Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica

Big Data
Tecniche e tool di analisi

Elaborato Finale

Laureando

Emanuele Gentiletti

Matricola: **090150**

Relatore

Prof. Diletta Romana Cacciagrano

Indice

Introduzione

Negli ultimi decenni, i Big Data hanno preso piede in modo impetuoso in una grande varietà di ambiti. Il fenomeno ha avuto un enorme impatto: settori come medicina, finanza, business analytics e marketing sfruttano i Big Data per guidare lo sviluppo in modi semplicemente non possibili prima.

L'innovazione che rende possibili questi risultati è guidata dal software molto più che dall'hardware. Ci sono stati dei grandi cambiamenti nel modo di pensare alla computazione e all'organizzazione dei suoi processi, che hanno portato a risultati notevoli nell'efficienza di elaborazione di grandi quantità di dati.

Uno dei più importanti fenomeni che hanno portato a questa spinta è stato lo sviluppo di Hadoop, un framework open source progettato per la computazione batch di dataset di grandi dimensioni. Utilizzando un'architettura ben congeniata, Hadoop ha permesso l'analisi in tempi molto rapidi di interi dataset di dimensioni nell'ordine dei terabyte, fornendo una capacità di sfruttamento di questi, e conseguentemente un valore molto più alti.

Una delle conseguenze più importanti di Hadoop è stata una democratizzazione delle capacità di analisi dei dati:

- Hadoop è sotto licenza Apache, permettendo a chiunque di utilizzarlo a scopi commerciali e non;
- Hadoop non richiede hardware costoso ad alta affidabilità, e incoraggia l'adozione di macchine più generiche e prone al fallimento per il suo uso, che possono essere ottenute a costi inferiori;
- Il design di Hadoop permette la sua esecuzione in cluster di macchine eterogenee nel software e nell'hardware che possono essere acquisite da diversi rivenditori, un altro fattore che permette l'abbattimento dei costi;
- I vari modelli di programmazione in Hadoop hanno in comune l'astrazione della computazione distribuita e dei problemi intricati che questa comporta, abbassando la barriere in entrata in termini di conoscenze e lavoro richiesti per creare programmi che necessitano di un altro grado di parallelismo.

Questi fattori hanno spinto a una vasta adozione di Hadoop e dell'ecosistema software che lo circonda, in ambito aziendale e scientifico. L'adozione di Hadoop, secondo un sondaggio fatto a maggio 2015[1], si aggira al 26% delle imprese negli Stati Uniti, e si prevede che il mercato attorno ad Hadoop sorpasserà i 16 miliardi di dollari nel 2020 [2].

Tutto questo accade in un'ottica in cui la produzione di informazioni aumenta ad una scala senza precedenti: secondo uno studio di IDC[3], la quantità di informazioni nell'"Universo Digitale" ammontava a 4.4 TB nel 2014, e la sua dimensione stimata nel 2020 è di 44 TB.

Data la presenza di questa vasta quantità di informazioni, lo sfruttamento efficace di queste è fonte di grandi opportunità. In questo documento si analizzano le varie tecniche che sono a disposizione per l'utilizzo effettivo dei Big Data, come queste differiscono tra di loro, e quali strumenti le mettono a disposizione. Si parlerà inoltre di come gli strumenti possano essere integrati in sistemi di produzione esistenti, le possibili architetture di un sistema di questo tipo e come …

La gestione di sistemi per l'elaborazione di Big Data richiede una configurazione accurata per ottenere affidabilità e fault-tolerance. Pur sottolineando che l'importanza di questi aspetti non è da sottovalutare, questa tesi si concentrerà più sul modello computazionale e di programmazione che gli strumenti offrono.

Parte 1

Big Data e Paradigmi di Elaborazione

Per Big Data si intendono collezioni di dati con caratteristiche tali da richiedere strumenti innovativi per poterli gestire e analizzare. Uno dei modelli tradizionali e più popolari per descrivere le caratteristiche dei Big Data si chiama **modello delle 3V**. Il modello identifica i Big Data come collezioni di informazione che presentano grande abbondanza in una più delle seguenti caratteristiche:

- Il **volume** delle informazioni, che può aggirarsi dalle decine di terabyte per arrivare all'ordine dei petabyte;
- La **varietà**, intesa come la varietà di *fonti* e di *possibili strutturazioni* delle informazioni di interesse;
- La **velocità** di produzione delle informazioni di interesse.

Ognuno dei punti di questo modello deriva da esigenze che vanno ad accentuarsi andando avanti nel tempo, in particolare:

- Il volume delle collezioni dei dati è aumentato esponenzialmente in tempi recenti, con l'avvento dei Social Media, dell'IOT, e degli smartphone muniti di molti sensori diversi. Generalizzando, i fattori che hanno portato a un grande incremento del volume dei data set sono un aumento della generazione automatica di dati da parte dei dispositivi e dei contenuti prodotti dagli utenti.

- L'aumento dei dispositivi e dei dati generati dagli utenti portano conseguentemente a un aumento delle fonti dei dati, ed essendo queste gestite da enti e persone diverse la struttura che le fonti presentano difficilmente sarà uniforme, l'una rispetto all'altra. Inoltre, l'utilizzo di dati non strutturati rigidamente è prevalente nelle tecnologie web (in particolare documenti JSON), che sono spesso un obiettivo desiderabile di analisi.

- Si possono fare le stesse considerazioni fatte per il volume dei dati per quanto riguarda la velocità. I flussi di dati vengono generati dai dispositivi e dagli utenti, che li producono a velocità molto maggiori rispetto agli operatori.

Per l'elaborazione di dataset con queste caratteristiche sono stati sviluppati molti strumenti, che usano diversi pattern di elaborazione a seconda delle esigenze dell'utente e del tipo di dati

con cui si ha a che fare. I modelli di elaborazione più importanti e rappresentativi sono il *batch processing* e lo *stream processing*.

Batch e Streaming Processing

Il batch processing è il pattern di elaborazione generalmente più efficiente, e consiste nell'elaborazione di un intero dataset in un'unità di lavoro, per poi ottenere i risultati al termine di questa.

Questo approccio è ottimale quando non c'è una necessità impendente di avere risultati, ma in alcuni casi è necessario avere i risultati a disposizione mano a mano che la computazione procede, e il batch processing non è adatto a questo scopo:

- Le fasi del batch processing richiedono la schedulazione dei lavori da parte dell'utente, con un conseguente overhead dovuto alla schedulazione in sé o alla configurazione di strumenti automatizzati che se ne occupino;
- Non è possibile accedere ai risultati prima del termine del job, che può avere una durata eccessiva rispetto alle esigenze dell'applicazione o dell'utente.

Per use case in cui questi fattori sono rilevanti, lo **stream processing** si presta come più adatto. In questo paradigma, i dati da elaborare vengono ricevuti da stream (nella maggior parte dei casi da Internet) e vengono processati mano a mano con il loro arrivo. I job in streaming molto spesso non hanno un termine prestabilito, ma vengono terminati dall'utente, e i risultati dell'elaborazione possono essere disponibili mano a mano che l'elaborazione procede, permettendo quindi un feedback più rapido rispetto ai lavori batch.

Data at Rest e *Data in Motion*

I due paradigmi si differenziano anche per il modo in cui i dati sono disponibili. Il processing batch richiede che l'informazione sia *data at rest*, ovvero informazioni salvate interamente in un mezzo di memoria accessibile al programma. I dati di input in una computazione batch sono determinati all'inizio dell'elaborazione, e non possono cambiare nel corso di questa. Questo significa che se nuova informazione arriva nel corso di un job batch, questa non può essere tenuta in conto nell'elaborazione finale.

Lo **stream processing**, invece, è progettato per *data in motion*, dati in arrivo continuo la cui quantità non è fissa a priori. È possibile utilizzare strumenti di processing in streaming anche per *data at rest*, rappresentando il dataset come uno stream. Questa proprietà è desiderabile, perché permette di utilizzare le stesse applicazioni per l'elaborazione di dati in arrivo dalla rete e quelli salvati. Come si vedrà, la Kappa architecture, ovvero una possibile architettura software per l'elaborazione di Big Data, utilizza questa proprietà per sfruttare un solo paradigma sia per computazioni in real-time di dati in arrivo da stream, che per i dati salvati storicamente,

permettendo di utilizzare gli stessi tool e interfacce di programmazione per entrambi i tipi di elaborazione e massimizzare il riutilizzo di codice.

Tabella 1.1: Differenze tra elaborazione batch e streaming

Caratteristiche	Batch	Streaming
Ottimizzazione	Alto throughput	Bassa latenza
Tipo di informazione	*Data at rest*	*Data in motion* e *Data at rest*
Accesso ai dati	Stabilito all'inizio	Dipendente dallo stream
Accesso ai risultati	Fine job	Continuo

Un esempio di *data at rest* sono i resoconti delle vendite di un'azienda, su cui si possono cercare pattern per identificare quali prodotti sono in trend nelle vendite. Per *data in motion* si può considerare l'invio di dati da parte di sensori IoT o le pubblicazioni degli utenti nei social media, che sono continui e senza una fine determinata.

Hadoop e modelli di elaborazione

Nella prossima sezione si discuterà di Hadoop, un progetto nato con l'intento di affrontare la computazione batch

Parte 2

Hadoop

Nell'ambito dei Big Data, Hadoop è il perno centrale su cui è basato un *ecosistema* di tool e tecnologie, tant'è che spesso il termine Hadoop viene utilizzato per riferirsi all'intero ecosistema di tool e tecnologie construiti attorno a questo.

La documentazione ufficiale[4] lo descrive come:

> …un framework che abilita l'elaborazione distribuita di grandi dataset in cluster di computer utilizzando semplici modelli di programmazione. Hadoop è progettato per essere scalato da server singoli a migliaia di macchine, dove ognuna di queste offre computazione e storage locale. Invece di affidarsi all'hardware per fornire un'alta affidabilità, Hadoop è progettato per rilevare e gestire i fallimenti [delle computazioni] a livello applicativo, mettendo a disposizione un servizio ad alta affidiabilità su cluster di computer proni al fallimento.

In questa definizione sono racchiusi dei punti molti importanti:

- **Semplici modelli di programmazione**

  Hadoop raggiunge molti dei suoi obiettivi fornendo un'interfaccia di livello molto alto al programmatore, in modo di potersi assumere la responsabilità di concetti complessi e necessari all'efficienza nella computazione distribuita, ma che hanno poco a che fare con il problema da risolvere in sé (ad esempio, la sincronizzazione di task paralleli e lo scambio dei dati tra nodi del sistema distribuito). Questo modello **pone dei limiti alla libertà del programmatore**, che deve adeguare la codifica della risoluzione del problema al modello di programmazione fornito.

- **Computazione e storage locale**

  L'ottimizzazione più importante che Hadoop fornisce rispetto all'elaborazione dei dati è il risultato dell'unione di due concetti: **distribuzione dello storage** e **distribuzione della computazione**.

  Entrambi sono importanti a prescindere dell'uso particolare che ne fa Hadoop: la distribuzione dello storage permette di combinare lo spazio fornito da più dispositivi e di

farne uso tramite un'unica interfaccia logica, e di replicare i dati in modo da poter tollerare guasti nei dispositivi. La distribuzione della computazione permette di aumentare il grado di parallelizzazione nell'esecuzione dei programmi.

Hadoop unisce i due concetti utilizzando cluster di macchine che hanno sia lo scopo di mantenere lo storage, che quello di elaborare i dati. Quando Hadoop esegue un lavoro, **quante più possibili delle computazioni richieste vengono eseguite nei nodi che contengono i dati da elaborare**. Questo permette di ridurre la latenza di rete, minimizzando la quantità di dati che devono essere scambiati tra i nodi del cluster. Il meccanismo è trasparente all'utente, a cui basta persitere i dati da elaborare nel cluster per usifruirne. Questo principio viene definito **data locality**.

- **Scalabilità**

  |||

- **Hardware non necessariamente affidabile**

  I cluster di macchine che eseguono Hadoop non hanno particolari requisiti di affidabilità rispetto ad hardware consumer. Il framework è progettato per tenere in conto dell'alta probabilità di fallimento dell'hardware, e per attenuarne le conseguenze, sia dal punto di vista dello storage e della potenziale perdita di dati, che da quello della perdita di risultati intermedi e parziali nel corso dell'esecuzione di lavori computazionalmente costosi. In questo modo l'utente è sgravato dal compito generalmente difficile di gestire fallimenti parziali nel corso delle computazioni.

Hadoop è composto da diversi moduli:

- **HDFS**, un filesystem distribuito ad alta affidabilità, che fornisce replicazione automatica all'interno dei cluster e accesso ad alto throughput ai dati
- **YARN**, un framework per la schedulazione di lavori e per la gestione delle risorse all'interno del cluster
- **MapReduce**, un framework e un modello di programmazione fornito da Hadoop per la scrittura di programmi paralleli che processano grandi dataset.

Installazione e Configurazione

Ogni versione di Hadoop viene distribuita in tarball, una con i sorgenti, da cui si può eseguire una build manuale, e una binaria, che può essere estratta e utilizzata così com'è. Per un approccio più strutturato, sono disponibili repository che forniscono versioni pacchettizzate di Hadoop, come il PPA per Ubuntu[5] e i pacchetti AUR per Arch Linux[6].

Ci sono anche distribuzioni di immagini virtuali Linux create appositamente con lo scopo di fornire un ambiente preconfigurato con Hadoop e vari componenti del suo ecosistema. I due

ambienti più utilizzati di questo tipo sono Cloudera QuickStart e HortonWorks Sandbox, disponibili per VirtualBox, VMWare e Docker. Gli esempi di questo documento sono eseguiti prevalentemente da Arch Linux e dalla versione Docker di HortonWorks Sandbox.

Hadoop è configurabile tramite file XML, che si trovano rispetto alla cartella d'installazione in `etc/hadoop`. Ogni componente di Hadoop (HDFS, MapReduce, Yarn) ha un file di configurazione apposito che contiene impostazioni relative al componente stesso, mentre un altro file di configurazione contiene proprietà comuni a tutti i componenti.

Tabella 2.1: Nomi dei file di configurazione per i componenti di Hadoop

Comuni	HDFS	YARN	MapReduce
`core-site.xml`	`hdfs-site.xml`	`yarn-site.xml`	`mapred-site.xml`

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://namenode/</value>
    </property>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>resourcemanager:8032</value>
    </property>
</configuration>
```

Listato 2.1: Esempio di file di configurazione personalizzato di Hadoop.

È anche possibile selezionare un'altra cartella da cui prendere i file di configurazione, impostandola come valore della variabile d'ambiente `HADOOP_CONF_DIR`. Un approccio comune alla modifica dei file di configurazione consiste nel copiare il contenuto di `etc/hadoop` in un'altra posizione, specificare questa in `HADOOP_CONF_DIR` e fare le modifiche nella nuova cartella. In questo modo si evita di modificare l'albero d'installazione di Hadoop.

Per molti degli eseguibili inclusi in Hadoop, è anche possibile specificare un file che contiene ulteriori opzioni di configurazione, che possono sovrascrivere quelle in `HADOOP_CONF_DIR` tramite lo switch `-conf`.

Esecuzione di software in Hadoop

I programmi che sfruttano il runtime di Hadoop sono generalmente sviluppati in Java (o in un linguaggio che ha come target di compilazione la JVM), e vengono avviati tramite l'eseguibile `hadoop`. L'eseguibile richiede che siano specificati il classpath del programma, e una classe contente un metodo `main` che si desidera eseguire (analogo all'entry point dei programmi Java).

Il classpath può essere specificato tramite la variabile d'ambiente `HADOOP_CLASSPATH`, che può essere il percorso di una directory o di un file `jar`. La classe con il metodo `main` da invocare viene messa tra i parametri del comando `hadoop`, seguita dagli argomenti che si vogliono passare in `args[]`.

(1) Volendo eseguire il seguente programma in Hadoop:

```
public class SayHello {
    public static void main(String args[]) {
        System.out.println("Hello " + args[0] + "!");
    }
}
```

Lo si può compilare e pacchettizzare in un file `jar`, per poi utilizzare i seguenti comandi:

```
$~ export HADOOP_CLASSPATH=say_hello.jar
$~ hadoop SayHello Josh

Hello Josh!
```

(2) In alternativa, si può eseguire il comando `hadoop jar`, e specificare il file `jar` direttamente nei suoi argomenti:

```
$~ hadoop jar say_hello.js SayHello Josh

Hello Josh!
```

In generale, i programmi eseguiti in Hadoop fanno uso della sua libreria client. La libreria fornisce accesso al package `org.apache.hadoop`, che contiene le API necessarie per interagire con Hadoop. Non è necessario che la libreria client si trovi nel classpath finale, in quanto il runtime di Hadoop fornisce le classi della libreria a runtime.

Per gestire le dipendenze e la pacchettizzazione dei programmi per Hadoop è pratico utilizzare un tool di gestione delle build. Negli esempi in questo documento si utilizza Maven a questo scopo, che permette di specificare le proprietà di un progetto, tra cui le sue dipendenze, in un file XML chiamato POM (Project Object Model). A partire dal POM, Maven è in grado di scaricare automaticamente le dipendenze del progetto, e di pacchettizzarle correttamente negli artefatti `jar` a seconda della configurazione fornita.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="..." >
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>say_hello</artifactId>
    <version>1.0</version>

    <dependencies>

        <!-- Libreria client di Hadoop -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>2.8.1</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>
</project>
```

Listato 2.2: Un esempio semplificato di un POM per il programma SayHello.

Maven è in grado di gestire correttamente la dipendenza della libreria client di Hadoop, attraverso un meccanismo chiamato *dependency scope*. Per ogni dipendenza è possibile specificare una proprietà *scope*, che indica in che modo la dipendenza debba essere gestita a tempo di build (in particolar modo, se debba essere inclusa nel classpath). Se non specificato, lo scope è impostato a `compile`, che indica che la dipendenza è resa disponibile nel classpath dell'artefatto. Per gestire correttamente la dipendenza dalla libreria client di Hadoop, è opportuno impostare lo scope della dipendenza a `provided`, che indica che le classi della libreria sono fornite dal container in cui è eseguito il programma.

HDFS

HDFS è un filesystem distribuito che permette l'accesso ad alto throughput ai dati. HDFS è scritto in Java, e viene eseguito nello userspace. Lo storage dei dati passa per il filesystem del sistema che lo esegue.

I dati contenuti in HDFS sono organizzati in unità logiche chiamate *blocchi*, come è comune nei filesystem. I blocchi di un singolo file possono essere distribuiti all'interno di più macchine all'interno del cluster, permettendo di avere file più grandi della capacità di storage di ogni singola macchina nel cluster. Rispetto ai filesystem comuni la dimensione di un blocco è molto più grande, 128 MB di default. La ragione per cui HDFS utilizza blocchi così grandi è minimiz-

zare il costo delle operazioni di seek, dato il fatto che se i file sono composti da meno blocchi, si rende necessario trovare l'inizio di un blocco un minor numero di volte. Questo approccio riduce anche la frammentazione dei dati, rendendo più probabile che questi vengano scritti contiguamente all'interno della macchina[1].

Il blocco, inoltre, è un'astrazione che si presta bene alla replicazione dei dati nel filesystem all'interno del cluster: per replicare i dati, come si vedrà, si mette uno stesso blocco all'interno di più macchine nel cluster.

HDFS è basato sulla specifica POSIX, ma non la implementa in modo rigido: tralasciare alcuni requisiti di conformità alla specifica permette ad HDFS di ottenere prestazioni e affidabilità migliori, come verrà descritto in seguito.

Principi architetturali

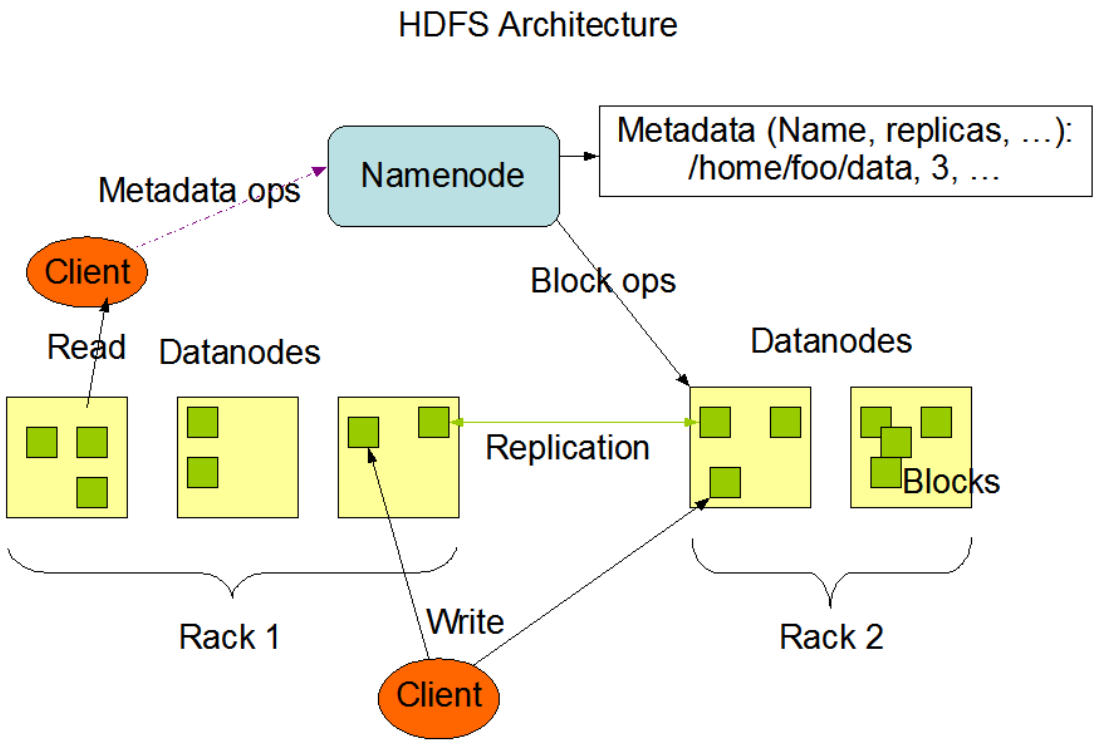

Figura 2.1: Schema di funzionamento dell'architettura di HDFS

La documentazione di Hadoop descrive i seguenti come i principi architetturali alla base della progettazione di HDFS:

- **Fallimento hardware come regola invece che come eccezione**

[1]Non è possibile essere certi della contiguità dei dati, perché HDFS non è un'astrazione diretta sulla scrittura del disco, ma sul filesystem del sistema operativo che lo esegue. Per cui la frammentazione effettiva dipende da come i dati vengono organizzati dal filesystem sottostante.

Un sistema che esegue HDFS è composto da molti componenti, con probabilità di fallimento non triviale. Sulla base di questo principio, HDFS da' per scontato che **ci sia sempre un numero di componenti non funzionanti**, e si pone di rilevare errori e guasti e di fornire un recupero rapido e automatico da questi.

Il meccanismo principale con cui HDFS raggiunge questo obiettivo è la replicazione: in un cluster, ogni blocco di cui un file è composto è replicato in più macchine (3 di default). Se un blocco non è disponibile in una macchina, o se non supera i controlli di integrità, una sua copia può essere letta da un'altra macchina in modo trasparente per il client.

Il numero di repliche per ogni blocco è configurabile, e ci sono più criteri con cui viene deciso in quali macchine il blocco viene replicato, principalmente orientati al risparmio di banda di rete.

- **Modello di coerenza semplice**

  Per semplificare l'architettura generale, HDFS fa delle assunzioni specifiche sul tipo di dati che vengono salvati in HDFS e pone dei limiti su come l'utente possa lavorare sui file. In particolare, **non è possibile modificare arbitrariamente file già esistenti**, e le modifiche devono limitarsi a operazioni di troncamento e di aggiunta a fine file. Queste supposizioni permettono di semplificare il modello di coerenza, perché i blocchi di dati, una volta scritti, possono essere considerati immutabili, evitando una considerevole quantità di problemi in un ambiente dove i blocchi di dati sono replicati in più posti:

  – Per ogni modifica a un blocco di dati, bisognerebbe verificare quali altre macchine contengono il blocco, e rieseguire la modifica (o rireplicare il blocco modificato) in ognuna di queste.

  – Queste modifiche dovrebbero essere fatte in modo atomico, o richieste di lettura su una determinata replica di un blocco invece che in un'altra potrebbe portare a risultati inconsistenti o non aggiornati.

  Le limitazioni che Hadoop impone sono ragionevoli per lo use-case per cui HDFS è progettato, caratterizzato da grandi dataset che vengono copiati nel filesystem e letti in blocco.

- **Dataset di grandi dimensioni**

  I filesystem distribuiti sono generalmente necessari per aumentare la capacità di storage disponibile oltre quella di una singola macchina. La distribuzione di HDFS, assieme alla grande dimensione dei blocchi

- **Accesso in streaming**

  HDFS predilige l'accesso ai dati in streaming, per permettere ai lavori batch di essere eseguiti con grande efficienza. Questo approccio va a discapito del tempo di latenza della lettura dei file, ma permette di avere un throughput in lettura molto vicino ai tempi di lettura del disco.

- **Portabilità su piattaforme software e hardware eterogenee**

  HDFS è scritto in Java, ed è portabile in tutti i sistemi che ne supportano il runtime.

L'architettura di HDFS è di tipo master/slave, dove un nodo centrale, chiamato **NameNode**, gestisce i metadati e la struttura del filesystem, mentre i nodi slave, chiamati **DataNode**, contengono i blocchi di cui file sono composti. Tipicamente, viene eseguita un'istanza del software del DataNode per macchina del cluster, e una macchina dedicata esegue il NameNode.

I *client* del filesystem interagiscono sia con il NameNode che con i DataNode per l'accesso ai file. La comunicazione tra il client e i nodi avviene tramite socket TCP ed è coordinata dal NameNode, che fornisce ai client tutte le informazioni sul filesystem e su quali nodi contengono i DataBlock dei file richiesti.

Comunicare con HDFS

Hadoop fornisce tool e librerie che possono agire da client nei confronti di HDFS. Il tool più diretto è la CLI, accessibile nelle macchine in cui è installato Hadoop tramite il comando `hadoop fs`.

```
% hadoop fs -help
Usage: hadoop fs [generic options]
    [-appendToFile <localsrc> ... <dst>]
    [-cat [-ignoreCrc] <src> ...]
    [-checksum <src> ...]
    [-chgrp [-R] GROUP PATH...]
    [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
    [-chown [-R] [OWNER][:[GROUP]] PATH...]
    [-copyFromLocal [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
    [-copyToLocal [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
    [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] <path> ...]
    [-cp [-f] [-p | -p[topax]] [-d] <src> ... <dst>]
...
```

La CLI fornisce alcuni comandi comuni nei sistemi POSIX, come `cp`, `rm`, `mv`, `ls` e `chown`, e altri che riguardano specificamente HDFS, come `copyFromLocal` e `copyToLocal`, utili a trasferire dati tra la macchina su cui si opera e il filesystem.

I comandi richiedono l'URI che identifica l'entità su cui si vuole operare. Per riferirsi a una risorsa all'interno di un'istanza di HDFS, si usa l'URI del namenode, con schema `hdfs`[2], e con

[2] Hadoop è abbastanza generale da poter lavorare con diversi filesystem, con lo schema definisce il protocollo di comunicazione, che non deve essere necessariamente `hdfs`. Ad esempio, un URI con schema `file` si riferisce al filesystem locale, e le operazioni eseguite su URI che utilizzano questo schema vengono effettuate sulla macchina dove viene eseguito il comando. Questo approccio può essere adatto nella fase di testing dei programmi, ma nella maggior parte dei casi è comunque desiderabile lavorare su un filesystem distribuito adeguato alla gestione dei Big Data, e un'alternativa ad HDFS degna di nota è MapR-FS[7].

il path corrispondente al percorso della risorsa nel filesystem. Ad esempio, è possibile creare una cartella `foo` all'interno della radice del filesystem con il seguente comando:

```
hadoop fs -mkdir hdfs://localhost:8020/foo
```

Per diminuire la verbosità dei comandi è possibile utilizzare percorsi relativi, e specificare l'opzione `dfs.defaultFS` nella configurazione di Hadoop all'URI del filesystem ai cui i percorsi relativi si riferiscono. In questo modo, si può accorciare l'esempio precedente a:

```
hadoop fs -mkdir foo
```

Ad esempio, data la seguente cartella:

```
[root@sandbox example_data]# ls
example1.txt  example2.txt  example3.txt
```

Si possono copiare i file dalla cartella locale della macchina al filesystem distribuito con il seguente comando:

```
[root@sandbox example_data]# hadoop fs -copyFromLocal example*.txt /example
```

Per verificare che l'operazione sia andata a buon fine, si può ottenere un listing della cartella in cui si sono trasferiti i file con il comando `ls`:

```
[root@sandbox example_data]# hadoop fs -ls /example
Found 3 items
-rw-r--r--   1 root hdfs         70 2017-06-30 03:58 /example/example1.txt
-rw-r--r--   1 root hdfs         39 2017-06-30 03:58 /example/example2.txt
-rw-r--r--   1 root hdfs         43 2017-06-30 03:58 /example/example3.txt
```

Il listing è molto simile a quello ottenibile su sistemi Unix. Una differenza importante è la seconda colonna, che non mostra il numero di hard link al file nel filesystem[3], ma il numero di repliche che HDFS ha a disposizione del file, in questo caso una per file. Il numero di repliche fatte da HDFS può essere impostato settando il fattore di replicazione di default, che per Hadoop in modalità distribuita è 3 di default. Si può anche cambiare il numero di repliche disponibili per determinati file, utilizzando il comando `hdfs dfs`:

```
[root@sandbox ~]# hdfs dfs -setrep 2 /example/example1.txt
Replication 2 set: /example/example1.txt
[root@sandbox ~]# hadoop fs -ls /example
Found 3 items
-rw-r--r--   2 root hdfs         70 2017-06-30 03:58 /example/example1.txt
-rw-r--r--   1 root hdfs         39 2017-06-30 03:58 /example/example2.txt
-rw-r--r--   1 root hdfs         43 2017-06-30 03:58 /example/example3.txt
```

HDFS è anche accessibile tramite *HDFS Web Interface*, un tool che fornisce informazioni sullo stato generale del filesystem e sul suo contenuto. Ci sono anche tool di amministrazione di cluster Hadoop che offrono GUI web più avanzate di quella fornita di default da HDFS. Due

[3] HDFS correntemente non supporta link nel filesystem.

esempi sono Cloudera Manager e Apache Ambari, che offrono un file manager lato web con cui è possibile interagire in modo più semplice, permettendo anche a utenti in ambito meno tecnico di lavorare con il filesystem.

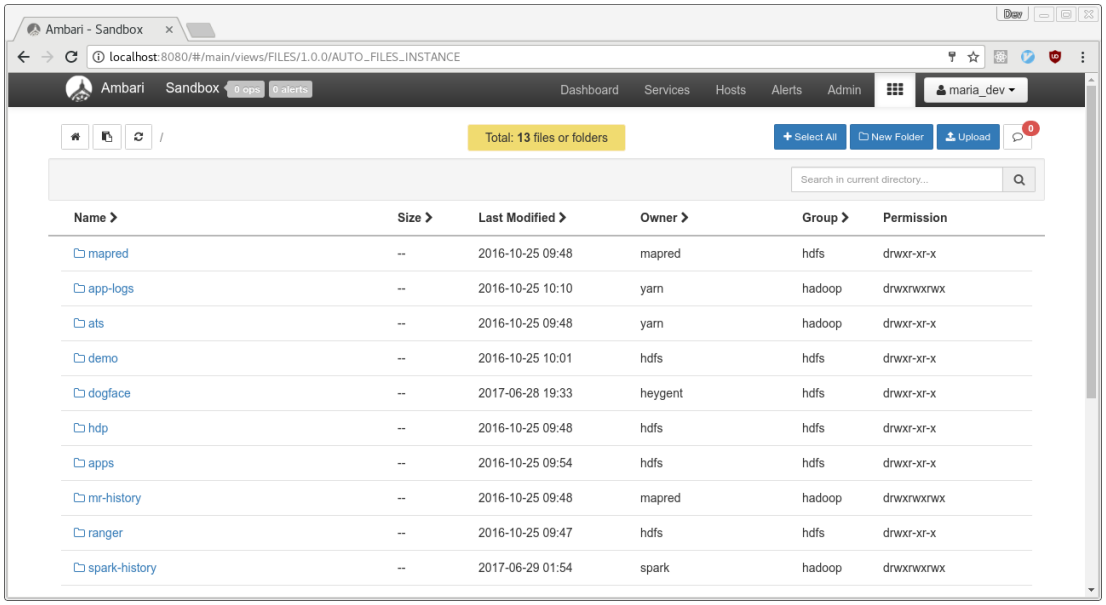

Figura 2.2: Screenshot del file manager HDFS incluso in Ambari

Un'altra interfaccia importante ad HDFS è l'API `FileSystem` di Hadoop, che permette un accesso programmatico da linguaggi per JVM a tutte le funzioni del filesystem. L'API è generale, in modo che possa essere utilizzata con filesystem diversi da HDFS.

Per linguaggi che non supportano interfacce Java, esiste un'implementazione in C chiamata `libhdfs`, che si appoggia sulla Java Native Interface per esporre l'API di Hadoop.

Esistono poi progetti che permettono il montaggio di HDFS in un filesystem locale. Alcune di queste implementazioni sono basate su FUSE, mentre altre su NFS Gateway. Questo metodo di accesso permette l'utilizzo di utilità native del sistema in uso in HDFS.

NameNode

Il NameNode è il riferimento centrale per i metadati del filesystem nel cluster, il che vuol dire che se il NameNode non è disponibile il filesystem non è accessibile. Questo rende il NameNode un *single point of failure* del sistema, e per questa ragione HDFS mette a disposizione dei meccanismi per attenutare l'indisponibilità del sistema in caso di non reperibilità del NameNode, e per assicurare che lo stato del filesystem possa essere recuperato a partire dal NameNode.

Il NameNode è anche il nodo a cui i client si connettono alla lettura del file. La connessione ha il solo scopo di fornire le informazioni sui DataNode che contengono i dati effettivi del file. I dati di un file non passano mai per il NameNode.

Tuttavia, il NameNode non salva persistentemente le informazioni sulle posizioni dei blocchi, che vengono invece mantenute dai DataNode. Perché il NameNode possa avere in memoria

le informazioni sui file necessarie per essere operativo, questo deve ricevere le liste dei blocchi in possesso dei DataNode, in messaggi chiamati **block report**. Non è necessario che il DataNode conosca la posizione di tutti i blocchi sin dall'inizio, ma basta che per ogni blocco conosca la posizione di un numero minimo di repliche, determinato da un'opzione chiamata `dfs.replication.min.replicas`, di default 1.

Questa procedura avviene quando il NameNode si trova in uno stato chiamato **safe mode**.

Namespace image ed *edit log*

Le informazioni sui metadati del sistema vengono salvate nello storage del NameNode in due posti, la ***namespace image*** e l'***edit log***. La *namespace image* è uno snapshot dell'intera struttura del filesystem, mentre l'*edit log* è un elenco di operazioni eseguite nel filesystem a partire dalla *namespace image*. Partendo dalla *namespace image* e applicando le operazioni registrate nell'*edit log*, è possibile risalire allo stato attuale del filesystem. Il NameNode ha una rappresentazione dello stato del filesystem anche nella memoria centrale, che viene utilizzata per servire le richieste di lettura.

Quando HDFS riceve una richiesta che richiede la modifica dei metadati, il NameNode esegue le seguenti operazioni:

1. registra la transazione nell'*edit log*
2. aggiorna la rappresentazione del filesystem in memoria
3. passa all'operazione successiva.

La ragione per cui i cambiamenti dei metadati vengono registrati nell'*edit log* invece che nella *namespace image* è la velocità di scrittura: scrivere ogni cambiamento del filesystem mano a mano che avviene nell'immagine sarebbe lento, dato che questa può avere dimensioni nell'ordine dei gigabyte. Il NameNode esegue un *merge* dell'*edit log* e della *namespace image* a ogni suo avvio, portando lo stato attuale dell'immagine al pari di quello del filesystem.

Dato che la dimensione dell'*edit log* può diventare notevole, è utile eseguire l'operazione di *merge* al raggiungimento di una soglia di dimensione del log. Questa operazione è computazionalmente costosa, e se fosse eseguita dal NameNode potrebbe interferire con la sua operazione di routine.

Per evitare interruzioni nel NameNode, il compito di eseguire periodicamente il *merge* dell'*edit log* è affidato a un'altra entità, il **Secondary NameNode**. Il Secondary NameNode viene solitamente eseguito su una macchina differente, dato che richiede un'unità di elaborazione potente e almeno la stessa memoria del NameNode per eseguire l'operazione di merge.

Avvio del NameNode e *Safe Mode*

Prima di essere operativo, il NameNode deve eseguire alcune operazioni di startup, tra cui attendere di aver ricevuto i block report dai DataNode in modo da conoscere le posizioni dei

blocchi. Durante queste operazioni, il NameNode si trova in uno stato chiamato *safe mode*, in cui sono permesse unicamente operazioni che accedono ai metadati del filesystem, e tentativi di lettura e scrittura di file falliscono. Prima di poter permettere l'accesso completo, il NameNode ha bisogno di ricevere le informazioni sui blocchi da parte dei DataNode.

Per ricapitolare, al suo avvio, il NameNode effettua il merge della *namespace image* con l'*edit log*. Al termine dell'operazione, il risultato del merge viene salvato come la nuova *namespace image*. Il Secondary NameNode non viene coinvolto in questo primo merge.

Prima di uscire dalla safe mode, il NameNode attende di avere abbastanza informazioni da poter accedere a un numero minimo di repliche di ogni blocco. A questo punto il NameNode esce dalla safe mode.

Si possono utilizzare dei comandi per verificare lo stato, attivare e disattivare la safe mode.

```
bash-4.1$ hdfs dfsadmin -safemode get
Safe mode is OFF
bash-4.1$ hdfs dfsadmin -safemode enter
Safe mode is ON
bash-4.1$ hdfs dfsadmin -safemode leave
Safe mode is OFF
```

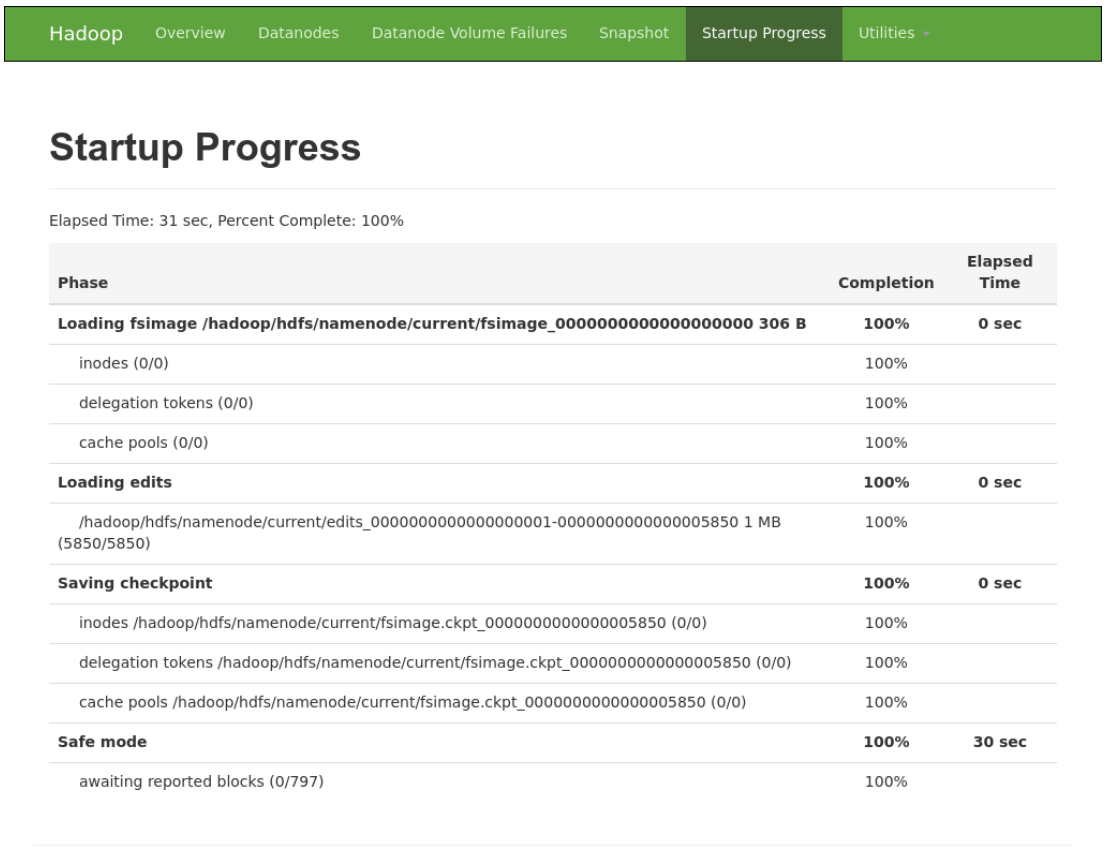

Figura 2.3: Lo stato dello startup di un'istanza di HDFS, mostrata da HDFS Web Interface.

Processo di lettura di file in HDFS

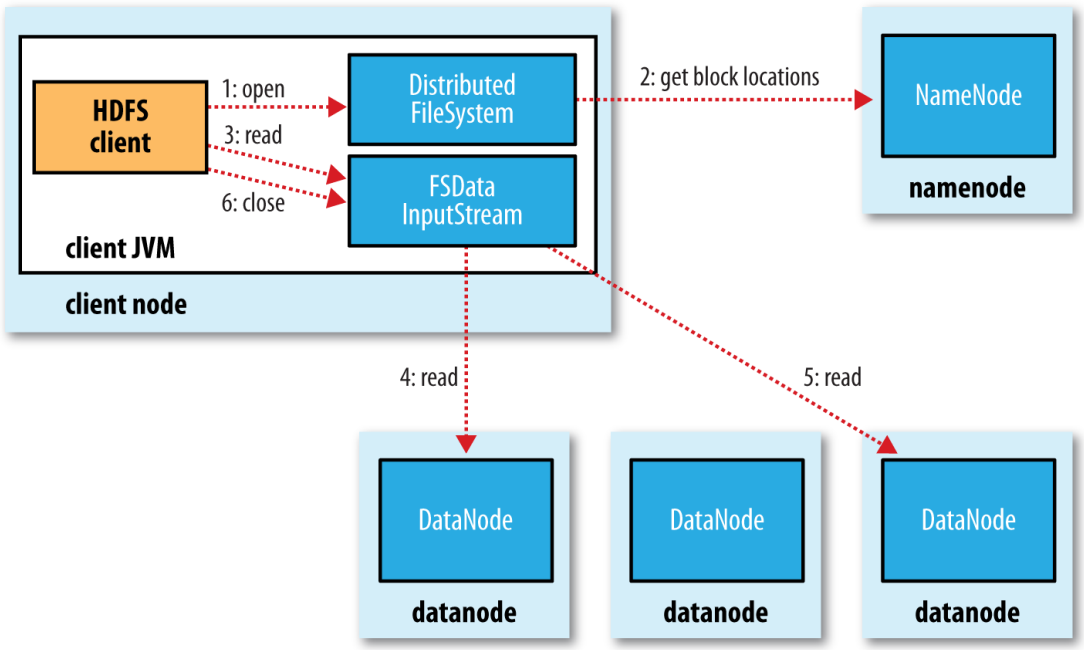

Figura 2.4: Diagramma delle operazioni eseguite nella lettura di un file in HDFS[8]

Per avere un quadro completo del funzionamento di HDFS, è utile osservare come avvenga il processo di lettura di un file. In questa sezione si prende in esame un programma di esempio che utilizza le API `FileSystem` di Hadoop per reimplementare una versione semplificata del comando `cat`, per poi esaminare come le operazioni specificate nel programma vengano effettivamente portate a termine in un'istanza di HDFS.

La reimplementazione del programma `cat` utilizza il primo parametro della linea di comando per ricevere l'URI del file che si vuole stampare nello standard output. L'URI deve contenere il percorso di rete del filesystem HDFS, ed essere quindi del formato `hdfs://[indirizzo o hostname del namenode]/[path del file]`. Di seguito vengono spiegati i passi eseguiti dal programma. Quando non qualificato, l'identificativo `hadoop` si riferisce al package Java `org.apache.hadoop`.

1. Si crea un oggetto `hadoop.conf.Configuration`. Gli oggetti `Configuration` forniscono l'accesso ai parametri di configurazione di Hadoop (impostati in file XML, come descritto in Installazione e Configurazione).

2. Si ottiene un riferimento `sourcefs` a un `hadoop.fs.FileSystem` (dichiarato come interfaccia Java), che fornisce le API che verranno usate per leggere e manipolare il filesystem. Il riferimento viene ottenuto tramite il metodo statico `FileSystem.get(URI source, Configuration conf)`, che richiede un URI che possa essere utilizzato per risalire a quale filesystem si vuole accedere. Un overload di `FileSystem.get` permette di specificare solo l'oggetto `Configuration`, e ottiene le informazioni sul filesystem da aprire dalla proprietà di configurazione `dfs.defaultFS`.

```java
import java.io.InputStream;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class MyCat {

    public static void main(String args[]) throws Exception {

        String source = args[0];
        Configuration conf = new Configuration();

        try(
            FileSystem sourcefs = FileSystem.get(URI.create(source), conf);
            InputStream in = sourcefs.open(new Path(source))
        ) {
            IOUtils.copyBytes(in, System.out, 4096, false);
        }
    }
}
```

Listato 2.3: Programma di esempio che reimplementa il comando `cat`.

3. Si apre il file il lettura, chiamando `sourcefs.open(Path file)`. Il metodo restituisce un oggetto di tipo `hadoop.fs.FSDataInputStream`, una sottoclasse di `java.io.InputStream` che supporta anche l'accesso a punti arbitrari del file. In questo case l'oggetto è utilizzato per leggere il file sequenzialmente, e il suo riferimento viene salvato nella variabile `InputStream in`.

4. Si copiano i dati dallo stream `in` a `System.out`, di fatto stampando i dati nella console. Questa operazione è eseguita tramite il metodo `hadoop.io.IOUtils.copyBytes(InputStream in, OutputStream out, int bufSize, bool closeStream)`. Il metodo copia i dati da uno stream d'ingresso a uno d'uscita, e non ha funzioni specifiche rispetto ad Hadoop, ma viene fornito per la mancanza di un meccanismo simile in Java.

5. Lo stream e l'oggetto `FileSystem` vengono chiusi. L'operazione avviene implicitamente utilizzando il costrutto try-with-resources di Java.

L'esecuzione del programma dà il seguente output:

```
$~ hadoop MyCat hdfs://sandbox.hortonworks.com:8020/example/example1.txt
This is the first example file
```

Nel caso di un URI con schema HDFS, l'istanza concreta di `FileSystem` che viene restituita da `FileSystem.get` è di tipo `DistributedFileSystem`, che contiene le funzionalità necessarie a comunicare con HDFS. Con uno schema diverso (ad esempio `file`), l'istanza concreta di `FileSystem` cambia.

Dietro le quinte, `FSDataInputStream`, restituito da `FileSystem.open(...)`, utilizza chiamate a procedure remote sul namenode per ottenere le posizioni dei primi blocchi del file. Per ogni blocco, il namenode restituisce gli indirizzi dei datanode che lo contengono, ordinati in base alla prossimità del client. Se il client stesso è uno dei datanode che contiene un blocco da leggere, il blocco viene letto localmente.

Alla prima chiamata di `read()` su `FSDataInputStream`, l'oggetto si connette al DataNode che contiene il primo blocco del file, e lo richiede (nell'esempio, `read` viene chiamato da `IOUtils.copyBytes`). Il DataNode risponde inviando i dati corrispondenti al blocco, fino al termine di questi. Al raggiungimento della fine di un blocco, `DFSInputStream` termina la connessione con il DataNode corrente e ne inizia un'altra con il più prossimo dei DataNode che contiene il blocco successivo.

In caso di errore dovuto al fallimento di un DataNode o alla ricezione di un blocco di dati corrotto, il client può ricevere il blocco dal nodo successivo della lista dei DataNode contenenti il blocco ricevuta dal NameNode.

I blocchi del file non vengono inviati tutti insieme, e il client deve periodicamente richiedere al NameNode i dati sui blocchi successivi. Questo avviene trasparentemente rispetto al client, che si limita a chiamare `read` su `DFSInputStream`.

Le comunicazioni di rete, in questo meccanismo, sono distribuite su tutto il cluster. Il NameNode riceve richieste che riguardano solo i metadati dei file, mentre il resto delle connessioni viene eseguito direttamente tra client e DataNode. Questo approccio permette ad HDFS di raggiungere un gran livello di scalabilità, evitando i colli di bottiglia dovuti a un punto di connessione centralizzato nel filesystem.

Parte 3

Batch Processing

Il Batch Processing è la *raison d'être* di Hadoop. Il primo paradigma di programmazione per Hadoop, MapReduce, è stato l'unico per molte release, e ha avuto il grande merito di astrarre la complessità della computazione batch in ambiente distribuito in funzioni che associano chiavi e valori a risultati, una grande semplificazione rispetto ai programmi che gestiscono granularmente l'intricatezza di ambienti distribuiti.

Pur essendo popolare, MapReduce è soggetto a molte limitazioni, che riguardano soprattutto la necessità di esprimere i programmi da eseguire con un modello che non lascia molto spazio alla rielaborazione dei risultati. Come si vedrà, queste limitazioni sono intrinseche al fatto che i risultati intermedi vengano salvati nello storage locale del nodo del cluster, e quelli finali in HDFS. Questi due fattori influenzano pesantemente le prestazioni che si possono ottenere da un algoritmo, perché vi introducono l'overhead della lettura e scrittura nel disco, o peggio in HDFS.

YARN è stato creato proprio per questo motivo: permettere che altri modelli di computazione diversi da MapReduce potessero essere eseguiti sfruttando HDFS. Le nuove versioni di MapReduce sono implementate al di sopra di YARN invece che direttamente in Hadoop, a testimoniare l'effettiva capacità di YARN di generalizzare i modelli di esecuzione nei cluster.

La sua alternativa più popolare, Apache Spark, ha API più espressive e funzionali rispetto a MapReduce, e prestazioni molto più elevate in molti algoritmi[9]. Tramite astrazioni che offrono un controllo più preciso sul comportamento dei risultati dell'elaborazione, Spark trova moltissime applicazioni pratiche sia negli ambiti , tra cui il machine learning

In questa sezione si esaminano MapReduce e Spark, quali sono le limitazioni di MapReduce che hanno richiesto la necessità di un nuovo modello computazionale, e quale soluzioni sono offerte da Spark.

MapReduce

Il modello computazionale di MapReduce è composto, nella sostanza, da due componenti, che sono intuitivamente il Mapper e il Reducer.

Il Mapper è una classe contenente una funzione `map`, che riceve in input una coppia composta da chiave e valore, e che restituisce a sua volta zero, uno, o più coppie di chiavi e valori[1]. Le chiavi e i valori ricevuti in input dal Mapper sono derivati direttamente dall'elemento letto in HDFS. Nel caso dei file di testo, ad esempio, la chiave è un intero che rappresenta la riga del file letto, e il valore è la riga di testo. È possibile configurare quali chiavi e valori vengano derivati dalla sorgente e come, creando una classe che implementa l'interfaccia `InputMapper` fornita nella libreria di Hadoop.

Le applicazioni MapReduce specificano un proprio Mapper estendendo la classe `Mapper` nella libreria di Hadoop, e specificando i tipi dei parametri generici opportunamente. La firma di `Mapper` è la seguente:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> extends Object
```

I tipi di `KEYIN` e `VALUEIN` sono gli input della funzione `map` del Mapper, e devono corrispondere ai tipi che l'`InputFormat` di riferimento restituisce. `KEYOUT` e `VALUEOUT` sono invece i tipi che il Mapper restituisce rielaborando le chiavi e i valori in input. `map` ha la seguente signature:

```
protected void map(KEYIN key, VALUEIN value, Context context)
    throws IOException, InterruptedException
```

Una volta restituiti dal Mapper, le coppie vengono date in input a una classe `Reducer`, che ha una signature simile:

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

Una classe che estende `Reducer` ha un metodo `reduce`, che diversamente dal metodo `map` riceve in input una chiave, e un iterabile di tutti i valori che hanno quella stessa chiave:

```
protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
    throws IOException, InterruptedException
```

Nella fase di reduce, quindi, i valori sono **aggregati** in base alla chiave. I valori a questo punto possono essere combinati a seconda dell'esigenza dell'utente per restituire un risultato finale.

Esempio di un programma MapReduce

Come esempio di programma per MapReduce, si prende in considerazione l'analisi di log di un web server. Il dataset su cui si esegue l'elaborazione è fornito liberamente dalla NASA[10], e

[1] Dato che Java non permette la restituzione di valori multipli da una funzione, per "restituire" i valori si usa il metodo `Context.write` dell'oggetto `Context` ricevuto in input da `map` e `reduce`. È comunque intuitivo pensare ai Mapper e ai Reducer come entità che eseguono associazioni da valore a valore.

corrisponde ai log di accesso al server HTTP del NASA Kennedy Space Center dal 1/07/1995 al 31/07/1995. Il log è un file di testo con codifica ASCII, dove ogni riga corrisponde a una richiesta e ognuna di queste contiene le seguenti informazioni:

1. L'host che esegue la richiesta, sottoforma di hostname quando disponibile o indirizzo IP altrimenti
2. Timestamp della richiesta, in formato "`WEEKDAY MONTH DAY HH:MM:SS YYYY`" e fuso orario, con valore fisso `-0400`.
3. La Request-Line HTTP tra virgolette
4. Il codice HTTP di risposta
5. La dimensione in byte della risposta.

```
ntp.almaden.ibm.com - - [24/Jul/1995:12:40:12 -0400]
    "GET /history/apollo/apollo.html HTTP/1.0" 200 3260

fsd028.osc.state.nc.us - - [24/Jul/1995:12:40:12 -0400]
    "GET /shuttle/missions/missions.html HTTP/1.0" 200 8678
```

Listato 3.1: Campione di due righe dal log da analizzare

A partire da questo log, si vuole capire quante richieste siano state ricevute da ogni risorsa HTTP. Un possibile approccio alla risoluzione del problema è eseguire il parsing di ogni riga del log nel Mapper utilizzando un'espressione regolare, per estrarre l'URI dalla richiesta. Il Mapper, per ogni riga, restituische l'URI come chiave e 1 come valore.

Dopo l'esecuzione dei Mapper, i Reducer riceveranno una coppia formata dall'URI delle richieste come chiave, e da un iterabile di valori 1, uno per ogni richiesta. È sufficiente sommare questi valori per ottenere il numero di richieste finale per l'URI chiave.

Come si può osservere da lst. 3.2, i tipi utilizzati dal Mapper non sono tipi standard Java, ma sono forniti dalla libreria. Hadoop utilizza un suo formato di serializzazione per lo storage e per la trasmissione dei dati in rete, le cui funzionalità sono accessibili tramite l'interfaccia `hadoop.io.Writable`. Le classi `LongWritable` e `Text` sono dei wrapper sui tipi `long` e `String` che forniscono i metodi richiesti dall'interfaccia di serializzazione, e i valori contenuti in questi tipi possono essere ottenuti rispettivamente con `LongWritable.get()` e `Text.toString()`.

Per il resto, le operazioni del Mapper sono intuitive: si utilizza l'espressione regolare per ottenere il token contenente l'URI della richiesta, e tramite `context.write` il Mapper invia la coppia URI e 1.

Il Reducer, mostrato in lst. 3.3, prende in input nel suo metodo `reduce` i valori aggregati in base alla chiave. Una volta sommati in una variabile accumulatore, questi vengono scritti in output in una coppia URI-accumulatore. L'insieme di tutti i valori restituiti dal Reducer costituiscono l'output finale del programma.

Prima di poter eseguire l'applicazione, è necessario creare un esecutore, ovvero una classe contenente un punto d'entrata `main` che utilizzi le API di Hadoop per eseguire il programma,

analogamente a come descritto in Esecuzione di software in Hadoop. I lavori MapReduce sono configurati tramite l'oggetto `hadoop.mapreduce.Job`, che richiede di specificare le classi da utilizzare come Mapper e Reducer, assieme ai percorsi dei file da elaborare. L'esecutore dell'analizzatore di log è mostrato in lst. 3.4.

L'oggetto `job` è il centro della configurazione del programma MapReduce. Tramite questo si specificano il `jar` contenente le classi dell'applicazione, il nome del Job, utilizzato per mostrare descrittivamente nei log e nell'interfaccia web lo stato di completamente di questo, le classi Mapper e Reducer e i tipi dei valori di output del Reducer. Vengono impostati anche i path del file di input e dei file di output, utilizzando i valori ricevuti come parametri in `args`. Il job viene effettivamente eseguito alla chiamata di `job.waitForCompletion(bool verbose)`, che restituisce `true` quando questo va a buon fine.

Al termine della compilazione e pacchettizzazione, il programma può essere eseguito con il comando `hadoop`:

```
$ hadoop LogAnalyzer /example/NASA_access_log_Jul95 /example/LogAnalyzerOutput

17/07/03 18:17:47 INFO Configuration.deprecation: session.id is deprecated.
    Instead, use dfs.metrics.session-id
17/07/03 18:17:47 INFO jvm.JvmMetrics: Initializing JVM Metrics with
    processName=JobTracker, sessionId=
17/07/03 18:17:47 WARN mapreduce.JobResourceUploader: Hadoop command-line
    option parsing not performed. Implement the Tool interface and execute
    your application with ToolRunner to remedy this.
17/07/03 18:17:48 INFO input.FileInputFormat: Total input files to process : 1
17/07/03 18:17:48 INFO mapreduce.JobSubmitter: number of splits:2
17/07/03 18:17:48 INFO mapreduce.JobSubmitter: Submitting tokens for
    job: job_local954245035_0001
17/07/03 18:17:48 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
17/07/03 18:17:48 INFO mapreduce.Job: Running job: job_local954245035_0001
17/07/03 18:17:48 INFO mapred.LocalJobRunner: OutputCommitter set in config null
...
```

Il metodo `job.waitForCompletion` è stato invocato con il parametro `verbose` impostato a `true`, per cui l'esecuzione stampa in output un log sul job in esecuzione. È anche possibile verificare lo stato di esecuzione dei job tramite un'interfaccia web fornita dal framework.

Al termine dell'esecuzione, i risultati sono disponibili in HDFS nella cartella `/example/LogAnalyzerOutput`, come specificato nei parametri d'esecuzione. I risultati si trovano in una cartella perché questi sono composti da più file, uno per ogni Reducer eseguito parallelamente dal framework. In questo caso, il job è stato eseguito da un solo reducer, per cui i risultati si trovano in un unico file. È possibile scegliere la quanitità di Reducer da eseguire parallelamente nel framework, mentre i Mapper, come si vedrà, sono stabiliti in base all'input.

Eseguendo `ls` nella cartella di output si può effettivamente verificare la presenza del file prodotto dal Reducer.

```
$ hadoop fs -ls /example/LogAnalyzerOutput
Found 2 items
-rw-r--r--   3 heygent hdfs          0 2017-07-03 18:17 /example/.../_SUCCESS
-rw-r--r--   3 heygent hdfs     804597 2017-07-03 18:17 /example/.../part-r-0000
```

Assieme al risultato della computazione, MapReduce salva un file vuoto chiamato `_SUCCESS`, di cui si può verificare la presenza in HDFS per capire se il job è andato a buon fine. Consultando il file, si può osservare il risultato della computazione eseguita.

```
...
/elv/DELTA/del181.gif   71
/elv/DELTA/del181s.gif  390
/elv/DELTA/deline.gif   84
/elv/DELTA/delseps.jpg  90
/elv/DELTA/delta.gif    1492
/elv/DELTA/delta.htm    267
/elv/DELTA/deprev.htm   71
/elv/DELTA/dsolids.jpg  84
/elv/DELTA/dsolidss.jpg 369
/elv/DELTA/euve.jpg     36
/elv/DELTA/euves.jpg    357
/elv/DELTA/rosat.jpg    38
/elv/DELTA/rosats.jpg   366
/elv/DELTA/uncons.htm   163
...
```

Astrazioni su MapReduce

Spark

```java
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

public class LogMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

    private final static Pattern logPattern = Pattern.compile(
        ".*\"[A-Z]+ (.*) HTTP.*"
    );

    private final static LongWritable one = new LongWritable(1);

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

        final String request = value.toString();
        final Matcher requestMatcher = logPattern.matcher(request);

        if(requestMatcher.matches()) {
            context.write(
                new Text(requestMatcher.group(1)),
                one
            );
        }
    }

}
```

Listato 3.2: Implementazione del Mapper utilizzato per analizzare il file di log.

```java
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class LogReducer extends Reducer<Text, LongWritable, Text, LongWritable> {
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {

        long accumulator = 0;

        for(LongWritable value: values) {
            accumulator += value.get();
        }

        context.write(key, new LongWritable(accumulator));

    }
}
```

Listato 3.3: Implementazione del Reducer per il programma di analisi dei log.

```java
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class LogAnalyzer {

    public static void main(String args[]) throws Exception {
        if(args.length != 2) {
            System.err.println("Usage: LogAnalyzer <input path> <output path>");
        }

        Job job = Job.getInstance();
        job.setJarByClass(LogAnalyzer.class);
        job.setJobName("LogAnalyzer");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(LogMapper.class);
        job.setReducerClass(LogReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Listato 3.4: Esecutore dell'analizzatore di log.

Parte 4

Stream Processing

Kafka

Spark Streaming

Storm

Parte 5

NoSQL e Big Data

Scalabilità e CAP Theorem

HBase

Query MapReduce su HBase

Bibliografia

1. Gartner Survey Highlights Challenges to Hadoop Adoption, http://www.gartner.com/newsroom/id/3051717

2. Hadoop Market Forecast 2017-2022, https://www.marketanalysis.com/?p=279

3. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

4. Apache Hadoop Documentation, https://hadoop.apache.org/

5. Hadoop Ubuntu Packagers PPA, https://launchpad.net/~hadoop-ubuntu/+archive/ubuntu/stable

6. AUR - Hadoop, https://aur.archlinux.org/packages/hadoop/

7. MapR-FS Overview, https://mapr.com/products/mapr-fs/

8. White, T.: Hadoop: The Definitive Guide. (2015)

9. Shi, J., Qiu, Y., Farooq Minhas, U., Jiao, L., Wang, C., Reinwald, B., Ozcan, F.: Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics, http://www.vldb.org/pvldb/vol8/p2110-shi.pdf

10. HTTP Logs from the KSC-NASA, http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html