TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

SDCC 2023-2024

PAGE RANK

HTTPS://GITHUB.COM/0334321-LSE/SDCC-DISTRIBUITED-PAGE-RANK

AGENDA

01 INTRODUZIONE

02 MAP REDUCE

03 ALGORITMO

04 CONTAINER & RUOLI

05 DOCKER COMPOSE

06 IMPLEMENTAZIONE

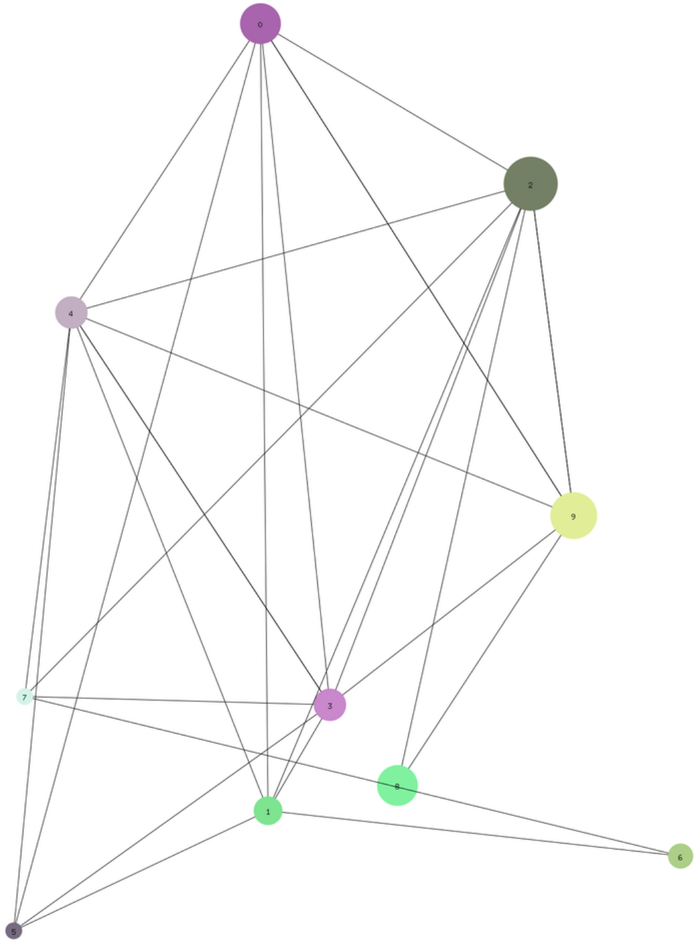
07 POSSIBILI MIGLIORAMENTI

INTRODUZIONE

Implementazione dell’algoritmo di Google PageRank, per la valutazione dell’importanza di nodi all’interno di un grafo.

Soluzione distribuita, caratteristiche:

- MapReduce
- Docker-Compose
- AWS-EC2



MAP REDUCE

Cos'è?

Un modello di programmazione ideato da Google.
Costruito per l'elaborazione in parallelo su grande volume di dati .
Garantisce:

- Elevata scalabilità, grazie alla separazione dei task
- Tolleranza ai guasti, grazie ai molteplici nodi

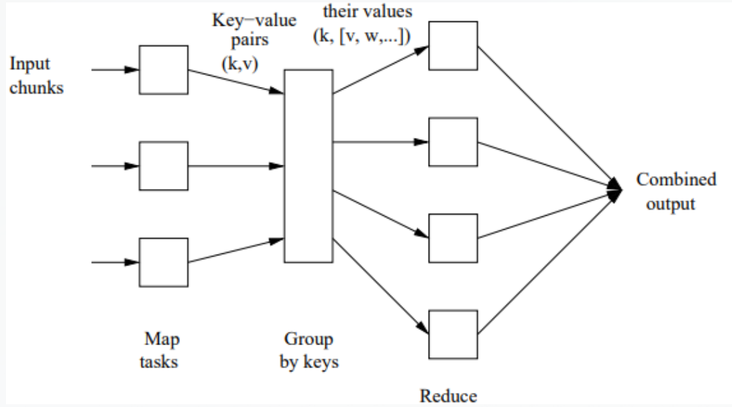
Basato su due funzionalità principali:

- **Map:** Prende in input una coppia <Chiave K, valore V> e produce delle coppie intermedie.
- **Reduce:** Riceve una coppia <Chiave K, Array V[]>, combinai valori v associandone uno soltanto alla chiave

Tra le due fasi c'è una terzo step.

- **Shuffle:** Raggruppa i dati ottenuti nella fase di map in base alla chiave, prepara l'input della fase di reduce.

Funzionamento



MAP REDUCE

In particolare

Dato un grafo G, ed un nodo N ∈ G:

- **Map:** In input <N.adjacency_list, N.page_rank> calcola lo share da distribuire tra tutti i vicini di N
- **Shuffle:** Raccoglie tutti gli share prodotti nella fase di map e li assegna alla chiave corrispondente (N.ID)
- **Reduce:** In input <N, {p1,p2,..pn}> e calcola il nuovo pagerank.

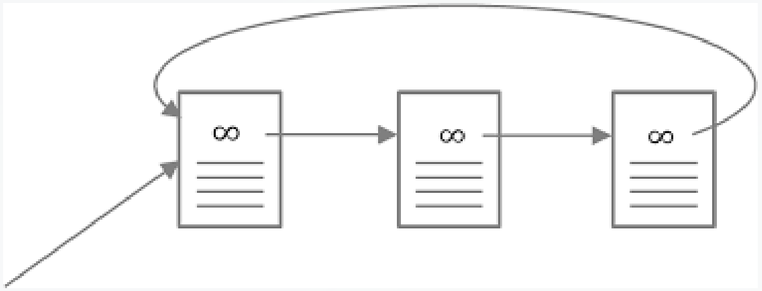
Necessario gestire due problematiche:

- **Sink**: nodi o gruppi di nodi che non hanno link uscenti, trattengono il valore del pagerank tra di loro.
- **Random Jump Factor:** possibilità che un navigatore non segua sempre un link esistente.

Aggiungere fase

- **CleanUp:** Tiene conto della massa persa a causa dei sink e della possibilità di random jump.

Esempio di Sink



ALGORITMO

Base

L'idea: l'importanza di una pagina si calcola in base al numero di link entranti e alla loro rilevanza.
Partendo dalla formula base, siano:
u: Singola pagina web.
V_u: Insieme di pagine che puntano a u.
N_v: Numero di link uscenti da v.

$$R_{i+1}(u) = \sum_{v \in V_u} \frac{R_i(v)}{N_v}$$

Scaled

Successivamente si modella il salto su pagine casuali, estendendo la formula base, siano:
E(*u*): Vettore di probabilità sulle pagine
c: Damping factor
Questa viene effettivamente usata nella fase di Reduce.

$$R_{i+1}(u) = \sum_{v \in V_u} \frac{R_i(v)}{N_v} + (1 - c)E(u)$$

ALGORITMO

CleanUp

Per gestire il contributo dei sink, ci sono due step nella fase di clean-up, rispettivamente:

- **step 1**: in cui si ottiene il contributo di tutti i sink nel grafo.
- **step 2**: si calcola nuovamente il PR aggiornando il precedente, usando:

$$p' = c\,(p + \frac{m}{N}) + (1 - c)\frac{1}{N}$$

Recap

Di seguito, tutti gli step dell'algoritmo:

- Inizializzazione del **PR** a 1/N.
- Distribuzione del **PR** tra i nodi vicini mediante funzione di Map.
- Calcolo del **PR** mediante scaled formula usando funzione di Reduce.
- Fase di clean-up per includere il contributo dei nodi sink.

Questi ripetuti fino alla convergenza o per num max.

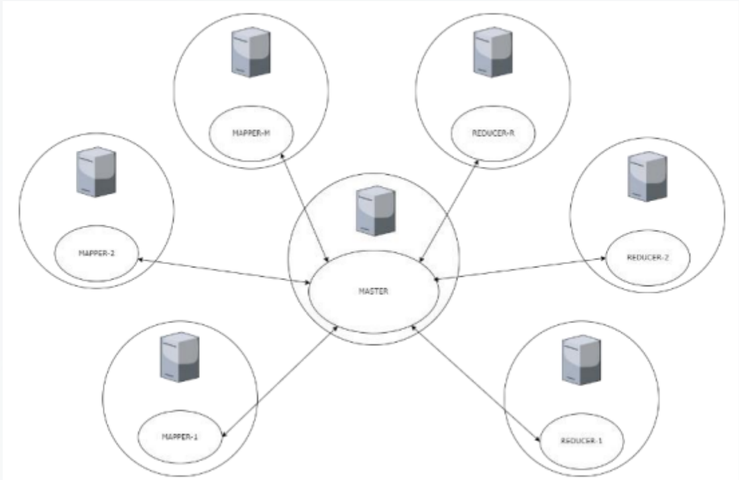
CONTAINER & RUOLI

MAPPER

Uno delle due tipologie di worker, numero arbitrario* di container in ascolto che svolgono la fase di map.

MASTER

Il client dell'applicazione, viene eseguito su un container ad hoc. Comunica con i nodi worker ai quali richiede l'esecuzione dei vari task.



REDUCER

Uno delle due tipologie di worker, numero arbitrario container in ascolto che svolgono la fase di reduce.

*nei limiti di Docker-Compose e della macchina host

DOCKER COMPOSE

Docker compose è lo strumento scelto per la gestione dei container. Ogni elemento dell'applicazione è istanziato su un container, la comunicazione tra di essi avviene tramite gRPC. L'utilizzo del paradigma di MapReduce combinato con i container ha reso l'applicazione:

Modificando il file di configurazione è possibile decidere quanti nodi worker. istanziare.

SCALABILE

Il crash di uno o più nodi worker non comporta il fallimento nell'applicazione, purché ne rimanga almeno 1 per tipologia.

FAULT TOLERANT

Tramite l'utilizzo di go-routine, le chiamate del client ai worker sono rese parallele. Questo ha comportato una velocizzazione nell'applicaizone.

PARALLELA

IMPLEMENTAZIONE

Ognuno dei tre componenti ha una propria porzione di codice:

- **Master:** è il più corposo. Instaura le connessioni con i vari worker ed esegue l'algoritmo: è dove viene eseguito il main loop. Gestisce le chiamate con uno scheduling round robin. Tramite un servizio di health checking verifica o meno che i container siano attivi prima di contattarli.

- **Mapper:** implementa il servizio di map e la sua porzione di clean-up. Ogni sua istanza è in ascolto su porte differenti, rimangono attive anche dopo che il client ha terminato.
- **Reducer:** Implementa il servizio di reduce e la sua porzione di clean-up. Ogni sua istanza è in ascolto su porte differenti, rimangono attive anche dopo che il client ha terminato.

Tutta l'applicazione viene eseguita su un'istanza micro di EC2 di AWS

POSSIBILI MIGLIORAMENTI

Migliorare la politica di scheduling, testare una soluzione come **RabbitMQ**. Lavorare su **Kubernetes** per poter incrementare la capacità di scaling e renderlo dinamico.

SCHEDULING & SCALING

Aggiornare il codice di stampa del grafo il quale risulta essere molto pesante e non in grado di gestire grafi più grandi di 20/30 nodi.

PRINTING

Applicare tecniche di vettorizzazione del codice per migliorarne le performance generali e rendere l'applicazione ancor più performante.

CODE OPTIMIZATION

GRAZIE PER L'ATTENZIONE

Luca Saverio Esposito - 0334321