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

Anno Accademico 2016/2017

Indice

Introduzione

Negli ultimi decenni, i Big Data hanno preso piede in modo impetuoso in una grande varietà di ambiti. Settori come medicina, finanza, business analytics e marketing sfruttano i Big Data per guidare lo sviluppo, utilizzando tecnologie che riescono a ricavare valore da grandi dataset in tempi eccezionalmente brevi rispetto al passato.

L'innovazione che rende possibili questi risultati è stata guidata dal software molto più che dall'hardware. Nuove idee sulla computazione distribuita e sull'organizzazione dei suoi processi hanno permesso un notevole progresso nell'efficienza di elaborazione di grandi quantità di dati.

Uno dei fattori più importanti ad aver dato slancio a questo fenomeno 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, il loro sfruttamento efficace può essere 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. Nella prima parte si analizzano i vari tipi di paradigmi di processing e di come differiscono tra loro, e le architetture software basate su di questi. Nella seconda parte si affronta Hadoop, il framework per la computazione distribuita più popolare per l'analisi di Big Data. Nella terza parte e quarta parte si osservano i paradigmi di elaborazione *batch* e *stream*, e due tool che li mettono a disposizione, MapReduce e Spark, e degli esempi pratici per illustrare il loro fuzionamento.

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 sulle interfacce 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**, che identifica i Big Data come collezioni di informazione che presentano grande abbondanza in una o più delle seguenti caratteristiche:

- Il **volume** delle informazioni, che può aggirarsi dalle decine di terabyte per arrivare fino ai 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. 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, gestite da enti e persone diverse. Per questa ragione, le strutture dei dati ricavati difficilmente saranno uniformi. Inoltre, l'utilizzo di dati non strutturati rigidamente è prevalente nelle tecnologie consumer, business e scientifiche (come documenti JSON, XML e CSV), che sono spesso un obiettivo auspicabile per l'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 a ritmi sempre più incalzanti.

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'elaborare un intero dataset in un'unità di lavoro, per poi ottenere i risultati al termine di questa.

Questo approccio è ottimale quando i dati da elaborare sono disponibili a priori, e non c'è necessità di ottenere i risultati in tempi immediati o con bassa latenza. Tuttavia, questo approccio ha dei limiti.

- 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 per automatizzare il processo;
- 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*, che rappresentano flussi di dati contigui in arrivo nel corso del tempo. Gli stream forniscono nuovi dati in modo *asincrono*, e il loro arrivo fa scattare eventi di ricezione a cui il software può reagire. I job in streaming molto spesso non hanno un termine prestabilito, ma vengono terminati dall'utente, e i risultati dell'elaborazione sono resi disponibili mentre questa 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 completamente accessibili a priori dal programma. I dati di input in una computazione batch sono determinati al suo inizio, e non possono cambiare durante il suo corso. Questo significa che se si rendesse desiderabile fornire in input una nuova informazione a un lavoro batch, l'unico modo per farlo è rieseguire interamente il lavoro.

Lo **stream processing**, invece, è progettato per *data in motion*, dati in arrivo continuo non necessariamente disponibili prima dell'inizio dell'elaborazione. Esempi di data in motion possono essere rappresentati dai dati ricevuti in un socket TCP inviati da reti di sensori IOT, o dall'ascolto di servizi di social media.

L'astrazione dello stream è abbastanza geneale da poter rappresentare anche *data at rest*. Questa proprietà è desiderabile, perché permette l'uso di tool di elaborazione in streaming per processare *data at rest*.

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 hanno un trend positivo 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.

Architetture di sistemi Big Data

Ad oggi, le architetture dei sistemi che sfruttano i Big Data si basano principalmente su due modelli, la **Lambda** e la **Kappa** architecture.

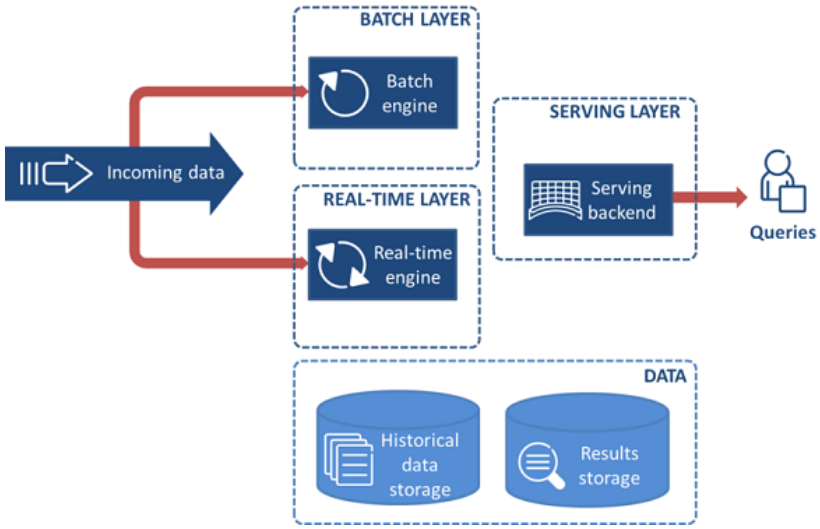

Figura 1.1: Diagramma della Lambda Architecture[4].

La Lambda architecture utilizza tre unità logiche, il **batch layer**, lo **speed layer** e il **serving layer**. Il serving layer è un servizio o un insieme di servizi che permettono di eseguire query sui dati elaborati dagli altri due layer. Il batch layer esegue framework per computazioni batch, mentre lo speed layer esegue computazioni stream. Questi due layer rendono disponibili i risultati delle loro computazioni al serving layer per la consultazione da parte degli utenti.

Il **batch layer** opera sui dati archiviati storicamente, e riesegue le computazioni periodicamente per integrare i nuovi dati ricevuti. Questo layer può eseguire velocemente computazioni sulla totalità dei dati. Lo **speed layer** invece elabora i dati asincronamente alla loro ricezione, e offre risultati con una bassa latenza.

Questo approccio è il più versatile, perché permette l'utilizzo di entrambi i paradigmi e della totalità degli strumenti progettati per batch e stream processing. Tuttavia, i layer batch e speed richiedono una gestione separata, e il mantenimento di due basi di codice scritte con API e potenzialmente linguaggi diversi, anche per applicazioni che eseguono le stesse funzioni. I sistemi che implementano architetture Lambda sono i più onerosi nello sviluppo e nella manutenzione.

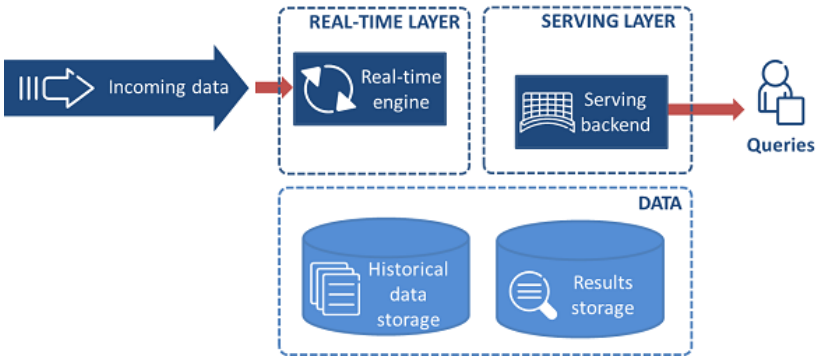

Figura 1.2: Diagramma della Kappa Architecture[4]

In contrapposizione, la Kappa architecture non utilizza un batch layer, e la tutte le computazioni vengono eseguite dallo speed layer. Per eseguire elaborazioni sui dati archiviati, questi vengono rappresentati come uno stream, che viene dato in ingestione allo speed layer. In questo modo gli strumenti e le basi di codice possono essere unificate, semplificando l'architettura e rendendo la gestione del sistema meno impegnativa.

Come regola generale, si può definire preferibile la Lambda architecture per l'efficienza delle computazioni, superiore nei sistemi di elaborazione batch. La Kappa architecture è invece preferibile quando le elaborazioni che si vogliono eseguire sui dati storici e quelli in arrivo sono identiche o molto simili, o si vuole ottenere un sistema architetturalmente più semplice. Spesso la scelta dipende da un tradeoff tra questi due fattori.

Parte 2

Hadoop

Hadoop è una piattaforma software utilizzata per lo storage e la computazione distribuita di dataset di grandi dimensioni. Hadoop viene eseguito in *cluster* di computer, che vengono coordinati dalla piattaforma per fornire delle API in grado di astrarre una parte importante della complessità insita nei sistemi distribuiti. La piattaforma fornisce delle interfacce per l'elaborazione diretta dei dati da parte degli utenti, e delle primitive di livello più basso che consentono l'implementazione di altri framework basati sulla sua infrastruttura di base. Grazie a quest'ultima caratteristica, Hadoop è diventato un perno centrale nell'ambito dei Big Data, su cui si è costruito un ecosistema di tool e tecnologie strettamente integrati con esso.

La documentazione ufficiale[5] 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 importanti:

- **Semplici modelli di programmazione**

  Hadoop raggiunge molti dei suoi obiettivi fornendo un'interfaccia di alto livello al programmatore, in modo di potersi assumere la responsabilità di molti concetti complessi e necessari alla correttezza e all'efficienza della computazione distribuita, ma che hanno poco a che fare con il problema da risolvere in sé (come la sincronizzazione di task paralleli e lo scambio dei dati tra nodi del sistema distribuito).

  Il framework fornisce una piattaforma di programmazione distribuita rivolta agli utenti chiamata MapReduce, e ne sono state sviluppate molte altre da terze parti.

- **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 parallelismo 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 già 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 e utilizzare un framework basato su Hadoop per usifruirne. Questo principio viene definito **data locality**.

- **Rack awareness**

  Nel contesto di Hadoop, *rack awareness* si riferisce a delle ottimizzazioni sull'utilizzo di banda di rete e sull'affidabilità che Hadoop fa basandosi sulla struttura del cluster.

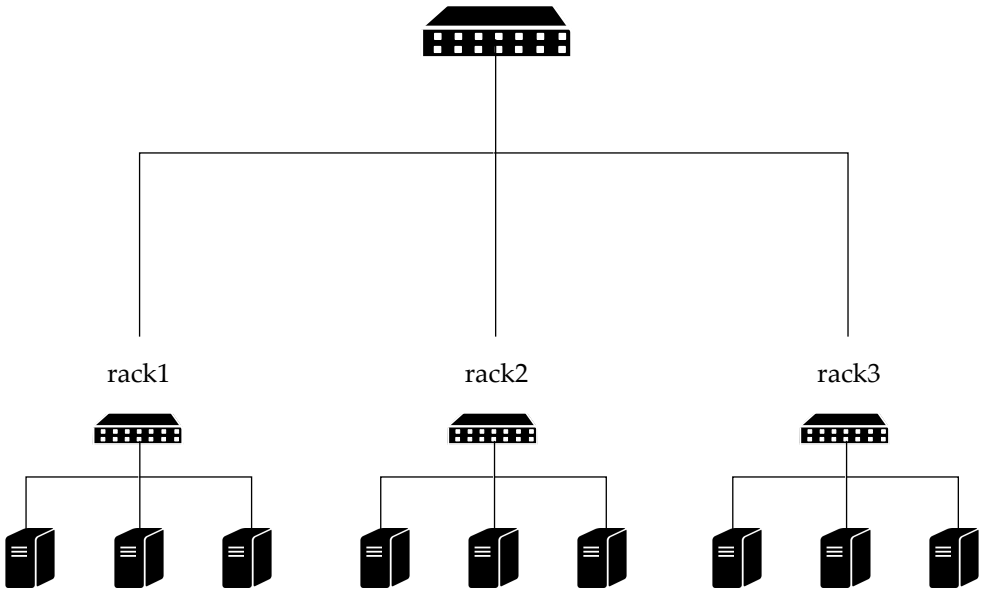

Figura 2.1: Topologia di rete tipica di un cluster Hadoop.

Quando configurato per essere *rack aware*, Hadoop considera il cluster strutturato come un insieme di *rack* che contengono i suoi nodi. I nodi di ciascun rack sono connessi a uno

switch di rete (o dispositivo equivalente), e uno switch centrale connette tutti gli switch dei vari rack.

A partire da questa struttura si può fare un'assunzione importante: la comunicazione tra nodi in uno stesso rack è meno onerosa in termini di banda rispetto alla comunicazione tra nodi in rack diversi, perché la comunicazione può essere commutata tramite un solo switch.

Quando possibile, Hadoop utilizza questo principio per minimizzare l'uso di banda tra nodi del cluster. Come si vedrà, i vari componenti di Hadoop fanno uso della configurazione di rete per ottimizzazare dell'uso della rete e per ottenere una migliore fault-tolerance.

- **Scalabilità**

  Hadoop è in grado di sostenere cluster composti da un gran numero di macchine. Yahoo riporta di eseguire un cluster Hadoop composto da circa 4500 nodi, utilizzato per il sistema pubblicitario e di ricerca[6].

- **Hardware non necessariamente affidabile**

  I cluster di macchine che eseguono Hadoop non hanno particolari requisiti di affidabilità. Il framework è progettato per essere eseguito su *commodity hardware* e per tenere in conto dell'alta probabilità di fallimento dell'hardware attenuandone le conseguenze, sia dal punto di vista dello storage e della potenziale perdita di dati, che da quello della perdita di risultati intermedi nel corso dell'esecuzione di lavori computazionalmente costosi. In questo modo l'utente è sgravato dal compito generalmente complesso 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. Per un approccio più strutturato, sono disponibili repository che forniscono versioni pacchettizzate di Hadoop, come il PPA per Ubuntu[7] e i pacchetti AUR per Arch Linux[8].

Ci sono anche distribuzioni di immagini virtuali Linux create appositamente con lo scopo di fornire un ambiente preconfigurato di prototipazione 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[9].

Pur essendo progettato per l'uso efficiente in cluster di macchine, Hadoop fornisce tre diverse modalità di esecuzione, di cui due locali:

- In modalità **standalone**, l'esecuzione avviene in una sola macchina e in un solo processo. Questa modalità è utilizzata per eseguire il debug di programmi scritti per Hadoop.
- La modalità **pseudodistribuita** è simile alla modalità standalone, ma l'esecuzione avviene in diversi processi.
- La modalità **distribuita** è la modalità operativa di Hadoop, e la sua esecuzione avviene in un cluster composto da più macchine.

La modalità di esecuzione e tutte le altre opzioni di Hadoop sono configurabili tramite file XML che si trovano, a partire dalla cartella d'installazione, in `etc/hadoop`. Ogni componente di Hadoop (HDFS, MapReduce, Yarn) ha un file di configurazione apposito che contiene le sue impostazioni, e un file di configurazione globale per il cluster 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`

È possibile far selezionare ad Hadoop una cartella diversa da `etc/hadoop` da cui leggere 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 il nome di una classe contente un metodo `main` che si desidera eseguire (un entry point dei programmi Java).

```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.

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:

```java
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 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, scritto in Java ed eseguito nello userspace. HDFS è stato studiato e progettato per fornire un sistema di storage distribuito che permetta un'efficiente elaborazione batch di grandi dataset, e che sia resiliente al fallimento delle singole macchine del cluster.

I dati contenuti in HDFS sono organizzati, a livello di storage, in unità logiche chiamate *blocchi*, nel senso comune del termine nel dominio dei 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 minimizzare 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].

HDFS è basato sulla specifica POSIX, e ha quindi una struttura gerarchica. L'utente può strutturare i dati salvati in directory, e impostare permessi di accesso in file e cartelle. Tuttavia, l'adesione a POSIX non è rigida, e alcune operazioni non sono rese possibili, come la modifica dei file in punti arbitrari. Queste restrizioni permettono ad HDFS di semplificare la sua architettura, e di implementare efficientemente funzioni specifiche del suo dominio (come il batch processing),

Principi architetturali

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**

  Un sistema che esegue HDFS è composto da molti componenti, con probabilità di fallimento non triviale. Sulla base di questo principio, HDFS dà 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.

  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.

[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. La frammentazione effettiva dipende da come i dati vengono organizzati dal filesystem del sistema operativo.

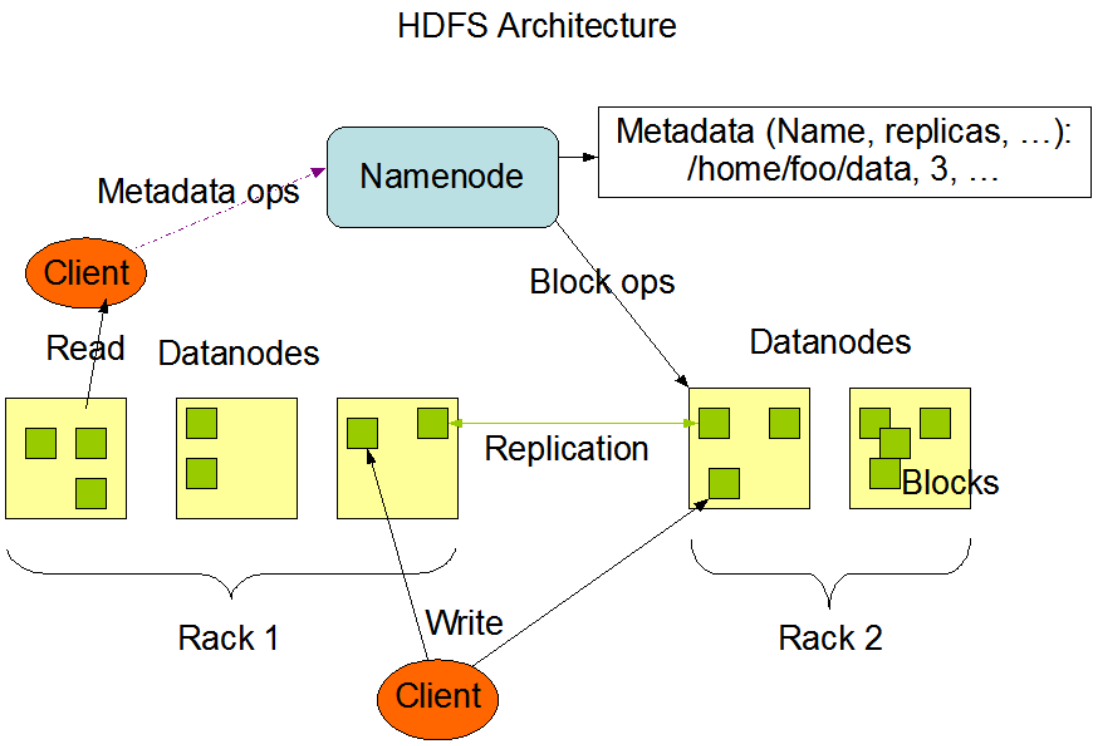

Figura 2.2: Schema di funzionamento dell'architettura di HDFS

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.

  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. Il modello del filesystem di Hadoop è definito **write once, read many**.

- **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, offre un supporto privilegiato ai file molto grandi, dato che questi sono i soggetti principali dell'elaborazione del framework.

- **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, mentre 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.

Replicazione e fault-tolerance

Il blocco è un'astrazione che si presta bene alla replicazione dei dati nel filesystem all'interno del cluster: per replicare i dati, HDFS persiste ogni blocco all'interno di più macchine nel cluster. HDFS utilizza le informazioni sulla configurazione di rete del cluster per decidere il posizionamento delle repliche di ogni blocco: considerando che i tempi di latenza di rete sono più bassi tra nodi in uno stesso rack, HDFS salva due copie del blocco in due nodi che condividono il rack. In questo modo, nell'eventualità in cui una delle copie del blocco non fosse disponibile o avesse problemi d'integrità, una sua replica può essere recuperata in un nodo che si trova all'interno dello stesso rack, minimizzando l'overhead di rete.

Per aumentare la fault-tolerance, HDFS è programmato per salvare un'ulteriore copia del blocco al di fuori del rack in cui memorizza le altre due. Questa operazione salvaguardia l'accesso al blocco in caso di fallimento dello switch di rete del rack che contiene le prime due copie, che renderebbe altrimenti inaccessibili tutte le macchine contenenti il blocco.

Il numero di repliche create da HDFS per ogni blocco è definito *replication factor*, ed è configurabile tramite l'opzione `dfs.replication`. Quando il numero di repliche di un certo file scende sotto la soglia di questa proprietà (eventualità che accade in caso di fallimento dei nodi) HDFS

riesegue trasparentemente la replicazione dei blocchi per raggiungere la soglia definita nella configurazione.

Inoltre, l'integrità dei blocchi è verificata trasparentemente da HDFS alla loro lettura e scrittura, utilizzando checksum CRC-32.

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 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, specificando nell'opzione `dfs.defaultFS` della configurazione del cluster l'URI del filesystem ai cui i percorsi

[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[10].

relativi si riferiscono. Gli URI riferiti a istanze di HDFS hanno schema `hdfs://`, seguito dall'indirizzo IP o dell'hostname della macchina che esegue il NameNode. Specificando l'URI, 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 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 non tecnicamente esperti di lavorare con il filesystem.

[3] Non è necessario mostrare i link dei file, perché HDFS correntemente non li supporta.

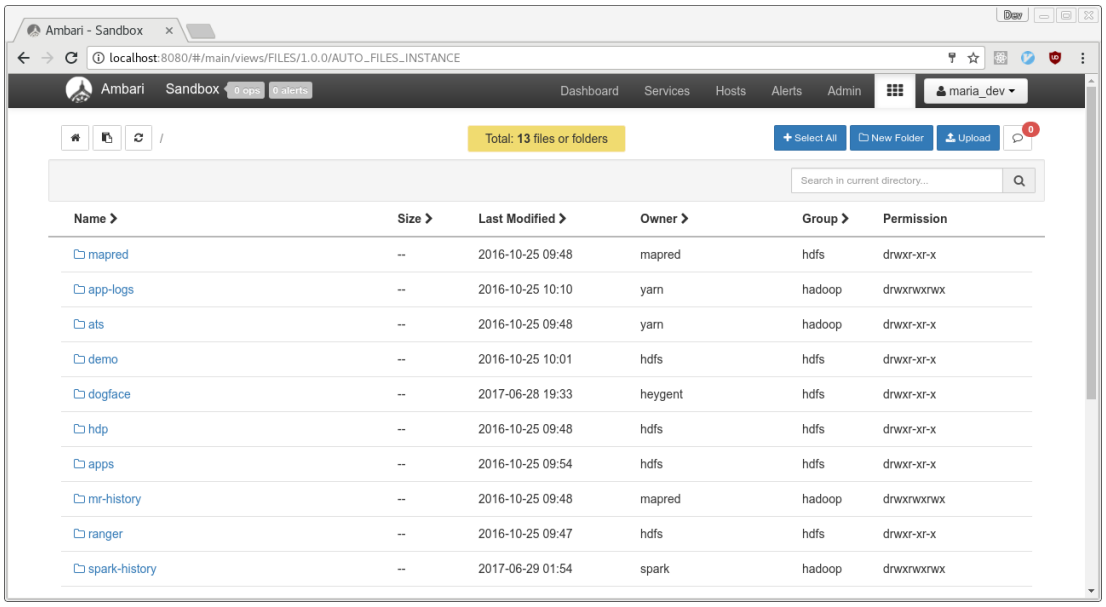

Figura 2.3: Screenshot del file manager HDFS incluso in Ambari

Un altro importante modo di interfacciarsi ad HDFS è l'API `FileSystem` di Hadoop, che permette un accesso programmatico alle sue funzioni da linguaggi per JVM.

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. Questi strumenti permettono l'utilizzo in HDFS di utilità native di gestione dei file del sistema client.

NameNode in dettaglio

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. Prima che il NameNode possa essere operativo, deve ricevere e salvare in memoria 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, ed è sufficiente che per ogni blocco conosca la posizione di un numero minimo di

repliche, determinato dall'opzione del cluster `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 transazioni eseguite nel filesystem a partire dallo stato registrato nella *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
```

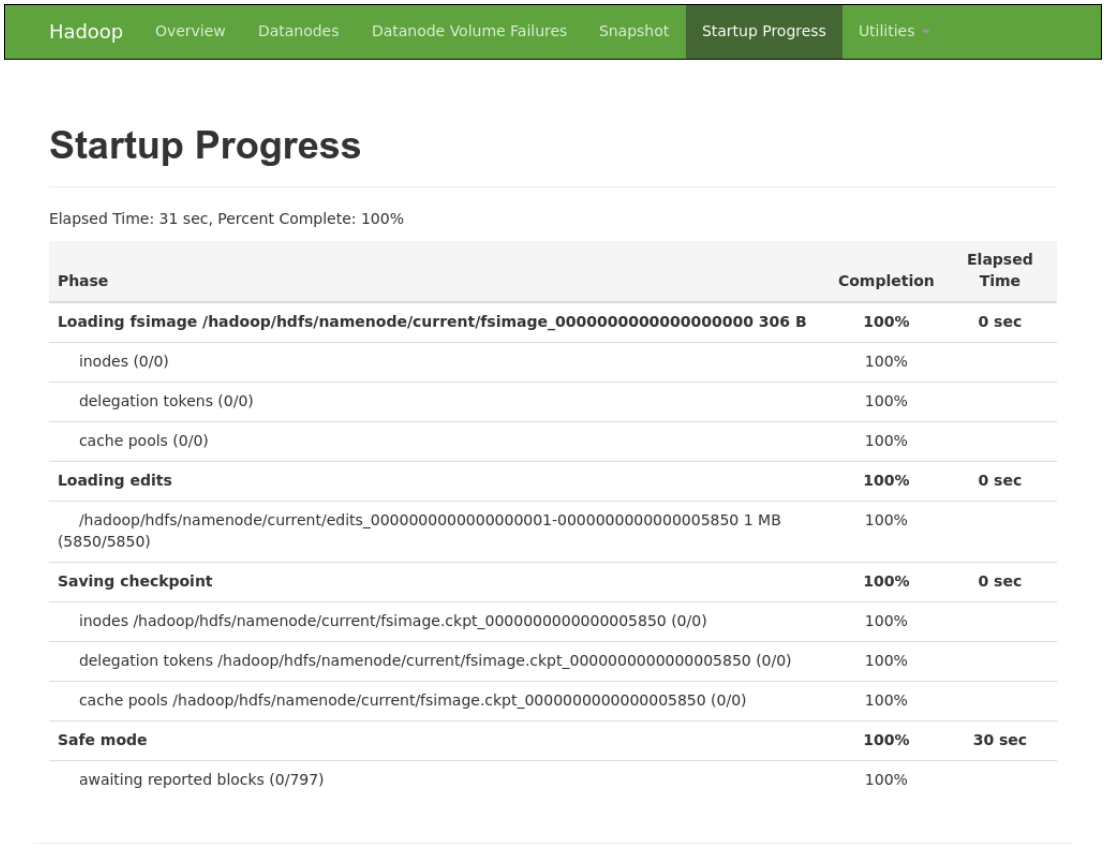

Figura 2.4: Lo stato dello startup di un'istanza di HDFS, mostrata da HDFS Web Interface.

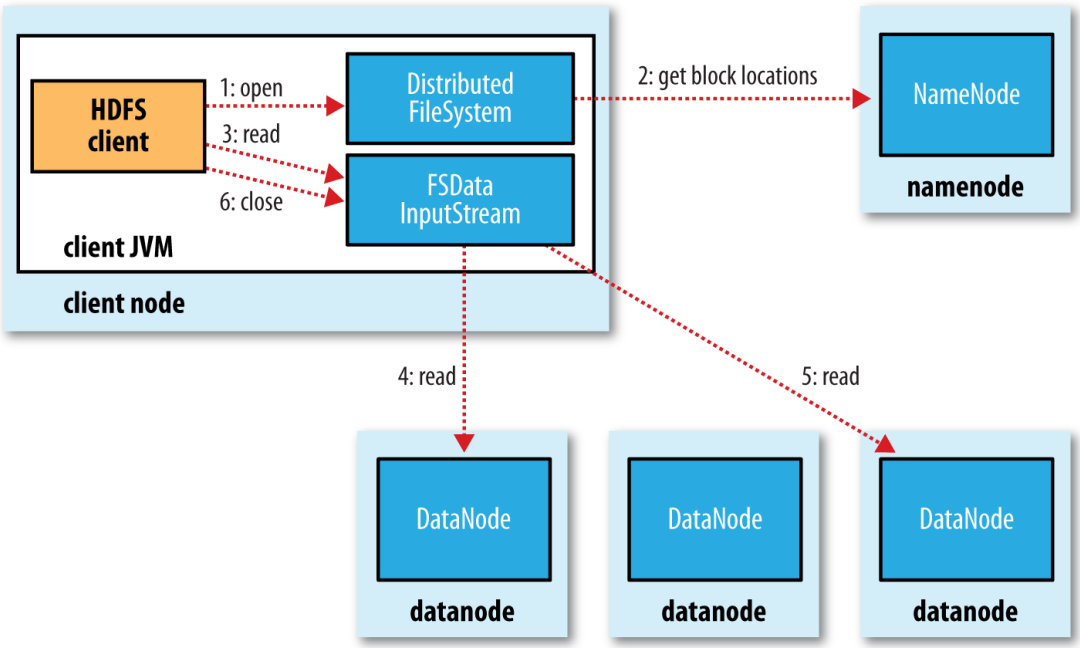

Figura 2.5: Diagramma delle operazioni eseguite nella lettura di un file in HDFS[11]

Processo di lettura di file in HDFS

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://` per filesystem locali), l'istanza concreta di `FileSystem` cambia per gestire opportunamente lo schema richiesto (se supportato).

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 un'altra copia del blocco dal nodo successivo della lista di DataNode 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 passaggio avviene trasparentemente rispetto all'interfaccia, in cui l'utilizzatore 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 evitare colli di bottiglia dovuti a un punto di connessione ai client centralizzato, distribuendo le comunicazioni di rete tra i vari nodi del cluster.

YARN

YARN è acronimo di Yet Another Resource Negotiator, ed è l'insieme di API su cui sono implementati framework di programmazione distribuita di livello più alto, come MapReduce e Spark. YARN si definisce un *"negotiator"* perché è l'entità che decide quando e come le risorse del cluster debbano essere allocate per l'esecuzione, e che gestisce le comunicazioni che riguardano le risorse con tutti i nodi coinvolti. Inoltre, YARN ha l'importante ruolo di esporre un'interfaccia che permette di imporre **vincoli di località** sulle risorse richieste dalle applicazioni, permettendo l'implementazione di applicazioni che seguono il principio di *data locality* di Hadoop.

I servizi di YARN sono offerti tramite *demoni* eseguiti nei nodi del cluster. Ci sono due tipi di demoni in YARN:

- i **NodeManager**, che eseguono su richiesta i processi necessari allo svolgimento delle applicazioni distribuite nel cluster. L'esecuzione dei processi avviene attraverso *container*, che permettono di limitare le risorse utilizzate da ogni processo eseguito. Il NodeManager viene eseguito in ogni nodo del cluster che prende parte alle computazioni distribuite.

- il **ResourceManager**, di cui è eseguita un'istanza per cluster che ha lo scopo di gestire le sue risorse. Il ResourceManager è l'entità che comunica con i NodeManager e che decide quali processi devono eseguire e quando.

I container in YARN possono essere rappresentativi di diverse modalità di esecuzione di un processo. Queste sono configurabili dall'utente tramite la proprietà `yarn.nodemanager.container-executor.clas` il cui valore identifica una classe che stabilisce come i processi debbano essere eseguiti. La configurazione permette l'uso di diversi container di virtualizzazione OS-level, come lxc e Docker[12].

L'esecuzione di applicazioni distribuite in YARN è richiesta dai client al ResourceManager. Quando il ResourceManager decide di avviare un'applicazione, alloca un container in uno dei NodeManager e lo utilizza per invocare un **application master**.

L'application master è specificato dalle singole applicazioni distribuite eseguite su YARN, ed ha i seguenti ruoli[13]:

- negoziare l'acquisizione di nuovi container con il ResourceManager nel corso dell'applicazione;
- utilizzare i container per eseguire i processi distribuiti di cui è costituita l'applicazione;
- monitorare lo stato e il progresso dell'esecuzione dei processi nei container.

Le richieste di container specificano CPU, memoria, e la specifica macchina dove si desidera l'esecuzione. Tra i parametri della richiesta è anche possibile specificare se si vuole permettere l'esecuzione in una macchina diversa da quella richiesta, qualora non fosse disponibile.

Mano a mano che la computazione procede, l'application master può riferire al ResourceManager di rilasciare determinate risorse. Quando il master decide di porre termine all'applicazione, lo riferisce al ResourceManager, in modo da permettere il rilascio del container in cui è eseguito.

Il ResourceManager è in grado di gestire più job contemporaneamente utilizzando diverse politiche di scheduling. L'esecuzione dei job può essere richiesta da diversi utenti ed entità che hanno accesso al cluster, e la scelta di una politica di scheduling adeguata permette di stabilire priorità di accesso diverse per ognuna delle entità coinvolte.

Tra gli scheduler forniti da Hadoop, i seguenti sono i più utilizzati:

- Lo scheduler **FIFO** esegue i job sequenzialmente in ordine di arrivo, e ogni job può potenzialmente utilizzare tutte le risorse del risorse del cluster.

- Il **Fair Scheduler** esegue i job concorrentemente, fornendo una parte delle risorse del cluster a ogni job. I job possono avere una *priorità*, ovvero un peso che determina la frazione di risorse che ricevono. Mano a mano che nuovi job arrivano, le risorse rilasciate dai job già in esecuzione vengono riassegnate al nuovo job per bilanciare la distribuzione delle risorse in base ai pesi[14]. È anche possibile configurare lo scheduler in modo che le risorse siano distribuite in base agli utenti che richiedono l'esecuzione dei job.
- Il **Capacity Scheduler** è il più adatto per condividere cluster tra organizzazioni. Lo scheduler viene configurato per avere diverse *code gerarchiche* di job, ognuna dedicata a un ente che fa uso del cluster. Per ogni coda è specificata una quantità minima di risorse del cluster che devono essere disponibili per l'uso in ogni momento, di cui lo scheduler garantisce la disponibilità.

Gli scheduler sono implementati in classi, e lo scheduler da istanziare viene scelto dal ResourceManager cercando, tramite reflection Java, la classe con il nome indicato in `yarn.resourcemanager.scheduler.class`. L'utente è libero di implementare un proprio scheduler e di specificarne l'identificatore in questa proprietà.

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 degli 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 come in passato, a testimoniare l'effettiva capacità di YARN di generalizzare i modelli di esecuzione nei cluster.

L'alternativa più popolare a MapReduce, Apache Spark, ha API più espressive e funzionali rispetto a MapReduce, ed è più performante in molti tipi di algoritmi[15]. Tramite astrazioni che offrono un controllo più preciso sul comportamento dei risultati dell'elaborazione, Spark trova applicazioni pratiche in vari ambiti, tra cui machine learning[16], graph processing[17] e elaborazione SQL[18].

In questa sezione si esaminano MapReduce e Spark, quali sono le limitazioni di MapReduce che hanno fatto sentire la necessità di un nuovo modello computazionale, e alcune delle soluzioni e modelli di programmazione offerti da Spark.

MapReduce

Il modello computazionale di MapReduce è composto, nella sostanza, da due componenti, il Mapper e il Reducer. Questi componenti sono specificati dall'utilizzatore del framework, e possono essere descritti come due funzioni.

$$Map(K_1, V_1) \mapsto Sequence[(K_2, V_2)]$$

La funzione Map è eseguita nello stadio iniziale della computazione su valori di input esterni. L'input della funzione Map è una coppia chiave-valore K_1 e V_1, i cui valori dipendono dal tipo di input letto. Ad esempio, nei file di testo, K_1 rappresenta il numero di riga di un file e V_1 la riga di testo corrispondente.

A partire da ogni coppia, Map elabora e restituisce una sequenza di nuove coppie chiave-valore di tipo K_2 e V_2. Queste coppie vengono poi rielaborate trasparentemente dal framework, che esegue due operazioni:

1. **ordina** tutte le coppie in base alla chiave;
2. **aggrega** le coppie che condividono la stessa chiave in una nuova coppia $(K_2, Sequence[V_2])$.

$$Reduce(K_2, Sequence[V_2]) \mapsto (K_2, V_3)$$

Ognuna delle coppie aggregate dal framework viene poi fornita in input alla funzione $Reduce$, che ha quindi a disposizione una chiave K_2 e tutti i valori restituiti da Map che hanno la stessa chiave. $Reduce$ esegue una computazione sui valori di input e restituisce (K_2, V_3), che andrà a far parte dell'output finale dell'applicazione assieme al risultato delle altre invocazioni di $Reduce$, una per ogni chiave distinta restituita da Map.

Sintetizzando, MapReduce permette di dividere e categorizzare l'input in diverse parti, e di elaborare un risultato per ognuna di queste.

MapReduce è un paradigma *funzionale*, dato che il framework richiede di ricevere in input le funzioni utili all'elaborazione dei dati. Per esprimere questo tipo di paradigma in Java si ricorre a classi che incapsulano le funzioni richieste dal framework, che vengono quindi chiamate Mapper e Reducer.

Il Mapper in un'applicazione MapReduce è una classe contenente un metodo `void map`, che riceve in input una chiave e un valore, e un oggetto `Context`, il cui ruolo più importante è fornire il metodo `Context.write(K, V)`, che viene utilizzato per scrivere i valori di output del Mapper.

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

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

Le chiavi e i valori ricevuti in input dal Mapper sono derivati direttamente dall'elemento letto in HDFS. 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. Nella libreria, Hadoop fornisce diversi `InputMapper` che corrispondono a comportamenti di lettura desiderabili per diversi tipi di file e sorgenti, come file con formati colonnari, o contenti coppie chiave-valore divise da marcatori.

I tipi ricevuti in input dal Mapper sono specificati nei parametri generici `KEYIN` e `VALUEIN`, 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. La signature della funzione `map` nella libreria di Hadoop è la seguente:

```
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 a quella del `Mapper`:

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

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

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

I valori possono quindi 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[19], e corrisponde ai log di accesso al server HTTP del 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 contiene le seguenti informazioni:

1. L'host che esegue la richiesta, sotto forma 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.

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

```
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

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, diverso dalla serializzazione integrata in Java. In questo modo il framework ha controllo preciso sulla fase di serializzazione, un fattore importante data la crucialità in termini di efficienza che questa può avere.

Le funzionalità di serializzazione di Hadoop sono rese accessibili dagli oggetti serializzabili tramite l'interfaccia `hadoop.io.Writable`. Le classi `LongWritable` e `Text` sono dei wrapper sui tipi `long` e `String` che implementano l'interfaccia `Writable`, e i valori contenuti in questi tipi possono essere ottenuti rispettivamente con `LongWritable.get()` e `Text.toString()`[1].

Nel Mapper, si utilizza l'espressione regolare `/.*"[A-Z]+ (.*) HTTP.*/` per ottenere il token contenente l'URI della richiesta, e tramite `context.write` si restituisce 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 tutte le coppie restituite dal Reducer costituiscono l'output finale del programma, che vengono scritte in un file di testo separando le chiavi dai valori con caratteri di tabulazioni, e ogni valore di restituzione con un nuova riga.

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.

[1] Le classi definite dagli utenti possono implementare a loro volta l'interfaccia `Writable` per essere supportate come tipi di chiavi e valori nei Mapper e nei Reducer. Inoltre, è possibile configurare Hadoop per utilizzare formati di serializzazione alternativi a quello fornito. Un'alternativa popolare è Apache Avro, un formato e una libreria di serializzazione che mira agli stessi obiettivi di efficienza della serializzazione Hadoop, e che fornisce una maggiore estensibilità e interoperabilità in linguaggi diversi da Java.

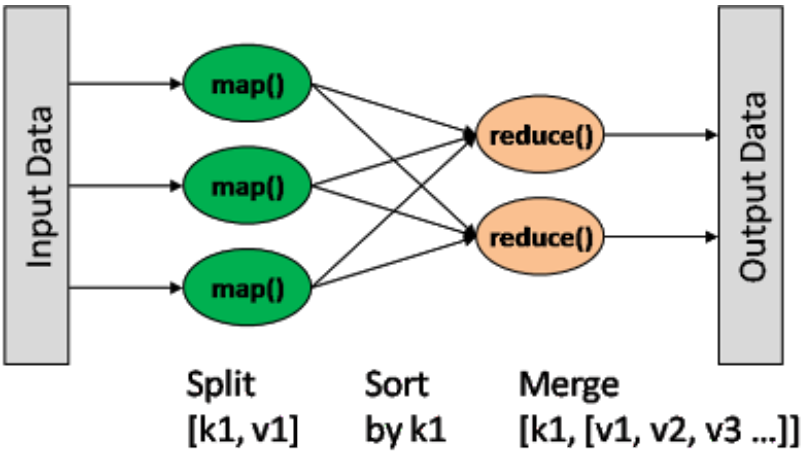

Figura 3.1: Diagramma di funzionamento di MapReduce[20]

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 completamento, 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
```

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. Lo stato di esecuzione dei job è anche consultabile tramite un'interfaccia web fornita dal framework.

```
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
```

```
...
```

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 possono essere composti da più file, uno per ogni Reducer eseguito parallelamente dal framework. Questa limitazione è dovuta ad HDFS, che restringe rigidamente l'accesso in scrittura ai file a un solo utilizzatore. In questo caso, il job è stato eseguito da un solo reducer, per cui i risultati si trovano in un unico file. L'output dei reducer è salvato in file testuali con il nome `part-r-` seguito da un numero sequenziale che identifica l'istanza del Reducer che lo ha prodotto.

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`, utilizzabile per verificare programmaticamente se il job è andato a buon fine. Consultando il file `part-r-0000`, 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
...
```

```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.

Modello di esecuzione di MapReduce

Il modello di programmazione di MapReduce è progettato per essere altamente parallelizzabile, e in modo che sia possibile processare diverse parti dell'input indipendentemente. Questo dato si riflette nel design del Mapper, che riceve come input piccole porzioni del file letto, permettendo al framework di assegnare l'elaborazione delle operazioni di Map a diversi processi indipendenti.

MapReduce è implementato in YARN, e utilizza le sue astrazioni per avvantaggiarsi della località dei dati, eseguendo i processi che riguardano una certa porzione di input nei nodi che contengono i corrispondenti blocchi HDFS. L'esecuzione dei lavori MapReduce avviene secondo i seguenti step:

1. I file vengono partizionati da MapReduce in frammenti chiamati *split*, e per ognuno di questi MapReduce esegue un *map task* in un determinato nodo del cluster. Ogni map task può eseguire uno o più processi nel nodo in cui si trova, a seconda delle risorse assegnate da YARN.

   La dimensione degli split è configurabile, e non corrisponde necessariamente alla dimensione di un blocco HDFS, pur essendo questa l'opzione di default (128 MB). Con split della stessa dimensione dei blocchi, la maggior parte dei dati può essere processata dai nodi che contengono il blocco nel loro storage locale. È possibile configurare MapReduce per utilizzare split più grandi, ma se una parte dello split non si trova nel nodo in cui viene eseguito il map task, questa deve essere ricevuta tramite rete da un altro nodo nel cluster che la contiene, riducendo quindi la *data locality*.

   Configurando MapReduce per utilizzare split più piccoli del blocco, si può ottenere un load balancing migliore, dato che macchine più veloci possono eseguire più task mentre le altre finiscono i precedenti. In questo scenario, bisogna considerare se l'overhead dovuto alla gestione di una quantità maggiore di task possa superare il guadagno ottenuto nei tempi di elaborazione.

2. In ogni *map task*, lo split corrispondente viene diviso in più *record*, che corrispondono alle coppie ricevute in input dal Mapper. Il map task esegue il Mapper in uno o più processi del nodo in cui si trova, per poi salvare il loro output nello storage locale del nodo. Lo storage locale è più efficiente per la scrittura rispetto ad HDFS, ma non offre fault-tolerance, per cui in caso di fallimento del nodo che contiene i risultati di un *map task*, l'application master dell'applicazione MapReduce deve schedulare la sua riesecuzione.

   Oltre all'esecuzione dei Mapper, i nodi in questa fase ordinano la parte di output in loro possesso in base alla chiave. Questo permette di eseguire parallelamente buona parte del sorting dell'input dei Reducer.

3. Quando non ci sono più *map task* da eseguire sull'input, l'application master inizia ad avviare i *reduce task*. I *reduce task* ricevono in input i risultati ordinati prodotti dai *map task* mano a mano che questi sono disponibili. Ogni *map task* può inviare coppie chiave-

valore a ogni *reduce task*, a condizione che coppie con la stessa chiave finiscano sempre nello stesso reducer.

Per fare questo, i nodi che eseguono *map task* dividono il loro output in partizioni, una per ogni Reducer. Ogni chiave delle coppie di output viene associata univocamente a una partizione, utilizzando la seguente funzione[21]:

$$partitionId(K_i) = hash(K_i) \bmod partitionCount$$

In questo modo, la stessa chiave è sempre associata alla stessa partizione in ogni nodo.

4. I nodi che eseguono i Reducer ricevono dai nodi Mapper diversi insiemi ordinati di coppie chiave-valore. Questi gruppi vengono uniti tramite un'operazione di *merge*, analoga alla stessa operazione nel contesto del *mergesort*. Una volta ricevuti tutti i valori, il *reduce task* esegue i Reducer che computano l'output finale dell'applicazione.

Spark

Le astrazioni fornite dal paradigma computazionale di MapReduce tolgono dall'utente l'onere di pensare al dataset in elaborazione, astraendo l'applicazione a una serie di elaborazioni su chiavi e valori. Questa astrazione ha tuttavia un costo: l'utente non ha il controllo sulla gestione del flusso dei dati, che è affidata interamente dal framework.

Il costo della semplificazione diventa evidente quando si cerca di utilizzare MapReduce per eseguire operazioni che richiedono la rielaborazione di risultati. Al termine di ogni job MapReduce, l'output viene salvato in HDFS, ed è quindi necessario rileggerlo dal filesystem per poterlo utilizzare.

Di per sé, MapReduce non contiene un meccanismo che permetta la schedulazione consecutiva di job che ricevono in input l'output di un altro job, e per eseguire elaborazioni che richiedono più fasi è necessario utilizzare tool esterni. Inoltre, l'overhead della lettura e scrittura in HDFS è alto, e MapReduce non fornisce metodi per rielaborare i dati direttamente nella memoria centrale.

Il creatore di Spark, Matei Zaharia[22], ha posto questo problema come dovuto alla mancanza di *primitive efficienti per la condivisione di dati* in MapReduce. Per come le interfacce di MapReduce sono poste, sarebbe anche difficile crearne di nuove, data la mancanza di un'API che sia rappresentativa del dataset invece che delle singole chiavi e valori.

Infine, la scrittura dei risultati delle computazioni in HDFS è necessaria per fornire fault-tolerance su di questi, che andrebbero persi nel caso di un fallimento di un nodo che mantiene i risultati nella memoria centrale. Un sistema di elaborazione che agisca sulla memoria centrale deve necessariamente avere un meccanismo di recupero da fault, per evitare che il fallimento di uno dei singoli nodi coinvolti nella computazione renda necessario rieseguire completamente l'applicazione.

Spark si propone come alternativa a MapReduce, con l'intenzione di offrire soluzioni a questi problemi. Le soluzioni derivano da un approccio funzionale, sfruttando strutture con semantica di immutabilibità per rappresentare i dataset e API che utilizzano funzioni di ordine superiore per esprimere concisamente le computazioni. L'astrazione principale del modello di Spark è il Resilient Distributed Dataset, o RDD, che rappresenta una collezione immutabile e distribuita di record di cui è composto un dataset o una sua rielaborazione.

Spark è scritto in Scala, e la sua esecuzione su Hadoop è gestita da YARN. YARN non è l'unico motore di esecuzione di Spark, che può essere eseguito anche su Apache Mesos o in modalità standalone, sia su cluster che su macchine singole. Le API client di Spark sono canonicamente disponibili in Scala, Java, R e Python.

Spark dispone anche di una modalità interattiva, in cui l'utente interagisce con il framework tramite una shell REPL Scala o Python. Questa modalità permette la prototipazione rapida di applicazioni, e abilita l'utilizzo di paradigmi come l'**interactive data mining**, che consiste nell'eseguire analisi sui dataset in via esploratoria, scegliendo quali operazioni intraprendere mano a mano che si riceve il risultato delle elaborazioni precedenti.

RDD API

I Resilient Distributed Dataset sono degli oggetti che rappresentano un dataset partizionato e distribuito, su cui è possibile eseguire operazioni parallelamente.

Gli RDD sono immutabili, e ogni computazione richiesta su di questi restituisce un valore o un nuovo RDD. Le computazioni sono eseguite tramite metodi chiamati sugli oggetti RDD, e si dividono in due categorie: **azioni** e **trasformazioni**.

Le trasformazioni creano un nuovo RDD, basato su delle operazioni deterministiche sull'RDD di origine. L'elaborazione del nuovo RDD è lazy, e non viene eseguita finché non viene richiesta l'esecuzione di un'azione.

Alcuni esempi di trasformazioni sono `map`, che associa a ogni valore del dataset un nuovo valore, e `filter`, che scarta dei valori nel dataset in base a un predicato. Spesso, per descrivere le computazioni, le trasformazioni richiedono in input funzioni pure (prive di side-effect).

Tabella 3.1: Alcune trasformazioni supportate da Spark

Trasformazione	Risultato
`map(fun)`	Restituisce un nuovo RDD passando ogni elemento della sorgente a `fun`.
`filter(fun)`	Restituisce un RDD formato dagli elementi che `fun` mappa in `true`.
`union(dataset)`	Restituisce un RDD che contiene gli elementi della sorgente uniti con quelli di `dataset`.

Trasformazione	Risultato
`intersection(dataset)`	Restitusce un RDD contente gli elementi comuni alla sorgente e a `dataset`
`distinct([numTasks]))`	Restituisce un RDD contentente gli elementi del dataset senza ripetizioni

Le azioni fanno scattare la valutazione dell'RDD, che porta quindi all'esecuzione di tutte le trasformazioni da cui questo è derivato. Alcuni esempi di azioni sono `foreach`, che esegue una funzione specificata dall'utente per ogni input del dataset, `reduce`, che utilizza una funzione di input per aggregare i valori del dataset, e `saveAsTextFile`, che permette il salvataggio di un RDD in un file testuale.

Ogni sessione interattiva e programma Spark utilizza un oggetto `SparkContext` per creare gli RDD iniziali. Lo `SparkContext` contiene le impostazioni principali sul programma, come il master di esecuzione (`local`, `yarn`, `mesos`) e l'identificativo con cui tracciare il job in esecuzione. Le sessioni interattive forniscono lo `SparkContext` automaticamente, in una variabile globale chiamata `sc`.

Le sessioni interattive Spark possono essere avviate tramite i comandi `spark-shell`, che mette a disposizione una shell REPL Scala, o `pyspark`, che ne mette a disposizione una Python. Tramite gli argomenti dell'eseguibile si può specificare il master (di default `local`).

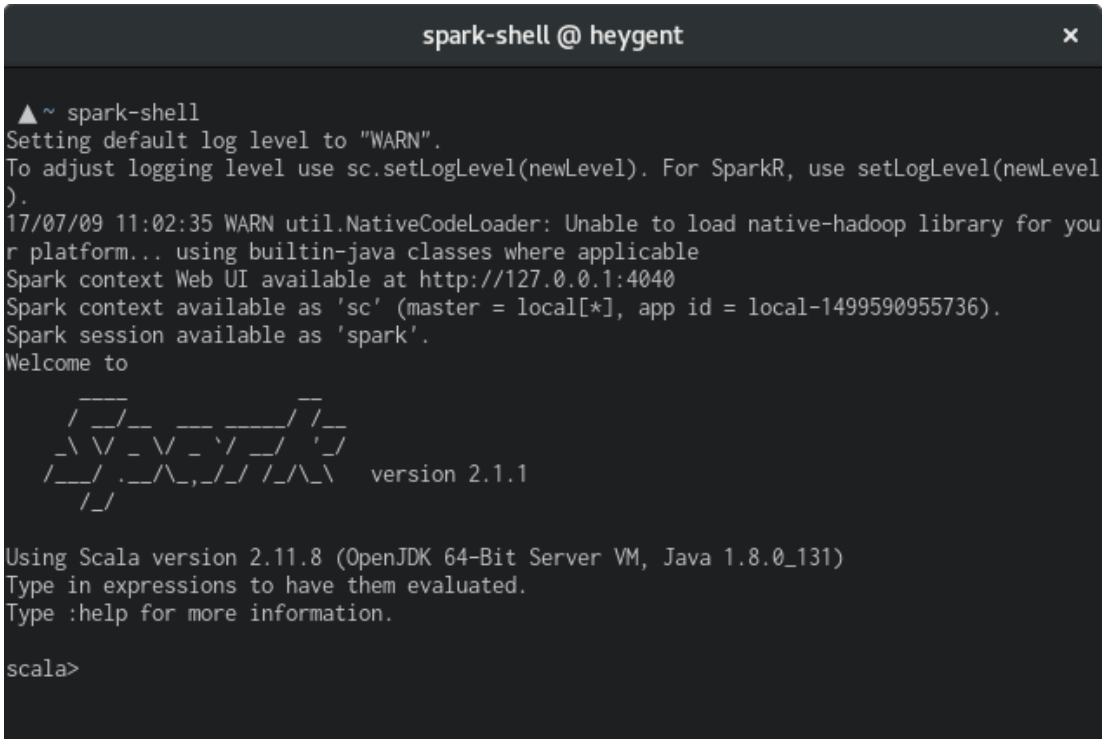

Figura 3.2: Avvio di una sessione interattiva Spark.

Tramite l'oggetto `sc`, si può creare un nuovo RDD a partire da diversi fonti. Il seguente codice crea un RDD partendo da un range inclusivo Scala, analogo allo stesso concetto in Python.

```
scala> val range = sc.parallelize(1 to 50)
range: org.apache.spark.rdd.RDD[Int] =
  ParallelCollectionRDD[0] at parallelize at <console>:24
```

Per creare un RDD a partire da un percorso, `SparkContext` fornisce metodi come `textFile(path: String)`, che permette la lettura di file di testo da storage locali e distribuiti, avvantaggiandosi della *data locality* quando possibile, o `hadoopRDD(job: JobConf)`, che permette l'utilizzo di qualunque `InputFormat` Hadoop per creare il dataset. Nel seguente esempio si crea un RDD a partire dalla versione testuale inglese del libro *Le metamorfosi* di Franz Kafka, offerto gratuitamente dal Progetto Gutenberg[23].

```
scala> val book = sc.textFile("/books/kafka-metamorphosis.txt")
book: org.apache.spark.rdd.RDD[String] =
    /books/kafka-metamorphosis.txt MapPartitionsRDD[28] at textFile
    at <console>:24
```

Una volta ottenuto l'RDD, è possibile iniziare a eseguirvi trasformazioni. È importante tenere in conto che ogni trasformazione restituisce un nuovo RDD, di cui è necessario salvare un riferimento per poterlo utilizzare in seguito. Nelle sessioni interattive Scala i risultati di tutte le espressioni valutate nella shell sono disponibili in variabili con il nome `res` seguito da un identificativo numerico sequenziale, utilizzabili per tenere traccia degli RDD valutati.

```
scala> val words = book.flatMap(_.split(' ')).filter(_ != "")
words: org.apache.spark.rdd.RDD[String] =
    MapPartitionsRDD[30] at filter at <console>:26
```

La trasformazione `flatMap` riceve in input una funzione[2] che restituisce un iterabile di elementi. La funzione viene chiamata su tutti gli elementi dell'RDD, e i valori contenuti negli iterabili restituiti sono raggruppati nell'RDD restituito. Con la funzione `_.split(' ')` si separano le parole in ogni riga del libro. Viene poi eseguita la trasformazione `filter` con il predicato `_ != ""`, per scartare le stringhe vuote che possono risultare dalla trasformazione precedente[3].

Se non vengono fatte specificazioni, l'esecuzione delle trasformazioni avviene ogni volta che viene chiamata un'azione su di un RDD. Per evitare ricomputazioni costose, è possibile specificare quali RDD persistere nella memoria dei nodi, in modo che i risultati computati possano essere riutilizzati in operazioni successive. Per richiedere al framework di salvare i valori computati di un RDD, è sufficiente chiamare il suo metodo `persist`.

```
scala> words.persist()
res10: words.type = MapPartitionsRDD[30] at filter at <console>:26
```

Se il dataset da salvare è molto grande le partizioni potrebbero non entrare completamente in memoria. Il comportamento di default di Spark in questo caso consiste nel mettere in cache solo

[2] Scala supporta una sintassi concisa per la creazione di funzioni anonime, definibili con espressioni che utilizzano l'identificativo _ come valori. A ogni utilizzo di _ corrisponde un parametro della funzione, che viene sostituito nella rispettiva posizione alla chiamata della funzione.

[3] Per semplificare l'esempio, si ignora il casing e la punteggiatura delle parole, che andrebbero altrimenti normalizzate per ottenere un risultato corretto sul conteggio delle parole.

parte della partizione, e ricomputare la parte restante quando viene richiesta. Spark può anche eseguire azioni alternative, come serializzare gli oggetti in modo che occupino meno spazio o eseguire parte del caching su disco. Nel gergo di Spark il comportamento da attuare in questi casi è definito **livello di persistenza**, ed è specificabile come argomento del metodo `persist`.

Tabella 3.2: Alcuni livelli di persistenza forniti da Spark.

Livello di persistenza	Effetto
`MEMORY_ONLY`	Salva l'RDD sotto forma di oggetti deserializzati nella JVM. Se l'RDD non entra in memoria, alcune partizioni non vengono persistite e vengono ricomputate al volo ogni volta che sono richieste. (default)
`MEMORY_AND_DISK`	Salva l'RDD sotto forma di oggetti deserializzati nella JVM. Se l'RDD non entra in memoria, alcune partizioni vengono scritte su disco e lette quando sono richieste.
`MEMORY_ONLY_SER` (solo Java e Scala)	Salva l'RDD come oggetti Java serializzati. Questa opzione è più efficiente in termini di spazio degli oggetti deserializzati, ma più computazionalmente intensiva.
`MEMORY_AND_DISK_SR`	Come `MEMORY_ONLY_SER`, ma le partizioni che non entrano in memoria sono salvate su disco invece di essere ricomputate.
`DISK_ONLY`	Salva le partizioni solo su disco.

Per avviare l'esecuzione delle trasformazioni, è necessario eseguire un'azione. Nel seguente esempio, l'azione eseguita è `take(n: Int)`, che restituisce i primi `n` elementi dell'RDD in un array Scala.

```
scala> words.take(20)
res19: Array[String] = Array(One, morning,, when, Gregor, Samsa, woke,
    from, troubled, dreams,, he, found, himself, transformed, in, his,
    bed, into, a, horrible, vermin.)
```

Il file di origine è stato diviso in parole, come specificato nelle trasformazioni. Dato che è stato chiamato `persist` sull'RDD `words`, i valori rielaborati si trovano ancora nella memoria dei nodi, ed è possibile riutilizzarli semplicemente eseguendo operazioni sull'RDD.

Tramite le interfacce di Spark si può facilmente rappresentare il modello computazionale di MapReduce. A partire da `words`, si può eseguire il conto delle parole all'interno del libro mappando il dataset a coppie chiave-valore, dove la chiave è la parola e il valore è 1. Per eseguire il conto, gli RDD forniscono il metodo `reduceByKey`, che esegue la stessa operazione effettuata dai Reducer nel modello MapReduce: aggrega i valori delle coppie con la stessa chiave.

Diversamente da MapReduce, in `reduceByKey` la funzione che rappresnta il Reducer non riceve un iterabile dei valori, ma un accumulatore e uno degli elementi aggregati. Per ogni gruppo di valori aggregati a una chiave, la funzione viene chiamata una volta per ogni valore del

gruppo, ricevendo in input il valore e l'accumulatore corrente. Il valore di restituzione viene utilizzato come accumulatore di input per l'invocazione sul valore successivo.

$$reducer(A_i, V_i) = A_{i+1}$$

Le coppie chiave-valore sono rappresentate da tuple Scala. La funzione di riduzione da utilizzare in questo caso è la somma.

```
scala> val wordCount = words.map(w => (w, 1)).reduceByKey(_ + _)
wordCount: org.apache.spark.rdd.RDD[(String, Int)] =
    ShuffledRDD[61] at reduceByKey at <console>:28

scala> wordCount.take(20)
res21: Array[(String, Int)] = Array(
    (swishing,,1), (pitifully,1), (someone,5), (better.,2), (propped,1),
    (nonetheless,3), (bone,1), (movements.,2), (order,7), (drink,,1),
    (experience,,1), (behind,15), (Father,,1), (wasn't,5), (been,99),'
    (they,,1), (Father.,1), (introduction,,1), ("Gregor,,3), (she's,1)
    )
```

Al termine della computazione, si rende utile salvare i valori in un mezzo di storage. Questa operazione è eseguibile tramite diverse azioni, come `saveAsTextFile(path: String)`, che salva i risultati come testo, o `saveAsObjectFile(path: String)`, che serializza efficientemente i valori, permettendone un rapido accesso programmatico.

```
scala> wordCount.saveAsTextFile("/tmp/results")
```

Come per MapReduce, i risultati possono essere sparsi per diversi file, a seconda di quanti task paralleli sono stati coinvolti nell'operazione di riduzione.

```
$ cd /tmp/results
$ ls
part-00000  part-00001  _SUCCESS
$ head part-00000
(swishing,,1)
(pitifully,1)
(someone,5)
(better.,2)
(propped,1)
(nonetheless,3)
(bone,1)
(movements.,2)
(order,7)
(drink,,1)
```

Sviluppo ed esecuzione di un Job

Le applicazioni Spark vengono sviluppate ed eseguite in modo simile a MapReduce, in cui ogni applicazione ha un punto di entrata `main` dove i job vengono configurati e avviati.

Riprendendo in considerazione l'esempio dell'analizzatore di log, questo è rappresentabile in maniera molto più succinta tramite le interfacce fornite da Scala e Spark. La prima operazione da eseguire è creare un oggetto di configurazione, come mostrato in lst. 3.5, righe 9-11. In questo caso, si specifica il nome del job come "Log Analyzer" e `yarn` come master di esecuzione. Nella riga 13 si crea uno `SparkContext` utilizzando la configurazione, che viene poi utilizzato per aprire un file HDFS il cui percorso è preso in input dagli argomenti del programma.

L'RDD ottenuto viene utilizzato per mappare ogni riga di richiesta alla risorsa corrispondente. La trasformazione `collect` riceve in input una funzione parziale Scala, che in questo caso tenta di eseguire il match delle righe con l'espressione regolare e di estrarre il valore corrispondente al gruppo di cattura. Sulle righe in cui il match è valido, la funzione restituisce una coppia URI-1. Per eseguire il calcolo finale, si utilizza l'azione `reduceByKey`.

Prima di salvare il file, il risultato viene mappato a una stringa, la cui formattazione permette all'output di essere un valido file TSV (Tab Separated Values).

Per essere eseguito, il file deve essere compilato e impacchettato in un file `jar`. La richiesta di esecuzione di un job può essere fatta tramite l'eseguibile `spark-submit`, distribuito con Spark. Come per l'eseguibile `hadoop`, è possibile specificare gli argomenti che si vogliono passare al programma.

```
$ spark-submit sparkdemo-assembly-1.0.jar /example/NASA_access_log_Jul95
    /tmp/results
```

Una volta inviato, lo stato di un Job può essere consultato tramite un'interfaccia web, come in MapReduce. L'interfaccia mostra la fase di esecuzione del Job, e può fornire visualizzazioni in forma di grafo diretto aciclico degli stage richiesti per la sua esecuzione.

I risultati possono essere quindi consultati nella cartella `/tmp/results` dell'istanza HDFS del cluster.

```
[root@sandbox ~]# hadoop fs -cat /tmp/results/part-00001 | head
/cgi-bin/imagemap/countdown?112,206 2
/shuttle/missions/41-b/images/  19
/cgi-bin/imagemap/countdown?292,205 2
/cgi-bin/imagemap/countdown70?248,269   1
/history/apollo/apollo-13.apollo-13.html    1
/cgi-bin/imagemap/countdown?220,280 1
/news/sci.space.shuttle/archive/sci-space-shuttle-7-feb-1994-87.txt 1
/htbin/wais.pl?current+position 1
/cgi-bin/imagemap/countdown?105,213 2
/cgi-bin/imagemap/fr?280,27 1
```

```scala
import org.apache.spark._

object LogAnalyzer {

  private val logURIRegex = """.*"[A-Z]+\s(.*)\sHTTP.*""".r

  def main(args: Array[String]) {

    val conf = new SparkConf()
      .setAppName("Log Analyzer")
      .setMaster("yarn")

    val sc = new SparkContext(conf)

    val logs = sc.textFile(args(0))

    val counts = logs
      .collect {
        case logURIRegex(uri) => (uri, 1)
      }
      .reduceByKey(_ + _)

    counts.map { case (k, v) => s"$k\t$v" }.saveAsTextFile(args(1))

  }
}
```

Listato 3.5: Analizzatore di log reimplementato in Scala e Spark.

DataFrame API

Spark fornisce un'altra API per l'elaborazione dei dati, basata sull'astrazione del DataFrame. Un DataFrame rappresenta dati strutturati o semistrutturati, come documenti JSON o CSV, di cui Spark è internamente consapevole della struttura. Utilizzando i DataFrame, Spark è in grado di eseguire ottimizzazioni e di fornire operazioni aggiuntive all'utente. Una delle funzioni notevoli dell'API DataFrame è la possibilità di eseguire query SQL sui dataset, utilizzando anche funzioni di aggregazione come `sum`, `avg` e `max`.

Le API DataFrame sono disponibili nelle sessioni interattive Spark tramite un oggetto di `SparkSession`, fornito nella variabile `spark`. In modo analogo agli RDD, le azioni eseguibili sui DataFrame possono essere azioni e trasformazioni.

La creazione di DataFrame è simile alla creazione degli RDD, e avviene utilizzando l'oggetto `spark` per caricare i dati da una sorgente. I dati di questo esempio sono forniti da Population.io, un servizio che fornisce dati aggiornati sulla popolazione mondiale[24]. Il dataset che

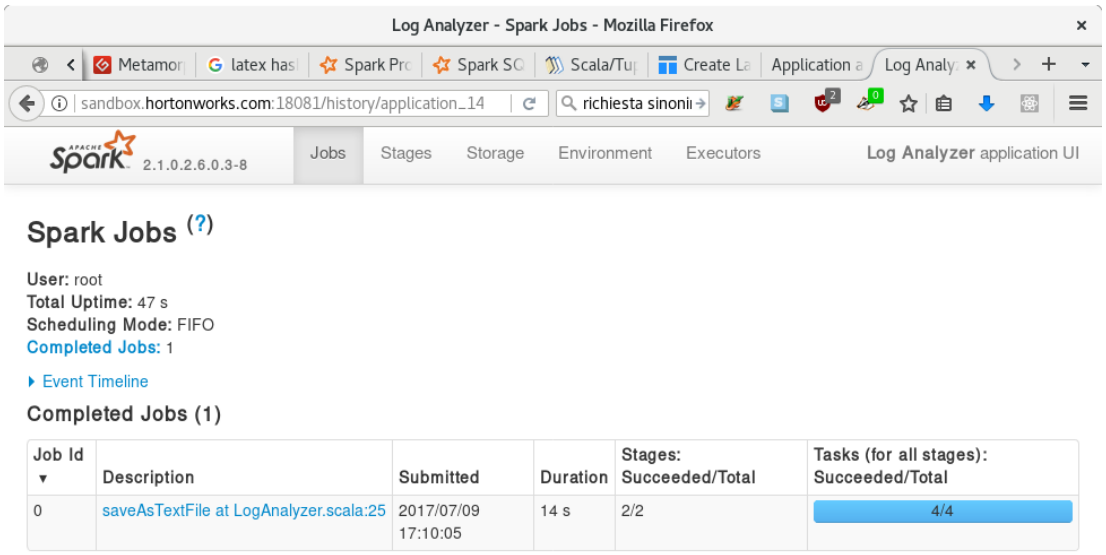

Figura 3.3: Interfaccia web di Spark, tracciamento dell'esecuzione dei job.

viene caricato è un file JSON contenente i dati sulla popolazione degli Stati Uniti nell'anno 2017, strutturato come un array di oggetti con i seguenti campi:

- `age`: Fascia di età a cui l'oggetto si riferisce
- `females`: Numero di donne
- `males`: Numero di uomini
- `total`: Totale di donne e uomini
- `country`: Nazione di riferimento (in questo caso sempre Stati Uniti)
- `year`: Anno di riferimento (2017)

Il seguente codice carica il DataFrame in memoria:

```
scala> val population = spark.read.json("us_population.json")
population: org.apache.spark.sql.DataFrame =
    [age: bigint, country: string ... 4 more fields]
```

La stringa rappresentativa del DataFrame visualizzata in risposta dà qualche indizio sulla struttura rilevata. Si può richiedere al DataFrame di visualizzare la sua intera struttura:

```
scala> population.printSchema()
root
 |-- age: long (nullable = true)
 |-- country: string (nullable = true)
 |-- females: long (nullable = true)
 |-- males: long (nullable = true)
 |-- total: long (nullable = true)
 |-- year: long (nullable = true)
```

Spark ha interpretato correntamente il file, distinguendo tra i campi numerici e stringa. Si può stampare il DataFrame utilizzando il metodo show:

```
scala> population.show(10)
+---+-------------+-------+-------+-------+----+
|age|      country|females|  males|  total|year|
+---+-------------+-------+-------+-------+----+
|  0|United States|1953000|2044000|3997000|2017|
|  1|United States|1950000|2041000|3991000|2017|
|  2|United States|1889000|1977000|3866000|2017|
|  3|United States|1918000|2006000|3925000|2017|
|  4|United States|1946000|2034000|3980000|2017|
|  5|United States|1972000|2060000|4032000|2017|
|  6|United States|1996000|2083000|4079000|2017|
|  7|United States|2018000|2104000|4123000|2017|
|  8|United States|2040000|2125000|4165000|2017|
|  9|United States|2055000|2139000|4194000|2017|
+---+-------------+-------+-------+-------+----+
only showing top 10 rows
```

I DataFrame sono implementati tramite RDD, il cui accesso è reso disponibile agli utenti. Gli RDD dei DataFrame sono composti da oggetti di tipo `spark.sql.Row`, che possono essere indirizzati in modo analogo agli array Scala. Il seguente esempio utilizza l'RDD del DataFrame per ottenere la terza colonna, corrispondente alla popolazione femminile, dei primi 10 elementi del DataFrame.

```
scala> population.rdd.take(10).map(_(2))
res30: Array[Any] = Array(1953000, 1950000, 1889000, 1918000,
    1946000, 1972000, 1996000, 2018000, 2040000, 2055000)
```

Un modo più idiomatico per accedere ai dati del DataFrame è utilizzare i metodi da esso forniti. I DataFrame espongono un DSL ispirato a SQL come metodo di accesso ai dati, che nel seguente esempio viene utilizzato per selezionare la popolazione di adulti maschi di età compresa tra i 20 e i 30 anni.

```
scala> val adult_males = population.select($"males")
    .filter($"age" >= 20 && $"age" <= 30)

adult_males: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
    [males: bigint]

scala> adult_males.show
+-------+
|  males|
+-------+
|2175000|
```

```
|2262000|
|2344000|
|2432000|
|2479000|
|2460000|
|2400000|
|2344000|
|2280000|
|2238000|
|2235000|
+-------+
```

I numeri così ottenuti possono essere sommati per ottenere il numero totale di adulti in questa fascia d'età.

```
scala> adult_males.agg(sum("males")).first.get(0)
res46: Any = 25649000
```

Oltre al DSL fornito dal DataFrame, Spark supporta l'esecuzione diretta di query SQL specificate tramite stringhe. Le query vengono eseguite su degli oggetti definiti view, che vengono mantenuti globalmente in memoria da Spark fino alla fine della sessione[4].

Per creare una view, si può chiamare `createOrReplaceTempView(name: String)` sul DataFrame di interesse. Una volta creata, la view è accessibile nelle query SQL con il nome specificato in `name`.

Le query possono essere eseguite chiamando il metodo `spark.sql(query: String)`, che restituisce il loro risultato sotto forma di DataFrame. Le query possono essere utilizzate per eseguire computazioni di vario tipo, utilizzando varie funzioni di aggregazione fornite dal framework. Il seguente codice esegue e mostra il risultato di una query, che richiede la somme delle popolazioni maschile e femminile di età compresa tra i 40 e i 60 anni.

```
scala> population.createOrReplaceTempView("population")

scala> spark.sql("""
     | SELECT SUM(males), SUM(females) FROM population
     | WHERE age >= 40 AND age <= 60
     | """).show
+----------+------------+
|sum(males)|sum(females)|
+----------+------------+
|  44632000|    45014000|
+----------+------------+
```

[4]Spark supporta anche l'integrazione con Hive, un engine SQL progettato per Hadoop, che può essere utilizzato per creare e mantenere tabelle persistenti.

Modello di esecuzione

Spark utilizza diversi componenti nella sua esecuzione[25]:

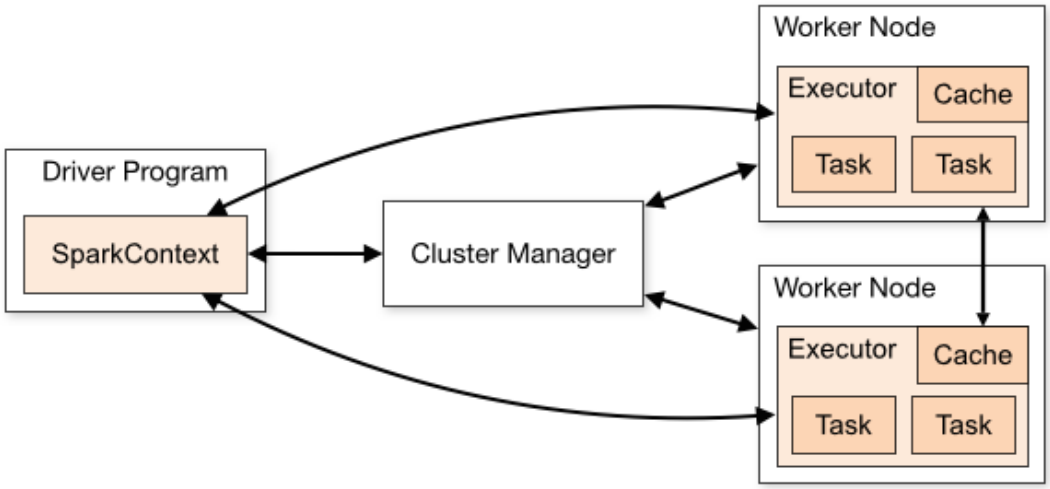

Figura 3.4: Diagramma del modello di esecuzione di Apache Spark[26].

- Il **driver** è il programma principale delle applicazioni, ed è definito dagli utenti tramite le interfacce della libreria client di Spark. Il driver coordina, tramite l'oggetto `SparkContext`, i processi in esecuzione nel cluster, comunicando con il *cluster manager* in utilizzo.
- Il **cluster manager** è l'entità che esegue allocazioni di risorse nel cluster, che può essere YARN, Mesos, o il cluster manager integrato in Spark.
- I **worker node** sono processi avviati nelle macchine coinvolte nella computazione distribuita nel cluster. Ogni worker node gestisce i processi della macchina in cui è eseguito.
- Gli **executor** sono processi allocati all'interno delle macchine worker per eseguire i task assegnati dal driver. Ogni applicazione Spark esegue gli *executor* al proprio avvio, e li termina alla sua conclusione.
- I **task** sono le unità di lavoro eseguite dai singoli worker, che vengono inviati sotto forma di funzioni serializzate. Gli *executor* deserializzano i *task*, per poi eseguirli su partizioni di dataset.

Le computazioni sono definite tramite le funzioni passate come parametro ad azioni e trasformazioni. Spark tiene traccia di queste tramite un grafo, definito **lineage**. Il grafo ha lo scopo di tracciare la provenienza di ogni RDD derivato da trasformazioni, rappresentando gli RDD come nodi e le trasformazioni come archi. Gli archi connettono gli RDD di origine a quelli derivati, tenendo anche traccia delle trasformazioni che li hanno prodotti.

Tramite il grafo di *lineage*, Spark crea un altro grafo aciclico, definito *execution plan*, per determinare come l'esecuzione debba essere organizzata nei nodi. Il criterio utilizzato è di eseguire

quante più computazioni possibili in uno stesso nodo, per ridurre gli spostamenti che richiederebbero banda di rete. Alcune operazioni, come `reduce`, richiedono necessariamente lo spostamento dei dati, visto che le operazioni di aggregazione richiedono che i dati da aggregare si trovino nello stesso nodo. L'operazione di spostare i dati degli RDD per eseguire operazioni di aggregazione è definita *shuffle*, ed è simile all'operazione eseguita da MapReduce per partizionare i risultati dei Mapper.

Per ogni job, Spark esegue una divisione logica sulle operazioni da eseguire, che vengono raggruppate nell'*execution plan* in nodi definiti *execution phases*. Ogni fase di esecuzione raggruppa quante più operazioni possibili, e le fasi sono separate l'una dall'altra solo da operazioni che richiedono l'esecuzione di uno *shuffle*. Il grafo delle fasi di esecuzione di ogni job è visibile nell'interfaccia web di monitoraggio dei job, come mostrato in fig. 3.5 per l'analizzatore di log.

Per fornire fault-tolerance, Spark utilizza il grafo di *lineage*: nell'eventualità in cui un nodo contenente una partizione di un RDD dovesse fallire, Spark può retrocedere agli RDD genitori sul grafo di lineage, fino a trovare degli RDD candidati da cui si può ricavare la partizione non più disponibile. Dal grafo si possono ricavare quali sono le operazioni che hanno prodotto la partizione dell'RDD nella memoria del nodo in stato di fault, che possono quindi essere rischedulate per riottenere la partizione persa. Gli RDD di cui è stato eseguito il caching sono buoni candidati per ricavare la partizione non più disponibile.

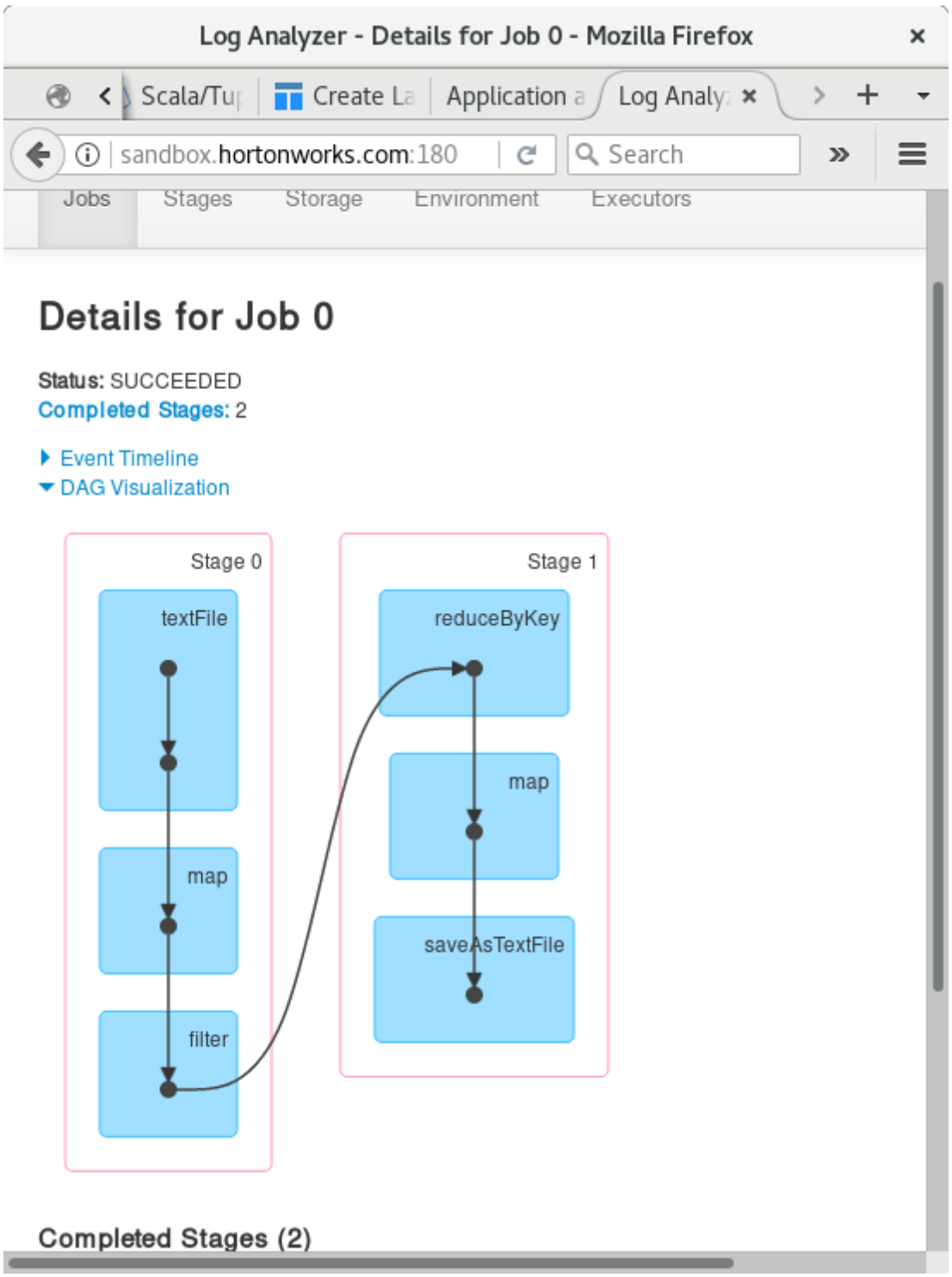

Figura 3.5: Interfaccia web di Spark, visualizzazione DAG delle operazioni.

Parte 4

Stream Processing

Il paradigma di elaborazione su stream permette di eseguire computazioni continuativamente su flussi di dati in arrivo nel corso del tempo, che non hanno limiti definiti di quantità. Nel batch processing, la dimensione dei dati da elaborare è il fattore che definisce il termine della computazione: dato il dataset da elaborare in batch, quando la totalità dei dati contenuti nel dataset sono stati processati l'elaborazione termina. Nello stream processing non si dispone di tale misura, dato che i flussi non definiscono necessariamente il loro termine.

Questa distinzione è importante, perché nel batch processing il termine della computazione corrisponde all'emissione definitiva dell'output dell'elaborazione. Dato che i flussi non hanno necessariamente una conclusione, i tool che gestiscono l'elaborazione su stream devono definire il tempo di emissione dell'output in termini diversi. Generalmente il tempo può essere configurato dall'utente, che deve comunque rispettare dei vincoli dovuti al modello di elaborazione utilizzato.

Ci sono due modelli di elaborazione importanti e riconosciuti nella stream computation, da cui dipende la modalità di emissione dell'output:

- L'approccio **real-time** esegue le computazioni su ogni record continuativamente con il loro arrivo nello stream, e l'emissione di nuovi output può essere fatta corrispondere al termine dell'elaborazione di qualunque record in arrivo.
- Il **microbatching** considera gli stream come una serie di piccoli batch, e i dati di ogni batch vengono processati insieme. L'emissione di output deve corrispondere al termine dell'elaborazione di un microbatch.

Il primo approccio favorisce una latenza minore, mentre il secondo un throughput più alto.

Lo stream processing può essere considerato come una *generalizzazione* del batch processing. Le elaborazioni batch possono essere considerate come elaborazioni su uno stream di record di un dataset, dove l'emissione dell'output viene fatta coincidere con il termine dell'elaborazione del suo ultimo record. I tool che eseguono stream processing possono eseguire computazioni batch seguendo questo modello, fornendo un paradigma di elaborazione unico per *data at rest* e *data in motion*.

In questa sezione si osserva Spark Streaming, un modulo di Apache Spark dedicato allo stream processing orientato al microbatching, che permette di riapplicare l'infrastruttura dedicata al batch processing di Spark su flussi di dati.

Spark Streaming

Spark Streaming è un'estensione di Spark che permette di lavorare con stream, che possono essere ottenuti tramite diverse fonti (socket, sistemi di ingestione come Kafka, HDFS…). I dati ricevuti in uno stream sono raccolti in piccoli batch in un arco di tempo specificato dall'utente, per poi essere rielaborati utilizzando l'engine di esecuzione di Spark.

L'astrazione utilizzata sui flussi è chiamata DStream, che sta per Discretized Stream. I DStream permettono l'esecuzione di azioni e trasformazioni come per gli RDD, aggiungendo alcune operazioni particolari dedicate agli stream. Internamente, i DStream sono rappresentati come sequenze di RDD, ciascuno corrispondente a un particolare microbatch. Gli RDD interni corrispondenti ai microbatch sono resi accessibili all'utente, per eseguire trasformazioni con le loro API specifiche.

I risultati dell'elaborazione possono essere salvati in diversi mezzi, come database, HDFS e filesystem locali, o dashboard per analytics in real-time. Come per gli RDD, il formato degli output può essere specificato, tuttavia la struttura è differente: per ogni microbatch elaborato, Spark Streaming crea una nuova cartella con i risultati. Le cartelle con i risultati vengono nominate con un prefisso e un suffisso specificati dall'utente, e un timestamp in formato epoca UNIX che indica il momento della computazione.

Gli stream possono essere creati con l'analogo dello `SparkContext` per Spark Streaming, ovvero uno `StreamingContext`. Nell'istanziazione di uno StreamingContext, si specifica un oggetto `SparkConf`, dello stesso tipo utilizzato per gli `SparkContext`, e un intervallo di batch, utilizzato per stabilire ogni quanto tempo i dati raccolti debbano essere elaborati.

```
val conf = new SparkConf()
   .setMaster(args(0))
   .setAppName("Some Spark Streaming job")

val ssc = new StreamingContext(conf, Seconds(1))
```

Le fonti da cui è possibile ricavare DStream si dividono in due categorie:

- Le sorgenti base possono essere create direttamente dallo `StreamingContext`, e rappresentano primitive semplici, come socket e file di testo.
- Le sorgenti avanzate richiedono l'uso di librerie apposite, e vengono create tramite classi d'utilità fornite da queste. Le librerie forniscono interfacce di alto livello a protocolli applicativi di diverse applicazioni e servizi, come Kafka e Twitter.

```
val lines = ssc.socketTextStream("localhost", 9999)
val tweets = TwitterUtils.createStream(ssc, None)
```

Listato 4.1: Creazione di DStream a partire da una sorgente base, un socket, e una sorgente avanzata, uno stream di tweet.

In lst. 4.2 viene mostrato un programma Spark Streaming che utilizza un flusso di tweet come input. Nella riga 13 la trasformazione `flatMap` viene utilizzata per mappare ogni tweet al paese da cui è stato inviato, e scartare i tweet di cui non si conosce la provenienza. Il valore è restituito in una coppia composta dal paese e il valore 1. Con `reduceByKey`, si aggregano le coppie per paese e si sommano i valore delle coppie.

`tranform`, rr. 15, è un metodo che riceve in input una funzione, che come parametro ottiene un RDD rappresentativo del microbatch. La funzione viene utilizzata per accedere al metodo `sortBy` dell'RDD, che ne ordina i valori in base a una funzione che restituisce una chiave di comparazione. La chiave utilizzata è il numero di tweet, e l'ordinamento viene specificato come decrescente. In questo modo, il risultato finale è ordinato in base al numero di tweet, creando una classifica dei paesi che hanno inviato più tweet. Infine, si richiede al framework di stampare nello standard output i primi cinque risultati.

Per avviare l'elaborazione in Spark Streaming, sono necessarie due chiamate finali a metodi dello `StreamingContext`: `ssc.start()`, che avvia la computazione, e `ssc.awaitTermination()`, una chiamata a funzione bloccante che fa sì che il programma non termini fino al termine del lavoro. La fine del lavoro può essere segnalata al framework chiamando il metodo `ssc.stop()`. Dato che nel programma questo metodo non viene chiamato, il job resta in esecuzione fino a quando non viene terminato dall'utente.

Il programma stampa un output ogni 60 secondi, nella forma mostrata in lst. 4.3. Per ogni output, il framework stampa automaticamente il momento di elaborazione sotto forma di UNIX epoch.

Operazioni Stateful

In molti contesti, è desiderabile essere in grado di mantenere uno stato relativo al flusso, in modo che i risultati delle elaborazioni possano riguardare periodi di tempo più estesi rispetto all'intervallo di batch. A questo scopo, i DStream forniscono un metodo `updateStateByKey`, che permette di mantenere uno stato arbitrario su una serie di chiavi. Il metodo riflette un meccanismo analogo a `reduceByKey`, dove coppie chiave-valore vengono raggruppate in base alla chiave e un'operazione specificata dall'utente aggrega i valori di ogni gruppo. In `updateStateByKey`, l'operazione coinvolge anche uno stato, che rappresenta il risultato dell'operazione di aggregazione precedente. Lo stato può essere unito al risultato dell'operazione di aggregazione corrente per ottenere un nuovo stato, che verrà a sua volta passato nell'operazione di aggregazione successiva.

```scala
object Main {

  def main(args: Array[String]) {

    val conf = new SparkConf()
      .setMaster(args(0))
      .setAppName("Some Spark Streaming App")

    val ssc = new StreamingContext(conf, Seconds(60))
    val tweets = TwitterUtils.createStream(ssc, None)

    tweets
      .flatMap(t => Try { (t.getPlace.getCountry, 1) }.toOption)
      .reduceByKey(_ + _)
      .transform(_.sortBy(_._2, ascending = false))
      .print(5)

    ssc.start()
    ssc.awaitTermination()

  }
}
```

Listato 4.2: Programma Spark Streaming che calcola, per ogni gruppo di tweet inviato nell'arco di 60 secondi, quanti ne sono stati inviati da ogni paese.

L'operazione di aggregazione viene specificata come una funzione, che prende in input una sequenza di valori del microbatch aggregati in base alla chiave, e lo stato precedente. Riprendendo l'esempio della classifica del numero di tweet per paese, `updateStateByKey` può essere utilizzato per eseguire il conto totale dei tweet a partire dall'esecuzione del job. La funzione di input di `updateStateByKey` è la seguente:

```scala
def updateTweetCount(newVals: Seq[Int], state: Option[Int]): Option[Int] =
  Some(state.getOrElse(0) + newVals.sum)
```

Lo stato preso in input dalla funzione è di tipo `Option`[1], per gestire il caso della computazione iniziale, in cui nessuno stato precedente è stato calcolato. Il metodo `getOrElse(default: T)` restituisce il valore dello stato, se esiste, altrimenti 0. Il risultato viene calcolato sommando lo stato precedente (oppure 0) con la somma dei nuovi tweet.

Il nuovo elenco di operazioni sul DStream dei tweet appare come segue.

```scala
    tweets
```

[1] `Option` è una classe astratta utilizzata in Scala per incapsulare un valore nullabile, e ha due sottoclassi concrete, `Some` e `None`, che rappresentano rispettivamente la presenza e l'assenza di un valore.

```
-------------------------------------------
Time: 1499720340000 ms
-------------------------------------------
(United States,4)
(España,3)
(Türkiye,2)
(Venezuela,1)
(United Arab Emirates,1)
...

-------------------------------------------
Time: 1499720400000 ms
-------------------------------------------
(United States,12)
(Brasil,8)
(United Kingdom,6)
(Australia,1)
(Kosovo,1)
...
```

Listato 4.3: Output del programma in lst. 4.2.

```
    .flatMap(t => Try { (t.getPlace.getCountry, 1) }.toOption)
    .updateStateByKey(updateTweetCount)
    .transform(_.sortBy(_._2, ascending = false))
    .print(5)
```

`updateStateByKey` restituisce un DStream contente le coppie chiave-stato, che può essere utilizzato per eseguire le stesse operazioni precedentemente descritte per ottenere la classifica dei tweet. L'output del programma, è esposto in lst. 4.4.

Operazioni su Finestre

Le operazioni su finestre sono trasformazioni che permettono di eseguire computazioni sugli input ricevuti entro un certo lasso di tempo. Il concetto è mostrato visivamente in fig. 4.1.

I *windowed DStream* vengono creati a partire da un DStream tramite una trasformazione, e hanno due parametri fondamentali:

- la *window duration*, che rappresenta l'arco di tempo antecedente al microbatch di cui si vogliono considerare gli input;
- la *sliding duration*, che indica ogni quanto tempo la finestra si sposta in avanti.

Entrambi questi parametri devono essere multipli dell'intervallo di microbatch. A ogni intervallo di tempo di durata *sliding duration*, i DStream emettono i valori di input ricevuti nel periodo

```
-------------------------------------------
Time: 1499722150000 ms
-------------------------------------------
(United States,34)
(Brasil,17)
(United Kingdom,5)
(Argentina,4)
(Brazil,3)
...

-------------------------------------------
Time: 1499722155000 ms
-------------------------------------------
(United States,35)
(Brasil,18)
(United Kingdom,5)
(Argentina,5)
(Brazil,3)
...
```

Listato 4.4: Output consecutivi del programma di conteggio dei tweet, configurato per l'operazione stateful con un intervallo di batch di cinque secondi.

di tempo che va dall'attimo che antecede il momento attuale di un tempo uguale alla *window duration*, fino al momento attuale. I valori ottenuti possono poi essere rielaborati come un normale DStream. In questo modo, si può ottenere una gestione dei tempi dell'elaborazione più flessibile rispetto alla sola configurazione dell'intervallo di batch, permettendo l'interleaving degli intervalli di elaborazione con quelli di ricezione dell'input.

Il contatore di tweet può essere riadattato facilmente a questo paradigma. Per creare un *windowed DStream* a partire da un DStream, la trasformazione più generale è `window(windowDuration: Duration, slidingDuration: Duration)`. Esistono metodi più specifici per fare uso delle operazioni su finestre, come `reduceByKeyAndWindow`, che esegue un'operazione di `reduce` nell'arco di una finestra. Il metodo `reduceByKeyAndWindow` si adatta bene allo use case del contatore, e può essere utilizzato per fare una classifica dei paesi che hanno inviato più tweet nell'arco dell'ultimo minuto.

```
tweets
  .flatMap(t => Try { (t.getPlace.getCountry, 1) }.toOption)
  .reduceByKeyAndWindow(_ + _, Minutes(1))
  .transform(_.sortBy(_._2, ascending = false))
  .print(5)
```

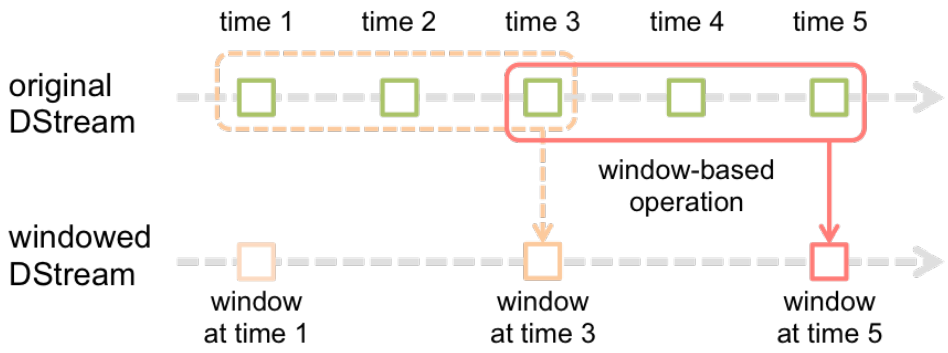

Figura 4.1: Schema rappresentante l'operazione su finestre di Spark Streaming[27]

```
-------------------------------------------
Time: 1499737107000 ms
-------------------------------------------
(United States,21)
(Brasil,9)
(Argentina,2)
(Peru,1)
(Nigeria,1)
...

-------------------------------------------
Time: 1499737108000 ms
-------------------------------------------
(United States,21)
(Brasil,9)
(Argentina,2)
(Japan,2)
(Peru,1)
...

-------------------------------------------
Time: 1499737109000 ms
-------------------------------------------
(United States,22)
(Brasil,9)
(Argentina,2)
(Japan,2)
(Peru,1)
...
```

Listato 4.5: Campione dell'output del programma di classifica dei paesi che hanno inviato più tweet, configurato con una sliding window di un minuto.

Bibliografia

1. Guess: Big Data, Governance, and Hadoop Adoption Rates, http://www.dataversity.net/big-data-governance-and-hadoop-adoption-rates/

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. Data processing architectures - Lambda and Kappa | Ericsson Research Blog, https://www.ericsson.com/research-blog/data-processing-architectures-lambda-and-kappa/

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

6. PoweredBy - Hadoop Wiki, https://wiki.apache.org/hadoop/PoweredBy#Y

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

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

9. HortonWorks Sandbox, https://it.hortonworks.com/products/sandbox/

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

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

12. Docker Container Executor, https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/DockerContainerExecutor.html

13. HortonWorks - Apache Hadoop YARN, https://it.hortonworks.com/apache/yarn/

14. YARN Fair Scheduler, https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

15. 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

16. Spark MLlib, https://spark.apache.org/mllib/

17. Spark GraphX, https://spark.apache.org/graphx/

18. Spark SQL, https://spark.apache.org/sql/

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

20. How Hadoop Map/Reduce works

21. hadoop/HashPartitioner.java at trunk - apache/hadoop, https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapred/lib/HashPartitioner.java

22. Matei, Z.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

23. Project Gutenberg - Metamorphosis by Franz Kafka, https://www.gutenberg.org/ebooks/5200

24. Population.io by World Data Lab, http://population.io/

25., Muhammad, Asif Abbasi: Learning Apache Spark 2. (2017)

26. Cluster Mode Overview - Spark, https://spark.apache.org/docs/latest/cluster-overview.html

27. Spark Streaming - Spark 2.1.1 Documentation, https://spark.apache.org/docs/latest/streaming-programming-guide.html