporto alla programmazione in questi linguaggi di applicativi MapReduce è possibile grazie alle librerie Hadoop Streaming. Generalmente comunque un programma MapReduce è un sorgente singolo compilato in Java (.class) o un insieme di sorgenti collezionati in un file con estenzione .jar.

3.3 Struttura di un Cluster Hadoop

L'architettura di Apache Hadoop è di tipo *master/slaves*, ovvero i nodi del cluster possono operare da coordinatori dell'esecuzione dei task (*master*) o da esecutori del loro processamento (*slaves*).

Un cluster Hadoop è composto da una o più macchine ***master*** e nessuna o più macchine ***slaves***.

3.3.1 configurazione del cluster

La configurazione di un cluster di n macchine definisce quali e quanti nodi opereranno come nodi *master* e quali come *slaves*.

La presenza di più nodi *master* consente la sostituzione del nodo che correntemente svolge le mansioni di *master*, nel caso questo subisca un crash o non sia più raggiungibile dalla rete, con uno tra gli altri candidati *masters*, mentre una buona disponibilità di slaves consente una maggiore suddivisione del carico di lavoro complessivo ed una maggiore tolleranza ai guasti a runtime. Il trade-off nel rapporto tra il numero m di nodi *master* ed il numero s di *slaves* dipende soprattutto dalla probabilità di avere crash o isolamenti nella rete reale del cluster in uno specifico caso d'uso.

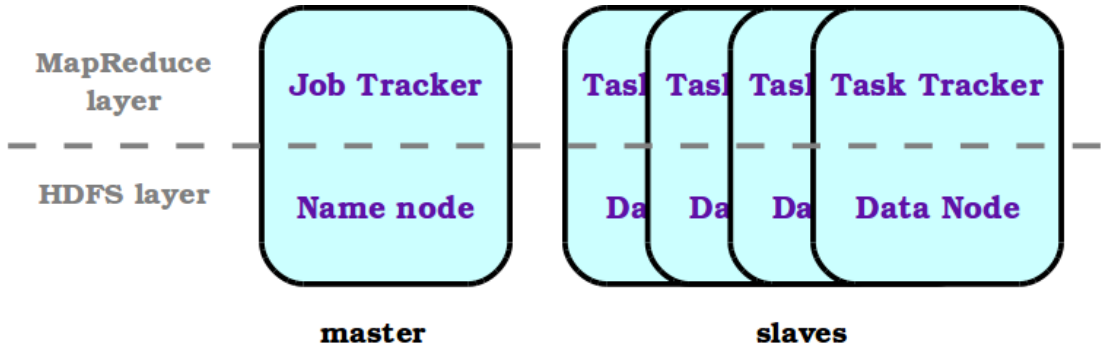
Nel caso in cui ci sia solo una macchina *master* e nessun nodo *slave* si ha una configurazione *single node*: in questo caso in realtà il nodo opera sia come *master* che come *slave*.

I nodi *master* ospitano i demoni dei servizi:

- JobTracker → **JT**;
- NameNode → **NN**.

I nodi *slave* ospitano i demoni dei servizi:

- TaskTracker → **TT**;
- DataNode → **DN**.



3.3.2 JobTracker e TaskTracker

Il TaskTracker

Un TaskTracker (TT) accetta ed esegue i job che gli vengono sottoposti dal JobTracker (JT). I nodi TT sono i nodi *slaves* che eseguono una porzione dell'elaborazione complessiva detta **task**.

Ogni TT esegue il task corrente in un processo separato da quello dello stesso demone server TT, in modo da impedire che un fallimento nel processo di esecuzione del task causi il termine dell'intero processo TT. Il TT cattura l'output e l'exit code di ogni processo così generato e una volta che uno di questi termina (correttamente o meno) lo notifica al JT.

Un TT può eseguire contemporaneamente un numero t di task, e suddivide la propria memoria locale in t slot, uno per ogni task che può accettare. Periodicamente il TT invia al JT un ping che lo informa che il processo è ancora in vita, e su questo ping viene fatta viaggiare l'informazione sul numero di slots ancora disponibili. Il JobTracker suddividerà il carico di lavoro rimanente sui TT con una maggiore disponibilità di slot.

Il JobTracker

Il JobTracker si occupa di assegnare i task MapReduce a specifici nodi del cluster, possibilmente gli stessi che mantengono nella memoria locale la porzione di dati dell'HDFS relativi al task. Il comportamento del JT può essere schematizzato così:

1. le applicazioni client sottopongono jobs al JobTracker;
2. il JobTracker chiede al NameNode (che gestisce l'HDFS) dove sono le locazioni dei dati;
3. il JobTracker trova i TaskTracker con slot di memoria disponibili che risiedono dove sono i dati o sono loro prossimi (come numero di *hop*) nella rete;
4. il JobTracker sottopone il job al TaskTracker scelto;
5. il JobTracker controlla che i TT inviino ping (*heartbeating*) ad una frequenza soddisfacente. In caso contrario considera il job del TT che non ha risposto come failed e lo rischedula assegnandolo ad un TT in buono stato con slots disponibili;
6. se un TaskTracker segnala al JT il fallimento di un job, il JobTracker può:
   - rischedulare il job e assegnarlo ad un altro task;
   - marcare lo specifico record che ha causato l'interrruzione dell'esecuzione come record da non processare e risottoporre il job;
   - inserire il TT in una blacklist.
7. il JobTracker aggiorna il proprio stato ogni volta che un job è terminato;
8. le applicazioni Client possono interrogare il JobTracker per ottenere informazioni sull'esecuzione del processo.

Com'è facile aspettarsi, il JobTracker è un point of failure per l'intero servizio MapReduce. In caso di fallimento del JT, tutti i job, completati o meno, vengono persi e l'esecuzione del lavoro complessivo viene fatta ripartire da un nuovo JT.

3.3.3 NameNode e DataNode

DataNode

Un DataNode (DN) mantiene dei dati dell'HDFS ed è un demone in servizio presso una macchina *slave*.

L'HDFS è composto preferibilmene da più di un DataNode su cui distribuisce i dati con ridondanza. All'avvio si connette al NameNode, da cui attende richieste di servizio, e termina quando il NN termina. Le applicazioni client possono comunicare direttamente col DataNode dopo che a questo sono state assegnate dei dati. In questo modo ad esempio il JobTracker decide quale task affidare ad un dato TaskTracker in funzione dei dati contenuti nel DataNode che risiede nella stessa macchina (se esiste) o in quelle vicine, di modo che l'esecuzione del processo MapReduce avvenga quanto più possibile vicino ai dati.

Esiste una comunicazione inter-DataNode, che avviene quando più DataNode mantengono gli stessi dati. I DataNode non hanno bisogno di memorizzare i dati secondo un qualche RAID > 0, in quanto la loro ridondanza è distribuita su più server invece che su più dischi dello stesso server.

Una configurazione ideale di un server *slave* è di avere un DataNode, un TaskTracker ed uno slot TaskTracker per CPU, permettendo così ad ogni processo che esegue un task di avere il 100% di tempo di CPU disponibile.

NameNode

Il NameNode è la componente principale dell'HDFS, mantiene il *directory tree* di tutti i file nel file system e mantiene le informazioni sulla locazione reale dei dati nella rete. Il servizio di NameNode non mantiene nessun dato.

Le applicazioni client invocano il NameNode ogni volta che vogliono scrivere, leggere, inserire o rimuovere un file dal filesystem, e in risposta ottengono una lista dei DataNode più rilevanti che contengono la risorsa richiesta.

Com'è facile notare, anche il NameNode, come il JobTracker, è un *Single Point of Failure* per il cluster. Nel caso cada il NameNode cade l'intero filesystem, a meno che non venga definito un **SecondaryNameNode** ospitato su un'altra macchina. Il *SecondaryNameNode* crea dei checkpoints del filesystem per consentire il rollingback del filesystem allo stato descritto nel checkpoint più recente.

La configurazione ideale per un NameNode è quella in cui:

- il demone server del NN gira su una macchina con molta RAM, fatto che consente di avere un DFS più grande e/o blocchi di dati più piccoli;
- nei file di configurazione sono presenti più di una directory per i metadati del sistema, ed ogni directory è un percorso verso un disco diverso dalle altre: nel caso di rottura di un disco il sistema non verrebbe danneggiato;
- è possibile verificare lo spazio disco disponibile al NameNode e nel caso si renda necessario, aggiungere nuovo spazio;
- il demone server del NN non gira su una macchina che già stà operando come JobTracker o TaskTracker.

3.4 Uso di Hadoop

Per eseguire programmi MapReduce in Hadoop si passa attraverso le fasi di:

1. configurazione del cluster;
2. diffusione del sistema hadoop nel cluster;
3. sviluppo e compilazione del programma MapReduce;
4. avvio dell'HDFS e dei processi MapReduce;
5. caricamento dei dati di input;
6. esecuzione del programma;
7. prelevamento dell'output.

3.4.1 Configurazione del cluster

Per configurare un cluster Hadoop si impostano i parametri all'interno dei file xml:

- *core-site.xml*;
- *mapred-site.xml*;
- *hdfs-site.xml*.

core-site.xml

In *core-site.xml* vengono impostati tutti i parametri di un generico ambiente di lavoro Hadoop. I parametri impostati sono quelli che generalmente si ritiene siano comuni a tutti gli scenari in cui si pensa di voler usare il framework Hadoop, ovvero quelli che non cambiano a seconda del tipo di applicazione e di cluster specifici di un caso d'uso particolare.

mapred-site.xml

In *mapred-site.xml* vengono definiti i parametri specifici di un particolare caso d'uso relativi all'esecuzione delle funzioni Map e Reduce, ovvero quei parametri che ineressano i processi JobTracker e TaskTracker. È possibile ridefinire parametri già impostati in *core-site.xml*: a runtime il sistema viene configurato secondo le impostazioni in *mapred-site.xml*, ed eventuali ridefinizioni degli stessi parametri in *core-site.xml* vengono ignorate.

hdfs-site.xml

In *hdfs-site.xml* vengono definiti i parametri specifici di un particolare caso d'uso relativi all'HDFS, ovvero quei parametri che ineressano i processi NameNode e DataNode. È possibile ridefinire parametri già impostati in *core-site.xml*: a runtime il sistema viene configurato secondo le impostazioni in *hdfs-site.xml*, ed eventuali ridefinizioni degli stessi parametri in *core-site.xml* vengono ignorate.

3.4.2 Diffusione del sistema Hadoop nel cluster

Per distribuire Hadoop sui nodi della rete, una volta terminata la fase di configurazione, è sufficiente caricare la cartella con i file binari e di configurazione in ogni macchina del cluster, collocandola all'interno di un percorso uguale per tutti. In ogni cartella, oltre ai file xml configurati nella fase precedente dovranno anche essere presenti i file ***masters*** e ***slaves***, in cui andranno scritti rispettivamente tutti gli indirizzi IP dei nodi *master* e di quelli *slave*, scritti uno per riga.

Quando il server *master* vorrà attivare i demoni TT e DN, instaurerà una connessione ssh con ogni *slave* e farà partire i processi. In mancanza di un servizio di DNS interno alla rete (scenario frequente all'interno di piccole reti locali) è importante che ogni *slave* risolva il nome dell'host master configurando opportunamente il proprio sistema operativo (ad esempio, in Unix si dovrà specificare il nome dell'host del *master* in */etc/hosts*).

3.4.3 Sviluppo e compilazione del programma MapReduce

Un programma MapReduce è un programma Java che implementa la classe ***Mapper*** in cui è definita la funzione *map* e la classe ***Reducer*** in cui è definita la funzione *reduce()*

Mapper

```
public static class AflMapper
 extends Mapper<Object, Text, Text, Text>{

   //Dichiarazione attributi di classe
   ...

   public void map(Object key, Text value,
    Context context)
     throws IOException, InterruptedException {
      //Definizione della funzione
      ...
      return;
   }
}
```

Reducer

```
public static class AFLReducer
 extends Reducer<Text, Text, Text, Text>{

   //Dichiarazione attributi di classe
   ...

   public void reduce(Text key, Iterable<Text> values,
    Context context)
    throws IOException, InterruptedException {
      //Definizione della funzione
      for (Text val:values){
          ...
      }
      ...
      return;
   }
}
```

I sorgenti di queste classi e quello contenente la funzione *main()* dovranno importare le librerie ***org.apache.hadoop.****.

Come già detto è possibile scrivere programmi anche nei linguaggi *C++* e *python* grazie al framework Hadoop Streaming, ma di seguito tratteremo solo la compilazione di programmi Hadoop MapReduce in *Java*.

Per compilare i sorgenti Java che fanno riferimento alle librerie *org.apache.hadoop.** è necessario indicare al compilatore *javac* il percorso per il file *hadoop-core-VERSIONE_HADOOP.jar* che le contiene.

Spesso un programma MapReduce P è composto da n sottoprogrammi MapReduce P_i, ognuno con le sue funzioni map m_i e r_i, che devono essere eseguiti sequenzialmente di modo che l'output in uscita al reducer r_i diventi l'input per m_{i+1} (per $0 < i < n$). Per fare ciò è necessario che P sia uno script, o comunque un programma "esterno" rispetto ai programmi P_i, che sequenzi l'esecuzione $P_1,\ P_2,\ ...,\ P_n$.

3.4.4 Avvio dell'HDFS e dei processi MapReduce

Prima di poter compiere qualunque operazione sul filesystem distribuito o qualunque esecuzione di programmi MapReduce, si devono avviare correttamente i demoni NameNode, DataNode, JobTracker e TaskTracker su tutti i nodi del cluster.

Per prima cosa è necessario avviare l'HDFS, attraverso l'esecuzione dello script *bash start-dfs.sh*. Questo script avvia i demoni DataNode sulle macchine il cui indirizzo è memorizzato nel file *slaves*, e NameNode sul server in *masters* tramite collegamenti ssh. Il programma hadoop *dfsadmin* permette di effettuare una diagnostica dei DataNode nella rete per verificare se si sono avviati correttamente, quanto spazio di memorizzazione offre ognuno ed il loro numero.

Se il filesystem distribuito si è avviato correttamente, si avviano i demoni TaskTracker sulle macchine indirizzate in *slaves* e JobTracker sul server in *masters*, eseguendo lo script *start-mapres.sh*. Da questo momento i server TaskTracker attendono l'arrivo di tasks in arrivo dal JobTracker, mentre questo è in attesa che le applicazioni client sottomettano jobs.

Sebbene sia preferibile procedere in quest'ordine nell'avvio dei demoni sui server, non esiste un vincolo che lo inponga. È implicito però notare che mentre il filesystem distribuito può essere acceduto in lettura e scrittura anche senza che siano ancora stati avviati i processi MapReduce JT e TT, non è possibile l'esecuzione di applicazioni MapReduce se non è prima stato avviato un HDFS.

3.4.5 Caricamento dei dati in input

Prima di eseguire un programma, dovranno essere caricati manualmente all'interno dell'HDFS i dati in input. Per fare ciò è necessario chiamare sul DFS il programma *put* per ogni file che contiene dati dell'input. Se si vuole che insiemi di dati siano letti sequenzialmente e non vengano divisi tra più *task* MapReduce, sono possibili due alternative:

- porre detti dati in uno stesso file di testo;
- comprimere i file contenenti detti dati in un unico file con estensione *.zip*.

Lo splitting dell'input è automatico, e generalmente, per grandi dimensioni dello stream in ingresso, ogni *split* ha una dimensione di circa 64 MB. La partizione dell'input in *splits* avviene suddividendo l'input in più files. Un solo file di input causerebbe l'esecuzione di un solo *task*, con un'elaborazione senz'altro non vantaggiosa rispetto alla normale computazione sequenziale.

Non esiste un modo unico per formattare l'input, ma questo dovrà essere formattato così come richiesto dal programma specifico del caso d'uso. Un mapper termina il suo *task* una volta esaurito il processamento del file in ingresso.

3.4.6 Esecuzione del programma MapReduce

Una volta avviati i server *master* e *slaves* e compilato un programma MapReduce P (o un set di programmi P_i da eseguire sequenzialmente), è possibile avviare l'esecuzione di questi. Per eseguire un programma compilato in bytecode Java *.class* o una collezione di sorgenti compilati *.jar* è sufficiente eseguirli come un qualunque programma Java, con la differenza che invece di chiamare

```
$java -jar applicazione.jar
   percorso_classe_con_main <parametri_applicazione>
```

si chiamerà il programma *hadoop*:

```
$hadoop jar applicazione_mapreduce.jar
    percorso_classe_con_main <parametri_applicazione>
```

Durante l'esecuzione verranno visualizzati i messaggi verso lo standard output e la percentuale di completamento delle operazioni complessive di *map* e di *reduce*, del tipo:

```
11/05/03 10:44:40 INFO mapred.JobClient:  map 0% reduce 0%
11/05/03 10:50:37 INFO mapred.JobClient:  map 50% reduce 5%
```

```
11/05/03 10:55:15 INFO mapred.JobClient:  map 70% reduce 20%
...
11/05/03 10:44:40 INFO mapred.JobClient:  map 100% reduce 100%
```

Al termine dell'esecuzione *hadoop* ritorna correttamente o con un codice d'errore.

3.4.7 Prelevamento dell'output

Gli output delle funzioni *reduce* vengono memorizzati ciascuno in un file di testo (con nome *part*_Id) nel filesystem distribuito. Per prelevare questi ed altri file dall'HDFS si invoca il sottoprogramma *get*. Il file richiesto è salvato nel percorso del disco locale specificato nel secondo parametro del programma. Ad esempio per prelevare la cartella `folder` occorre inserire da shell il comando:

```
~hadoop-dir/bin/$ ./hadoop dfs -get folder/ destination_dir_in_local_fs/
```

Nel caso in cui un output debba essere riciclato come input di un successivo programma MapReduce, non si deve estrarlo dal filesystem distribuito, ma basta specificarne il percorso all'interno di HDFS come input all'esecuzione del prossimo programma.

Capitolo 4

Implementazione di At First Look - Hitsura

Poiché l'obiettivo del presente lavoro è l'analisi dell'algoritmo At First Look implementato secondo il paradigma MapReduce, come prima cosa abbiamo avuto bisogno di implementare AFL come semplice programma Java composto da un solo processo e un solo thread, ovvero un programma che operi in modo sequenziale su uno stream in input di dimensioni relativamente contenute ($\approx$ 250 MB).

In questo capitolo daremo una descrizione ad alto livello della nostra implementazione dell'algoritmo in Java, il programma **Hitsura**, soffermandoci in particolare su alcuni accorgimenti che ci hanno permesso di migliorarne sensibilmente le prestazioni.

4.1 Funzionamento di Hitsura

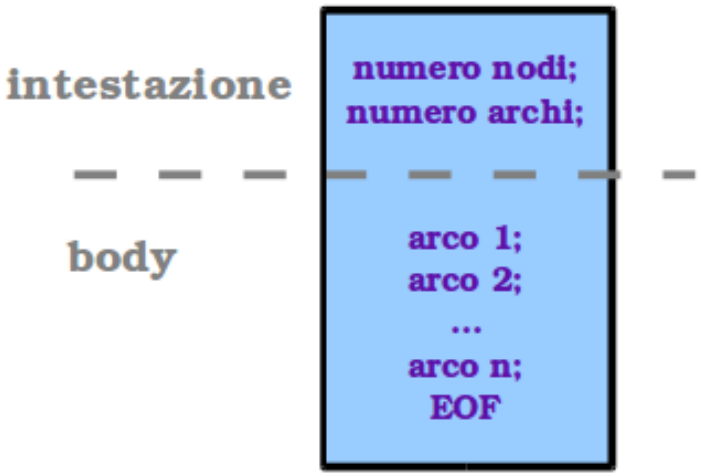
Per l'intera descrizione e analisi del comportamento di At First Look rimandiamo il lettore al capitolo 1. Nel nostro programma viene implementato l'algoritmo nell'approccio in streaming, ma ci siamo scostati dalla definizione originale di AFL relativamente a:

- la struttura del Navigational Sketch;
- le funzioni di *union()* e *find()*;
- la struttura dati che rappresenta la foresta.

In linea generale le modifiche che abbiamo apportato sostituiscono l'*union-by-size* con delle funzioni iterative che non tengono conto delle dimensioni degli alberi (per effettuare le rotazioni preliminari all'unione tra due alberi o sottoalberi non viene scelto quello con numero di nodi minore ma uno a

caso) **preferendo non aggiornare tutti i nodi che appartengono agli alberi coinvolti nell'unione**. Giustificheremo più avanti questo approccio ed i vantaggi prestazionali che ne derivano, analizzandone la correttezza.

4.1.1 L'input



l'input di Hitsura

L'input del programma è un file singolo in cui è descritto un grafo generico non orientato $G = (V, E)$, dove V è l'insieme dei nodi e E quello degli archi. La struttura di detto file è composta da

1. un'intestazione - righe 0 e 1;
2. un *body* - dalla riga 2 alla fine del file.

Intestazione: È composta da due righe contenenti ognuna un solo valore intero: nella prima è definito il numero di nodi presenti nel grafo ($\#V$), nella seconda il numero complessivo di archi ($\#E$).

Body: È formattato di modo che ogni riga sia costituita da una coppia di valori $< [v_1]\ [v_2] >$: $v_1, v_2 \in V$ rappresentante un arco $e : e \in E$ tra v_1 e v_2. Fanno eccezione le prime due righe del file, in cui è presente un solo valore: Possono essere presenti nel *body* anche righe vuote (utili nel caso si voglia poter leggere l'input come blocchi di record) o righe di commento. I

commenti sono righe che iniziano col carattere '#' seguito da note utili alla lettura "umana" del file. Sia le righe vuote che i commenti una volta letti non vengono processati dal programma.

L'input deve rispettare il vincolo sullo spazio di memoria richiesto dal Navigational Sketch di At First Look, ovvero, considerato il numero di nodi $n = \#V$ del grafo in input e M la quantità di memoria disponibile espressa in bit, deve valere

$$O(n \log n) < M.$$

Nota: Se nel grafo sono presenti archi riflessivi del tipo $< [v_i][v_i] >$ questi possono essere omessi, in quanto non contribuiscono a formare componenti connesse o biconnesse.

4.1.2 L'esecuzione

L'esecuzione del programma opera seguendo i passi:

1. allocazione del Navigational Sketch in memoria;
2. processamento degli archi in input;
3. stampa dell'output.

Allocazione del Navigational Sketch

Il programma legge le prime due righe del file in input:

- la prima riga contiene il numero $n = \#V$: viene allocato un array statico di n nodi rappresentante il Navigational Sketch in memoria;
- la seconda riga contiene il numero $m = \#E$: viene memorizzato m in una variabile locale.

Processamento degli archi

Viene avviato un ciclo *for* di m passi. Ad ogni passo viene letta una riga in input che viene sottoposta ad un parser. Il parser legge gli identificativi dei due nodi v_1 e v_2 e aggiunge un arco $e(v_1, v_2)$ al Navigational Sketch.

A seconda delle relazioni tra v_1 e v_2 nel NS possono verificarsi i casi:

- v_1 e v_2 appartengono a due alberi distinti: questo può accadere se almeno uno dei due nodi è letto per la prima volta oppure se non è ancora stato processato alcun arco tra un nodo dell'albero di v_1 ed uno dell'albero di v_2. In questo caso l'albero di v_2 viene ruotato invertendo tutte le relazioni padre-figlio dell'albero nel percorso dalla radice a v_2.

Alla fine di questa procedura v_2 è la radice del suo albero. Fatto questo vengono uniti i due alberi, impostando v_1 come nodo padre di v_2.

- v_1 e v_2 appartengono allo stesso albero: in questo caso distinguiamo altre due possibilità:
  - v_1 e v_2 sono nodi fratelli oppure sono uno il nodo padre dell'altro: il Navigational Sketch non viene modificato, in quanto o v_1 e v_2 sono già nella stessa componente biconnessa oppure è già stata catturata la relazione padre-figlio tra di essi.
  - v_1 e v_2 non sono nodi fratelli nè sono uno il padre dell'altro: in questo caso si considera il percorso $p(v_1, v_2)$. Il Navigational Sketch viene modificato aggiungendo ogni nodo n nel percorso tra v_1 e v_2 ed ogni nodo fratello di n ad un'unica catena di fratelli. Un solo nodo all'interno della catena viene scelto, estratto dalla catena e posto come nodo padre del nodo più a sinistra della catena.

Stampa dell'output

Quando il programma ha letto e processato l'ultima riga del file, il Navigational Sketch è completo e rappresentativo di tutte le componenti connesse e biconnesse dell'intero grafo G descritto dall'input, e viene stampato in un file di output.

Come dimostrato in 1.5, una volta che è stato completato il Navigational Sketch:

- ogni albero nel NS contiene tutti e soli i nodi di una stessa componente connessa in G;
- i nodi che nel NS appartengono ad una stessa catena di fratelli in unione col loro nodo padre sono tutti e soli i nodi di una specifica componente biconnessa in G;
- ad ogni arco nel NS tra un nodo padre f ed un nodo figlio s senza nodi fratelli corrisponde un arco $e(f, s)$ che è un *bridge* di G;
- ogni nodo non adiacente ad una catena di fratelli in NS è un *articulation point* in G.

4.2 Union-by-Size? No, grazie!

Abbiamo anticipato all'inizio della sezione 4.1 che l'implementazione di Hitsura si distingue dalla definizione originale di AFL riguardo alle operazioni di *union/find*, proponendo un approccio che cerca di leggere e aggiornare il minor numero possibile di nodi.

4.2.1 Costo della ricerca di percorsi nel Navigational Sketch

Come abbiamo visto in 4.1.2, quando viene prelevato dall'input un arco tra due nodi v_1, v_2 che nel Navigational Sketch appartengono a due alberi distinti oppure a due catene di fratelli distinte in uno stesso albero, si devono unire (***union***) due alberi o sottoalberi in uno unico. Per fare questo è necessario dapprima applicare le opportune trasformazioni degli alberi (o sottoalberi) di partenza, operazione che si può rivelare piuttosto costosa.

Consideriamo come esempio l'informazione *size* del modello *union-by-size*. In questo caso infatti vengono visitati e aggiornati tutti i nodi che appartengono agli alberi (sottoalberi) coinvolti per sovrascrivere l'informazione sulla dimensione della nuova componente connessa (biconnessa). È possibile ridurre drasticamente il numero di aggiornamenti necessari se le informazioni sul *size* non vengono scritte in ogni nodo ma solo nel nodo di riferimento dell'albero (il nodo radice) o sottoalbero (il nodo padre). Con un approccio di questo tipo, il costo dell'operazione non è più quello dell'aggiornamento di ogni nodo degli alberi (sottoalberi) da unire quanto piuttosto "solo" quello della visita di tutti i nodi direttamente coinvolti, ovvero tutti quei nodi che:

- o, nel caso di unione di più alberi, appartengono al percorso da $v \in \{v_1, v_2\} : \#albero_v = min\,(\#albero_{v_1}, \#albero_{v_2})$, ovvero al percorso tra quel nodo in $\{v_1, v_2\}$ il cui albero ha dimensione minore e la radice dell'albero stesso;
- o, nel caso di unione di sottoalberi, sono i nodi "n" che appartengono al percorso da v_2 a v_1 oppure nodi fratelli di un nodo n.

Anche attraverso questi accorgimenti però l'*union* rimane un'operazione potenzialmente costosa, in quanto un Navigational Sketch è una foresta di alberi generici. Il costo della ricerca di un percorso tra due nodi nel caso peggiore dovrà attraversare $n-1$ archi, dove n è il numero di nodi del grafo di ingresso G. Questo accade quando il NS è composto da un unico albero che ha una struttura in cui ogni nodo ha per padre un nodo che non ha un nodo fratello a destra. Ad esempio la ricerca in un NS siffatto della componente connessa del nodo foglia n_{worst}, ovvero il nodo più a destra della catena di fratelli più in profondità nell'albero, dovrà percorrere tutti e $n-1$ gli archi che separano n_{worst} dalla radice.

4.2.2 Il Navigational Sketch in Hitsura

Il programma è composto da tre classi Java che implementano At First Look:

- AFLNode;

- AncestorFlag;
- AtFirstLook.

Il Navigational Sketch di Hitsura è un array statico di n istanze della classe AFLNode, allocato in memoria all'inizio dell'esecuzione. Ogni nodo possiede gli attributi:

- *father*;
- *left_brother*;
- *right_brother*;
- *strongLeftBrother*;
- *strongRightBrother*;

Come si può notare, oltre alle informazioni necessarie alla descrizione della posizione del nodo nella foresta (gli attributi *father*, *left_brother*, *right_brother*, ne sono presenti alcune aggiuntive rispetto alla definizione originale di un Navigational Sketch in At First Look mentre sono assenti gli attributi sulle dimensioni della componente connessa e delle eventuali componenti biconnesse del nodo.

4.2.3 Find in Hitsura

Abbiamo detto che in un Navigational Sketch la ricerca di un percorso nel worst case può costare anche $n - 1$ dove n è il numero di nodi letti dal grafo. Per com'è definito At First Look, notiamo anche che modellando grafi con molti nodi senza memorizzare tutte le informazioni in ciascuno di essi, è necessario introdurre nodi di riferimento in cui queste informazioni sono custodite. Ad esempio, per identificare il nodo padre f di un nodo n è necessario risalire l'intera catena di fratelli di n. Poiché possiamo considerare che dopo l'inserzione di un certo numero di nodi nel NS questo tipo di operazione viene compiuta praticamente per ogni lettura di un nuovo *edge* dall'input, il tempo per la sua elaborazione aumenta considerevolmente a mano a mano che l'input viene consumato.
Sempre dal comportamento di AFL abbiamo:

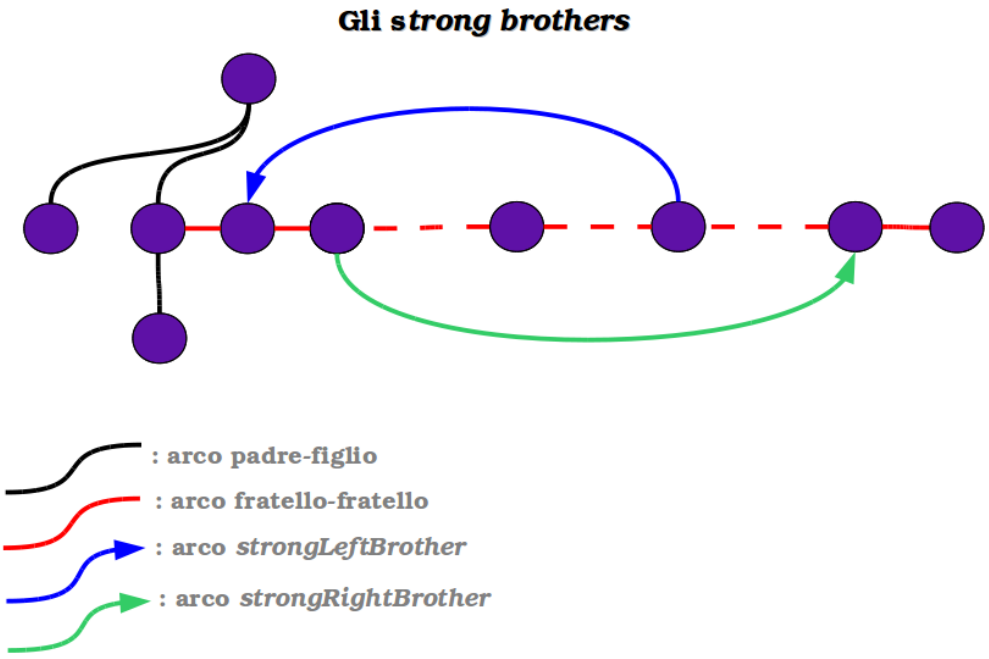
lemma 1: tutti i nodi n che appartengono ad uno stesso albero T del Navigational Sketch, corrispondente ad una stessa componente connessa CC del grafo in input G, dopo il processamento di i archi di G, apparterranno ad uno stesso albero $T' \supseteq T$ (corrispondente alla componente connessa $CC' \supseteq CC$ di G) al passo $i + 1$;

lemma 2: tutti i nodi n che appartengono ad uno stessa catena di fratelli B del Navigational Sketch, corrispondente ad una stessa componente biconnessa BCC nel grafo in input G dopo il processamento di i archi di G, apparterranno ad uno stessa catena di fratelli $B' \supseteq B$ (corrispondente alla componente biconnessa $BCC' \supseteq BCC$ di G) al passo $i + 1$;

A partire da questi lemmi osserviamo ancora che gli alberi e le catene di fratelli, che sono insiemi di nodi, possono al più aggregarsi e nessun nodo può venire più rimosso da essi una volta che ne fa parte.

Gli *Strong Brothers*

Poiché nella quasi totalità dei casi d'uso reali di At First Look il grafo G che si vuole analizzare è composto da componenti biconnesse con molti nodi al proprio interno, tipicamente si hanno Navigational Sketch con alberi relativamente poco profondi e molto "larghi", ovvero con catene di fratelli molto lunghe. Siccome le stesse catene di fratelli vengono lette molte volte (una catena deve essere risalita fino almeno al nodo padre per la ricerca della radice dell'albero, di un nodo padre, di un generico precorso tra nodi, ecc.), come anticipato in 4.2.2, gran parte del tempo di esecuzione è occupato dal "risalire" lungo queste catene. Abbiamo dunque inserito nella classe che modella il nodo (AFLNode) informazioni aggiuntive su nodi fratelli non contigui a sinistra e a destra (rispettivamente *strongLeftBrother* e *strongRightBrother*), che fungono da "scorciatoie" durante la ricerca.



I campi *strongLeftBrother* e *strongRightBrother* di un generico nodo n vengono sovrascritti solo "in lettura", cioè ogni volta che il nodo viene attraversato da una ricerca su nodi fratelli più distanti (a sinistra o a destra). Ad esempio, quando viene chiamata su n la funzione *getFather()*, questa viene rieseguita ricorsivamente, operando nel modo seguente:

- se n ha uno *strongLeftBrother* sl diverso da se stesso, $\Longrightarrow$ ritorna il valore $sl.getFather()$ dopo averlo usato per aggiornare sl;
- se $sl = n$ ma n è diverso dal suo *leftBrother* lb, $\Longrightarrow$ ritorna il valore $lb.getFather()$ dopo averlo usato per aggiornare sl;
- se la funzione $getFather()$ è stata chiamata ricorsivamente per almeno $k-1$ volte, alla k-esima la funzione aggiorna il campo *strongLeftBrother* di tutti i noti incontrati con l'identificativo di quello corrente e prosegue chiamando una nuova funzione sullo *strongLeftBrother* (*leftBrother*) del nodo corrente.
  In questo modo si impedisce che la ricorsività della funzione causi il termine del programma dovuto all'esaurimento dello spazio riservato in memoria al suo processo.

Il vantaggio di questo approccio è evidente, in quanto evita la rilettura dei nodi fratelli intermedi aggiornandone il minor numero possibile.

4.2.4 Abbandono di union-by-size

Abbiamo visto come dato un grafo G in input il suo Navigational Sketch sarà una foresta con alberi poco sviluppati in altezza (relazioni padre-figlio) e molto in larghezza (relazioni difratellanza).

A questo punto possiamo permetterci alcune osservazioni sui costi dell'operazione mediamente più costosa sul Navigational Sketch di $G = (V, E)$ (con $n = \#V, m = \#E$), ovvero la *find*, dopo avere introdotto l'utilizzo degli *strong brothers*:

- non viene rieseguita due volte sulle stesse porzioni di albero nel NS, per cui sebbene il suo *worst case* resta quello di una ricerca che attraversi $n - 1 = m$ archi, il suo verificarsi pagherà il costo dell'intero attraversamento una sola volta nel corso dell'elaborazione;
- la ricerca di percorsi all'interno di un albero richiede un costo in altezza che resta grossomodo costante (in realtà logaritmico con base grande) e trascurabile, ed in larghezza un costo ammortizzato sulla lunghezza b_i delle catene di fratelli.

Dopo questi accorgimenti quale miglioramento apporterebbe l'utilizzo del modello *union-by-size*? Ovvero esiste ancora un vantaggio nello scegliere il più piccolo albero (sottoalbero) da modificare in un Navigational Sketch tra due di essi che devono essere uniti? Sicuramente se occorresse manipolare l'informazione che risiede in ogni nodo delle strutture coinvolte nell'unione ci sarebbe un vantaggio notevole nel modificare solo la struttura minore. Per le osservazioni fatte però non abbiamo la necessità di aggiornare tutti i nodi, ma solo quelli che fino ad ora abbiamo chiamato *di riferimento* (la radice degli alberi e i *leftestBrother* per catene di nodi fratelli).

Dimostriamo come l'introduzione di *union-by-size* non apporti benefici relativamente ai due tipi di unione previsti in At First Look:

1. l'unione di due alberi tra loro, ovvero l'unione di due componenti connesse → funzione **CCUnion()**:
   richiede la rotazione di uno dei due alberi con la sostituzione del nodo radice (funzione *CCUnionSetup()*);

2. l'unione di più catene di fratelli, ovvero l'unione di due o più componenti biconnesse di nodi → funzione **BCCUnion()**:
   richiede l'apposizione di una catena di nodi fratelli o nodi singoli in coda ad altri nodi o catene di nodi fratelli (funzione *BCCUnionSetup()*).

Union-by-Size & CCUnion

Per predisporre l'unione di due alberi della foresta del Navigational Sketch del grafo $G = (V, E)$ (operazione *CCUnionSetup*), durante il processamento di un arco in input e del tipo $e =< v_1, v_2 >$, con $v_1, v_2 \in V \wedge albero\,(v1) \neq albero\,(v2)$, si deve ruotare uno dei due alberi di modo da rendere radice il nodo adiacente a e. Ad esempio, volendo ruotare l'albero di v_2, si invertiranno tutte le relazioni padre-figlio che vanno da v_2 fino alla radice dell'albero.

Il costo di quest'operazione preliminare all'unione è proporzionale alla profondità rispetto alla radice del nodo v adiacente ad e dell'albero che si intende modificare, ovvero un costo logaritmico rispetto al *size* dell'albero, con, come base approssimativa, il numero medio di figli di un nodo (tipicamente alto).

Supponiamo di adottare il modello *union-by-size*. Poiché l'altezza media h degli alberi è molto minore di n e in generale piuttosto ridotta, alberi con *size* molto differenti avranno altezze comunque simili (perché logaritmiche rispetto al *size*). Quindi, nella scelta dell'albero da ruotare tra due alberi distinti CC_i e CC_j, *union-by-size* apporta un miglioramento, seppur minimo, sul tempo di elaborazione, quantificabile come

$$O\log(sizeCC_j) - O\log(sizeCC_i)$$

(dove $size(CC_i) < size(CC_j)$). Per memorizzare il *size* del nuovo albero però occorre comunque risalire anche lungo la genealogia dell'altro nodo adiacente ad e fino alla radice dell'albero con dimensione maggiore. È facile convincersi che questa operazione annulli il vantaggio introdotto dalla scelta dell'albero da ruotare in funzione del *size*.

Union-by-Size & BCCUnion

L'unione di nodi ad un'unica catena di fratelli, ovvero ad un'unica componente biconnessa, è causata dal processamento di un arco $e = (v_1, v_2)$ tale che v_1 e v_2 appartengono ad uno stesso albero ma non ad una stessa catena di fratelli, e richiede:

1. una ricerca del *common ancestor* di entrambi v_1 e v_2;
2. la fusione dei nodi e delle componenti biconnesse incontrate nel percorso $p(v_1, v_2)$ in un'unica componente biconnessa.

La prima operazione si comporta indifferentemente sia se stiamo utilizzando *union-by-size* sia altrimenti, per cui non la analizzeremo. Notiamo però che dopo averla compiuta non ripercorreremo una seconda volta gli stessi percorsi da v_1 e v_2 verso il nodo *ancestor*, ma salteremo le catene di fratelli. Resta quindi da pagare solo il costo della lettura dei percorsi dai due nodi verso il *common ancestor* ed il costo relativo all'accodamento di catene di nodi fratelli o nodi singoli a catene di fratelli già esistenti. Per fare questo si utilizzeranno solo i nodi di riferimento delle eventuali catene: questo rende di fatto inutile conoscere la dimensione effettiva della catena di nodi fratelli *BCCSize*, in quanto l'unione di una catena ad un altra è effettuata operando solo sui nodi alle estremità destra e sinistra di questa. Anche in questo caso dunque l'introduzione del modello *union-by-size* non risulta vantaggiosa.

4.3 Il codice di Hitsura

In questa sezione daremo una descrizione ad alto livello delle porzioni principali del codice di Hitsura. Per una visione più dettagliata rimandiamo alla lettura del codice di Hitsura. Il programma Hitsura è composto complessivamente da quattro classi, le prime tre per l'implementazione di At First Look ed una per l'analisi del tempo di esecuzione:

1. *AtFirstLook.Java*;
2. *AFLNode.java*;

3. *AncestorFlag.java*;

4. *DateUtils.java*.

4.3.1 *AtFirstLook.java*

La classe *AtFirstLook.java* contiene le strutture dati e le funzioni necessarie:

- al mantenimento in memoria un Navigational Sketch nel corso dell'elaborazione;
- alla lettura dello stream in input;
- alla stampa dell'output sullo standard output o su un file in output.

Navigational Sketch

Tutti gli attributi ed i metodi principali dell'intera classe sono quelli relativi al mantenimento e aggiornamento del NS in memoria.

: Un oggetto *AtFirstLook* ha per attributi:

- un array statico `private AFLNode[] forest` che alloca in memoria i nodi *AFLNode* del Navigational Sketch;
- un intero `private int size` inizializzato con il numero complessivo dei nodi del grafo in input nel caso sia stato specificato come parametro, oppure a un milione nel caso non siano stati specificati parametri.

Metodi per la ricerca e l'inserzione dei nodi:

- `public AFLNode search(int id)`, restituisce il nodo, se esiste, corrispondente all'id specificato nel parametro, altrimenti ritorna `null`.
- `public int add_node(int id)`, inserisce un nuovo nodo, se non è già presente, all'interno del Navigational Sketch.

Metodi per le operazioni di *union*:

- `private AFLNode commonAncestorSearch(AFLNode and, AFLNode bnd, AncestorFlag brotherFlag)` ritorna il *least common ancestor - LCA* tra i nodi `and` e `bnd` passati per parametri, con un flag `brotherFlag` di tipo AncestorFlag che riporta informazioni sull'eventuale relazione di fratellanza tra i nodi ancestor di un determinato livello.

- `private void ccUnionSetup(AFLNode nd)` provvede a ruotare l'albero del nodo specificato nel parametro `nd`, invertendo tutte le relazioni padre-figlio da `nd` fino al nodo radice. Al termine dell'operazione la nuova radice dell'albero è `nd`.

- `private void bccUnionSetup(AFLNode and,AFLNode bnd)` fonde tra loro tutti i nodi e le componenti biconnesse "toccati" dal percorso che va dal nodo `and` ed il nodo `bnd` specificati come parametri, in un'unica catena di nodi fratelli.

Lettura dell'input

Metodi per la lettura dell'input:

- `public void add_edge(String edge)`, è la funzione che si occupa di gestire il corretto processamento di un arco in input. Ognuno di questi viene letto dallo stream in input in modo sequenziale e passato come parametro alla funzione. *add_edge* fa il parsing della stringa corrispondente all'arco ottenendo gli identificativi numerici dei due nodi, ed aggiorna correttamente il NS.

Stampa dell'output

Metodi di stampa dell'output:

- `public void printNavigationalSketch()` stampa i campi di tutti i nodi del Navigational Sketch sullo standard output.

- `public void printFileNavigationalSketch(PrintStream out)` stampa i campi di tutti i nodi del Navigational Sketch sul PrintStream `out` passato per parametro.

4.3.2 *AFLNode.java*

Questa classe modella un nodo del Navigational Sketch di At First Look. Offre i metodi per la ricerca dei percorsi all'interno del Navigational Sketch, che come si è già visto è strutturato attraverso le relazioni tra i nodi da cui è formato.

AFLNode

Un oggetto *AFLNode* possiede i parametri:

- `private int node_id`: un intero che identifica univocamente il nodo;

- `private AFLNode father`: il nodo padre. Questo campo è posto a `null` se il nodo è collocato in mezzo ad una catena di nodi fratelli (ovvero fa parte di una componente biconnessa);

- `private AFLNode left_brother`: l'adiacente nodo fratello sinistro se ne esiste uno, altrimenti il nodo corrente stesso;
- `private AFLNode right_brother`: l'adiacente nodo fratello destro se ne esiste uno, altrimenti il nodo corrente stesso;
- `private AFLNode strongLeftBrother`: un nodo fratello a sinistra nella catena di nodi fratelli ma non adiacente al nodo corrente;
- `private AFLNode strongRightBrother`: un nodo fratello a destra nella catena di nodi fratelli ma non adiacente al nodo corrente;
- `private Boolean controlled`: un valore che segnala se un nodo è già stato letto all'interno della sua componente biconnessa;
- `private Boolean inBcc`: un valore che segnala se la componente biconnessa del nodo corrente è già stata letta.

Metodi per la ricerca di percorsi nel Navigational Sketch

- `protected AFLNode getFather()`: ritorna il nodo padre risalendo ricorsivamente la catena dei nodi fratelli;
- `protected AFLNode getCc()`: ritorna il nodo radice dell'albero del nodo corrente, rappresentativo dell'intera componente connessa;
- `protected AFLNode getBcc()`: ritorna il nodo più a sinistra della catena di nodi fratelli, rappresentativo di un'intera componente biconnessa del nodo corrente;
- `protected AFLNode getStrongLeftBrother()`: ritorna il nodo più a sinistra della catena di nodi fratelli;
- `protected AFLNode getStrongRightBrother()`: ritorna il nodo più a destra della catena di nodi fratelli;
- `protected AFLNode getStrongLeftBrotherRaw()`: ritorna il nodo fratello sinistro non contiguo memorizzato nel campo `strongLeftBrother`;
- `protected AFLNode getStrongRightBrotherRaw()`: ritorna il nodo fratello destro non contiguo memorizzato nel campo `strongRightBrother`.

Altri metodi

- `protected int getDepth()`: ritorna il numero di archi del tipo padre-figlio che separano il nodo corrente dal nodo radice del suo albero;

- `protected Boolean hasLeftBrother(AFLNode target_node)`: ritorna true se tra i nodi a sinistra nella catena di fratelli rispetto al nodo corrente è presente il nodo `target_node` passato come parametro;
- `protected Boolean hasRightBrother(AFLNode target_node)`: ritorna true se tra i nodi a destra nella catena di fratelli rispetto al nodo corrente è presente il nodo `target_node` passato come parametro.

4.3.3 *AncestorFlag.java*

La classe modella una struttura con informazioni sulle relazioni di fratellanza di nodi *ancestors* di due nodi di partenza.

Gli attributi di un AncestorFlag:

- `private int flag`: il valore che assume dipende dalla relazione di fratellanza tra due antenati dei nodi di partenza alla stessa profondità nell'albero:
  - **0** se i due *ancestors* non sono fratelli tra loro;
  - **1** se i due *ancestors* sono il primo un fratello a sinistra del secondo;
  - **2** se i due *ancestors* sono il primo un fratello destro del secondo.
- `private int depth`: la profondità alla quale si stanno analizzando i due *ancestors*;
- `private int adepth`: la profondità nell'albero del primo nodo di cui si vogliono analizzare gli *ancestors*;
- `private int bdepth`: la profondità nell'albero del secondo nodo di cui si vogliono analizzare gli *ancestors*;
- `private int diff`: la differenza tra le profondità nell'albero rispettive dei due nodi di partenza.

Capitolo 5

Hitsura su Hadoop - HitSoop

HitSoop è l'oggetto di questa tesi, ovvero l'implementazione di At First Look secondo il paradigma Map Reduce. La differenza sostanziale di HitSoop con At First Look sta nel fatto di non essere un programma composto da un unico processo sequenziale sullo stram in input, ma un programma distribuito su un cluster di macchine che eseguono un set di processi paralleli e indipendenti, ognuno con una porzione casuale di input da elaborare. La difficoltà del problema risiede nel dover scomporre l'algoritmo nelle due fasi del paradigma MapReduce, in cui :

1. durante l'esecuzione dei Mapper vengono elaborate in modo esaustivo porzioni distinte dell'input complessivo e viene emesso per ognuna di esse un output intermedio che ne rappresenta il Navigational Sketch;

2. durante l'esecuzione dei Reducer vengono uniti tutti i risultati intermedi (contenenti ognuno un Navigational Sketch parziale) in un unico Navigational Sketch rappresentativo dell'intero stream in input.

In questo capitolo vedremo la struttura generale del programma MapReduce su Hadoop, come viene processato l'input, come vengono prodotti gli output intermedi, la loro correttezza sia secondo l'algoritmo che relativamente al paradigma MapReduce, la funzione *Reduce()*, la correttezza generale dell'output, l'esecuzione con i relativi tempi di elaborazione ed una breve descrizione del codice.

5.1 Struttura di HitSoop

HitSoop è un programma MapReduce per Hadoop scritto in Java. Utilizza in massima parte le funzioni e le classi di Hitsura, modificandone e aggiungendone alcune. Come per Hitsura, l'input di HitSoop è un grafo generico G descritto da uno stream molto grande, mentre l'output è un Navigational Sketch che descrive G. Continuiamo a riferirci all'input di HitSoop con la

parola *stream* sebbene non stiamo rispettando pedissequamente la sua definizione nella teoria del *classical streaming*: infatti, nel *classical streaming*, lo *stream* in input può essere letto solo sequenzialmente mentre nel nostro caso verrà sezionato in più porzioni lette parallelamente.

5.1.1 Architettura MapReduce di HitSoop

HitSoop calcola il Navigational Sketch di G in un unico ciclo MapReduce. L'architettura MapReduce che abbiamo implementato prevede l'esistenza di $m > 0$ Mappers ed un solo Reducer.

Sia i Mapper che il Reducer condividono lo stesso codice. Sebbene la funzione *Reduce()* fa uso delle stesse funzioni di *Map()* (che sono quelle usate da Hitsura) più quelle relative all'elaborazione di più Navigational Sketch, ogni Mapper esegue anche una funzione *Combiner()* identica a *Reduce()*.

5.1.2 Il ciclo MapReduce

All'avvio del ciclo, l'input I è suddiviso in n splits (per i concetti di *split*, *task*, ed altri rimandiamo il lettore a 2.2) corrispondenti ad n tasks e rappresentativi di porzioni I_i tali che $\sum_1^n I_i = I$. Ogni task viene sottoposto ad un processo TaskTracker (che adesso si comporta da processo Mapper) che ne calcola il Navigational Sketch. Quando tutto lo split è stato elaborato, il Mapper emette come output il Navigational Sketch NS_i di I_i. Il Reducer riceve gli NS_i di tutti gli split, li unisce in un NS unico, rappresentativo dell'intero grafo G, e lo scrive nel file di output definitivo.

Nel caso in cui il numero di tasks t è maggiore del numero di TaskTracker (ovvero del numero m di Mappers), la funzione *combiner()*, che condivide il codice della funzione *reduce()*, unisce i NS prodotti come output intermedio dal Mapper in un unico NS intermedio. Il Reducer riceve così al più m NS intermedi invece di t.

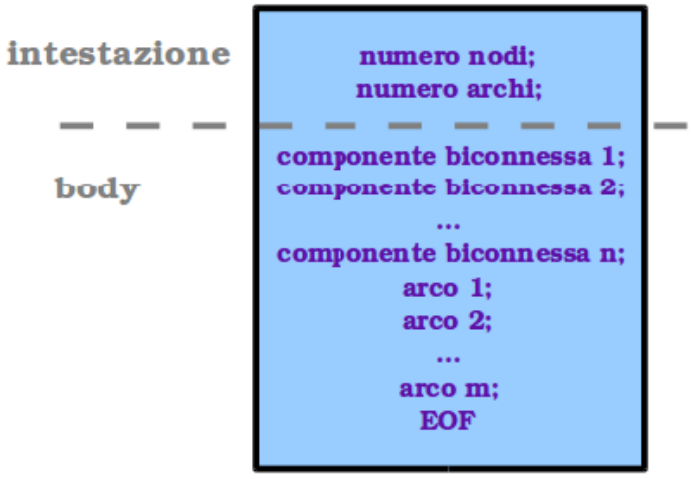
5.2 La funzione *Map()*

Il comportamento della funzione *Map()* è identico a quello del programma Hitsura, ad eccezione della stampa dell'output che in *Map()* deve rispettare i requisiti di Hadoop MapReduce. Non ripetiamo la descrizione di come viene creato un Navigational Sketch At First Look del grafo in input, rimandando il lettore all'analisi del comportamento di Hitsura in 4.1.

5.3 Le funzioni *Combine()* e *Reduce()*

Poiché *Combine()* e *Reduce()* sono di fatto la stessa funzione, descriviamo il comportamento di entrambe riferendoci a *Reduce()*, anticipando che ogni osservazione che faremo per *Reduce()* sarà identicamente valida per *Combine()*.

5.3.1 L'input di *Reduce()*



l'input di Hitsoop

L'input di *Reduce()* è un set di coppie $< chiave,\ valore >$ in cui come abbiamo visto il campo *chiave* è uguale per tutti gli output intermedi (la stringa `grafo`), mentre il campo *valore* di ogni coppia è una stringa suddivisa in due parti:

1. nella prima parte sono riportate tutte le componenti biconnesse del grafo rappresentato dal file in ingresso f_i, ottenute risalendo le catene di nodi fratelli in $NS_i = Map(f_i)$.

   Ogni componente biconnessa è rappresentata con una riga che inizia con l'identificativo numerico del nodo padre della catena seguito da tutti gli identificativi dei nodi della catena di fratelli. Uno spazio separa tra loro identificativi contigui distinti, ogni riga rappresenta una ed una sola componente biconnessa;

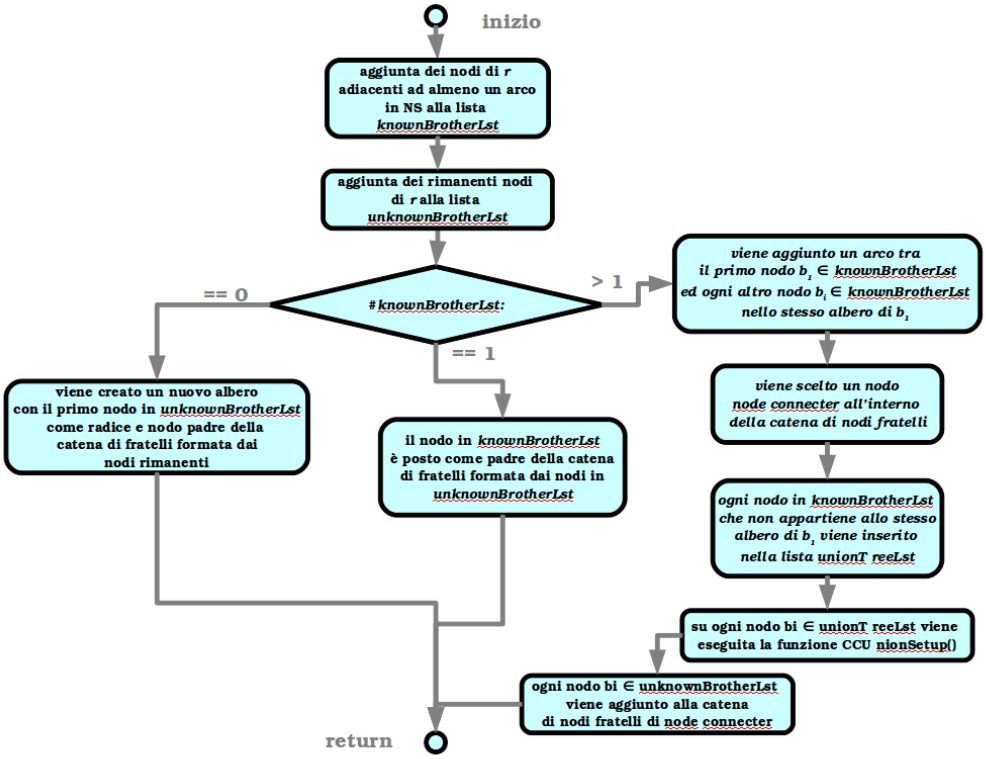
2. nella seconda parte sono riportati tutti gli archi semplici (quelli che in At First Look sono descritti come archi *solid*), ovvero tutti quegli

archi che modellano una relazione padre-figlio tra due nodi v_1 e v_2 in cui nè v_1 nè v_2 è in una catena di fratelli.

Ogni arco del tipo descritto viene riportato nell'output come una nuova riga con al suo interno gli identificativi numerici dei due nodi separati da uno spazio. Ogni riga di questo tipo ha solo due argomenti, righe distinte rappresentano archi distinti. Com'è facile notare la seconda parte di *valore* è, sia nel formato che nella semantica, uguale all'input di Hitsura.

5.3.2 Funzionamento di *Reduce()*

Reduce() fa uso delle stesse strutture e funzioni di Hitsura, ma a differenza di questo deve gestire l'aggiunta al Navigational Sketch di catene di fratelli già costruite da *Map()*. Senza ripeterci, descriviamo come viene processata la prima parte dei campi *valore* delle coppie in input tralasciando la seconda parte di cui abbiamo già visto l'elaborazione in Hitsura (4.1). Di seguito faremmo riferimento indistintamente ad una catena di nodi fratelli come una componente biconnessa e a un albero come componente connessa.



Reduce() legge ogni riga r di ogni *valore* e la passa come parametro `brotherList` della funzione `public void add_brother_list (String brotherList)`: se r ha un numero di argomenti $argc > 2$ allora r contiene una intera catena di nodi fratelli $Blist$. Ogni nodo $b_i \in Blist$ viene aggiunto in modo esclusivo:

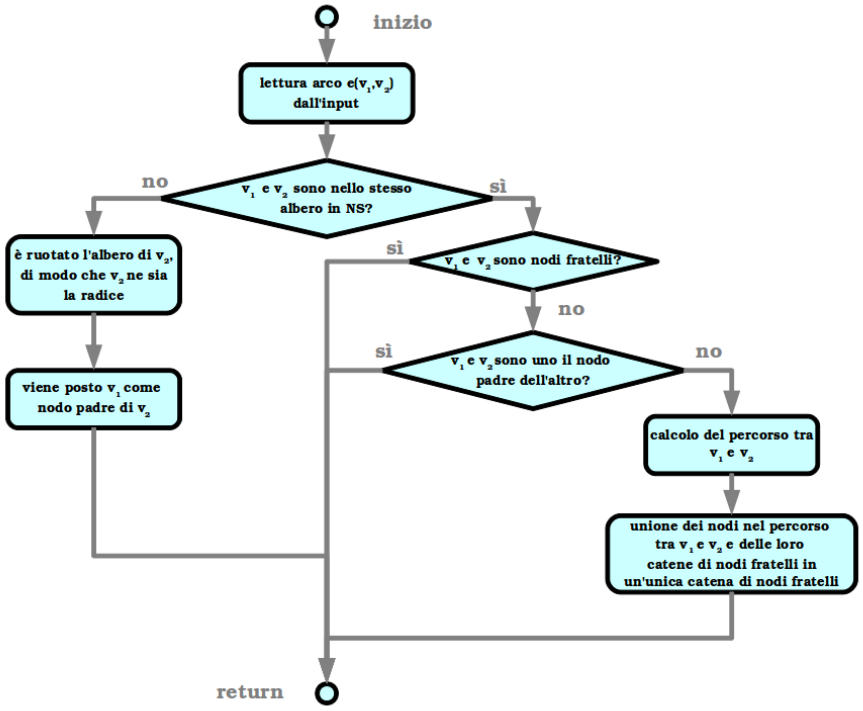
- alla lista $knownBrotherLst$ se v_i è già presente nel Navigational Sketch;
- alla lista $unknownBrotherLst$ altrimenti.

Possono verificarsi tre casi:

1. $\#knownBrotherLst = 0$: viene creato un nuovo albero con il primo nodo b_1 di $Blist$ come radice e aggiunto il resto della catena di nodi fratelli (da b_2 a b_last) ponendo come nodo padre b_1;
2. $\#knownBrotherLst = 1$: l'unico nodo già presente nel Navigational Sketch b_{known} viene posto come padre di una catena di nodi fratelli formata dai nodi $b_i : b_i \in unknownBrotherLst$;
3. $\#knownBrotherLst > 1$:
   (a) viene aggiunto un arco tra il primo nodo $b_1 \in knownBrotherLst$ ed ogni altro nodo in $b_i \in knownBrotherLst$ nello stesso albero di b_1, costruendo così una componente biconnessa che comprende tutti i nodi del percorso tra i nodi dell'arco con le relative catene di fratelli;
   (b) dalla componente biconnessa così formatasi viene scelto un nodo *node_connecter* all'interno della catena di fratelli (ovvero un nodo che non sia il nodo padre della catena);
   (c) ogni nodo $b_i \in knownBrotherLst$ che non appartiene allo stesso albero di b_1 viene inserito nella lista $unionTreeLst$;
   (d) su ogni nodo $b_i \in unionTreeLst$ viene eseguita la funzione $CCUnionSetup()$ se non appartiene ancora alla componente connessa di *node_connecter*, altrimenti viene aggiunto un arco tra b_i e *node_connecter*;
   (e) ogni nodo $b_i \in unknownBrotherLst$ viene aggiunto alla catena di nodi fratelli di *node_connecter*.

Al termine della funzione la componente biconnessa B passata come parametro risulta inserita in una componente biconnessa del Navigational Sketch di HitSoop che include anche tutte le componenti biconnesse dei nodi $b_i \in B$.

5.4 Prova di correttezza



Come abbiamo fatto per At First Look in 1.5, proviamo la correttezza di HitSoop dimostrando che dopo ogni chiamata della funzione *add_brother_list*() valgano gli invarianti:

1. un albero nel *Navigational Sketch* è una componente connessa;
2. tutti i nodi che sono inclusi in una catena di nodi fratelli B oppure il nodo padre di una catena di nodi fratelli B appartengono ad una stessa componente biconnessa;
3. ogni arco padre-figlio tra due nodi in cui il nodo figlio non appartiene ad una catena di nodi fratelli è un *bridge*;
4. ogni nodo v, adiacente ad un set di n archi, che verifichi $n > 2$, oppure, se è un nodo padre, $n >= 2$, è un *articulation point*;
5. al termine dell'esecuzione di *Reduce()* il Navigational Sketch NS, ottenuto da un input costituito da un set di Navigational Sketch $NS_i = Map\,(G_i)$ dove G_i è un generico grafo non orientato, è equivalente al Navigational Sketch $NS_G = Map\,(\sum_{i=1} G_i)$.

5.4.1 Alberi e componenti connesse

Teorema 1: *Ad ogni passo un albero nel* Navigational Sketch *è una componente connessa.*

Dimostrazione: poiché abbiamo già dimostrato in 1.5.2 la correttezza del teorema per il processamento di righe dell'input rappresentanti semplici archi del grafo da modellare, proviamo la correttezza del teorema anche per righe che modellano componenti biconnesse dimostrandola per induzione sul numero di righe processate:

Nel *caso base* non sono ancora state processate righe dell'input ed il Navigational Sketch è un grafo privo di archi in cui ogni nodo è un albero singleton. Il teroema è valido in quanto il Navigational Sketch modella un grafo senza archi, in cui ogni nodo è una componente connessa a sè stante.

Assumendo verificata la correttezza del teorema al passo k, dimostriamo la sua validità al passo $k+1$. Quando viene processata una nuova componente biconnesssa B del grafo in input G sappiamo che possono verificarsi tre casi a seconda del numero di nodi in B che sono già stati coinvolti in qualche relazione con altri nodi del NS:

1. nessun nodo $b_i \in B$ è in relazione con alcun nodo del NS: la funzione crea un albero che ha per radice il primo nodo presente in B, ponendolo come nodo padre della catena di fratelli composta dai restanti nodi in B. Il *teorema 1* vale, in quanto tutti ii nodi b_i non sono adiacenti ad altri archi in G se non quelli che formano la componente biconnessa B, pertanto non esiste alcun percorso del tipo $percorso\,(b_i, b_j) : b_i \in B \wedge b_j \notin B$, da cui il nuovo albero formatosi con tutti e soli i nodi di B è anche una componente connessa completa di G;

2. un solo nodo $b \in B$ è in relazione con almeno un nodo di NS: b diventa il nodo padre della catena di nodi fratelli costituita da tutti i nodi $b_i \in B : b_i \neq b$. Il *teorema 1* vale, in quanto non vengono creati nuovi alberi ed i nodi $b_i \neq b$, che appartengono alla stessa componente biconnessa di b, vengono aggiunti alla stessa componente connesssa di b;

3. più nodi $b_i \in B$ sono in relazione con almeno un nodo di NS:

   (a) viene creata una componente biconnessa unica composta da tutti i nodi $b_i : b_i \in B$ adiacenti ad almeno un arco in NS e appartenenti alla stessa componente connessa di b_1, dove b_1 è il primo nodo in B, più tutti i nodi che coinvolti nei percorsi $percorso\,(b, b_i)$ nel NS. Il *teorema 1* non è più valido in quanto i nodi $b_j : b_j \neq b_i \wedge$

$b_j \in NS$ possono appartenere ad alberi distinti pur essendo nodi appartenenti alla stessa componente biconnessa, e pertanto gli alberi in NS possono non essere più rappresentativi di componenti connesse;

(b) vengono ruotati gli alberi dei nodi $b_i : b_i \in NS \wedge b_i.getCC() \neq b.getCC()$ di modo da rendere ogni b_i la radice del proprio albero; vengono poi aggiunti i nodi b_i alla catena di nodi fratelli della componente biconnessa di b creata al passo precedente. A questo punto tutti gli alberi con più di un nodo che contengono nodi appartenenti a B vengono uniti in unico albero T. Il *teorema 1* non è verificato in quanto esiste un albero distinto da T per ogni nodo b_i che non è ancora stato messo in alcuna relazione con un nodo di NS;

(c) ogni nodo $b_i \in B$ che non è in relazione con alcun altro nodo in NS viene aggiunto alla componente biconnessa di b.

A questo punto tutti gli alberi dei nodi appartenenti a B sono inclusi nell'albero T. Poiché i nodi di una componente biconnessa devono appartenere necessariamente alla stessa componente connessa, il *teorema 1* è valido.

5.4.2 Nodi fratelli e componenti biconnesse

Teorema 2: *tutti i nodi che sono inclusi in una catena di nodi fratelli B oppure il nodo padre di una catena di nodi fratelli B appartengono ad una stessa componente biconnessa.*

Dimostrazione: Dimostriamo anche questo teorema per induzione sul numero di righe lette dall'input relativamente all'aggiunta di una nuova componente biconnessa B, senza ripeterci nel trattare l'aggiunta di semplici archi tra due nodi, la cui correttezza è già stata dimostrata in 1.5.3.

Nel *caso base* non sono ancora state lette righe dall'input, ed il grafico modellato dal NS è privo di archi tra i nodi. Il NS è composto dai nodi di G e da nessun arco. Non sono presenti pertanto catene di nodi fratelli ed il *teorema 2* è banalmente verificato.

Supponiamo di aver dimostrato la validità del teorema fino a quando il programma abbia processato k righe dell'input, verifichiamo se tale validità è ancora preservata dopo aver elaborato una riga $k+1$ del tipo componente biconnessa ($\#argomenti\,(riga_{k+1}) > 2$) nei tre casi:

1. nessun nodo $b_i \in B$ è in relazione con alcun nodo del NS: viene creato un nuovo albero nel NS con il primo nodo $b_1 \in B$ posto come radice e

come nodo padre della catena di nodi fratelli composta da tutti i nodi $b_j : b_j \in B \wedge b_j \neq b_1$. Il nuovo albero così formato è composto da tutti e soli i nodi che appartengono alla nuova componente biconnessa descritta dalla riga $k+1$, e, sempre per come abbiamo costruito l'albero, il *teorerma 2* risulta verificato;

2. un solo nodo $b \in B$ è in relazione con almeno un nodo di NS: tutti i nodi $b_i : b_i \in B \wedge b_i \neq b$ sono inseriti in una nuova catena di nodi fratelli con nodo padre b. Il *teorema 2* vale in quanto per come abbiamo costruito la nuova catena di fratelli essa è composta da tutti e soli i nodi di B meno b che ne è il nodo padre;

3. più nodi $b_i \in B$ sono in relazione con almeno un nodo di NS:

   (a) vengono unite in un'unica catena di nodi fratelli tutte le catene di nodi fratelli ed i singoli nodi che sono coinvolti in un percorso dal primo nodo di B, b_1, ad ognuno dei nodi $b_j : b_j \in B \wedge b_i.getCC() == b_1.getCC() \wedge b_i \neq b_1$. Consideriamo detti nodi b_j: questi sono quei nodi in B che già appartengono allo stesso albero di b nel NS. Per poterli unire in una stessa componente biconnessa come richiesto dal processamento di B, come quando in At First Look viene aggiunto un arco tra due nodi nello stesso albero, è necessario unire anche le componenti biconnesse (se esistono) di tutti i nodi coinvolti nel percorso da b a b_j. Tale unione è effettuata appendendo in coda ad una catena di nodi fratelli che partecipa del percorso $P = Percorso(b, b_j)$ tutte le altre catene di nodi fratelli e tutti i nodi coinvolti in P, ottenendo così una nuova catena di fratelli composta da tutti i nodi e le relative componenti biconnesse "toccati" da P appesa in coda ad una catena preesistente sempre tangente P. Abbiamo visto in 1.5.3 che ogni volta che viene formata un sottoalbero T' formato da un nodo padre ed una catena di nodi fratelli nel modo sopra descritto, per la definizione di componente biconnessa (1.1.2), T' contiene tutti e soli i nodi di una singola componente biconnessa verificando così il *teorema 2*.

   (b) vengono ruotati gli alberi in cui sono presenti nodi $b_i : b_i \in B \wedge b_i.getCC() \neq b_1.getCC()$: tali nodi b_i vengono posti come nodi radice dei rispettivi alberi se ancora questi non sono stati uniti, e poi aggiunti alla catena di fratelli della componente biconnessa di b_1. L'insieme composto dalla catena di nodi fratelli di b_1 e dal suo nodo padre è popolato da tutti e soli i nodi della componente biconnessa di b_1.

   (c) ogni nodo $b_i \in B$ che non è adiacente a nessun arco di NS viene aggiunto alla componente biconnessa di b_1, venendo aggiunto

in coda alla catena di nodi fratelli. Il *teorema 2* è banalmente verificato.

Al ritorno della funzione *Reduce()* il teorema è verificato.

5.4.3 Archi padre-figlio e *bridges*

Teorema 3: *Ogni arco padre-figlio tra due nodi in cui il nodo figlio non appartiene ad una catena di nodi fratelli è un bridge.*

Dimostrazione: Dimostriamo questo teorema per induzione sul numero di righe in input lette del tipo $r : argc(r) > 2$, basandoci ancora su quanto già visto in 1.5.4 per quanto riguarda le righe $r : argc(r) == 2$:

Nel *caso base* non esistono ancora archi nel NS ed il teorema è banalmente verificato.

Assumendo che sia stata verificata la correttezza del teorema per la funzione *Reduce()* dopo averla eseguita k volte, vogliamo dimostrare che il teorema è ancora valido dopo la $k+1$-esima chiamata della funzione:

1. non esistono archi tra un qualunque nodo $b_i \in B$ ed un altro nodo nel NS: viene creato un nuovo albero contenente la sola componente biconnessa definita in B. Non vengono aggiunti archi del tipo considerato dal teorema (l'unico arco padre-figlio è adiacente ad un nodo in una catena di fratelli) che quindi al termine dell'operazione è ancora verificato;

2. un solo nodo $b \in B$ ha almeno un arco con altri nodi nel NS: viene creata una nuova catena di nodi fratelli con padre b. Non vengono aggiunti archi del tipo considerato nel *teorema 3* e pertanto al termine dell'operazione il teorema è valido;

3. più di un nodo $b_i \in B$ sono adiacenti ad almeno un arco in NS:

   (a) vengono uniti i nodi, con le loro eventuali componenti biconnesse, che appartengono al percorso tra il primo nodo di B, b_1, ed ogni altro nodo che appartenga sia a B che allo stesso albero di b_1. Per ogni percorso di questo tipo vengono eliminati tutti gli archi padre-figlio al suo interno, ovvero tutti e soli quegli archi per cui la nuova componente biconnessa introduce almeno un percorso alternativo.

   (b) vengono uniti gli alberi in cui sono presenti nodi $b_i \in B$ che non sono nello stesso albero di b_1. Gli alberi sono uniti collegando

il rispettivo nodo radice alla componente biconnessa di b_1. Come si può notare, non vengono introdotti nè rimossi archi del tipo parde-figlio, pertanto il teorema è ancora valido al termine dell'operazione.

(c) ogni nodo $b_i \in B$ che non è ancora adiacente a nessun arco nel NS viene aggiunto alla catena di nodi fratelli della componente biconnessa di b_1. Ogni arco introdotto nel NS è del tipo fratello, il *teorema 3* è ancora verificato.

Al termine della $k+1$-esima chiamata a *Reduce()* il *teorema 3* vale.

5.4.4 Nodi del Navigational Sketch e *articulation points*

Teorema 4: *ogni nodo v, adiacente ad un set di n archi, che verifichi $n > 2$, oppure, se è un nodo padre, $n >= 2$, è un articulation point.*

Dimostrazione: Abbiamo visto in (1.5.5) come ad ogni passo di At First Look ogni nodo v con *color degree* $d_c(v) > 1$ sia un *articulation point*. Dato che un passo di AFL corrisponde in *Reduce()* al processamento di una riga r del tipo $r : argc(r) == 2$, vogliamo verificare che l'invariante venga rispettato anche quando si processino righe r del tipo $r : argc(r) > 2$. Procediamo dimostrando il teorema per induzione sul numero di righe in input su cui viene chiamata *Reduce()*:

Nel *caso base* non ci sono archi nel Navigational Sketch, e pertanto nessun nodo verifica l'ipotesi del *teorema 4*. Il teorema è verificato.

Assumendo che il *teorema 4* sia stato provato fino al processamento di k righe dell'input, vogliamo verificare che sia valido anche dopo che è stata elaborata la riga $k+1$-esima rappresentante la componente biconnessa B:

1. nessun nodo $b_i \in B$ è adiacente ad almeno un arco in NS: viene creato un albero composto dalla sola componente biconnessa in B, e composto da un nodo radice b che è padre della catena di fratelli composta da tutti i nodi $b_i : b_i \in B \wedge b_i \neq b$. Notiamo che ogni nodo non padre, nell'albero appena costruito, è adiacente al più a 2 archi, mentre il nodo radice, che è l'unico nodo padre, è adiacente ad un solo arco.Nessun nodo rientra nell'ipotesi del teorema, che risulta così verificato;

2. un solo nodo b è adiacente ad almeno un arco nel NS: tutti i nodi $b_i : b_i \in B \wedge b_i \neq b$ sono appesi ad una nuova catena di nodi fratelli che ha per padre il nodo b. Come nel caso precedente, non sono introdotti archi che verifichino l'ipotesi del teorema ed il teorema vale;

3. più di un nodo $b_i \in B$ è adiacente ad almeno un arco nel NS:

(a) viene aggiunto un arco tra b_1, il primo nodo letto da B, ed ogni altro nodo $b_i \in B$ che appartiene allo stesso albero di b_1. Vengono eliminati tutti gli archi singoli che collegano quei nodi che divengono fratelli dopo l'aggiunta di un arco. Ogni nodo *articulation point* che viene inserito nella catena di nodi fratelli rimane *articulation point* solo se mantiene un numero di archi adiacenti superiore a tre. Gli archi padre-figlio rimossi sono quelli che in G collegano nodi all'interno della stessa componente biconnessa. Ogni nodo che sia un*articulation point* prima di essere aggiunto alla componente biconnessa, smette di esserlo subito dopo se non ha archi verso nodi esterni ad essa. Il teorema risulta così valido per la definizione di *articulation point*;

(b) vengono aggiunti tutti gli alberi (anche singleton) contenenti nodi $b_i \in B$ che non sono nello stesso albero di b_1: non vengono nè introdotti nè rimossi archi del tipo padre-figlio, l'asserzione del teorema è valida;

Al ritorno di *Reduce()* il *teorema 4* è valido.

5.4.5 Somma di Navigational Sketch

Teorema 5: *Al termine dell'esecuzione di* Reduce() *il Navigational Sketch NS, ottenuto da un input costituito da un set di Navigational Sketch $NS_i = Map(G_i)$ dove G_i è un generico grafo non orientato, è equivalente al Navigational Sketch $NS_G \equiv Map(\sum_{i=1} G_i)$.*

Dimostrazione: Essendo il Navigational Sketch NS prodotto come output da *Reduce()* rappresentativo di tutte le caratterristiche dei Navigational Sketch NS_i in input, per i teoremi *1, 2, 3, 4*, vale:

$$NS \equiv \sum_{i=1} NS_i$$

(dove abbiamo definito l'operazione *somma* tra Navigational Sketch con la funzione *Reduce()*), da cui è banale la dimostrazione del *teorema 5*.

5.5 L'input

L'input I di HitSoop è simile a quello di Hitsura relativamente alla formattazione con cui è descritto il grafo G che si vuole analizzare, ma mentre in Hitsura G è definito in un unico file residente nella stessa macchina del processo, in HitSoop G è descritto da un insieme di t file corrispondenti ognuno ad uno specifico task e residenti nel filesystem distribuito HDFS. Per la descrizione sulla formattazione dell'input rimandiamo il lettore a 4.1.1, mentre

esaminiamo di seguito i vincoli di commutatività e associatività rispetto all'elaborazione di HitSoop che deve rispettare I per poter essere compatibile con il paradigma MapReduce.

5.5.1 L'input ai Mapper

I task che vengono sottoposti ai Mapper sono in forma di file di testo f_i di cui l'input complessivo è la collezione completa. Ogni Mapper sottopone alla propria funzione *Map()* una coppia $< chiave,\ valore >$ in cui la chiave è il nome del file, mentre il valore è il suo contenuto. Ad esempio, se uno specifico Mapper deve eseguire un task composto da uno split di F files chiamerà la sua funzione *Map()* F volte.

5.5.2 Commutatività dell'input

Il requisito di commutatività di I in HitSoop è necessario in quanto non è possibile prevedere l'ordine con cui l'insieme degli I_i viene sottoposto ad un'applicazione Hadoop MapReduce.

Dimostrare questa proprietà è banale dalla definizione dell'algoritmo At First Look: notiamo infatti che l'input rappresenta il grafo $G = (V, E)$ come un insieme di archi $e_i : e_i \in E$, e poiché G è un grafo generico non orientato, è ininfluente l'ordine con cui tali archi e_i vengono elencati. Al variare dell'ordine di come vengono sottoposti gli e_i, AFL strutturerà Navigational Sketch tra loro diversi ma semanticamente equivalenti e ugualmente rappresentativi di G.

5.5.3 Associatività dell'input

Il requisito di associatività degli I_i in Hadoop garantisce che valga

$$Map\,(I_1) + Map\,(I_2) \equiv Map\,(I_1 + I_2).$$

Per dimostrare l'associatività dell'input è necessario però definire un'operazione sul dominio degli output intermedi NS_i, chiamiamola "somma" $(+)$, per la quale dati due NS_i NS_1 e NS_2, generati a partire rispettivamente dai due split I_1 e I_2 attraverso la funzione $Map(I_i) = NS_i$, si abbia $NS_1 + NS_2 = NS_{1,2}$. L'associatività dell'input dunque vale se l'operazione di somma appena definita verifica $NS_1 + NS_2 \equiv Map\,((I_1 + I_2))$. L'operazione con le caratteristiche descritte è implementata dalla funzione *Reduce()*. Per concludere la dimostrazione sull'associatività dell'input serve dunque provare la correttezza di *Reduce()*, argomento che abbiamo affrontato in 5.4.

5.6 I Mapper output

L' output di un Mapper in HitSoop è il risultato della chiamata in cascata della funzione *Map()* sui file f_i di un task e della chiamata a *Combine()* sugli output di *Map()*.

5.6.1 L'output di *Map()*

La funzione *Map()* produce in uscita un set di coppie $< chiave,\ valore >$, una coppia per ogni file f_i di ogni task T_j del Mapper.

In ogni coppia il parametro *chiave* ha un valore costante per tutti i Mappers (in HitSoop è la stringa "`grafo`"), mentre il parametro *valore* contiene il Navigational Sketch $NS_i = Map\,(f_i)$, sotto forma di testo, il cui formato è descritto in 5.3.1. Ci riferiamo agli output di questo tipo come NS_i.

5.6.2 L'output di *Combine()*

La funzione *Combine()*, che abbiamo già detto condividere lo stesso codice di *Reduce()*, prende in input tutti gli NS_i prodotti dal Mapper su cui viene eseguita e produce come output un nuovo Navigational Sketch.

Combine() implementa la definizione della funzione "*somma*" $(+)$ già descritta in 5.5.3:

$$\begin{gathered} Map\,(f_1) = NS_1,\ Map\,(f_2) = NS_2 \\ \Longrightarrow \\ NS_1 + NS_2 \equiv Map\,(f_1 + f_2). \end{gathered}$$

L'introduzione di *Combine()*, facoltativa in Hadoop MapReduce, permette di diminuire la quantità di traffico nella rete dovuta all'invio degli output dei Mappers al Reducer, associando prima di inviarli quei valori che comunque verrebbero poi associati dalla funzione *Reduce()*.

Supponendo che ad un Mapper sia stato assegnato un task che prevede l'elaborazione di F file, per ogni file f_i verrà calcolato $NS_i = Map\,(f_i)$. Se non fosse definita *Combine()*, ogni NS_i così prodotta verrebbe trasmessa al Reducer. Poiché vale

$$size\,(NS_1 + NS_2) \leq size\,(NS_1) + size\,(NS_2),$$

Combine() fa in modo che il Mapper invii meno bytes senza che l'output perda informazioni.

Per quanto riguarda la formattazione dell'output, *Combine()* produce un Navigational Sketch $NS_{combine}$ formattato allo stesso modo degli NS_i che riceve in ingresso.

5.6.3 Commutatività dell'output intermedio

Anche gli output di *Map()* che di *Combine()* devono verificare di essere commutativi rispetto alla funzione cui devono essere sottoposti (rispettivamente *Combine()* e *Reduce*), in quanto non è possibile definire l'ordine con cui vengono processati. Si rende quindi necessario dimostrare la commutatività dell'operazione "*somma*", implementata da *Reduce()*, rispetto agli operandi NS_i.

5.7 L'output

Reduce() produce un file di output con il Navigational Sketch definitivo ottenuto "sommando" tutti gli NS_j prodotti dai Mapper dopo l'esecuzione di *Combine()*. Tale output è formattato sempre al modo già descritto per un generico NS, ma in aggiunta riporta alcune righe di commento (nell'intestazione e prima delle componeti biconnesse) che consentono una migliore facilità di comprensione ad una lettura "umana".

5.7.1 Scrittura dell'output

Una volta terminata l'operazione di *Reduce*, l'output viene scritto nel file `part-00000` all'interno della directory specificata come secondo parametro al momento del lancio del programma. Nel caso in cui il job richiesto da HitSoop fallisca, `part-00000` non viene creato.

5.7.2 Prelevamento dell'output

Per prelevare l'output è sufficiente chiamare l'apposita routine di Hadoop (in 3.4.7). Per esempio, se l'output è nella directory del *HDFS* `/user/username/output/`, allora il comando da shell sarà:

```
~hadoop-dir/bin/$ ./hadoop dfs -get output destination_dir_in_local_fs
```

5.8 Esecuzione di HitSoop su cluster

Per raccogliere i dati sul funzionamento di HitSoop, abbiamo compiuto diverse esecuzioni su un cluster di 5 server Amazon Web Services EC2. Una volta caricato e configurato il sistema Hadoop, si sono compiute due esecuzioni per ogni dataset in input. Abbiamo sottoposto al programma i dataset corrispondenti ai file *dutchElite* e *cnr-2005* (B), una volta facendo operare HitSoop inibendo la funzione *Combiner()* ed una volta attivandola, per un totale di quattro esecuzioni. I dati dimostrano che l'introduzione di *Combiner()* diminuisce la mole del traffico nella rete, e che il tempo di esecuzione speso complessivamente ai Mapper è molto maggiore rispetto a quello speso dal Reducer. A questo risultato, che già di per sè dimostra la netta diminuzione del tempo di esecuzione necessario in HitSoop rispetto a Hitsura (ovvero rispetto ad un'elaborazione strettamente sequenziale di un singolo processo a singolo thread), si deve poi aggiungere il vantaggio dato dal fatto che il Reducer si attiva non appena sono stati inviati i primi output intermedi dei Mapper, ovvero prima che l'intero *map step* sia terminato. Abbiamo raccolto nell'appendice A i dati sulle esecuzioni di HitSoop sul cluster EC2, confrontandole tra loro e mettendo particolarmente in evidenza, sia nel complesso che per ogni fase (Map e Reduce):

- i bytes letti in input;
- i bytes scritti nell'output;
- i bytes letti e scritti dal/nel HDFS;
- il flusso di records all'interno del framework Hadoop MapReduce;
- il flusso di bytes all'interno del framework Hadoop MapReduce;

Capitolo 6

Vantaggio dell'elaborazione MapReduce

Come si vede in 5.8, il vantaggio dell'adozione di un cluster nella risoluizone del problema è nell'aumento delle prestazioni dell'elaborazione in termini di tempo. Nel seguito del capitolo facciamo alcune osservazioni sulle ragioni per cui otteniamo un miglioramento consistente nel tempo di elaborazione complessiva, e sull'uso di cluster nella risoluzione di problemi nel classical streaming.

6.1 Perché funziona

Il motivo può non essere subito evidente se si considera che l'operazione che compiono i Mappers di fatto non è altro che aggregare tra loro una parte dei nodi in input in un'unica catena di fratelli, che per essere elaborata dal Reducer necessita la chiamata alle stesse funzioni che l'hanno formata. Notiamo però anche che una generica catena $brothers\_chain$ di n nodi fratelli potrebbe essersi formata, nel caso peggiore, dopo il processamento di $n \cdot (n-1) \equiv n^2$ archi. Quando ciò si verifica, il Mapper chiama la funzione *add_edge()* $n \cdot (n-1)$ volte. La catena deve poi essere sottoposta al Reducer: notiamo che il costo della sua elaborazione da parte di *reduce* dipende da quanti sono i nodi $b_i : b_i \in brothers\_chain$ che sono già adiacenti ad almeno un arco nel Navigational Sketch del Reducer: infatti per tali nodi è necessario chiamare *add_edge()*, mentre per i nodi b_i che non sono ancora adiacenti ad alcun arco nel NS viene speso un tempo costante (quello richiesto dalla loro aggiunta ad una catena già formata).

Ad esempio, supponiamo di avere una catena di nodi fratelli $< n_1,\ n_2,\ n_3,\ n_4,\ n_5 >$. Questa potrebbe essere stata generata a partire dagli archi $e(n_1, n_2)$, $e(n_1, n_2)$, ..., $e(n_5,\ n_4)$, nel cui caso sarebbero state richieste $5 \cdot 4 = 20$ chiamate alla funzione

add_edge(), mentre per il suo processamento da parte di *reduce()* ne sono sufficienti al più 5 (nel caso peggiore, ovvero quello in cui ogni nodo sia già adiacente ad un arco nel NS del Reducer).

6.2 Vantaggi nel modello streaming

Poiché i problemi risolvibili secondo il modello del classical streaming devono elaborare una mole di dati di dimensione molto grande, tipicamente impiegano molto tempo per la loro soluzione. Se un problema di questo tipo può essere risolto parallelizzandone la computazione, il tempo richiesto dalla computazione complessiva può essere considerevolmente ridotto.

Notiamo che l'insieme dei problemi risolvibili in MapReduce è contenuto propriamente in quello dei problemi nel *classical streaming*, in quanto ad esso non appartengono i problemi per i quali è necessario leggere lo stream in modo sequenziale. È anche possibile che l'adozione del paradigma MapReduce richieda un'implementazione che richieda lo stesso tempo, o un tempo addirittura maggiore, rispetto alla computazione sequenziale.

Capitolo 7

Futuri sviluppi

Poiché MapReduce è un paradigma appena nato, sono ancora pochi gli algoritmi che ne sfruttano le potenzialità: l'arricchimento di questa classe di algoritmi e il sempre maggiore interesse verso il calcolo distribuito cui assistiamo in moltissimi scenari, permettono di rispondere alla necessità di processare quantità di dati sempre maggiori riducendo in modo considerevole i tempi ed i costi.

Al crescere dell'insieme dei problemi che possono essere risolti seguendo il paradigma MapReduce, aumentano gli scenari applicativi per i quali tali soluzioni sono congeniali: ad oggi non solo i maggiori motori di ricerca come Google e Yahoo, ma anche molti social network (tra cui Facebook) e servizi web sono strutturati basandosi su MapReduce. La pubblicazione di progetti opensource e free come Apache Hadoop consente di poter sfruttare i vantaggi che derivano dal poter sviluppare applicativi distribuiti anche da parte di aziende e sviluppatori che non hanno una grande disponibilità di risorse, consentendo lo sviluppo a basso costo di servizi e ricerche altrimenti dispendiosi.

Capitolo 8

Conclusioni

Abbiamo esposto la nostra soluzione alla ricerca di componenti connesse, biconnesse, *articulation points* e *bridges* basata sul paradigma MapReduce per il calcolo parallelo, la sua implementazione HitSoop basata sul framework opensource Apache Hadoop e l'esecuzione di HitSoop su un cluster di macchine Amazon Web Services EC2.

Bibliografia

[1] Introduction to parallel programming and mapreduce.

[2] Apache Foundation.

[3] Apache Foundation.

[4] Luigi Laura Giorgio Ausiello, Donatella Firmani. Datastream computation of graph biconnectivity: Articulation points, bridges, and biconnected components. 2010.

[5] Luigi Laura Giorgio Ausiello, Donatella Firmani. Real-time anomalies detection and analysis of network structure, with application to the autonomous system network. 2011.

[6] Luigi Laura Giorgio Ausiello, Donatella Firmani. Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components. 2011.

[7] Michael T. Goodrich. Simulating parallel algorithms in the mapreduce framework with applications to parallel computational geometr.

[8] Sergei Vassilvitskii Howard Karloff, Siddharth Suri. A model of computation for mapreduce.

[9] Sanjay Ghemawat Jeffrey Dean. Mapreduce: A flexible data processing tool. 2010.

[10] Jelena Pjesivac-Grbovic Jerry Zhao. Mapreduce: Simplified data processing on large clusters.

[11] Yahoo Developer Network.

[12] Jason Venner. *ProHadoop*. Apress, first edition edition, 2009.

[13] Tom White. *Hadoop, the Definitive Guide*. O'Reilly, first edition edition, June 2010.

Appendice A

Dati sulle esecuzioni di HitSoop

A.1 Esecuzione su dutchElite senza Combiner

Tabella A.1: Job counters

Metrica	Map	Reduce	Total
SLOTS_MILLIS_MAPS	0	0	59.015
Launched reduce tasks	0	0	1
Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
Total time spent by all maps waiting after reserving slots (ms)	0	0	0
Launched map tasks	0	0	8
Data-local map tasks	0	0	8
SLOTS_MILLIS_REDUCES	0	0	10.854

Tabella A.2: File Output Format Counters

Metrica	Map	Reduce	Total
Bytes Written	0	39.987	39.987

Tabella A.3: File Input Format Counters

Metrica	Map	Reduce	Total
Bytes Read	49.998	0	49.998

Tabella A.4: FileSystemCounters

Metrica	Map	Reduce	Total
FILE_BYTES_READ	0	90.780	90.780
HDFS_BYTES_READ	51.174	0	51.174
FILE_BYTES_WRITTEN	260.438	111.951	372.389
HDFS_BYTES_WRITTEN	0	39.987	39.987

Tabella A.5: Map-Reduce Framework

Metrica	Map	Reduce	Total
Reduce input groups	0	1	1
Map output materialized bytes	90.822	0	90.822
Combine output records	0	0	0
Map input records	5.236	0	5.236
Reduce shuffle bytes	0	90.822	90.822
Reduce output records	0	2	2
Spilled Records	4.768	4.768	9.536
Map output bytes	81.234	0	81.234
SPLIT_RAW_BYTES	1.176	0	1.176
Map output records	4.768	0	4.768
Combine input records	0	0	0
Reduce input records	0	4.768	4.768

A.2 Esecuzione su dutchElite con Combiner

Tabella A.6: Job counters

Metrica	Map	Reduce	Total
SLOTS_MILLIS_MAPS	0	0	58.435
Launched reduce tasks	0	0	1
Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
Total time spent by all maps waiting after reserving slots (ms)	0	0	0
Launched map tasks	0	0	8
Data-local map tasks	0	0	8
SLOTS_MILLIS_REDUCES	0	0	14.194

Tabella A.7: File Output Format Counters

Metrica	Map	Reduce	Total
Bytes Written	0	40.768	40.768

Tabella A.8: File Input Format Counters

Metrica	Map	Reduce	Total
Bytes Read	49.998	0	49.998

Tabella A.9: FileSystemCounters

Metrica	Map	Reduce	Total
FILE_BYTES_READ	0	91.777	91.777
HDFS_BYTES_READ	51.174	0	51.174
FILE_BYTES_WRITTEN	262.211	113.045	375.256
HDFS_BYTES_WRITTEN	0	40.768	40.768

Tabella A.10: Map-Reduce Framework

Metrica	Map	Reduce	Total
Reduce input groups	0	1	1
Map output materialized bytes	91.819	0	91.819
Combine output records	4.792	0	4.792
Map input records	5.236	0	5.236
Reduce shuffle bytes	0	91.819	91.819
Reduce output records	0	2	2
Spilled Records	4.792	4.792	9.584
Map output bytes	81.234	0	81.234
SPLIT_RAW_BYTES	1.176	0	1.176
Map output records	4.768	0	4.768
Combine input records	4.768	0	4.768
Reduce input records	0	4.792	4.792

A.3 Esecuzione su cnr-2005 senza Combiner

Tabella A.11: Job counters

Metrica	Map	Reduce	Total
SLOTS_MILLIS_MAPS	0	0	116.595
Launched reduce tasks	0	0	1
Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
Total time spent by all maps waiting after reserving slots (ms)	0	0	0
Launched map tasks	0	0	9
Data-local map tasks	0	0	9
SLOTS_MILLIS_REDUCES	0	0	78.900

Tabella A.12: File Output Format Counters

Metrica	Map	Reduce	Total
Bytes Written	0	2.788.529	2.788.529

Tabella A.13: File Input Format Counters

Metrica	Map	Reduce	Total
Bytes Read	42.672.048	0	42.672.048

Tabella A.14: FileSystemCounters

Metrica	Map	Reduce	Total
FILE_BYTES_READ	0	4.272.260	4.272.260
HDFS_BYTES_READ	42.673.326	0	42.673.326
FILE_BYTES_WRITTEN	4.463.000	4.293.417	8.756.417
HDFS_BYTES_WRITTEN	0	2.788.529	2.788.529

Tabella A.15: Map-Reduce Framework

Metrica	Map	Reduce	Total
Reduce input groups	0	1	1
Map output materialized bytes	4.272.308	0	4.272.308
Combine output records	0	0	0
Map input records	3.129.116	0	3.129.116
Reduce shuffle bytes	0	3.585.271	3.585.271
Reduce output records	0	2	2
Spilled Records	118.544	118.544	237.088
Map output bytes	4.034.375	0	4.034.375
SPLIT_RAW_BYTES	1.278	0	1.278
Map output records	118.544	0	118.544
Combine input records	0	0	0
Reduce input records	0	118.544	118.544

A.4 Esecuzione su cnr-2005 con Combiner

Tabella A.16: Job counters

Metrica	Map	Reduce	Total
SLOTS_MILLIS_MAPS	0	0	143.783
Launched reduce tasks	0	0	1
Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
Total time spent by all maps waiting after reserving slots (ms)	0	0	0
Launched map tasks	0	0	9
Data-local map tasks	0	0	9
SLOTS_MILLIS_REDUCES	0	0	84.197

Tabella A.17: File Output Format Counters

Metrica	Map	Reduce	Total
Bytes Written	0	2.789.405	2.789.405

Tabella A.18: File Input Format Counters

Metrica	Map	Reduce	Total
Bytes Read	42.672.048	0	42.672.048

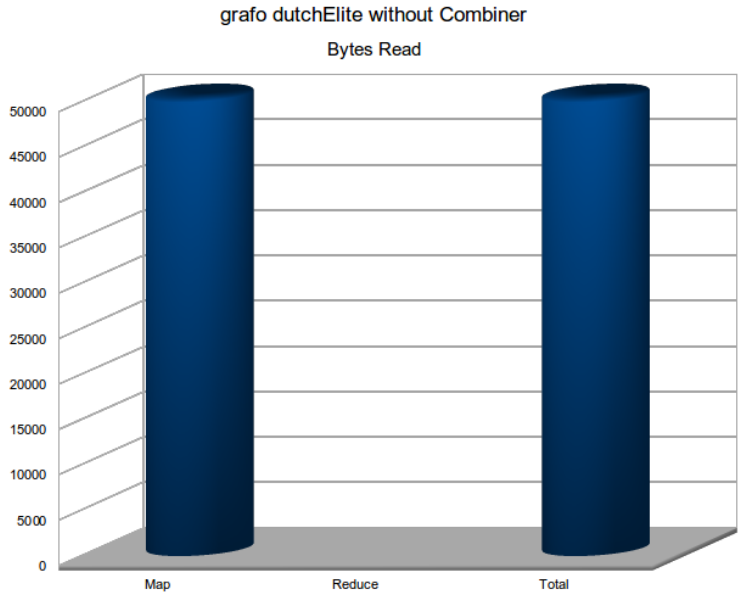
Tabella A.19: FileSystemCounters

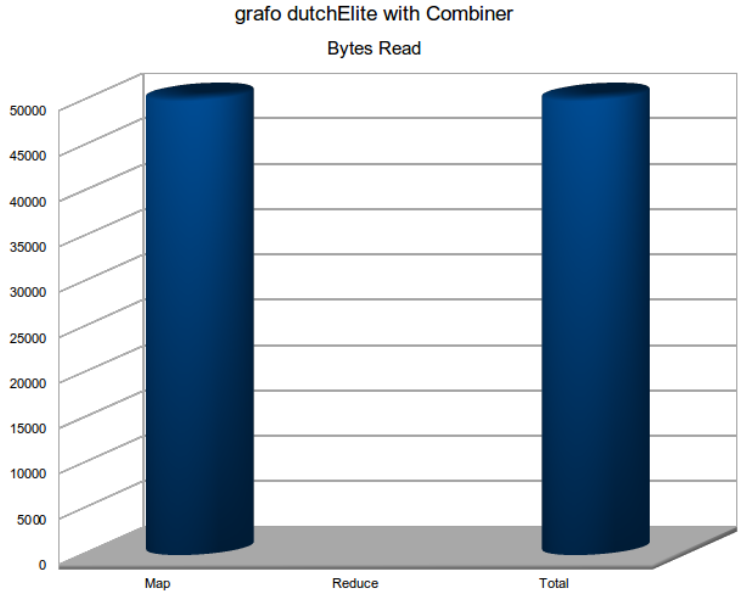
Metrica	Map	Reduce	Total
FILE_BYTES_READ	0	4.273.395	4.273.395
HDFS_BYTES_READ	42.673.326	0	42.673.326
FILE_BYTES_WRITTEN	4.465.044	4.294.653	8.759.697
HDFS_BYTES_WRITTEN	0	2.789.405	2.789.405

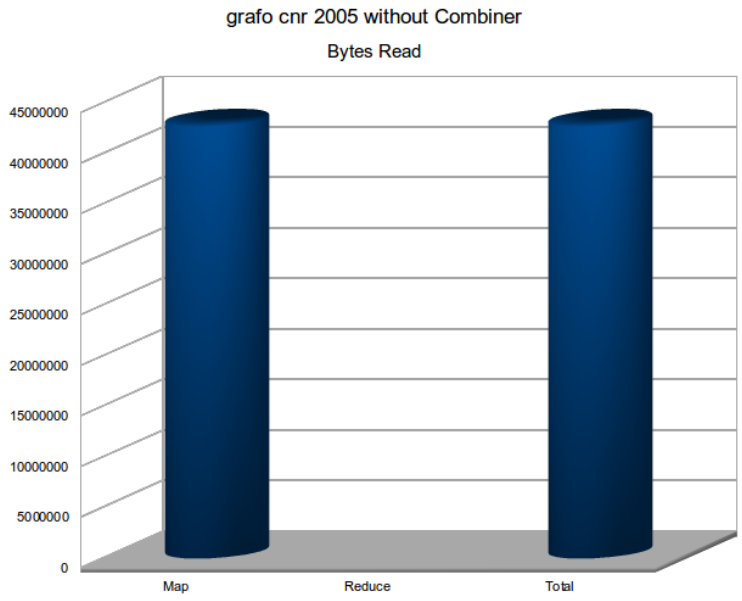
Tabella A.20: Map-Reduce Framework

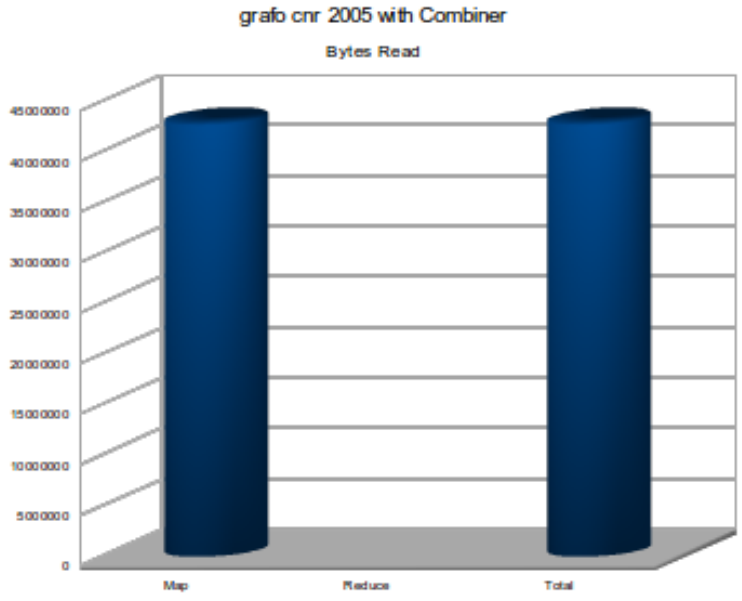
Metrica	Map	Reduce	Total
Reduce input groups	0	1	1
Map output materialized bytes	4.273.443	0	4.273.443
Combine output records	118.571	0	118.571
Map input records	3.129.116	0	3.129.116
Reduce shuffle bytes	0	4.273.443	4.273.443
Reduce output records	0	2	2
Spilled Records	118.571	118.571	237.142
Map output bytes	4.034.376	0	4.034.376
SPLIT_RAW_BYTES	1.278	0	1.278
Map output records	118.544	0	118.544
Combine input records	118.544	0	118.544
Reduce input records	0	118.571	118.571

A.5 Bytes letti in input

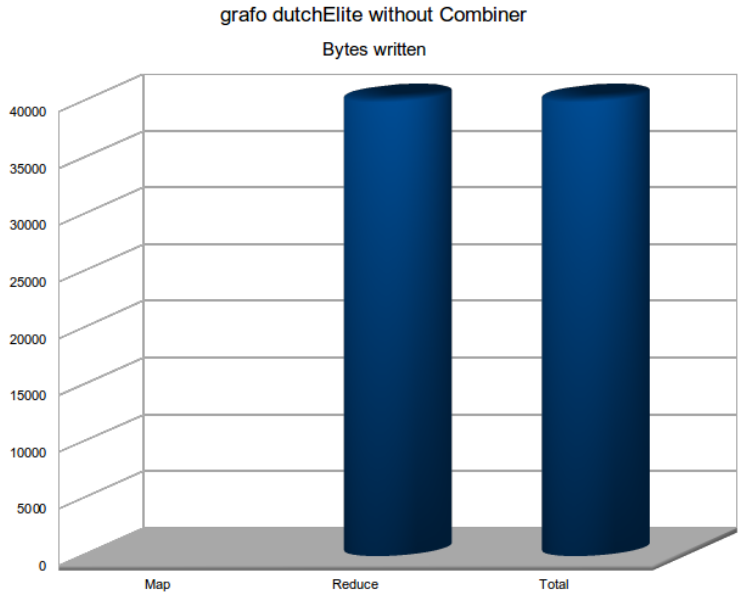


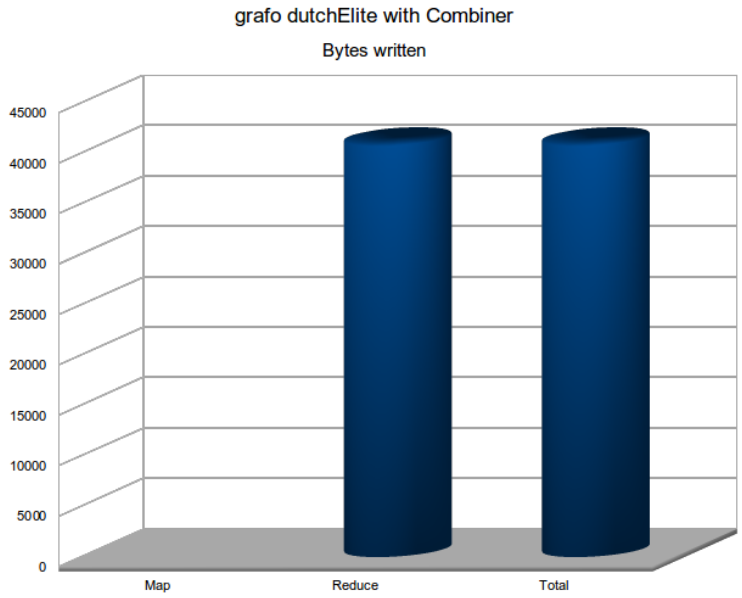


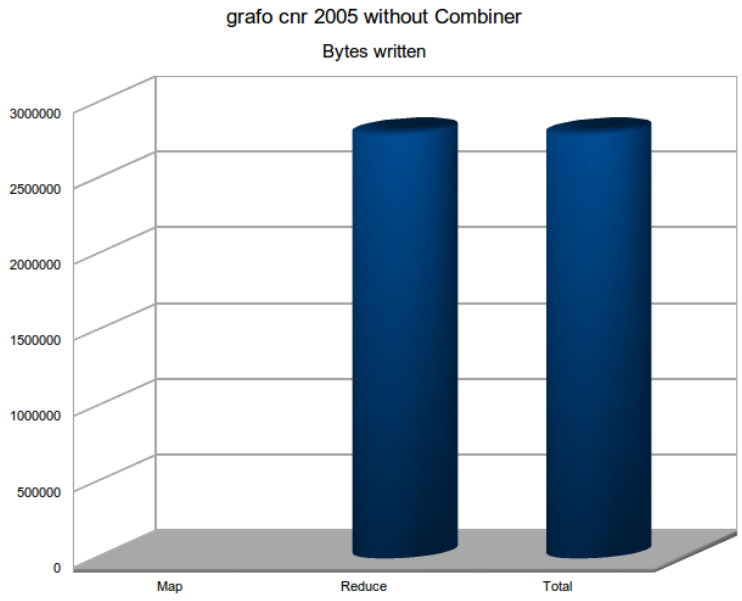

grafo cnr 2005 without Combiner
Bytes Read
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Map
Reduce
Total

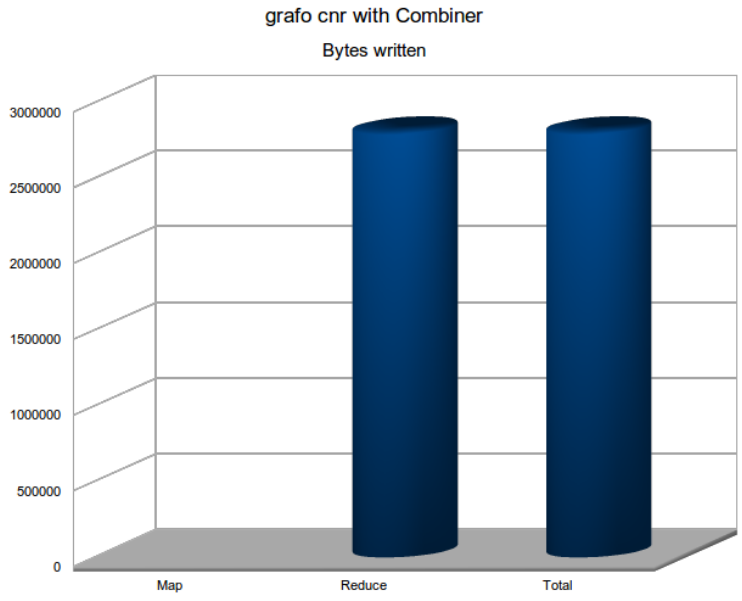

grafo cnr 2005 with Combiner
Bytes Read
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Map
Reduce
Total

A.6 Bytes scritti in otput

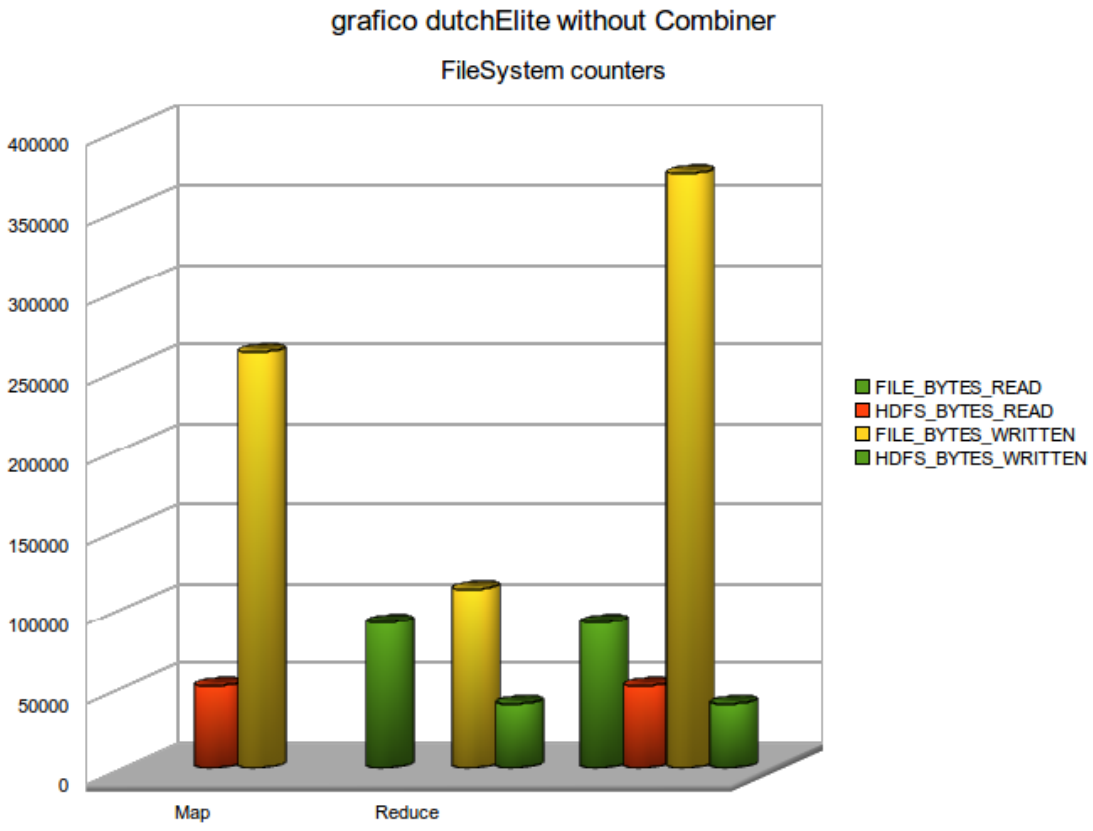





grafo cnr 2005 without Combiner
Bytes written
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total


grafo cnr with Combiner
Bytes written
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total

A.7 Bites letti e scritti dal/sul HDFS



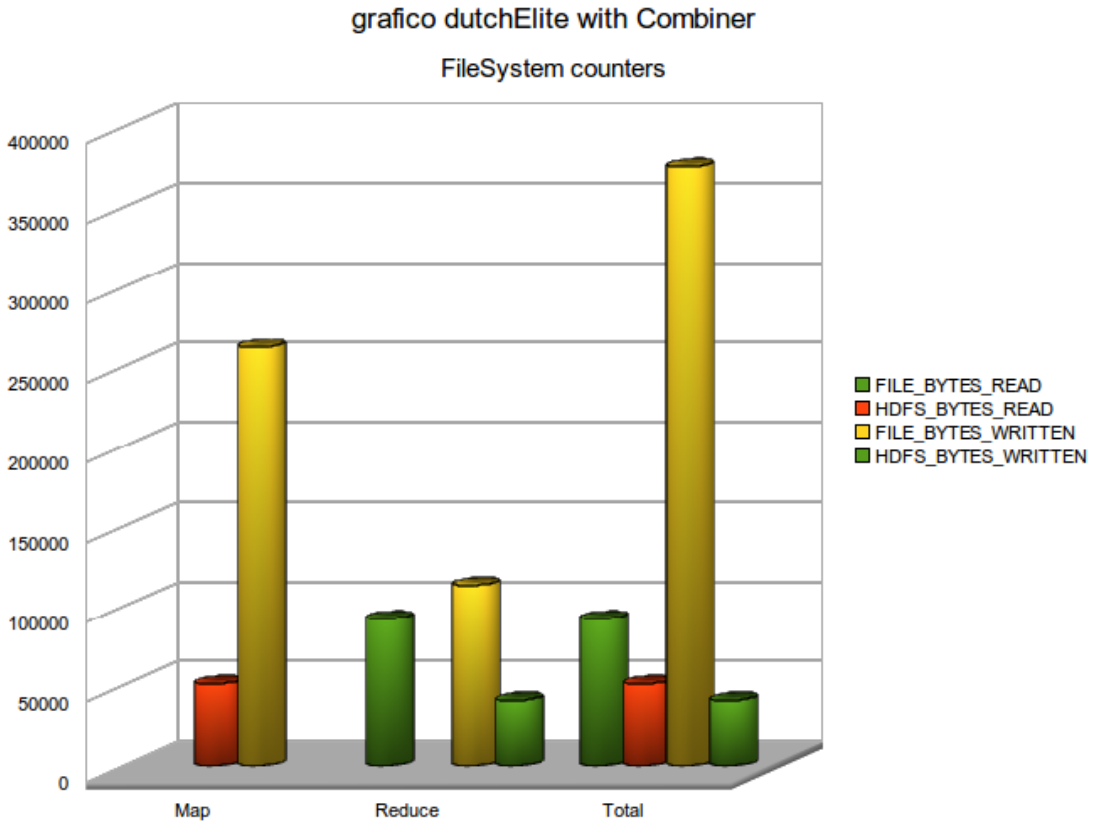


grafico dutchElite with Combiner
FileSystem counters
400000
350000
300000
250000
200000
150000
100000
50000
0
Map
Reduce
Total
FILE_BYTES_READ
HDFS_BYTES_READ
FILE_BYTES_WRITTEN
HDFS_BYTES_WRITTEN

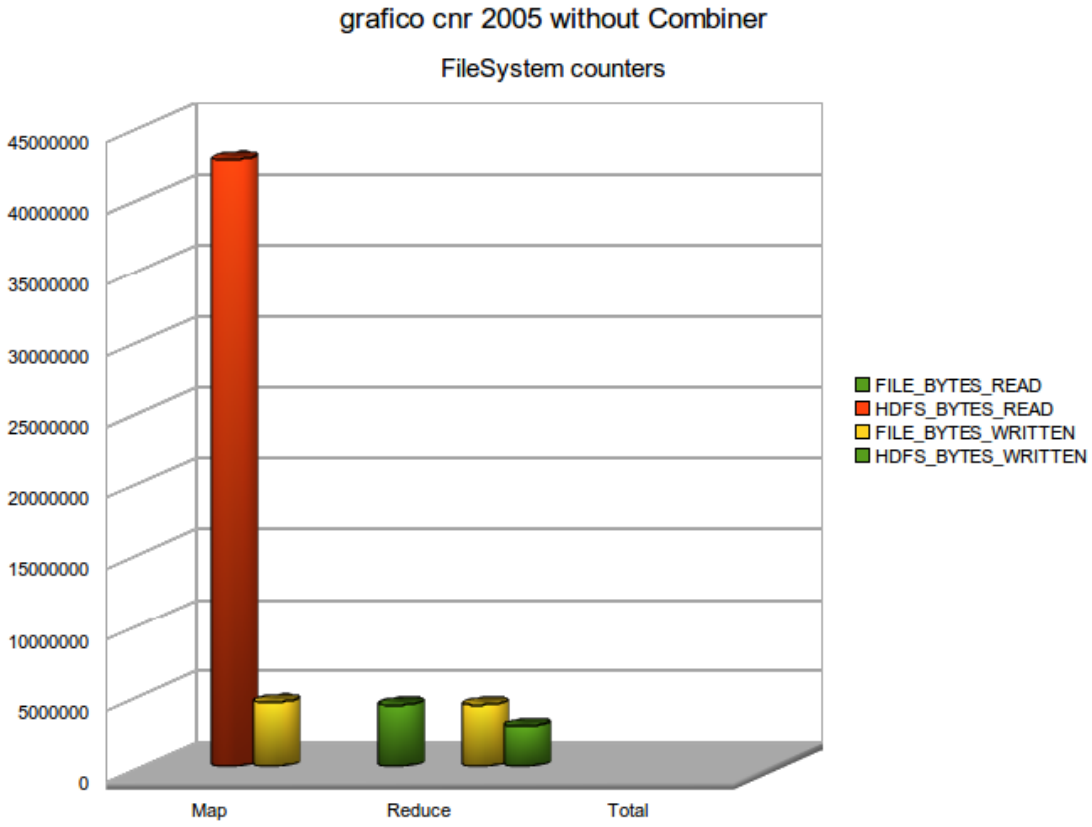


grafico cnr 2005 without Combiner
FileSystem counters
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Map
Reduce
Total
FILE_BYTES_READ
HDFS_BYTES_READ
FILE_BYTES_WRITTEN
HDFS_BYTES_WRITTEN

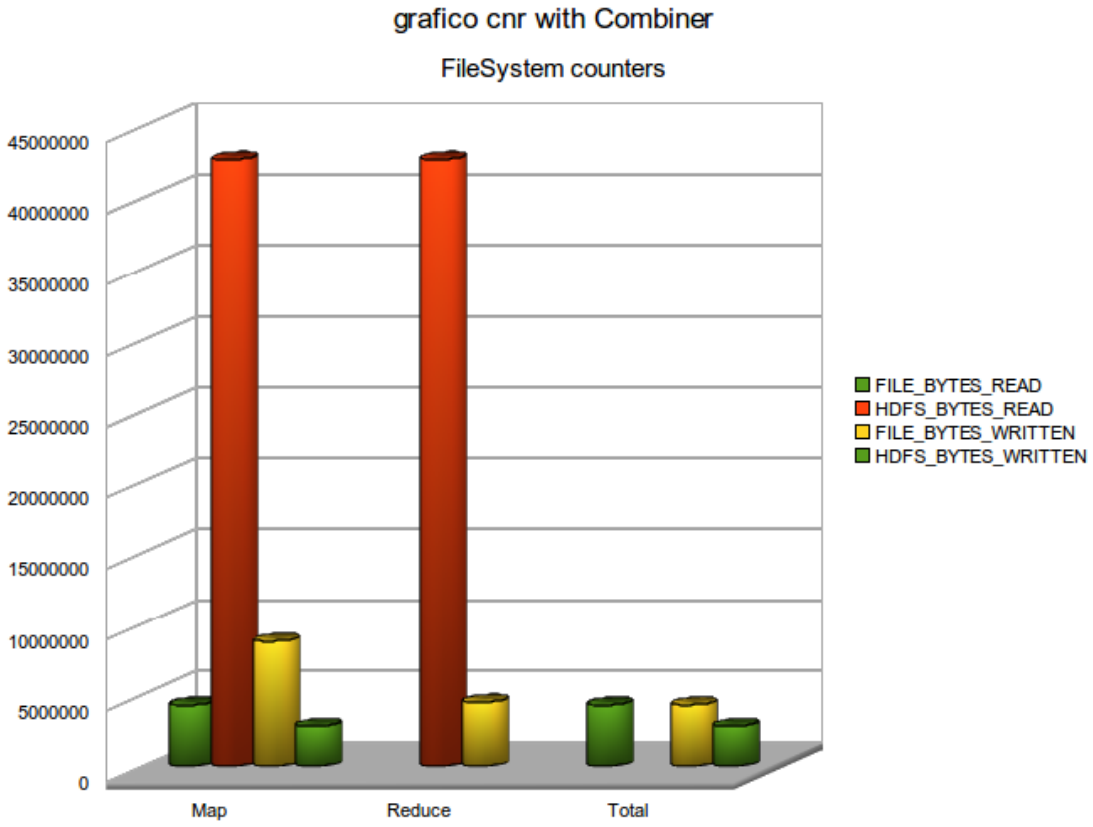
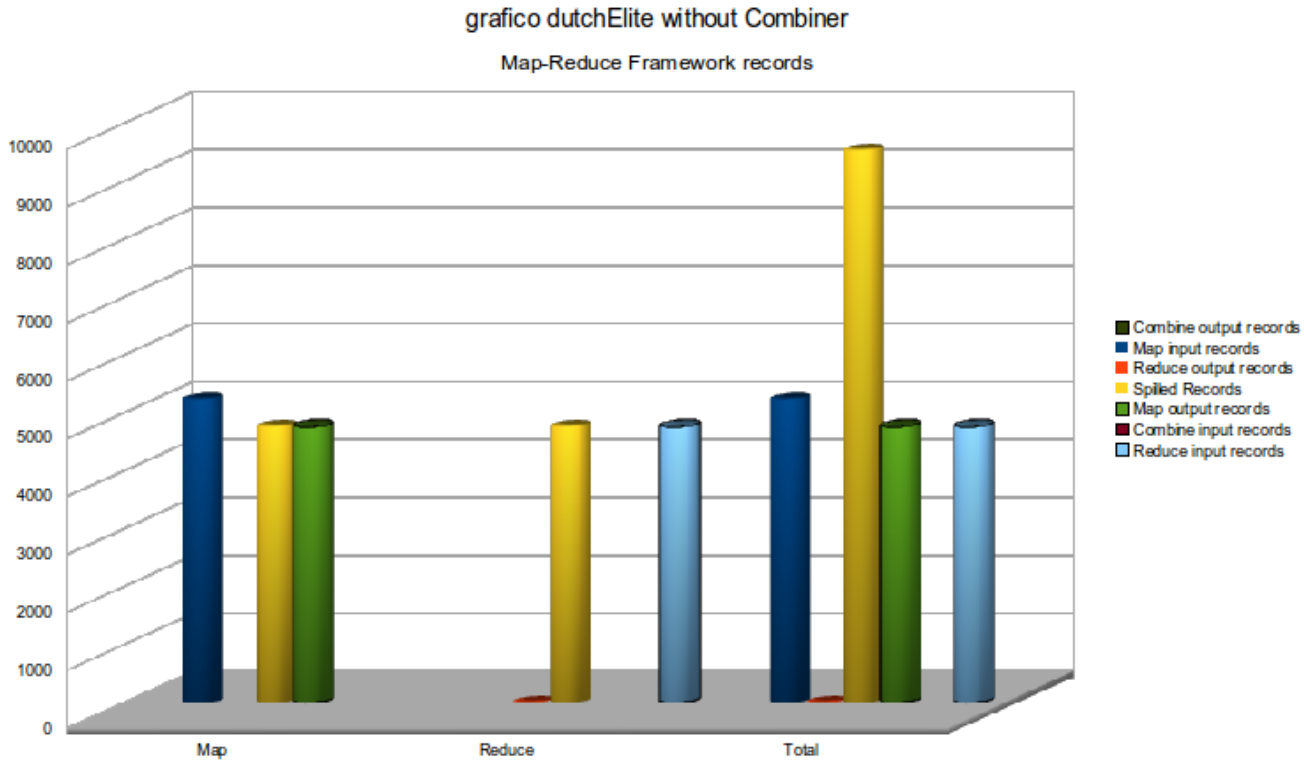


grafico cnr with Combiner
FileSystem counters
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Map
Reduce
Total
FILE_BYTES_READ
HDFS_BYTES_READ
FILE_BYTES_WRITTEN
HDFS_BYTES_WRITTEN

A.8 Flussi di records nel framework Hadoop MapReduce



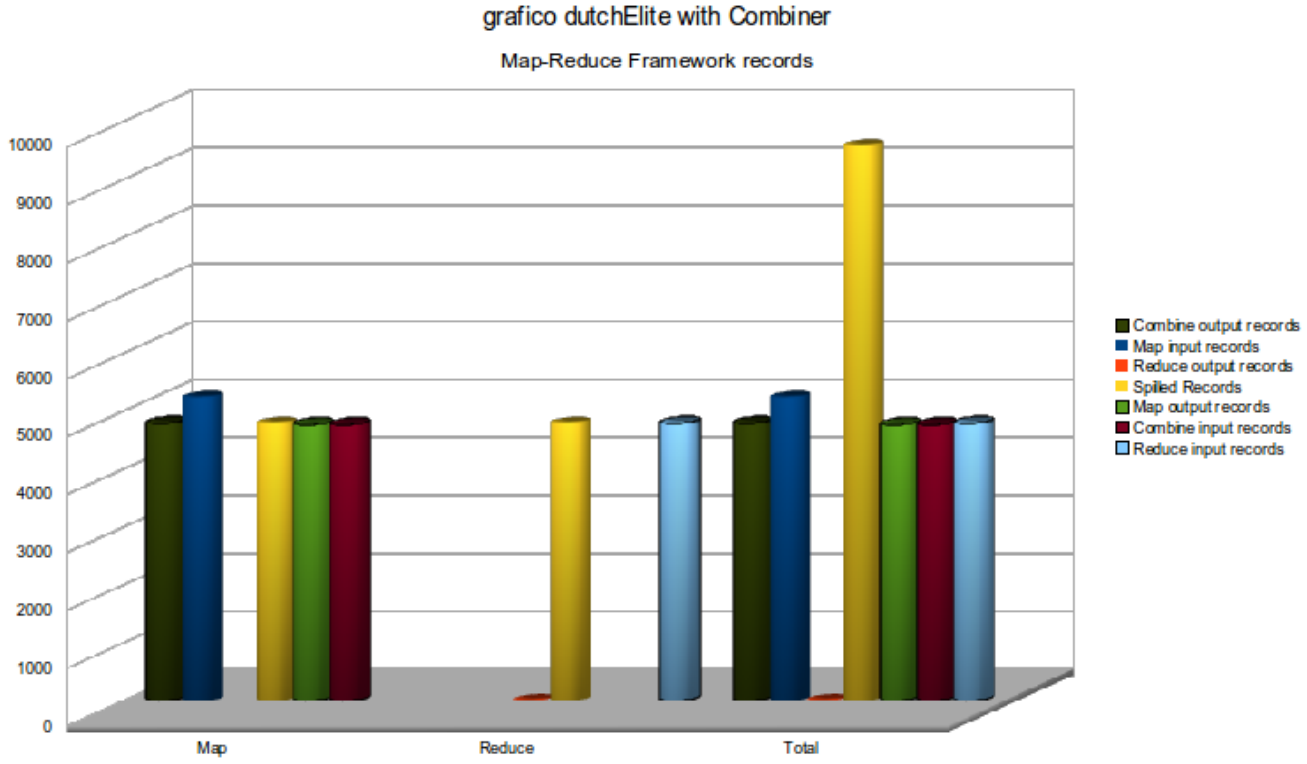


grafico dutchElite with Combiner
Map-Reduce Framework records
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Map
Reduce
Total
Combine output records
Map input records
Reduce output records
Spilled Records
Map output records
Combine input records
Reduce input records

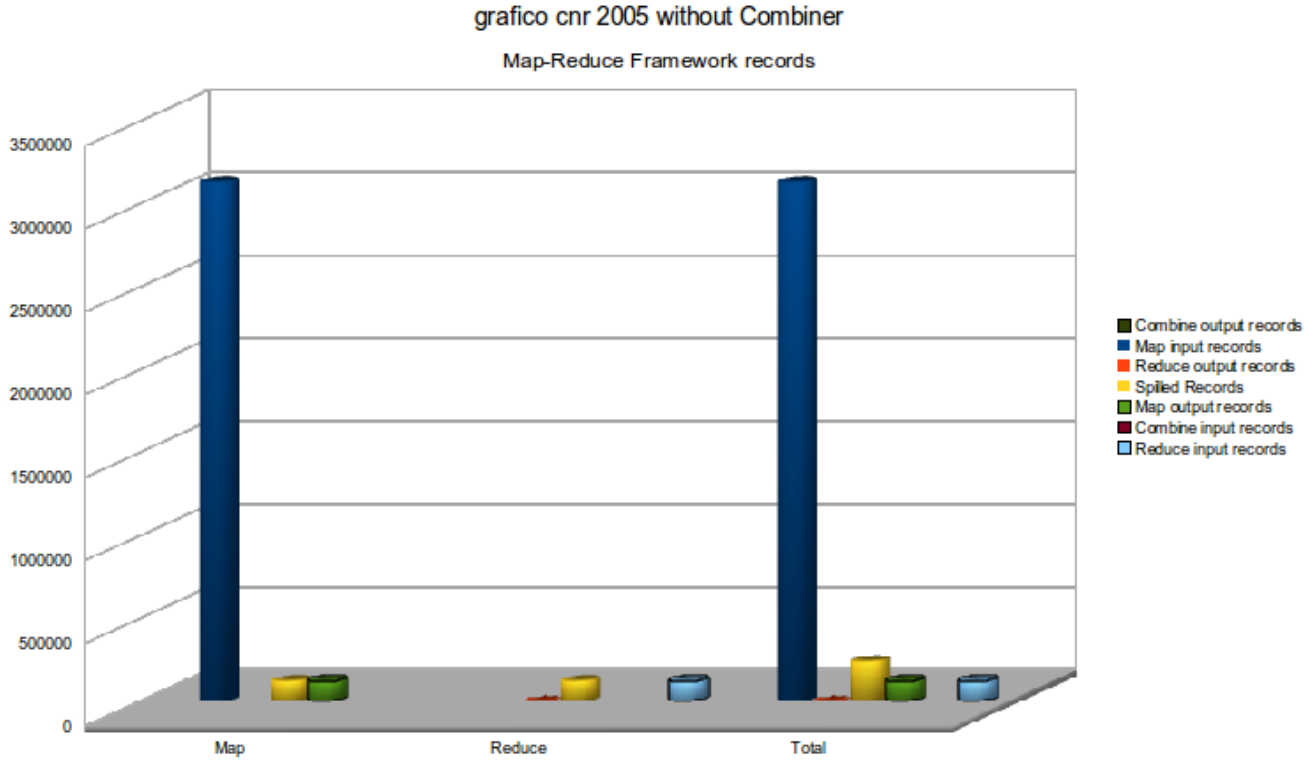


grafico cnr 2005 without Combiner
Map-Reduce Framework records
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total
Combine output records
Map input records
Reduce output records
Spilled Records
Map output records
Combine input records
Reduce input records

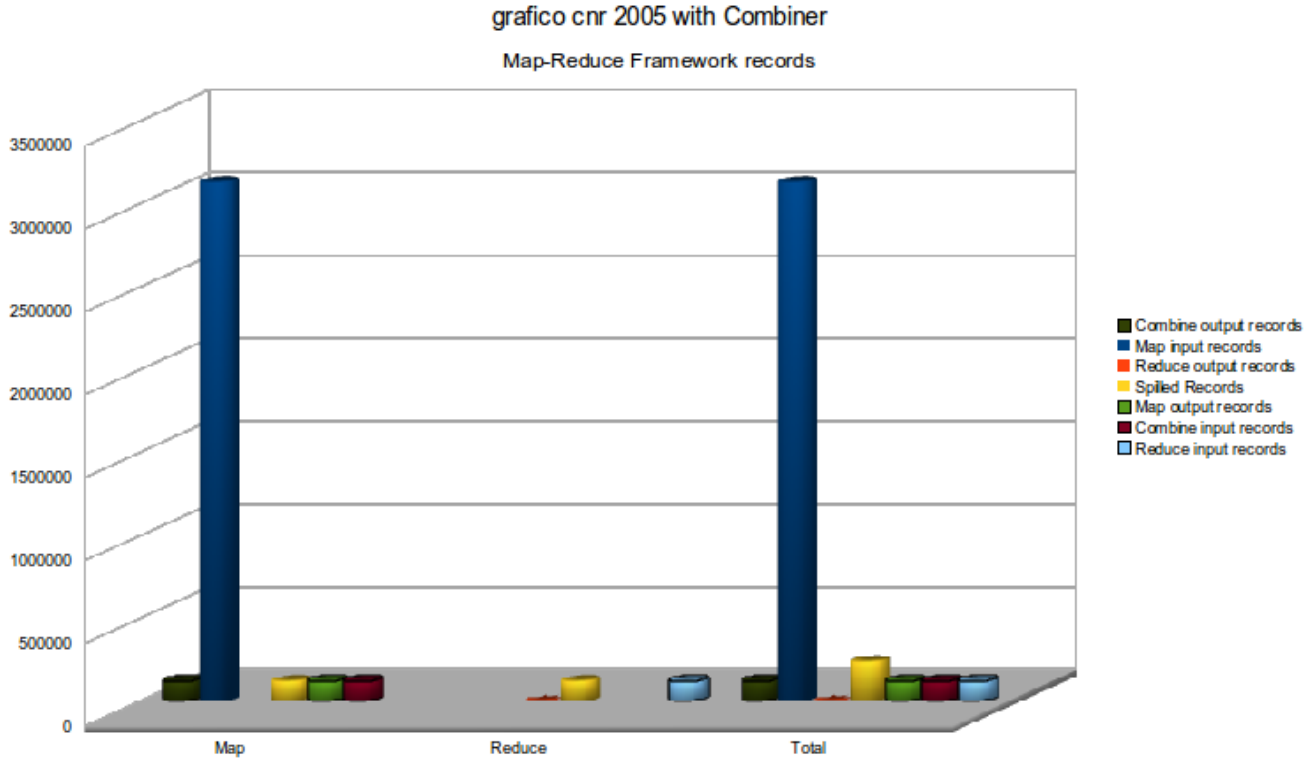
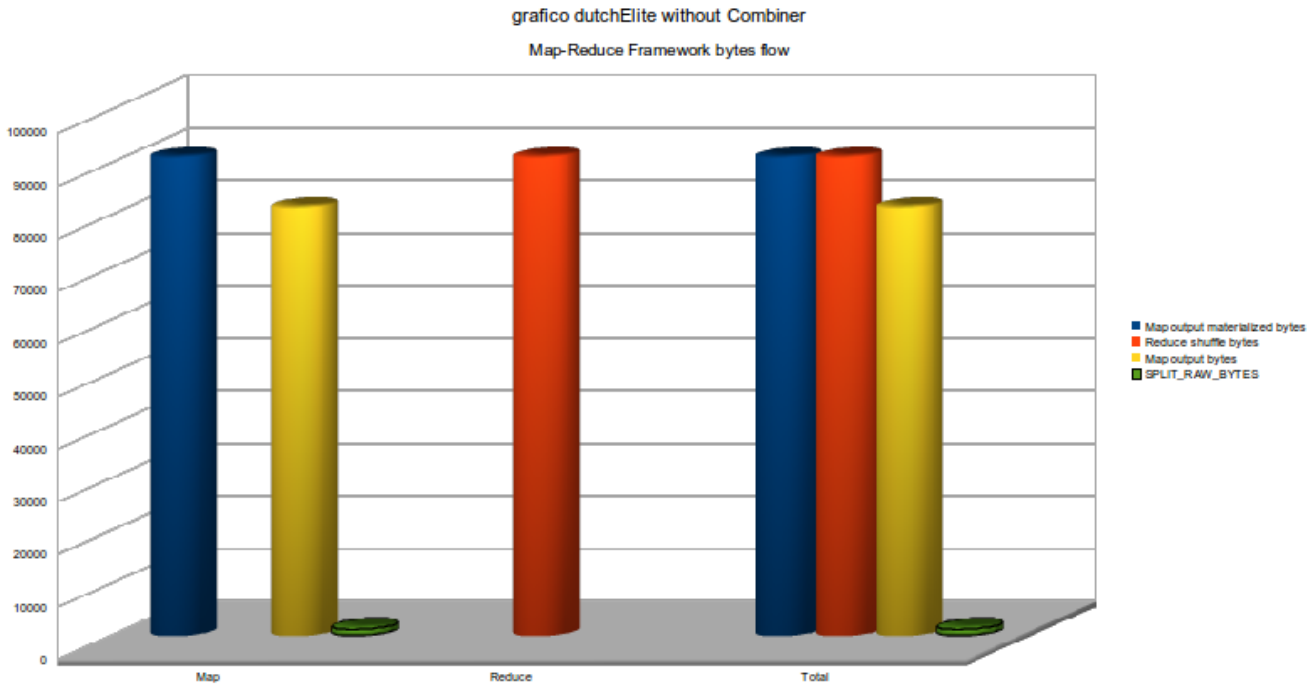


grafico cnr 2005 with Combiner
Map-Reduce Framework records
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total
Combine output records
Map input records
Reduce output records
Spilled Records
Map output records
Combine input records
Reduce input records

A.9 Flussi di bytes nel framework Hadoop MapReduce



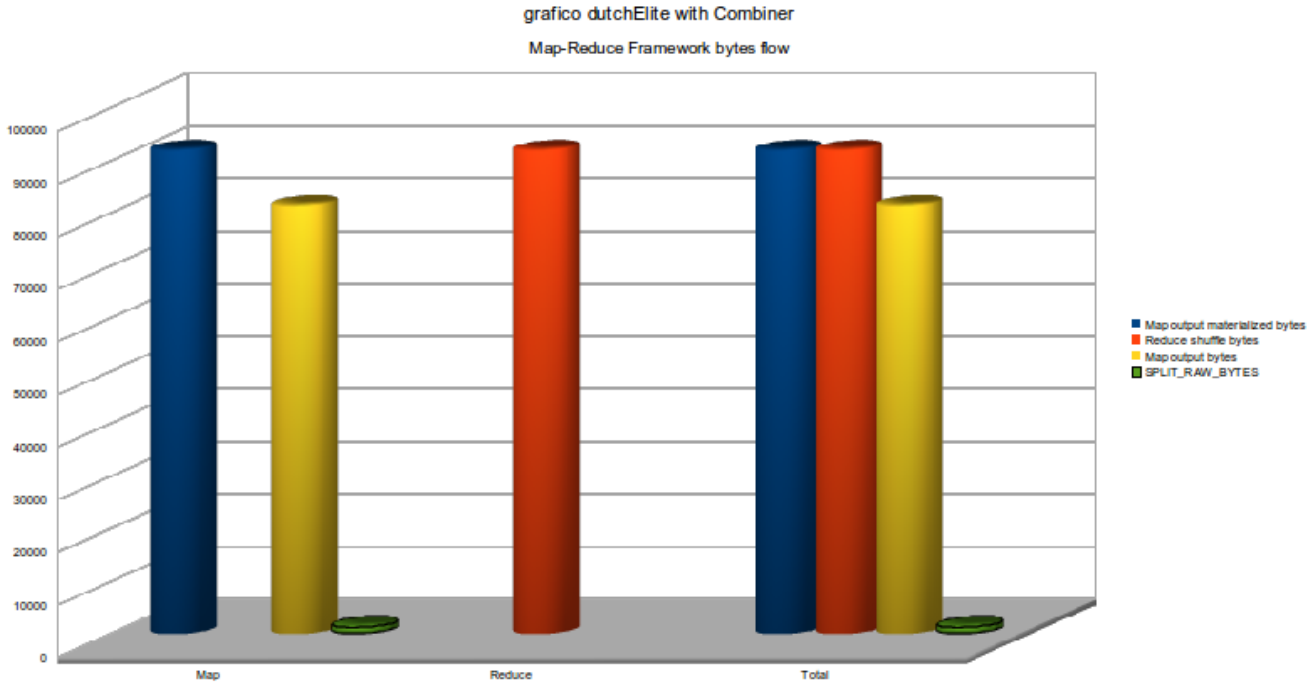


grafico dutchElite with Combiner
Map-Reduce Framework bytes flow
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Map
Reduce
Total
Map output materialized bytes
Reduce shuffle bytes
Map output bytes
SPLIT_RAW_BYTES

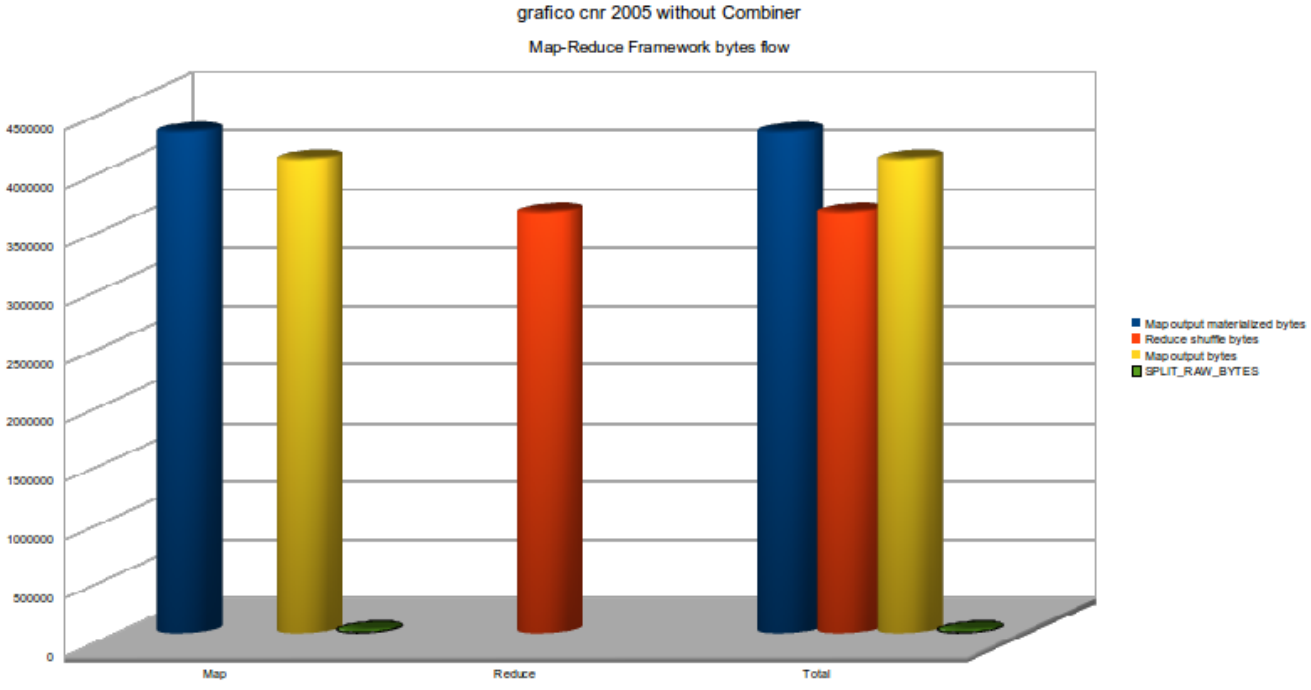


grafico cnr 2005 without Combiner
Map-Reduce Framework bytes flow
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total
Map output materialized bytes
Reduce shuffle bytes
Map output bytes
SPLIT_RAW_BYTES

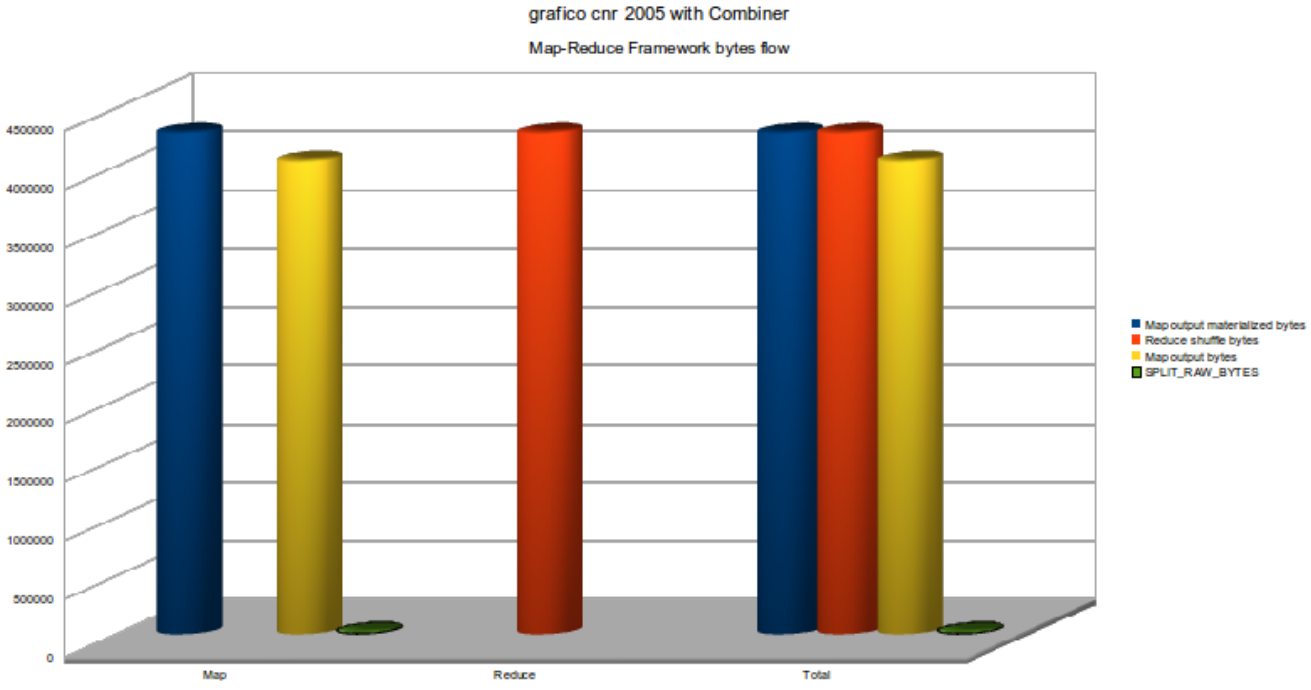


grafico cnr 2005 with Combiner
Map-Reduce Framework bytes flow
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Map
Reduce
Total
Map output materialized bytes
Reduce shuffle bytes
Map output bytes
SPLIT_RAW_BYTES

Appendice B

Fonti degli input

Tabella B.1: Fonti dei grafi che sono stati usati per costruire l'input di HitSoop

Nome	Autori	Url	Descrizione
dutchElite	Pajek datasets (V. Batagelj and A. Mrvar)	`http://vlado.fmf.uni-lj.si/pub/networks/data/`	Grafo di dati dell'amministrazione pubblica Olandese
cnr-2005	P. Boldi and S. Vigna	`http://law.dsi.unimi.it/`	rete del dominio del Centro Nazionale di Ricerca Italiano

Ringraziamenti

Voglio ringraziare il Professor Luigi Laura e l'ing. Donatella Firmani per avermi seguito e consigliato nel corso di questo lavoro di tesi, mia mamma Stefania per la grande pazienza che ha dovuto conservare nello starmi vicino, e Blessy per l'infinito sostegno e tifo che mi hanno sostenuto durante i mille crash del progetto e "caricato" quando finalmente sono apparsi risultati positivi.