Big Data and Reasoning

Daniele Avolio

Anno 2023/24

Contents

1 Teoria

In questi appunti ci saranno semplicemente gli appunti del corso di Big Data and Reasoning, tenuto dal professor Francesco Ricca presso l'Università della Calabria.

Non aspettatevi 100% di correttezza, sono appunti presi a lezione e quindi potrebbero contenere errori!

2

3 Lezioni di Laboratorio

Domande dell'esame possibili

In questa sezione ci sono domande che mi vengono fuori quando leggo i PDF e che quindi mi aspetto che possano essere inserite all'interno delle prove intermedie orale.

Buona fortuna a tutti!

4 Domande: Big Data Introduction

Domanda 4.1. *Cosa sono i Big Data?*

I Big Data sono un insieme di dati che viene differenziato dai comuni dati dal fatto che per essere gestiti non sono piu' abbastanza le vecchie tecnologie e metodi di gestione dei dati. I fattori che influenzano sono la velocita con cui vengono generati, il volume di dati che viene genegrato e la varieta' di dati che ci sono all'interno del dataset.

Domanda 4.2. *Quali sono le 5 V dei big data?*

Le 5 V dei big data fanno riferimento alle caratteristiche di essi, che sono:

- **Volume**: La quantita' di dati enorme che vanno gestiti. La scala che va fino ai Zettabyte.
- **Varieta'**: I vari tipi di danni che sono all'interno del database da analizzare e gestire. Possono essere immagini, audio, video, testo, ecc...
- **Velocita'**: La velocita' con cui i dati vengono generati e devono essere analizzati.
- **Veracita'**: L'incertezza e l'imprecisione dei dati che vengono generat e che quindi devono essere analizzati per capirne la *qualita'*.
- **Valore**: Il valore che i dati hanno per l'azienda che li possiede. Ad esempio vantaggi di business e analisi possibili da condurre.

Domanda 4.3. *Qual e' stata la prima rivoluzione nei database?*

Boh era quella iniziale dove praticamente si usava un modello gerarchico e veniva utilizzato Cobol. Diciamo che per gestirne uno sarebbe servito un esperto di database.

Domanda 4.4. *Qual e' stata la seconda rivoluzione nei database?*

L'introduzione del **modello relazionale** che porta i dati da essere tutti in unici tabelle a diverse con una struttura ben specifica.

Qui nasce anche il concetto di **transzione** che deve essere ACID:

- A: Atomica
- C: Consistente
- I: Isolata
- D: Duratura

Domanda 4.5. *Qual e' stata la terza rivoluzione nei database?*

Quando sono stati inventati i social network, grandissime moli di dati sono cominciate ad essere generate. Le strutture di un tempo non erano ancora abbastanza solide da poter gestire delle moli di dati tali. Per questo motivo **nuove tecniche** sono state inventate. Quindi nascono anche i database non relazionali. Ci sono state varie invenzioni come **lo sharding**. Sono stati creati altri tipi di database come quelli **(chiave, valore)**. Principalmente, pero', fu Google con il **distributed file system** a dare il via a questa rivoluzione.

Un concorrente a google fu Yahoo con HDFS (Hadoop Distributed File System).

5 Domande: GoodBye SQL

Domanda 5.1. *Differenza tra Scale Up e Scale Out*

Scale Up o scalare verticalmente significa *aggiungere risorse ad una macchina sola*. Ad esempio, significa potenziare una macchina tale da poter gestire piu carico di lavoro. Questo e' un approccio che e' stato utilizzato per molto tempo e che ha portato a creare macchine sempre piu potenti. Il problema e' che questo approccio ha un limite fisico, ovvero non si puo' andare avanti all'infinito. Inoltre, questo approccio e' molto costoso.

Scale Out o scalare orizzontalmente e' una cosa piu economicamente sostenibile, poiche' si basa sul concetto di *aggiugnere piu macchine che lavorano tra loro per gestire il carico di lavoro*. Questo e' notevolmente piu economico perche' i costi di una macchina sono molto piu bassi rispetto a quelli di una macchina potenziata.

Domanda 5.2. *Quali sono stati i primi tentativi di scalare orizzontalmente*

I primi tentativi furono quelli di costruire dei **Memcached server**, in cui gli utenti potessero effettuare letture senza andare a toccare i database principali. Un'altra tecnica era quella della **replicazione dei dati** su piu' databases.

Quando va bene questo? Quando le letture sono notevolmente maggiori rispetto alle scritture. Questo perche' se ci fossero tante scritture, ci sarebbe un bottleneck architetturale, dato che il database su cui vengono effettuate le scritture continua a rimanere 1 e 1 soltanto.

Domanda 5.3. *Cosa significa Sharding e come viene usato?*

Sharding significa **tagliare orizzontalmente un database** per splittarlo su diverse macchine. Questo comporta un aumento delle performance sopratutto in scrittura, poiche' aumenta il numero di macchine dove si puo' scrivere, ma aumenta anche la **complessita' di gestione del sistema.**

Bisogna anche effettuare il taglio con un criterio, poiche' poi quando i dati devono essere recuperati si vuole un tempo di accesso comunque accettabile senza dover effettuare una ricerca su tutti e n i database.

Problemoni:

- Complessita': Come gia' detto, gestire un sistema del genere diventa complesso e richiede una conoscenza elevata
- SQL: Non si puo usare SQL su diverse shard, quindi ci vuole un meccanismo dietro le quinte che gestisca il tutto
- ACID LOSS: Si perdono le transizioni ACID perche' su piu macchine non esiste, poiche' si ritornerebbe ad avere un bottleneck

Domanda 5.4. *Spiega il teorema di CAP*

Il teorema di CAP che sta per **Consistentcy, Availability e Partition Tolerance** dice che **non si possono avere tutti e 3 i requisiti** in un sistema distribuito. Provando ad ottenere tutti e 3 contemporaneamente, si andra' comunque a perdere 1 dei 3.

- Consistentcy: Ogni utente deve avere la stessa visualizzazione del contenuto in qualsiasi istante
- Availability: Ogni richiesta deve essere servita
- Partition Tolerance: Il sistema deve funzionare anche se alcune macchine non sono disponibili

Un esempio pratico: *Immagina di avere un servizio di database distribuito per un'applicazione di shopping online. In questo sistema, hai utenti che effettuano ordini e aggiornano il proprio carrello degli acquisti in tempo reale. Il database è distribuito su più server in diverse località geografiche per garantire la ridondanza e la tolleranza ai guasti.*

Coerenza (Consistency): Supponiamo che tu abbia implementato una forte coerenza dei dati, il che significa che ogni aggiornamento del carrello degli acquisti di un utente deve essere immediatamente riflesso su tutti i server. Questo assicura che tutti i server abbiano sempre una vista coerente dei dati. Tuttavia, quando si verifica una partizione di rete tra due gruppi di server, il sistema deve scegliere tra attendere la risoluzione della partizione (rendendo il servizio non disponibile per un certo periodo) o accettare il rischio di avere carrelli degli acquisti temporaneamente non coerenti tra le due parti del sistema.

Disponibilità (Availability): Se desideri massimizzare la disponibilità, il sistema dovrebbe continuare a permettere agli utenti di aggiornare i propri carrelli degli acquisti, anche se si verifica una partizione di rete tra i server. Tuttavia, ciò può comportare il rischio di avere dati non coerenti tra i due lati della partizione durante il periodo di separazione.

Tolleranza alla partizione (Partition Tolerance): Il sistema deve essere in grado di tollerare le partizioni di rete. Questo significa che il servizio dovrebbe funzionare in presenza di guasti di rete o partizioni geografiche, ma potrebbe comportare una temporanea mancanza di coerenza o disponibilità in situazioni di partizione.

Quindi, in questo esempio, puoi vedere che il teorema CAP si applica. Quando si verifica una partizione di rete, devi fare una scelta tra coerenza e disponibilità. Non è possibile garantire contemporaneamente entrambe le proprietà. Il sistema deve tollerare la partizione ma potrebbe sacrificare la coerenza o la disponibilità a seconda delle decisioni di progettazione prese.

Domanda 5.5. *Spiega: No Go Zone, Eventual Consistentcy, Strict Consistentcy*

- No Go Zone:

$$\text{Consistentcy} \cap \text{Availability} \cap \text{Partition Tolerance}$$

- Strict Consistentcy:

$$\text{Consistentcy} \cap \text{Partition Tolerance}$$

- Eventual Consistentcy:

$$\text{Availability} \cap \text{Partition Tolerance}$$

Domanda 5.6. *Come funziona Amazon Dynamo?*

- Database non relazionale alternativo.
- Accesso basato su chiave primaria.
- Dati recuperati da una chiave sono oggetti binari non strutturati.
- La maggior parte degli oggetti è piccola, sotto 1 MB.
- Dynamo permette di sacrificare la coerenza per garantire disponibilità e tolleranza alle partizioni.

Caratteristiche architetturali chiave

- Consistent hashing: usa l'hash della chiave primaria per distribuire i dati tra i nodi del cluster in modo efficiente.
- Coerenza regolabile: l'applicazione può bilanciare la coerenza, le prestazioni di lettura e scrittura.
- Versionamento dei dati: permette la gestione di più versioni di oggetti nel sistema, richiedendo la risoluzione delle versioni duplici da parte dell'applicazione o dell'utente.

Principali caratteristiche di Dynamo

- Hashing consistente: assegna le chiavi ai nodi in modo efficiente, consentendo l'aggiunta o la rimozione di nodi con minimo riassestamento.
- Coerenza regolabile: permette di scegliere tra coerenza, prestazioni di lettura e scrittura.
- Versionamento dei dati: gestisce più versioni di oggetti, richiedendo la risoluzione delle versioni duplici da parte dell'applicazione o dell'utente.

Domanda 5.7. *Qual e' la differenza tra SQL e NOSQL*

Nei database NOSQL si perde la strictness del modello relazionale e si parla piu' di **modello a documenti**. Questo significa che i dati sono salvati con una struttura diversa tra loro, non obbligatoriamente. Il formato con cui si salvano e' spesso **xml** o **json**.

Ogni documento possiamo dire che rappresenti una riga di un database relazionale. All'interno del documento ci potrebbero essere anche documenti innestati e liste.

6 Domande: Google Stack e Hadoop

Domanda 6.1. *Come funziona il file system di Google*

Il GFS o Google File System si basa sul concetto di avere costantemente i loro servizi attivi, avendo a disposizione un **numero enorme** di macchine e server che garantiscono l'**availability** costante.

La consistenza non e' un qualcosa che viene garantito, ma viene garantito una **fault tolerance** costante. Anche perche' avendo 1.000.000 di macchine, il fatto che 1 non funzioni diventa la normalita'. Ci si aspetta di avere anche dei sistemi che ripristino il sistmea quando questo accade.

- Faul tolerance: tante macchine che permettono di avere un'espansione orizzontalme
- Banda sostenuta: garantire sempre il servizio a discapito della latenza
- Atomicita' con minimi problemi di sincronizzazione
- **Availability**

Si basa su assunzioni di dover gestire tanti file dell'ordine di GB, di avere *poche letture random e tante letture a batch.*

Domanda 6.2. *Qual e' la struttura del GFS*

- Tanti client: accedono al servizio
- Un Master: Gestisce i metadati e comunica con i chunkserver
- Tanti chunckserver: Salva i dati e sul disco locale. Spesso ogni chunck viene duplicato per garantire la fault tolerance e availability.

Diciamo che la procedura di comunicazione dell'applicazione e' che:

1. Il client chiede al master quale chunckserver contattare
2. Il client contatta il chunck server
3. Vengono effettuate le operazioni sui chunk
4. Il master controlla lo stato dei chunckserver e aggiorna i metadati

Domanda 6.3. *Come funziona il paradigma MapReduce?*

Nota: IN un programma parallelo quali sono i maggiori bottleneck? **I LOCK**

Il paradigma si basa su due fasi:

1. Map: I dati vengono partizionati in chunks che saranno poi gestiti in parallelo

2. Reduce: Si combina l'output dei vari mappers per avere il risultato finale

Da notare che viene effetuato **con bruteforce** e il codice fa parecchio schifo mi sa.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Domanda 6.4. *Come funziona BigTable?*

BigTable e' un database distribuito che si basa su GFS e MapReduce. E' un database **non relazionale** che si basa su una struttura di **colonne** e **righe**.

Sparsa distribuita persistente multi dimensionale ordinata MAPPA

(riga: stringa, colonna:stringa, tempo: int64) $\rightarrow$ stringa

Le pagine web sono salvate con URL al contrario perche per distribuire i dati in modo uniforme sui server, BigTable deve utilizzare una chiave di partizionamento. In questo caso, la chiave di partizionamento è l'URL della pagina web.

Per evitare che tutti i dati vengano memorizzati su un unico server, BigTable utilizza una funzione di hash per trasformare l'URL in una chiave di partizionamento. Tuttavia, se l'URL fosse utilizzato come chiave di partizionamento così com'è, ci sarebbe il rischio che tutti i dati relativi a un singolo sito web finiscano sullo stesso server.

Per evitare questo problema, Google salva gli URL al contrario. In questo modo, l'hash della chiave di partizionamento viene distribuito in modo più uniforme sui server, migliorando le prestazioni del database.

- Righe: sono stringhe arbitrarie che identificano la riga
- Colonne: sono stringhe arbitrarie che identificano la colonna. Si chiamano famiglie. Spesso i dati della stessa famiglia sono dello stesso tipo.
- Timestamp: Viene usato per poter avere una versione dei dati e poter fare rollback

Domanda 6.5. *Implementazione di BigTable*

Ha 3 elementi principali.

Il client che usa le API per comunicare. Un **Master** che gestisce i tablet servers e garantisce il load balancing. **I tablet servers** che gestiscono un insieme di tablets.

I tablet servers gestiscono il partizionamento dei dati e le richieste in lettura e scrittura.

Notare che i **dati del client non passano dal master ma dai tablet servers**

Domanda 6.6. *Come funziona Hadoop*

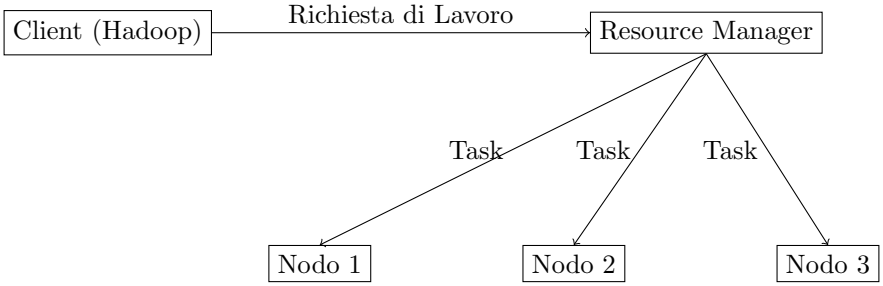
Hadoop e' un framework open source che permette di gestire grandi quantita' di dati in modo distribuito. Si basa su HDFS e MapReduce.

Lo **schema on read** permette di non dover progettare uno schema per i dati. Lo schema dipende dai dati che si andranno a gestire. Anche questo si basa sull'aggiungere hardware comodo ed economico per scalare.

- Scale out
- Chiave VALORE
- Functional Programming
- Offline batch (non realtime)

Domanda 6.7. *La struttura di Hadoop*

- NameNode: gestisce i metadati e i file system (**UNO E UNICO**)
- DataNode: ogni macchina slave avra' un datanode che gestisce i dati. (**SONO TANTI**)
- DataNode secondario: Usato come backuo diciamo (**tipicamente uno**)
- JobTracker: gestisce i job e le richieste dei client (**Uno solo, master**)
- TaskTracker: gestisce i task sui vari slave node(**tanti, slave**)



7 Domande: Column and in Memory database

Domanda 7.1. *Per quale motivo sono nati i database a grafo?*

I database relazionali immagazzinavano dati e le relazioni che ci sono tra loro. La controparte di queste relazioni e' il **grafo**. Ma per modellare un grafo non era facile con i sistemi di database relazionali. Ancora peggio i database non relazionali non riuscivano a modellare per bene i grafi, sopratutto quando si parla di relazioni tra oggetti (non ci sono i join).

Comunque in generale per questa parte non c'e' molto da dire. Cioe' provarono a costruire dei database basati sui grafi. Avevano alcune proprieta' decenti come la **index free adjacency** che permetteva di muoversi all'interno del grafo con senza dover cercare gli indici. Comunque c'erano dei problemi.

C'erano alcuni engine che permettevano di calcoalre questi grafi:

- Apache Giraph
- GraphX
- Titan

Domanda 7.2. *Cos'e RDF*

RDF significa **resource description framework RDF**. E' un framework per rappresentare le informazioni sul web.

$$entity : attribute : value \tag{1}$$

Utilizzala sintassi XML e il linguaggio SPARQL

```
1  SELECT ?object
2      FROM <http://example.org>
3      WHERE {
4          <http://example.org> <http://example.org/property> ?
   object
5      }
```

Listing 1: Esempio di query SPARQL

Domanda 7.3. *Differenza tra OLTP e OLAP*

OLTP sta per **online transaction processing** e OLAP sta per **online analytical processing**. OLTP e' un sistema che permette di gestire le transazioni ed e' comodo sopratutto quando si ha un carico elevato di **transizioni in scrittura**. Gli ecommerce ad esempio sfruttano molto i sistemi OLTP. I sistemi OLTP hanno uno schema **row oriented**.

I sistemi OLAP sono piu' utili in caso di sistemi che hanno un elevato numero di richieste in **lettura** come ad esempio i software che devono calcolare spesso medie e misure, come ad esempio i software di business intelligence e data mining. I sistemi OLAP hanno uno schema **column oriented**.

Domanda 7.4. *Per cosa e' buono avere database row o column*

Ad esempio, poniamo il caso di avere un database con 10 colonne e a noi interessa la colonna 'eta'

id	nome	cognome	eta	indirizzo
1	Mario	Rossi	20	Via Roma
2	Luca	Bianchi	30	Via Milano
3	Marco	Verdi	40	Via Napoli

Se a noi interessa ad esempio la media delle eta', con un database row oriented dobbiamo scorrere tutte le righe e prendere la colonna eta' e calcolare la media. Con un database column oriented invece, abbiamo gia' la colonna eta' e quindi possiamo calcolare la media direttamente.

Domanda 7.5. *A cosa serviva C-Store Storage e come funzionava*

C-store sfruttava la ridondanza degli elementi con le proiezioni. Una proiezione e' una ripetizione di alcune colonne che sono accesse frequentemente e salvate sul disco. Questo aumentava le performance.

Aveva una pesante compressione delle colonne. Cioe, le colonne venivano comprese e salvate sul disco. Questo permetteva di avere un accesso piu' veloce.

- C-Store storage utilizza la compressione delle colonne per migliorare le prestazioni.
- I dati sono archiviati in ordine ordinato e le colonne sono pesantemente compresse.
- Il sistema utilizza un Writeable (Delta) Store e un Read-Optimized Store per gestire le operazioni di scrittura e lettura.
- C-Store Storage utilizza la tecnica K-safety per garantire l'affidabilità dei dati.

Domanda 7.6. *Per quale motivo i sistemi heavy write non funzioano bene su ssd?*

La risposta e' molto semplice: Per scrivere su SSD bisogna cancellare e riscrivere. Quindi se si ha un sistema che scrive molto, si avranno molti cicli di scrittura e cancellazione. Questo ha un effetto di bottleneck sul sistema.

Domanda 7.7. *Database cacheless*

I database in memoria principale (Main Memory DB) sono progettati per funzionare senza cache, ovvero senza la necessità di memorizzare temporaneamente i dati su memoria cache per minimizzare l'accesso ai dati su disco.

A differenza dei database tradizionali basati su disco, i database in memoria principale memorizzano tutti i dati direttamente in memoria principale, senza la

necessità di accedere ai dati su disco. In questo modo, l'accesso ai dati è molto più veloce e non è necessario utilizzare una cache per minimizzare l'accesso ai dati su disco.

In altre parole, la cache è inutile in un database in memoria principale perché tutti i dati sono già memorizzati in memoria principale. Non c'è bisogno di memorizzare temporaneamente i dati su cache perché non c'è bisogno di minimizzare l'accesso ai dati su disco

Domanda 7.8. *Cosa e' Berkeley Analytics Data Stack*

BDAS è un insieme di strumenti open source per l'analisi dei dati formato da:

- Spark: un framework per l'elaborazione distribuita
- Mesos: un sistema operativo per i data center (Gestisce i cluster)
- Tachyon: un sistema di file distribuito

Domanda 7.9. *Cosa e' SPARK*

Spark e' un framework che permette di lavorare sui dati in modo distribuito.

Tratta i dati come RDD, cioe' *resilient distributed datasets*. Questi RDD sono immutabili e sono distribuiti sui nodi del cluster. Spark permette di fare operazioni su questi RDD in modo parallelo.

Non abbiamo visto ancora niente di Spark, quindi non posso dire altro ad essere onesto.

8 Domande: MapReduce Introduction

Domanda 8.1. *Per quale motivo Google ha inventato Map Reduce?*

Perche' aveva bisogno di un sistema che permettesse il processamento di un quantitativo enorme di file tra diverse macchine, in modo parallelo, con una tolleranza ai guasti e senza usare codici troppo complessi. Per questo motivo, nasce questo sistema che permette di fare tutto cio'. Ah questo andava fatto pero' su diverse macchine che non costassero un patrimonio, quindi doveva essere anche economico.

> I programmatori che non erano esperti di programmazione distribuita dovevano essere in grado di effettuare operazioni su sistema distribuiti a larga scala.

Domanda 8.2. *Cos'e' MapReduce*

MapReduce e' un modello di programmazione che soddisfa i seguenti punti:

- Programma per processare grandi insiemi di dati
- Partizionare i dati di input
- Schedulare l'esecuzione su un cluster di macchine
- Gestire i guasti
- Gestire la comunicazione tra le macchine

Domanda 8.3. *Quali sono i componenti principali di MapReduce*

I principali componenti di MapReduce sono due funzioni:

- Funzione mapping
- Funzione reducing

$$map(k1, v1) \rightarrow list(k2, v2)$$

Prende in **input** una coppia chiave valore e restituisce in **output** una lista di coppie chiave valore.

$$reduce(k2, list(v2)) \rightarrow list(v2)$$

Prende in **input** una chiave e una lista di valori e restituisce in **output** una lista di valori. (Oppure una coppia chiave valore)

Praticamente il workflow e':

1. Map: Si prende un file e lo si divide in blocchi
2. Map: Si applica la funzione map a tutti i blocchi
3. I risultati vengono salvati sulle macchine locali
4. Reduce: Si prendono i risultati e si applica la funzione reduce
5. Reduce: Si salvano i risultati finali sul GFS o HDFS

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Domanda 8.4. *Cosa succede in questo caso se il file per word count ha solamente 1 parola ripetuta 1000 volte?*

In questo caso, il lavoro del mapping sara' super parallelo, ma il lavoro del reducing sara' sequenziale, perche' tutti i risultati del mapping verranno mandati al reducer che si occupera' di fare il reduce.

Domanda 8.5. *La fase di shuffle cosa fa?*

La fase di shuffle e' la fase che si occupa di **mandare i risultati del mapping al reducer corretto**. In questo caso, il reducer corretto e' quello che si occupa di fare il reduce della parola che e' stata mappata.

Domanda 8.6. *Workflow di ma reduce nel dettaglio*

1. Viene splittato il file di input in diversi file
2. Si passa il programma al cluster di macchine
3. La macchina master divide il lavoro tra i worker
4. Ci sono M task di mapping e R task di reducing
5. I worker leggono il contenuto del file di input assegnato dal master
    - Per ogni coppia chiave valore, viene applicata la funzione map
    - Le coppie vengono salvate nella memoria buffer
    - I risultati vengono salvati in memoria locale periodicamente leggendo dalla memoria buffer in R regioni.
6. Il master viene informato di dove sono queste R regioni
7. Manda le informazioni sulla regione ai reducer
8. I reducer leggono i risultati dal disco locale tramite procedure remote
    - I risultati vengono ordinati per chiave
    - I risultati vengono passati alla funzione reduce
    - I risultati vengono aggiunti alla fine del file output (*appending*)
9. Tutte le macchine comunicano con il master che hanno terminato il lavoro
10. Il master manda il segnale di terminazione al client
11. Il file output e' disponibile per il client

Domanda 8.7. *Come fa MapReduce ad essere fault tolerant?*

Dobbiamo dire innanzitutto che abbiamo due tipi di fallimenti:

- Worker Failure

- Master Failure

Worker Failure: Il master tenta di tanto in tanto di pingare il worker. Se esso non risponde entro un certo limite di tempo, allora il master **segna il worker come inattivo**, e quindi questo implica che:

- Le task che dovevano essere eseguite dal worker e anche quelle gia completate vengono riassegnate ad altri worker
- Le task gia completate vengono eseguite dinuovo da altri worker: questo perche' ogni worker scrive sulla memoria locale, e quei file sono ormai perduti e non piu accessibili
- Discorso diverso se il worker era un **reducer**: Se il reducer ha gia' completato il suo lavoro ma non risponde, la sua task non va riassegnata perche' ha gia' inserito l'output della funzione reduce nel file di output corretto, che e' disponbiile sul filesystem.

Master Failure: Se il master fallisce gli facciamo il funerale. Si ferma la computazione di MapReduce.

Domanda 8.8. *Spiega la locality di MapReduce*

In MapReduce si vuole risparmiare banda piu' che si puo'. Per questo motivo, i file hanno un fattore di replicazione che dipende dalla configurazione del cluster. I dati iniziali vengono salvati su dischi locali delle macchine e in multipla copia.

Quando viene **assegnato un task di map**, si tende ad assegnarlo alle macchine che hanno i file *vicini* oppure gia' salvati all'interno della propria memoria come copia. Questo minimizza il consumo di banda.

Domanda 8.9. *Spiega la task granularity di MapReduce*

Diciamo che MapReduce tende a scegliere un numero di Mappers M tale che ogni compito di mapping richieda circa 16-64 MB di input.

Sceglie R un piccolo multiplo delle macchine del cluster, per bilanciarne il carico.

Sceglie il numero di M e R in modo che siano molto piu' grandi del numero di macchine nel cluster.

Domanda 8.10. *Spiega backup tasks in MapReduce*

Allevia il problema dei **ritardatari** o stragglers. Cioe', il master schedula delle esecuzioni backup delle task in corso. La task viene segnata come completa indipendentemente dal fatto che la task originale sia completata o meno. Cioe' la cosa che gli interessa di piu' e' che la task sia completata da qualche worker.

9 Domande: MapReduce Hadoop Basics

Domanda 9.1. *I data types di Hadoop*

Hadoop non ammette qualsiasi tipo di dati, poiche' per essere mandati i valori, devono essere **serializzati**.

Definizione 9.1. *(Serializzazione) La serializzazione e' il processo di conversione di un oggetto in un formato che puo' essere memorizzato (ad esempio, in un file o in un buffer di memoria) o trasmesso (ad esempio, attraverso una connessione di rete) e ripristinato in un oggetto con le stesse caratteristiche dell'originale.*

Ci sono diverse interfacce che devono essere implementate per le **chiavi** e per i **valori**.

- Valori: I valori devono implementare l'interfaccia **Writable**
- Chiavi: Le chiavi devono implementare l'interfaccia **WritableComparable**$\langle T \rangle$. C'e' bisogno di questo comparable perche' successivamente si andra' a fare un sort sulle chiavi.

Classe	Descrizione
BooleanWritable	Valore booleano
ByteWritable	Valore byte
DoubleWritable	Valore double
FloatWritable	Valore float
IntWritable	Valore intero
LongWritable	Valore long
Text	Valore stringa formato UTF8
NullWritable	Valore nullo usato quando la chiave non e' richiesta

Table 1: Tipi di classe di Hadoop

Domanda 9.2. *Si possono creare classi personalizzate?*

La risposta e; **si**. Bisogna fare l'override dei metodi:

- readFields(DataInput in)
- write(DataOutput out)
- compareTo(T o)

Domanda 9.3. *Cosa fa la classe MAPPER?*

La classer Mapper e' una classe astratta che deve essere estesa per implementare il metodo **map**.

public class **Mapper**$\langle KEYIN, VALUEIN, KEYOUT, VALUEOUT \rangle$

```
public class TokenCounterMapper extends Mapper<Object, Text,
Text, IntWritable>
```

Esistono gia' dei **mappers standard**.

Classe	Descrizione
IdentityMapper	Mapper che emette le coppie chiave-valore in input senza modificarle.
InvertMapper	Mapper che inverte le coppie chiave-valore in input.
RegexMapper	Mapper che emette coppie chiave-valore in base a un'espressione regolare.
TokenCountMapper	Mapper che emette coppie chiave-valore in base a un'espressione regolare.

Domanda 9.4. *Cosa fa la classe REDUCER?*

La classe reducer e' una classe astratta che deve essere estesa per implementare il metodo **reduce**.

Dato un insieme di dati che condividono la **stessa chiave**, riduce l'insieme ad un numero inferiore di elementi.

- Shuffle and sort: Prende i valori dai mapper con delle **richieste http** ed effettua un merge sort sulle chiavi
- Sort Secondario: Estende la chiave con la chiave seocndaria e definisce un comparatore di gruppo

public clas Reducer$< KEYIN, VALUEIN, KEYOUT, VALUEOUT >$

```
public class IntSumReducer<Key> extends Reducer<Key,
                        IntWritable, Key,IntWritable>
```

Ci sono anche reducer standard:

IdentityReducer	Reducer che emette le coppie chiave-valore in input senza modificarle.
LongSumReducer	Reducer che somma i valori in input.

Domanda 9.5. *Cosa sono i Combiner?*

I **Combiner** non sono altro che dei **reducer locali**. In parole povere, sono dei worker che diminuiscono il lavoro a carico dei **reducer**. *Implementa la stessa interfaccia* **Reducer** Quando ci sono molti dati, possono diminuire il traffico di rete. Non e' sempre possibile applicarlo, attenzione!

Come fanno a sapere dove prendere i dati, pero'? Gli viene detto dal **master.**

Domanda 9.6. *Cosa sono i Partitioner?*

I partitioner determinano dove vanno mandate le coppie chiave-valore che vengono sputate fuori dall'output dei mapper. **Attenzione**: I partitioner esistono solamente quando **c'e' piu' di un reducer!**

Di default, c'e' **HashParitioner** come classe di Hadoop.

Domanda 9.7. *InputFormat classe e interfaccia*

L'interfaccia definisce come l'input viene diviso e letto in Hadoop. Esistono diverse implementazioni e puoi fartene quante ne vuoi .

La classe descrive come specificare i dati di input per un lavoro di MapReduce. Divide i file di input in **InputSplits** che vengono assegnati ai Mapper individuali. Viene utilizzato un'implementazione chiamata **RecordReader** per estrarre i record di input dagli InputSplits.

Classe
TextInputFormat
KeyValueTextInputFormat
SequenceFileInputFormat
NLineInputFormat

Domanda 9.8. *Come si crea un custom input format*

Per farlo, la funzione InputFormat deve identificare tutti i file usati come dati di input e dividerli in splits.

Bisogna fornire un oggetto *RecordReader* dove bisogna iterare sui record e restituire la chiave e il valore.

```
1     public classe CustomRecordReader extends RecordReader {
2         @Override
3         public void initialize(InputSplit split, TaskAttemptContext
       context) throws IOException, InterruptedException {
4             //inizializza il record reader
5         }
6
7         @Override
8         public boolean nextKeyValue() throws IOException,
      InterruptedException {
9             //itera sui record e restituisci la chiave e il valore
10        }
11
12        @Override
13        public Object getCurrentKey() throws IOException,
      InterruptedException {
14            //restituisci la chiave corrente
15        }
16
17        @Override
18        public Object getCurrentValue() throws IOException,
      InterruptedException {
19            //restituisci il valore corrente
20        }
```

Classe	Descrizione
TextOutputFormat	Scrive i file di testo.
SequenceFileOutputFormat	Custom separator
NullOutputFormat	Non scrive alcun output.

```
21
22          @Override
23          public float getProgress() throws IOException,
        InterruptedException {
24              //restituisci il progresso
25          }
26
27          @Override
28          public void close() throws IOException {
29              //chiudi il record reader
30          }
31      }
```

Domanda 9.9. *Formati di output*

Gli output file di mapreduce usano la classe **OutputFormat**. Non ha nessuno split e ogni reducer scrive un singolo file di output. L'oggetto **RecordWriter** formatta l'output.

Domanda 9.10. *La classe Configuration*

Questa classe permette l'accesso ai parametri di configurazione.

```
1   public class Configuration extends Object
2   implements iterable<Map.Entry<String,String>>, Writable
```

Domanda 9.11. *Cosa sono i job?*

I job sono le unita' di lavoro di Hadoop. L'utente crea l'applicazione e descrive come deve essere eseguita tramite i **Job**. Manda poi i job ad essere eseguiti.

```
1   public class Job extends JobContextImpl
2    implements JobContext, AutoCloseable
```

Dei job si possono modificare:

- Il nome
- Il jar che contiene il codice da eseguire
- Configurare input e output
- Configurare mapper(s) e reducer(s)

Per eseguire e controllare l'esecuzione: *job.waitForCompletion(true)*

```
    //Create il job
    Job job = Job.getInstance();
    job.setJarByClass(MyJob.class);

    //Settare alcuni parametri
    job.setName("JobProva");

    job.setInputPath(new Path("input"));
    job.setOutputPath(new Path("output"));

    job.setMapperClass(MyJob.MyMapper.class);
    job.setReducerClass(MyJob.MyReducer.class);

    //Mandare il job e pullare i risultati fino alla fine del job
    job.waitForCompletion(true);
```