ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

CORSO DI LAUREA IN INFORMATICA PER IL MANAGEMENT

Modelli di programmazione scalabile per Big Data: analisi comparativa e sperimentale

---

Presentata da
BERTI Matteo

Relatore
Chiar.mo Prof.
ZAVATTARO Gianluigi

II SESSIONE

ANNO ACCADEMICO 2017/2018

Indice

Elenco delle figure

Capitolo 1

MapReduce

1.1 Introduzione a MapReduce

MapReduce è un modello di programmazione ed un framework software per elaborare e generare grandi insiemi di dati, tramite un algoritmo distribuito e parallelo su un cluster. Si specifica una funzione *map* che processa una coppia chiave-valore generando un insieme di coppie chiave-valore intermedie, ed una funzione *reduce* che unisce tutti i valori associati ad ogni chiave intermedia [1].
Il modello è una specializzazione della strategia split-apply-combine per l'analisi dei dati [2]. I contributi chiave di MapReduce non sono in realtà le funzioni map e reduce, ma la scalabilità e la tolleranza ai guasti raggiunta da molte applicazioni ottimizzando il motore di elaborazione.
In quanto tale, un'implementazione single-threaded di MapReduce non sarà più veloce di una tradizionale implementazione non-MapReduce; i veri benefici sono visibili solo con implementazioni multi-thread. L'utilizzo di questo modello migliora considerevolmente quando entrano in gioco l'operazione di shuffle distribuito e la tolleranza ai guasti che caratterizzano il framework MapReduce [3].
Le librerie MapReduce sono state scritte in molti linguaggi di programmazione, con diversi livelli di ottimizzazione. Una popolare implemen-

tazione open-source che supporta shuffles distribuiti è parte di Apache Hadoop. Il nome MapReduce, era originariamente riferito alla tecnologia Google proprietaria, tuttavia dal 2014 il colosso di Mountain View non utilizza più MapReduce come modello principale per processare big data [4].

1.2 Funzionamento

MapReduce è un framework utilizzato per elaborare problemi parallelizzabili utilizzando un elevato numero di computer, o nodi, collettivamente riferiti ad un *cluster*, se tutti i nodi sono sulla stessa rete locale ed utilizzano hardware simile, o un *sistema grid*, se i nodi sono distribuiti a livello geografico ed amministrativo e utilizzano hardware eterogeneo. MapReduce può sfruttare la località dei dati, processando i dati vicino al luogo in cui sono memorizzati, minimizzando così l'overhead di comunicazione.
Un framework MapReduce è tipicamente composto da tre operazioni:

1. *Map*: i worker node applicano la funzione map specificata ai dati locali e scrivono l'output su una memoria locale. Il master node si assicura che solo una copia ridondante di input sia processata.

2. *Shuffle*: i worker node redistribuiscono i dati basandosi sulle chiavi degli elementi prodotti in output dalla funzione map, in modo tale che tutti i dati appartenenti ad una certa chiave siano collocati sullo stesso worker node.

3. *Reduce*: i worker node processano ogni gruppo di dati in output, per chiave, in parallelo.

Anche la mappatura può essere eseguita in parallelo, purchè ogni operazione di mapping sia indipendente dalle altre; nella pratica quindi si è limitati dal numero di sorgenti di dati indipendenti e dal numero di CPU vicine ad ogni sorgente.

Analogamente, un insieme di reducers può eseguire la fase di riduzione in parallelo, purchè tutti gli output della funzione map che condividono la stessa chiave vengano presentati allo stesso reducer nello stesso momento o che la funzione di riduzione sia associativa.
MapReduce può essere applicato a dataset significativamente più grandi di quelli che server "commodity" possano gestire, una grande server farm può utilizzare MapReduce per ordinare un petabyte di dati in poche ore [5]. Il parallelismo offre inoltre l'eventuale possibilità di recuperare i dati da fallimenti parziali dei server o della memoria durante le operazioni di map o reduce.
Più nel dettaglio la computazione parallela e distribuita di questo framework avviene nel seguente modo: il sistema identifica i processori Map ed assegna ad ognuno una chiave in input K_1, ad esempio il nome di un documento, fornendo poi i dati di input associati, come può essere il documento stesso.
Successivamente la funzione Map è eseguita esattamente una volta per ogni chiave K_1, generando un output organizzato su chiavi K_2, nell'esempio tali chiavi potrebbero essere tutte le parole all'interno del documento. Il sistema MapReduce indica poi i processori Reduce, assegna delle chiavi K_2 ad ogni processore, e fornisce tutti i dati generati da Map associati a quella chiave, ad esempio il numero di occorrenze di ogni parola. La funzione Reduce viene eseguita esattamente una volta per ogni chiave K_2 prodotta da Map. Il sistema infine raccoglie tutto l'output di Reduce, e lo ordina per ogni chiave K_2 producendo il risultato finale.
Nella pratica il flusso di esecuzione di MapReduce non è necessariamente sequenziale, può essere interfogliato a condizione che il risultato finale non cambi.
In molte situazioni, i dati di input potrebbero essere già distribuiti su server differenti, in tal caso l'assegnazione delle chiavi in input K_1 potrebbe essere ottimizzata assegnando tali chiavi a server Map che possiedono già i dati localmente. In modo analogo, l'assegnazione delle chiavi K_2 può

essere velocizzata assegnando i processori Reduce che si trovano il più vicino possibile ai dati generati dai processori Map.

1.3 Implementazione

Le implementazioni variano molto in base all'ambiente su cui si vuole eseguire MapReduce, la descrizione seguente si riferisce all'implementazione adottata da Google nel 2003, l'ambiente disponibile consiste in grandi cluster di commodity PCs connessi tra loro tramite switch ethernet. Le

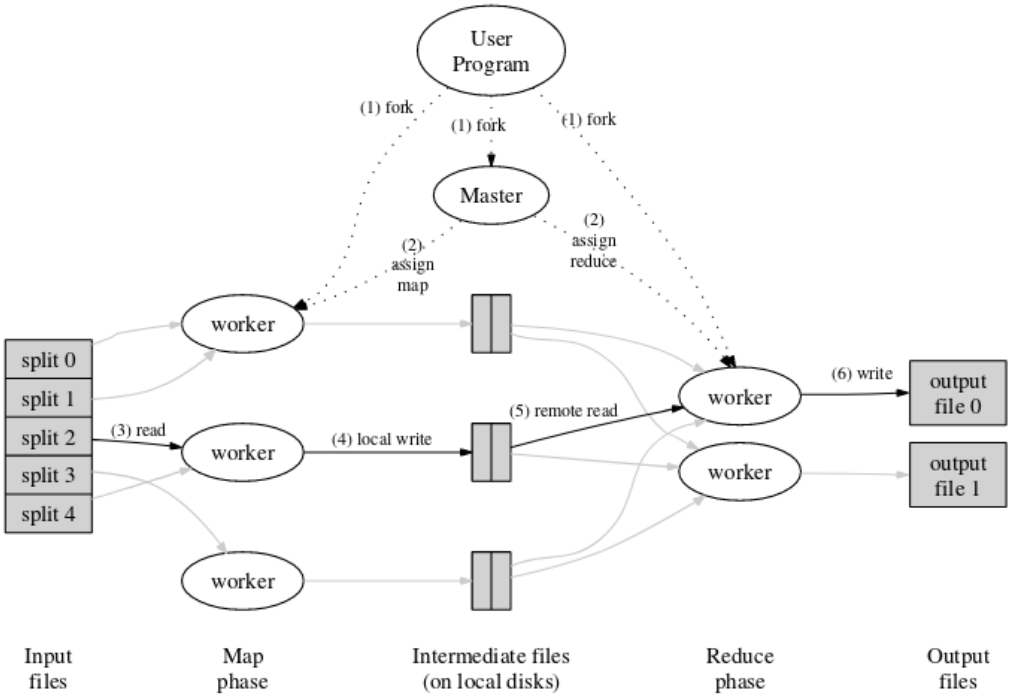

Figura 1.1: Schema di esecuzione di un lavoro MapReduce

elaborazioni Map sono distribuite tra varie macchine, partizionando automaticamente i dati di input in un insieme di M split. Gli split di input possono essere processati in parallelo da macchine differenti. Le elaborazioni Reduce sono distribuite partizionando lo spazio delle chiavi intermedio

in R parti tramite una funzione di partizione come ad esempio *hash(key) mod R*. Il numero di partizioni R e la funzione di partizione sono specificate dall'utente. Quando il programma dell'utente chiama la funzione MapReduce, avviene la seguente sequenza di azioni:

1. La libreria MapReduce nel programma utente divide prima di tutto il file di input in M parti, tipicamente da 16MB a 64MB ognuna, la dimensione può essere definita dall'utente tramite parametri opzionali. Successivamente manda in esecuzione varie copie del programma su un cluster di macchine.

2. Una delle copie del programma viene definita come master. I restanti sono workers che vengono assegnati dal master con lo scopo di elaborare i dati. Vi sono M map task e R reduce task da assegnare, il master assegna ad ogni worker inattivo una task map o reduce.

3. Un worker a cui è assegnata una map task legge il contenuto del corrispondente split di input. Effettua un parse delle coppie chiave-valore dei dati di input e passa ogni coppia alla funzione Map definita dall'utente. Le coppie chiave-valore intermedie prodotte dalla funzione Map sono bufferizzate in memoria.

4. Periodicamente, le coppie bufferizzate vengono scritte sul disco locale, partizionato in R regioni tramite la funzione di partizionamento. Le locazioni di queste coppie bufferizzate sul disco locale sono ritornate al master, che è responsabile di inoltrarle ai reduce workers.

5. Quando un reduce worker è notificato dal master riguardo a queste locazioni, esso utilizza delle RPC per leggere i dati bufferizzati dai dischi locali dei map workers. Quando un reduce worker ha letto tutti i dati intermedi, li ordina per chiavi intermedie in modo che tutte le occorrenze della stessa chiave siano raggruppate insieme. L'ordinamento è necessario in quanto solitamente varie chiavi diverse vengono mappate nella stessa reduce task. Se la quantità di

dati intermedi è troppo grande per stare in memoria, viene utilizzato un ordinamento esterno.

6. I reduce workers iterano tra i dati intermedi ordinati e per ogni chiave univoca intermedia incontrata viene passata la chiave e il corrispettivo insieme di valori alla funzione Reduce definita dall'utente. Il risultato di tale funzione viene aggiunto a un file di output finale per questa partizione di reduce.

7. Quando tutte le task map e reduce sono state completate, il master risveglia il programma utente. A questo punto la chiamata MapReduce ritorna al codice utente.

Dopo che l'intero processo viene completato con successo, l'output dell'esecuzione MapReduce è disponibile in R file di output (uno per ogni reduce task, con il nome di ogni file specificato dall'utente). Spesso, gli utenti non hanno bisogno di combinare questi R file di output in un unico file, li passano nuovamente come input ad un'altra chiamata MapReduce, o li utilizzano per un'altra applicazione distribuita in grado di gestire un input partizionato su più file [1].

1.4 Esempio

Si prenda ad esempio il problema di contare il numero di occorrenze di ogni parola in una vasta collezione di documenti. I codici delle funzioni Map e Reduce potrebbero seguire la logica dei seguenti pseudocodici [1]:

Algorithm 1 : **map** (String **key**, String **value**)

key: nome del documento
value: testo del documento
for each Word w **in** value **do**
  $EmitIntermediate(w, "1")$

Algorithm 2 : **reduce** (String **key**, Iterator **values**)

key: una parola
values: una lista di conteggi
int $result \Leftarrow 0$
for each Value v **in** values **do**
  $result \mathrel{+}= ParseInt(v)$
$Emit(AsString(result))$

Supponiamo di avere tre task map e due task reduce, utilizzando un semplice testo: "*a long time ago in a galaxy far, far away*", il risultato è il seguente:

Map Worker 1

a	→	<a, 1>
long	→	<long, 1>
time	→	<time, 1>
ago	→	<ago, 1>

Map Worker 2

in	→	<in, 1>
a	→	<a, 1>
galaxy	→	<galaxy, 1>
far	→	<far, 1>

Map Worker 3

far	→	<far, 1>
away	→	<away, 1>

Reduce Worker 1

<galaxy, 1> | <in, 1> | <far, 1, 1> | <ago, 1>
Riduzione: <galaxy, 1>; <in, 1>; <far, 2>; <ago, 1>

Reduce Worker 2

<a, 1, 1> | <long, 1> | <time, 1> | <away, 1>
Riduzione: <a, 2>; <long, 1>; <time, 1>; <away, 1>

Alla fine dell'esecuzione di MapReduce si ottiene il numero di occorrenza di ogni parola all'interno della collezione di documenti, nel nostro caso di una semplice frase.

Capitolo 2

Big Data Frameworks

2.1 Introduzione

Secondo una recente definizione, i Big Data rappresentano le risorse informative caratterizzate da un volume, una velocità e una varietà così elevati da richiedere tecnologie e metodi analitici specifici per la loro trasformazione in valore [6]. Le caratteristiche di questi dati sono [7]:

- *Volume*: la quantità di dati generati e memorizzati. La dimensione dei dati determina il valore e la potenziale intuizione.
- *Varietà*: il tipo e la natura dei dati. Questo aiuta gli analisti a utilizzare efficacemente l'intuizione risultante.
- *Velocità*: in questo contesto, la velocità con cui i dati vengono generati ed elaborati.
- *Veracità*: la qualità dei dati acquisiti può variare notevolmente, influenzando l'accuratezza dell'analisi.

Un framework è un'astrazione in cui il software che fornisce certe funzionalità generiche può essere modificato selettivamente da un ulteriore codice scritto dall'utente, fornendo così un software specifico per l'applicazione.

I framework e i motori di elaborazione sono responsabili del calcolo dei dati in un data system. Si può definire "motore" l'effettivo componente responsabile dell'elaborazione dei dati e "framework" l'insieme dei componenti progettati per fare lo stesso, ad esempio Apache Hadoop può essere considerato un framework di elaborazione con MapReduce come motore principale. Motori e framework possono spesso essere scambiati o combinati, un esempio è Apache Spark, un altro framework, che può essere agganciato ad Hadoop per sostituire MapReduce. Questa interoperabilità tra componenti è una ragione per la quale i big data system hanno una grande flessibilità.
Mentre i sistemi che gestiscono questa fase del ciclo di vita dei dati, possono essere complessi, gli obiettivi a livello generale sono molto simili: elaborare i dati in modo da aumentarne la comprensione, far emergere pattern presenti e ottenere informazioni su complesse interazioni tra essi. Spesso questi componenti vengono raggruppati in base allo stato dei dati che andranno ad elaborare. Alcuni sistemi gestiscono i dati in gruppi, altri li elaborano in un flusso continuo mentre attraversano il sistema e altri ancora combinano queste due tecniche in una gestione ibrida.

2.2 Sistemi di elaborazione batch

L'elaborazione batch consiste in uno o più calcolatori che lavorano attraverso una porzione di lavori separati su un set di dati definito, senza interattività, ovvero senza alcun intervento manuale. Questo genere di elaborazione ha molti vantaggi, è possibile ad esempio spostare il tempo di elaborazione del lavoro a quando le risorse di calcolo sono meno occupate, o evitare che le risorse rimangano inutilizzate tramite un intervento manuale e una supervisione costante. Inoltre mantenendo un alto tasso generale di utilizzo si ammortizza il calcolatore, soprattutto se costoso. Infine invece di eseguire un programma più volte elaborando una transazione alla volta, i processi batch eseguono il programma una sola volta raggruppan-

do molte transazioni, riducendo così il sovraccarico del sistema.
Questi sistemi di elaborazione hanno anche vari svantaggi, uno dei quali è l'impossibilità per l'utente di terminare un processo durante l'esecuzione senza perdere i progressi raggiunti, è quindi necessario attendere il completamento dell'esecuzione. L'architettura dell'elaborazione batch ha vari componenti logici [8]:

- *Archivio dati*: tipicamente un archivio di file distribuito che può fungere da repository per elevati volumi di file di grandi dimensioni in vari formati.

- *Elaborazione in batch*: gestendo moli di dati di grandi dimensioni significa che le soluzioni devono elaborare i file utilizzando lavori in batch di lunga durata per filtrare, aggregare e preparare i dati per l'analisi. Di solito queste mansioni comportano la lettura dei file di input, l'elaborazione e la scrittura dell'output in nuovi file.

- *Archivio dati analitici*: molte soluzioni di big data sono progettate per preparare dati per l'analisi e quindi è necessario avere dati elaborati in modo strutturato per facilitarne l'interrogazione tramite strumenti analitici.

- *Analisi e reporting*: l'obiettivo della maggior parte delle soluzioni di big data è quello di fornire approfondimenti sui dati attraverso analisi e reporting.

- *Orchestrazione*: con l'elaborazione batch è in genere necessaria un'orchestrazione per migrare o copiare i dati in memoria, nella computazione, nell'archivio dati analitici e nei livelli di reporting.

Il batch processing è particolarmente adatto per i calcoli in cui è richiesto l'accesso a un set completo di record, ad esempio, quando si calcolano totali o medie i set di dati devono essere trattati in modo olistico invece che come raccolta di record individuali.

Le elaborazioni che richiedono volumi di dati estremamente grandi sono spesso gestite meglio attraverso operazioni di batch, inoltre se i dataset vengono processati direttamente dalla memoria permanente o caricati in memoria, i sistemi batch sono progettati tenendo conto di dover gestire grandi quantità di dati e dispongono delle risorse per gestirli. Tuttavia per elaborare questi dati è spesso necessario un tempo di calcolo lungo, ciò rende il batch processing non appropriato in situazioni in cui il tempo di elaborazione è particolarmente significativo.

2.2.1 Apache Hadoop

La libreria del software Apache Hadoop è un framework che consente l'elaborazione distribuita di grandi set di dati su un cluster di computer utilizzando semplici modelli di programmazione [9]. Originariamente concepito per cluster basati su commodity hardware, tuttora in uso, ha anche trovato impiego su cluster di hardware di fascia alta. È progettato per passare da singoli server a migliaia di macchine, in cui ognuna delle quali offre calcolo e archiviazione locali. Anziché affidarsi all'hardware per garantire un'elevata disponibilità, la libreria stessa è progettata per rilevare e gestire i guasti a livello di applicazione, offrendo così un servizio altamente disponibile su un cluster di computer, ognuno dei quali potenzialmente esposto a guasti. Apache Hadoop è scritto principalmente nel linguaggio di programmazione Java, con un codice nativo in C e utility a linea di comando scritte come shell scripts. Il framework include vari moduli [9]:

- *Hadoop Common*: le utility comuni che supportano gli altri moduli Hadoop.
- *Hadoop Distributed File System* (HDFS): un file system distribuito che fornisce accesso ad alta velocità ai dati delle applicazioni.
- *Hadoop YARN*: un framework per la pianificazione del lavoro e la gestione delle risorse del cluster.

- *Hadoop MapReduce*: un motore basato su YARN per l'elaborazione parallela di set di dati di grandi dimensioni.

Hadoop Core

Apache Hadoop è costituito dal pacchetto Hadoop Common, che fornisce astrazioni a livello di file system e sistema operativo. Questo pacchetto contiene i file e gli script Java ARchive (JAR) necessari per avviare Hadoop.
Per una pianificazione efficace del lavoro, ogni file system compatibile con Hadoop dovrebbe fornire location awareness, ovvero il nome dello switch di rete in cui si trova un worker node. Le applicazioni Hadoop possono utilizzare queste informazioni per eseguire codice sul nodo in cui si trovano i dati e, in mancanza, sullo stesso switch per ridurre il traffico sulla dorsale. HDFS utilizza questo metodo durante la replica dei dati per la ridondanza su più rack. Questo approccio riduce l'impatto di un'interruzione dell'alimentazione del rack o di un guasto dello switch; se si verifica uno di questi guasti hardware, i dati rimarranno comunque disponibili [10].
Un piccolo cluster Hadoop include un singolo master e più nodi di lavoro. Il nodo master è costituito da un Job Tracker, Task Tracker, NameNode e DataNode. Un nodo slave o worker node agisce sia come DataNode che come TaskTracker. Sebbene sia possibile avere nodi di lavoro per i soli dati o per la sola elaborazione, questi sono normalmente usati solo in applicazioni non standard.
Hadoop richiede un Java Runtime Environment (JRE) di versione 1.6 o successive. Gli script di avvio e arresto standard richiedono che il sistema di Secure Shell (SSH) sia impostato tra i nodi del cluster.
In un cluster più grande, i nodi HDFS sono gestiti attraverso un server NameNode dedicato che ospita l'indice del file system e un NameNode secondario che può generare snapshot delle strutture di memoria del Na-

meNode, impedendo in tal modo il danneggiamento del file system e la perdita di dati. Allo stesso modo, un server JobTracker autonomo può gestire la pianificazione dei lavori tra i nodi. Quando Hadoop MapReduce viene utilizzato con un file system alternativo, l'architettura NameNode, NameNode secondario e DataNode di HDFS vengono sostituiti dagli equivalenti specifici del file system.

Hadoop Distributed File System

HDFS è un file system distribuito, scalabile e portatile scritto in Java per Apache Hadoop. Fornisce comandi di shell e funzioni API (Application Programming Interface) Java simili a quelli di altri file system. Un cluster Hadoop ha nominalmente un singolo NameNode più un cluster di DataNode, anche se sono disponibili opzioni di ridondanza per il namenode a causa della sua criticità. Ciascun DataNode fornisce blocchi di dati sulla rete usando un protocollo a blocchi specifico per HDFS. Il file system utilizza socket TCP/IP per la comunicazione e i client utilizzano chiamate remote (RPC) per comunicare tra loro.
HDFS memorizza file di grandi dimensioni, solitamente nell'ordine di gigabyte e terabyte, su più computer [11]. Raggiunge l'affidabilità replicando i dati su più host e quindi, in teoria, non è richiesta una memorizzazione tramite RAID sugli host. Tuttavia per aumentare le prestazioni input-output alcune configurazioni RAID sono comunque utili. Con il valore di replica predefinito, i dati vengono archiviati su tre nodi: due sullo stesso rack e uno su un rack diverso. I datanode possono comunicare tra loro per riequilibrare i dati, spostare le copie e mantenerne elevata la replica.
Il file system HDFS include un cosiddetto NameNode secondario, un termine che potrebbe essere erroneamente interpretato come un NameNode di backup quando il NameNode primario va offline. Il NameNode secondario si collega regolarmente al NameNode primario e crea snapshots delle informazioni della directory di quest'ultimo, che il sistema salva in directory locali o remote. Queste immagini di checkpoint possono essere

utilizzate per riavviare un NameNode primario in caso di fallimento senza dover rieseguire l'intero journal delle azioni del file system, e successivamente modificare il log per creare una struttura di directory aggiornata.
Poiché il NameNode è un punto unico per la memorizzazione e la gestione dei metadati, può diventare un collo di bottiglia per il supporto di un elevato numero di file, in particolare molti file di piccole dimensioni. Un vantaggio dell'utilizzo di HDFS è la conoscenza dei dati tra il Job Tracker e il Task Tracker. Il Job Tracker effettua il mapping o il reducing dei lavori con la consapevolezza della posizione dei dati. Ad esempio: se il nodo A contiene dati (a, b, c) e il nodo X contiene dati (x, y, z), il job tracker pianifica il nodo A per eseguire il mapping o la riduzione delle attività su (a, b, c) e nodo X dovrebbe essere programmato per eseguire il mapping o la riduzione delle attività su (x, y, z). Ciò riduce la quantità di traffico che attraversa la rete e impedisce trasferimenti di dati non necessari. Quando Hadoop viene utilizzato con altri file system, questo vantaggio non è sempre disponibile. Ciò può avere un impatto significativo sui tempi di completamento del lavoro, come dimostrato con i lavori ad alta intensità di dati [12].
Hadoop funziona direttamente con qualsiasi file system distribuito che può essere montato dal sistema operativo sottostante. Per ridurre il traffico di rete, Hadoop deve sapere quali server sono più vicini ai dati, informazioni che possono essere fornite dai bridge di file system specifici di Hadoop.
Nel maggio 2011, l'elenco dei file system supportati in bundle con Apache Hadoop era:

- *HDFS*: eseguito sui file system dei sistemi operativi sottostanti e progettato per scalare fino a decine di petabyte di storage.
- *FTP*: i dati sono archiviati su server FTP accessibili a distanza.
- *Amazon S3* (Simple Storage Service): si rivolge a cluster ospitati sull'infrastruttura server-on-demand di Amazon Elastic Compute Cloud.

Non c'è consapevolezza del rack in questo file system, poiché è tutto remoto.

- *Windows Azure Storage Blob* (WASB): questa è un'estensione di HDFS che consente alle distribuzioni di Hadoop di accedere ai dati negli archivi BLOB di Azure senza spostare permanentemente i dati nel cluster.

Yet Another Resource Negotiator

Apache YARN è una tecnologia per la gestione delle risorse distribuita su un cluster Hadoop introdotta dalla versione 2 del framework. L'idea fondamentale di YARN è quella di suddividere le funzionalità di gestione delle risorse, di pianificazione e monitoraggio del lavoro in processi daemon separati, estendendo la potenza di Hadoop ad altre tecnologie in evoluzione, in modo che possano sfruttare i vantaggi dell'HDFS e di cluster economici, formati ad esempio da commodity hardware.
L'architettura basata su YARN fornisce una piattaforma di elaborazione dati di uso generale, non solo limitata a MapReduce, permette infatti di eseguire diversi framework sullo stesso hardware in cui viene distribuito Hadoop. L'architettura è formata principalmente da un daemon master noto come Resource Manager, un daemon slave per ogni nodo slave chiamato Node Manager e un Application Master per ogni applicazione [13].
Il Resource Manager (RM) è il principale daemon di YARN, gestisce l'assegnamento globale delle risorse sulla CPU e sulla memoria, tra tutte le applicazioni e coordina le risorse di sistema tra le applicazioni concorrenti.
Il Resource Manager ha due componenti principali:

- *Scheduler*: responsabile per l'allocazione delle risorse all'applicazione in esecuzione. Questo elemento è un semplice pianificatore, pertanto non esegue alcun monitoraggio ne tracciamento nell'applicazione e, inoltre, non garantisce il riavvio delle attività non riuscite in caso di errore dell'applicazione o di errori hardware.

- *Application Manager*: responsabile dell'esecuzione dell'Application Master nel cluster, ovvero si occupa dell'avvio dei master, del loro monitoraggio e del riavvio su nodi diversi in caso essi producano errori.

Il Node Manager è il daemon slave di YARN, responsabile per i container che monitorano l'utilizzo delle risorse degli slave e riportano i dati al ResourceManager. Tiene anche traccia dello stato del nodo su cui è in esecuzione. Consente inoltre di collegare servizi ausiliari di lunga durata al Node Manager; questi sono servizi specifici dell'applicazione, specificati come parte delle configurazioni e caricati dal Node Manager durante l'avvio. Per le applicazioni MapReduce su YARN, un shuffle è un tipico servizio ausiliario caricato dai Node Manager.
L'Application Master negozia le risorse dal Resource Manager e lavora con il Node Manager, in generale gestisce il ciclo di vita dell'applicazione. Questo elemento acquisisce i containers dallo Scheduler del Resource Manager prima di contattare i Node Manager corrispondenti per avviare le singole task dell'applicazione.

Motore MapReduce

Sopra al file system vi è il motore MapReduce, che consiste in un JobTracker, a cui le applicazioni client inviano i lavori da eseguire. Il motore segue l'algoritmo map, shuffle, reduce utilizzando coppie chiave-valore. In modo astratto il flusso è il seguente: si legge dal file system HDFS il dataset, diviso in parti e distribuito tra i nodi disponibili. Successivamente si effettua la computazione del sottoinsieme di dati su ogni nodo, scrivendo i risultati intermedi nuovamente nel HDFS. Si redistribuiscono i risultati intermedi in gruppi divisi per chiave, e si "riduce" il valore di ogni chiave sommando e combinando i risultati calcolati dai singoli nodi. Si riscrivono infine i risultati finali nell'HDFS.
Poiché questa metodologia sfrutta pesantemente la memorizzazione permanente, molte letture e scritture ad ogni job, tendono a rendere il proces-

so piuttosto lento. Tuttavia lo spazio su disco è in genere una delle risorse più abbondanti nei server, ciò significa che MapReduce può gestire enormi moli di dati. MapReduce ha un incredibile potenziale di scalabilità ed è stato utilizzato in produzione su decine di migliaia di nodi. Come target per lo sviluppo, MapReduce è noto per avere una curva di apprendimento piuttosto ripida.
Nella pratica il JobTracker sottopone il lavoro ai nodi TaskTracker disponibili nel cluster, cercando di mantenerlo il più vicino possibile ai dati. Con un file system rack-aware, il JobTracker conosce quale nodo contiene i dati e quali altre macchine si trovano nelle vicinanze. Se il lavoro non può essere ospitato sul nodo in cui si trovano effettivamente i dati, viene data priorità ai nodi nello stesso rack, riducendo il traffico sulla rete dorsale. Se un TaskTracker fallisce o va in timeout, la parte di lavoro a lui competente viene riprogrammata. Il TaskTracker genera un processo JVM separato su ciascun nodo, per evitare di terminare totalmente se il lavoro in esecuzione genera un crash sulla JVM. Un heartbeat viene inviato dal TaskTracker al JobTracker ogni pochi minuti per verificarne lo stato. Lo stato e le informazioni di Job Tracker e TaskTracker sono resi disponibili da Jetty e possono essere visualizzati da un browser web.
Vi sono varie limitazioni a questo approccio ad esempio se un TaskTracker è molto lento, può ritardare l'intero lavoro di MapReduce, specialmente verso la fine, quando è necessario attendere che l'attività più lenta termini. Con l'esecuzione speculativa abilitata, tuttavia, una singola attività può essere eseguita su più nodi slave.
Per quanto riguarda la programmazione dei lavori, di default, Hadoop utilizza la pianificazione FIFO e facoltativamente 5 priorità di pianificazione per programmare i lavori da una work queue.

Struttura

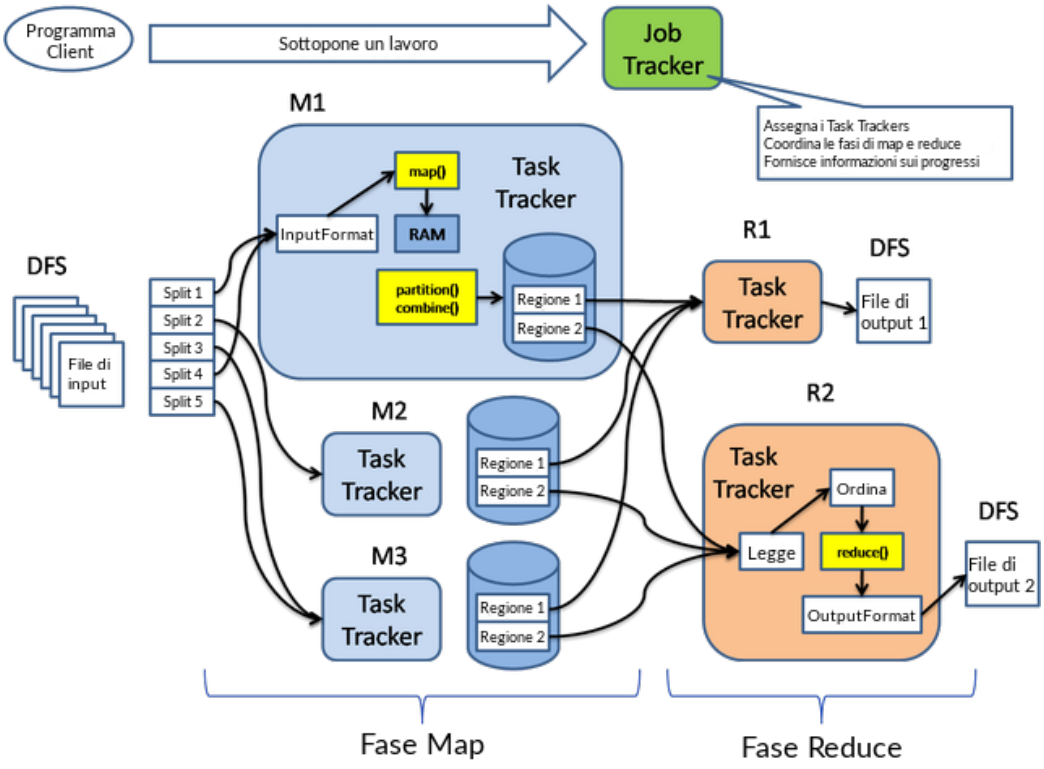

Figura 2.1: Schema di esecuzione di un lavoro Hadoop

Il flusso di esecuzione è chiaro nello schema, per prima cosa il lavoro viene sottoposto al Job Tracker, il quale assegna i Task Trackers e coordina le fasi di map e reduce. Nella fase di mapping ogni Task Tracker, all'interno di un DataNode analizza uno split di input, preso dal DFS, applica la funzione di mappatura e salva i risultati in memoria, nella propria macchina, suddividendoli in tante regioni quanti sono i reducer nodes. A questo punto il Task Tracker nel reducer DataNode legge la propria porzione di memoria nei mapper, ordina per chiave ed applica la funzione di riduzione, producendo un output. Questo verrà salvato su file nel DFS o in un database.

Casi d'uso

Un esempio è il caso di Hotels.com, un sito per il confronto di pernottamenti in hotel, appartenente al gruppo Expedia. Il problema principale era quello di convertire enormi moli di dati che entravano nel sistema ogni secondo in informazioni utili. Hotels.com ha quindi scelto di integrare Hadoop, in aggiunta all'utilizzo del cloud per supportare semplici funzioni come la ricerca automatica nel momento in cui l'utente digita nella casella di ricerca. Durante il periodo di alta stagione le ricerche si sono intensificate, creando la necessità di gestire una quantità di dati crescente senza rallentare le prestazioni del sito. Hotels.com ha iniziato allora ad utilizzare database NoSQL e Apache Cassandra. Integrando poi Hadoop, sono state semplificate le funzionalità di utilizzo di Cassandra, aumentandone l'efficienza e offrendo un servizio migliore.
Un altro caso importante è British Airways, il cui scopo era raccogliere dati sui clienti e analizzare il loro comportamento. Ha quindi introdotto il programma "Know Me", un piano unico inteso a comprendere meglio il cliente rispetto ai suoi concorrenti. Attraverso questo programma, sono stati contrassegnati i clienti che gli sono rimasti fedeli e li hanno premiati con offerte o benefici come la proposta di un posticipo del volo in caso il cliente sia bloccato in autostrada, ad esempio. British Airways è riuscita a raggiungere questi risultati con l'aiuto di Hadoop, eccellente per l'elaborazione di moli di dati così grandi.
Un ultimo esempio viene fornito da Yahoo, la cui necessità era quella di risparmiare milioni di dollari in costi hardware. Si dice che ogni giorno oltre 150 terabyte di dati macchina passino attraverso i loro data warehouse. Yahoo utilizza Hunk, uno strumento Hadoop di Splunk. Questa società ha infatti iniziato a dedicarsi ad Hadoop molto prima della maggior parte delle aziende. L'intenzione precedente era di accelerare il ritmo di indicizzazione delle pagine web tramite web crawler. Ci sono circa 4500 nodi nel più grande cluster Hadoop di proprietà di Yahoo. E gestisce il framework su oltre 100.000 CPU in oltre 40.000 server. Hadoop svolge un ruolo impor-

tante nel rilevamento dei messaggi di spam e nel loro blocco, fornendo il meglio ai clienti a seconda dei loro interessi e hobby. Yahoo invia pacchetti a valore aggiunto ai propri iscritti combinando analisi automatizzate e veri editori per definire gli interessi dei clienti. Yahoo utilizza Hadoop in collaborazione con altre tecnologie per offrire i migliori risultati anche agli inserzionisti e ai marketer [14].

2.2.2 Caratteristiche framework

Linguaggio implementazione	Java
Coordinamento del cluster	YARN
Archiviazione	HDFS
Modello di elaborazione	Batch
Computazione e trasformazione	Map, Reduce, Shuffle
Linguaggi utilizzabili	Qualsiasi
Operazioni con stato	Si
Supporto modalità interattiva	No
Facilità di sviluppo	Difficile
Modello di affidabilità	Elaborazione esattamente una volta
Performance di elaborazione	Lento
Utilizzo di memoria	Basso

2.3 Sistemi di elaborazione stream

I sistemi di stream processing elaborano i dati nel momento in cui entrano nel sistema. Ciò richiede un diverso modello di elaborazione rispetto al paradigma batch: invece di definire operazioni da applicare ad un intero dataset, i processori stream definiscono operazioni che verranno applicate ad ogni singolo dato, appena entra nel sistema. Nello stream processing i dataset vengono considerati illimitati, ciò ha varie implicazioni:

- Il dataset totale è definito come la quantità di dati che è entrata nel sistema in un dato momento nel tempo.
- Il dataset di lavoro è limitato ad un singolo elemento alla volta.
- L'elaborazione è basata su eventi e non termina finché non è esplicitamente fermata, i risultati sono immediatamente disponibili e verranno continuamente aggiornati quando subentrano nuovi dati.

I sistemi di stream processing possono gestire una quantità pressoché illimitata di dati, ma processano solo un elemento, real stream processing, o pochi elementi, micro-batch processing, alla volta, mantenendo uno stato minimo tra i record. L'architettura dell'elaborazione real time ha vari componenti logici [15]:

1. *Ingestione di messaggi in tempo reale*: l'architettura deve includere un modo per acquisire e archiviare i messaggi in tempo reale, consumati da un utente che elabora il flusso. In casi semplici, questo servizio potrebbe essere implementato come un semplice archivio di dati in cui i nuovi messaggi vengono depositati in una cartella. Ma spesso la soluzione richiede un broker di messaggi, che funge da buffer. Tale elemento dovrebbe supportare l'elaborazione scalabile e una consegna affidabile.
2. *Elaborazione del flusso*: dopo aver acquisito i messaggi in tempo reale, la soluzione deve elaborarli filtrando, aggregando e preparando i dati per l'analisi.

3. *Archivio dati analitici*: molte soluzioni di big data sono progettate per preparare i dati per l'analisi e quindi servire dati elaborati in un formato strutturato che possa essere interrogato utilizzando strumenti analitici.

4. *Analisi e reporting*: l'obiettivo anche qui, come per l'elaborazione in batch è quello di fornire approfondimenti sui dati attraverso analisi e reporting.

Mentre molti sistemi forniscono metodi per mantenere uno stato più dettagliato, lo stream processing è altamente ottimizzato per un'elaborazione più funzionale con pochi effetti collaterali. Le operazioni funzionali si concentrano su passi discreti che limitano lo stato e gli effetti collaterali: effettuare la stessa operazione sullo stesso insieme di dati produrrà lo stesso output indipendentemente da altri fattori. Questo tipo di elaborazione si adatta bene ai flussi, in quanto lo stato tra gli elementi è spesso una combinazione difficile e limitata. Quindi mentre alcuni tipi di gestione dello stato sono possibili, questi frameworks risultano più semplici ed efficienti senza.

2.3.1 Apache Storm

Apache Storm è un framework per lo stream processing che fornisce una latenza estremamente bassa, ciò lo rende l'opzione migliore per carichi di lavoro che richiedono un'elaborazione quasi in tempo reale.
Il progetto, scritto principalmente in Java e in Clojure, è stato reso open source dopo l'acquisto da parte di Twitter. Un'applicazione Storm è progettata come una topologia modellata tramite un grafo aciclico orientato (DAG) con "Bolts" e "Spouts" che agiscono come vertici del grafo. Gli archi sono denominati flussi e indirizzano i dati da un nodo ad un altro. Nel complesso, la topologia agisce come una pipeline per la trasformazione dei dati. A livello superficiale, la struttura generale della topologia è simi-

le a un lavoro MapReduce, con la differenza che i dati vengono elaborati in tempo reale anziché in singoli lotti [16].

Panoramica

Il cluster Apache Storm è composto principalmente da due componenti:

- *Nodi*: sono suddivisi in due tipi, Master Node e Worker Node. Il primo esegue un daemon chiamato Nimbus che assegna le task alle macchine e ne monitora le prestazioni. Il secondo esegue un daemon chiamato Supervisor che assegna le attività ai worker node e le gestisce secondo le necessità. Poiché Storm non è in grado nativamente di monitorare lo stato e la salute del cluster, integra Apache ZooKeeper per risolvere il problema, connettendo Nimbus ai Supervisor.
- *Componenti*: vi sono tre componenti importanti, gli Streams, ovvero flussi di dati che entrano in modo continuo nel sistema, gli Spouts, nonché le fonti dei flussi di dati ai margini della topologia che producono gli elementi su cui operare, questi possono essere API, code o altro. Infine vi sono i Bolts, fasi di elaborazione che applicano una qualche operazione sui flussi, generando il risultato nuovamente sotto forma di flusso.

I Bolt sono collegati a ciascun Spout e insieme organizzano tutta l'elaborazione necessaria. Alla fine della topologia, l'output del Bolt finale può essere utilizzato come input per un sistema collegato.
L'idea dietro a Storm è definire piccole operazioni discrete utilizzando i componenti sopraccitati per comporre una topologia. Di default, Storm offre la garanzia di almeno un'elaborazione, quindi ogni messaggio verrà elaborato almeno una volta, tuttavia in alcuni casi di fallimento potrebbero esserci dei duplicati. Questo framework tuttavia non garantisce l'ordine di elaborazione dei messaggi.
Per ottenere un'unica elaborazione, che conservi lo stato, è disponibile un'astrazione chiamata Trident. Più precisamente, Storm senza Trident

è spesso indicato come Core Storm, Trident altera in modo significativo le dinamiche di elaborazione di Storm, aumentando la latenza, aggiungendo lo stato all'elaborazione e implementando un modello di micro-batching invece di un sistema di streaming puro item-by-item. Gli utenti Storm raccomandano in genere l'uso di Core Storm quando possibile, per evitare tali penalità. Alla luce di questo, la garanzia di Trident di elaborare gli elementi esattamente una volta è utile nei casi in cui il sistema non è in grado di gestire in modo intelligente i messaggi duplicati. Trident è anche l'unica opzione disponibile in Storm quando è necessario mantenere lo stato tra gli elementi. Quest'astrazione fornisce flessibilità a Storm, anche se non sfrutta i punti di forza naturali del framework. Le topologie Trident sono composte da:

- *Stream batches*: ovvero micro-batch di dati stream che vengono suddivisi in parti per fornire semantica di elaborazione batch.

- *Operazioni*: si tratta di procedure batch che possono essere eseguite sui dati.

Storm è probabilmente la migliore soluzione attualmente disponibile per l'elaborazione quasi in tempo reale. È in grado di gestire i dati con una latenza estremamente bassa per i carichi di lavoro che vanno elaborati con un ritardo minimo. In termini di interoperabilità, Storm può integrarsi con il negoziatore di risorse YARN di Hadoop, facilitando l'accesso a una distribuzione Hadoop esistente. A differenza della maggior parte dei framework di elaborazione, Storm ha un supporto molto ampio di linguaggi di programmazione, offrendo agli utenti varie opzioni per definire le topologie.

Struttura

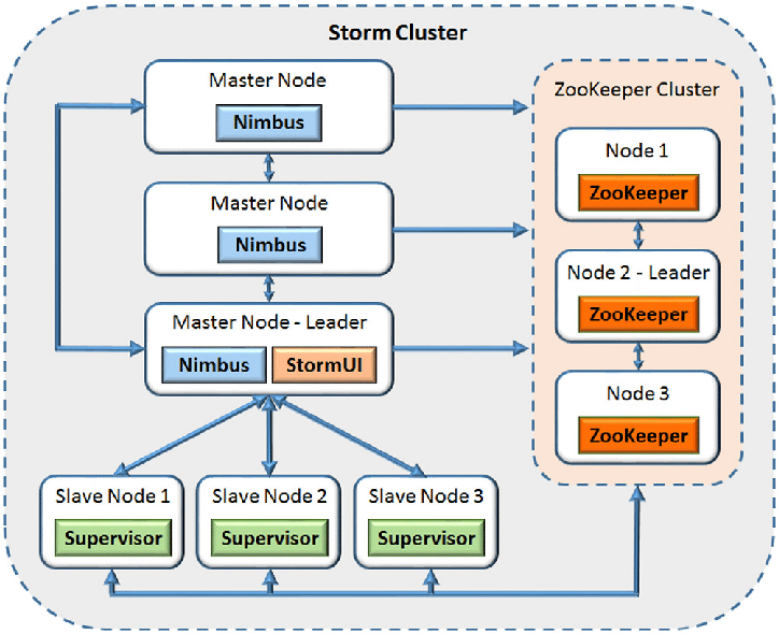

Figura 2.2: Schema di un cluster Storm

Nello schema è chiara la struttura di un cluster Storm. I master node eseguono il daemon Nimbus che si occupa della suddivisione delle task alle macchine del cluster, monitorandone le prestazioni. Il daemon Supervisor è eseguito all'interno dei nodi slave, con lo scopo di assegnare e gestire le attività. Tutti i nodi slave comunicano solo con il Master Node Leader, il quale scambia informazioni con gli altri nodi master, se presenti. Se si vogliono informazioni sul cluster nel suo complesso è necessario contattare il nodo leader di ZooKeeper, il cui scopo è monitorare lo stato generale del cluster, unendo le informazioni sui master node e sugli slave node. L'interfaccia utente risiede chiaramente sul nodo che coordina il cluster, ovvero il Master Node Leader. In generale un singolo processo di lavoro esegue solo attività da una singola topologia; pertanto, se si desidera eseguire più topologie sono necessari più worker JVM. La ragione di questa progettazione è la tolleranza ai guasti, ovvero l'isolamento delle topologie:

se una topologia fallisce la JVM anomala non influisce su altre topologie in esecuzione.

Casi d'uso

Storm è utilizzato per alimentare una varietà di sistemi Twitter come analisi in tempo reale, personalizzazione, ricerca, ottimizzazione dei guadagni e molti altri. Il framework si integra con il resto dell'infrastruttura di Twitter che include sistemi di database come Cassandra, Memcached e l'infrastruttura di messaggistica, Mesos e i sistemi di monitoraggio e avviso. Lo scheduler di isolamento di Storm rende possibile l'utilizzo dello stesso cluster sia per le applicazioni in produzione che per le applicazioni in sviluppo.
Yahoo! sta lavorando su una piattaforma di nuova generazione che consente l'unione di Big Data e l'elaborazione a bassa latenza. Sebbene Hadoop sia la tecnologia principale utilizzata dall'azienda per l'elaborazione in batch, Apache Storm consente l'elaborazione in streaming di eventi utente, feed di contenuto e registri delle applicazioni.
Flipboard è una piattaforma per esplorare, raccogliere e condividere notizie. Questa società utilizza Apache Storm per una vasta gamma di servizi come ricerca di contenuti, analisi in tempo reale, feed personalizzati, e molto altro. Apache Storm è integrato con l'infrastruttura che include sistemi come ElasticSearch, Hadoop, HBase e HDFS, per creare una piattaforma di dati altamente scalabile. Ooyala è una società privata venture-backed che fornisce prodotti e servizi di tecnologia video online per alcune delle più grandi reti, marchi e società di media del mondo. Ooyala ha un motore di analisi che elabora oltre due miliardi di eventi legati ad analisi ogni giorno, generati da quasi 200 milioni di spettatori. Ooyala utilizza Apache Storm per fornire ai propri clienti, analisi in streaming in tempo reale sul comportamento di visualizzazione dei consumatori e tendenze dei contenuti digitali. Storm consente data mining sui datasets di video online per fornire informazioni di business intelligence come la visualiz-

zazione di modelli in tempo reale, suggerimenti di contenuto personalizzati, guide di programmazione e importanti informazioni per aumentare le entrate [17].

2.3.2 Apache Samza

Apache Samza è un framework open source per l'elaborazione distribuita di flussi di eventi. Lo scopo principale è quello di supportare un throughput elevato per una vasta gamma di modelli di elaborazione, fornendo allo stesso tempo robustezza operativa. Samza raggiunge questo obiettivo attraverso un piccolo numero di astrazioni accuratamente progettate: log partizionati per messaggistica, stato locale tollerante ai guasti e pianificazione delle attività basata sul cluster.
Utilizza Apache Kafka per la messaggistica e Apache Hadoop YARN per fornire tolleranza ai guasti, isolamento del processore, sicurezza e gestione delle risorse.
Gli stream di eventi possono essere costituiti da molti tipi di dati, ad esempio l'attività degli utenti su un sito web, lo spostamento di merci o veicoli o la scrittura di record in un database. I processi di elaborazione in stream sono processi a esecuzione prolungata che consumano continuamente uno o più flussi di eventi, invocando alcune logiche di applicazione su ogni evento, producendo flussi di output derivati ed eventualmente scrivendo l'output in un database per interrogazioni successive.
Sviluppato originariamente da LinkedIn, è stato poi donato alla Apache Software Foundation nel 2013 e nel 2015 è diventato uno tra i principali progetti di Apache. Samza viene utilizzato in produzione da molte società, tra cui LinkedIn e TripAdvisor. Questo framework è progettato per situazioni in cui è richiesto un throughput molto elevato: con alcune impostazioni di produzione elabora milioni di messaggi al secondo o trilioni di eventi al giorno. Di conseguenza, la progettazione di Samza privilegia principalmente la scalabilità e la robustezza operativa rispetto ad altre caratteristiche [18].

Elaborazione dei log partizionati

Un lavoro Samza consiste in un insieme di istanze JVM (Java Virtual Machine), chiamate task, che elaborano ciascuna un sottoinsieme dei dati di input. Il codice in esecuzione in ogni JVM comprende il framework Samza e il codice utente che implementa le funzionalità specifiche dell'applicazione richiesta. L'API primaria per il codice utente è l'interfaccia Java StreamTask, che definisce un metodo process(). Una volta distribuito e inizializzato un lavoro Samza, il framework effettua una chiamata al metodo process() per ogni messaggio presente in uno qualsiasi dei flussi di input. L'esecuzione di questo metodo può avere vari effetti, ad esempio effettuare una query, aggiornare lo stato locale o inviare messaggi ai flussi di output. Questo modello di calcolo è strettamente analogo a una map task nel ben noto modello di programmazione MapReduce, con la differenza che l'input di un lavoro di Samza è in genere illimitato.
Analogamente a MapReduce, ogni attività Samza è un processo con un unico thread che scorre su una sequenza di record di input. Gli input di un lavoro Samza sono partizionati in sottoinsiemi disgiunti e ciascuna partizione di input viene assegnata esattamente a una singola task di elaborazione. È possibile assegnare più di una partizione alla stessa task di elaborazione, in questo caso l'elaborazione di tali partizioni viene interfogliata sul thread della task. Tuttavia, il numero di partizioni dell'input determina il massimo grado di parallelismo del lavoro. L'interfaccia di log presuppone che ogni partizione dell'input sia una sequenza di record completamente ordinata e che ogni record sia associato a un numero di sequenza o un identificativo progressivo in modo monotono crescente (noto come offset). Poiché i record di ogni partizione vengono letti in modo sequenziale, un processo può tracciare il suo avanzamento periodicamente memorizzando l'offset dell'ultimo record letto. In questo modo se viene riavviata un'attività di elaborazione stream, riprende a consumare l'input dall'ultimo offset salvato. Solitamente Samza viene utilizzato insieme ad Apache Kafka. Quest'ultimo fornisce un log partizionato, fault-tolerant

che consente ai "produttori" di aggiungere messaggi a una partizione dei log e ai "consumatori" di leggere in sequenza i messaggi in una partizione. Kafka consente inoltre ai processi di elaborazione in stream di rielaborare i record visti in precedenza reimpostando l'offset del consumatore su una posizione precedente, un'operazione utile durante il recupero dai guasti. In generale l'interfaccia di streaming di Samza è estendibile, può utilizzare qualsiasi sistema di archiviazione o di messaggistica come input, a condizione che il sistema possa aderire all'interfaccia di log partizionato. Per impostazione predefinita, Samza può anche leggere i file di input da Hadoop Distributed File System (HDFS).
Mentre ogni partizione di un flusso di input viene assegnata a una particolare attività di un lavoro Samza, le partizioni di output non sono associate alle attività. Cioè, quando uno StreamTask emette messaggi di output, può assegnarli a qualsiasi partizione del flusso di output. Questo fatto può essere utilizzato per raggruppare gli elementi di dati correlati nella stessa partizione. Questo aspetto garantisce, ad esempio nel caso del numero di occorrenze di ogni parola, che quando diverse attività incontrano occorrenze della stessa parola, vengano tutte scritte sulla stessa partizione di output, da cui un lavoro downstream potrà leggere e aggregare le occorrenze. Quando le attività di flusso sono composte in pipeline di elaborazione a più stadi, l'output di un'attività viene trasformato in input per un'altra attività [18].

Messaggistica

A differenza di molti altri framework di elaborazione dei flussi, Samza non implementa il proprio livello di trasporto dei messaggi; per recapitare i messaggi tra gli operatori di streaming, viene invece usato Kafka. Questa piattaforma scrive tutti i messaggi su disco, fornisce un ampio buffer tra le fasi della pipeline di elaborazione, limitata solo dallo spazio disponibile su tutti i nodi (o brokers). Gli elementi principali di Apache Kafka sono:

- *Produttori*: qualsiasi applicazione che invia messaggi a Kafka. Il produttore fornisce la chiave utilizzata per partizionare un topic.
- *Messaggi*: piccole quantità di dati che vengono inviate al sistema.
- *Consumatori*: qualsiasi applicazione che legge dei dati da Kafka. I consumatori sono responsabili della conservazione delle informazioni relative al proprio offset, in modo che siano a conoscenza di quali record sono stati elaborati in caso di errore.
- *Brokers*: i singoli nodi, o server, che costituiscono un cluster Kafka.
- *Topics*: un topic è un nome arbitrario dato ad un insieme di dati, per rendere tali dati identificabili. I produttori popolano i vari topic e i consumatori richiedono i dati di topic specifici.
- *Partizioni*: i dati di un topic possono essere molto voluminosi, è necessario quindi partizionarli tra i vari nodi del cluster Kafka. Un partizione quindi non è altro che una parte di topic distribuita su un nodo.
- *Offset*: un numero di sequenza di un messaggio all'interno di una partizione. Una volta che tale numero è assegnato è immutabile, così i messaggi sono memorizzati in ordine di arrivo all'interno di una partizione. Quindi tramite il nome del topic, il nome della partizione e l'offset si riesce a recuperare un messaggio preciso.

In genere, Kafka è configurato per conservare diversi giorni o settimane i messaggi di ciascun argomento. Pertanto, se uno stadio di una pipeline di elaborazione fallisce o inizia a rallentare, Kafka può semplicemente memorizzare l'input su quel livello lasciando al tempo stesso ampio spazio per la risoluzione del problema. A differenza dei progetti di sistema basati su backpressure, che richiedono un rallentamento del produttore se il consumatore non riesce a tenere il passo, il fallimento di un lavoro Samza non influisce sui lavori a monte che ne producono l'input. Questo fatto

è cruciale per il funzionamento robusto di sistemi su larga scala, poiché fornisce il contenimento dei guasti: per quanto possibile, un guasto in una certa parte del sistema non ha un impatto negativo su altre parti del sistema. I messaggi vengono eliminati solo se la fase di elaborazione fallita o lenta non viene riparata entro il periodo di conservazione dell'argomento di Kafka. Pertanto, la progettazione di Samza di utilizzare i registri su disco di Kafka per il trasporto dei messaggi è un fattore cruciale nella sua scalabilità: in una grande organizzazione, spesso accade che uno stream di eventi prodotto dal lavoro di un team venga consumato da uno o più lavori che sono amministrati da altri team. Infine, un ulteriore vantaggio dell'utilizzo di Kafka per il trasporto dei messaggi è che ogni flusso di messaggi nel sistema è accessibile per il debug e il monitoraggio: è infatti possibile in qualsiasi momento aggiungere un ulteriore consumatore per ispezionare il flusso dei messaggi [18].

Stato locale tollerante ai guasti

L'elaborazione stream stateless, in cui ogni messaggio può essere elaborato indipendentemente da qualsiasi altro messaggio, è facilmente implementabile e scalabile. Tuttavia, molte applicazioni richiedono che le attività di elaborazione dei flussi mantengano lo stato. Molti framework di elaborazione dello stream utilizzano lo stato transitorio conservato in memoria nell'attività di elaborazione, ad esempio in una tabella hash. Tuttavia, tale stato viene perso quando un'attività viene arrestata in modo anomalo o quando viene riavviato un processo di elaborazione, ad esempio per avviare una nuova versione.
Per rendere lo stato tollerante ai guasti, Samza consente a ciascuna attività di mantenere lo stato sul disco locale del nodo di elaborazione, con una cache in memoria per gli elementi di accesso frequente. Per impostazione predefinita, Samza utilizza RocksDB, un archivio di elementi chiave-valore incorporato che viene caricato nel processo JVM dell'attività di stream. Per carichi di lavoro con una buona localizzazione, RocksDB con cache of-

fre prestazioni vicine all'elaborazione in-memory; per carichi di lavoro ad accesso casuale su larga scala, rimane significativamente più veloce rispetto all'accesso ad un database remoto.
Se un lavoro viene arrestato e riavviato non per causa di guasti, nella maggior parte dei casi lo stato sopravvive al riavvio dell'attività senza ulteriori azioni. Tuttavia, in alcuni casi, ad esempio, se un nodo di elaborazione subisce un errore di sistema, lo stato sul disco locale può essere perso o reso inaccessibile. Per sopravvivere alla perdita di spazio su disco, Samza si affida nuovamente a Kafka. Per ogni spazio di memoria che contiene lo stato di un'attività di flusso, Samza crea un argomento di Kafka chiamato changelog che funge da log di replica per lo spazio di memoria. Quando un'attività di Samza deve recuperare il suo stato dopo la perdita della memoria locale, legge tutti i messaggi nella partizione appropriata del topic sul changelog e li applica nuovamente su RocksDB. Al termine di questo processo, il risultato è una nuova copia di memoria che contiene gli stessi dati di quella persa.
Poiché Kafka replica tutti i dati su più nodi, è adatto per un'archiviazione del changelog duratura e tollerante ai guasti. Se un'attività di flusso scrive ripetutamente nuovi valori per la stessa chiave nella propria memoria locale, il changelog rischia di contenere molti messaggi ridondanti, è infatti necessario solo il valore più recente di una determinata chiave per ripristinare la memoria locale. Per rimuovere questa ridondanza, Samza utilizza una funzionalità di Kafka chiamata log compaction sul topic changelog. Con questa funzione abilitata, Kafka esegue un processo in background che ricerca i messaggi con la stessa chiave e li scarta tutti tranne il più recente. Pertanto, ogni volta che una chiave in memoria viene sovrascritta con un nuovo valore, il vecchio valore viene rimosso dal log delle modifiche [18].

Pianificazione delle attività

Quando viene avviato un nuovo processo di elaborazione dello stream, è necessario assegnare le risorse di elaborazione: core della CPU, RAM, spazio su disco e larghezza di banda della rete. È possibile che queste risorse debbano essere modificate di volta in volta in base alla variazione del carico e recuperate quando un lavoro viene interrotto. Per le grandi organizzazioni, centinaia o migliaia di lavori devono essere eseguiti in modo coerente, con numeri del genere non è pratico assegnare manualmente le risorse: l'attività di pianificazione e l'allocazione delle risorse è necessario siano automatizzate. Per massimizzare l'utilizzo dell'hardware, molti lavori e applicazioni vengono distribuiti in un pool condiviso di macchine, con ogni macchina multi-core che esegue tipicamente una combinazione di attività da molti lavori diversi. Questa architettura richiede un'infrastruttura per la gestione delle risorse e per la distribuzione del codice di elaborazione dei lavori sulle macchine in cui deve essere eseguito. Samza supporta due modalità di funzionamento distribuito [18]:

- Un lavoro può essere distribuito in un cluster gestito da Apache Hadoop YARN. Uno scheduler di risorse e un gestore cluster di uso generico in grado di eseguire processori stream, lavori batch di MapReduce, motori di analisi dei dati e varie altre applicazioni su un cluster condiviso. I lavori di Samza possono essere distribuiti ai cluster YARN esistenti senza richiedere alcuna configurazione a livello di cluster o allocazione delle risorse.

- Samza supporta anche una modalità autonoma in cui le istanze JVM di un lavoro vengono distribuite ed eseguite attraverso un processo esterno che non è sotto il controllo di Samza. In questo caso, le istanze utilizzano Apache ZooKeeper per coordinare il loro lavoro, come l'assegnazione delle partizioni dei flussi di input. Inoltre, l'interfaccia di gestione dei cluster di Samza è estendibile, consentendo ulteriori integrazioni con altre tecnologie come Mesos.

Struttura

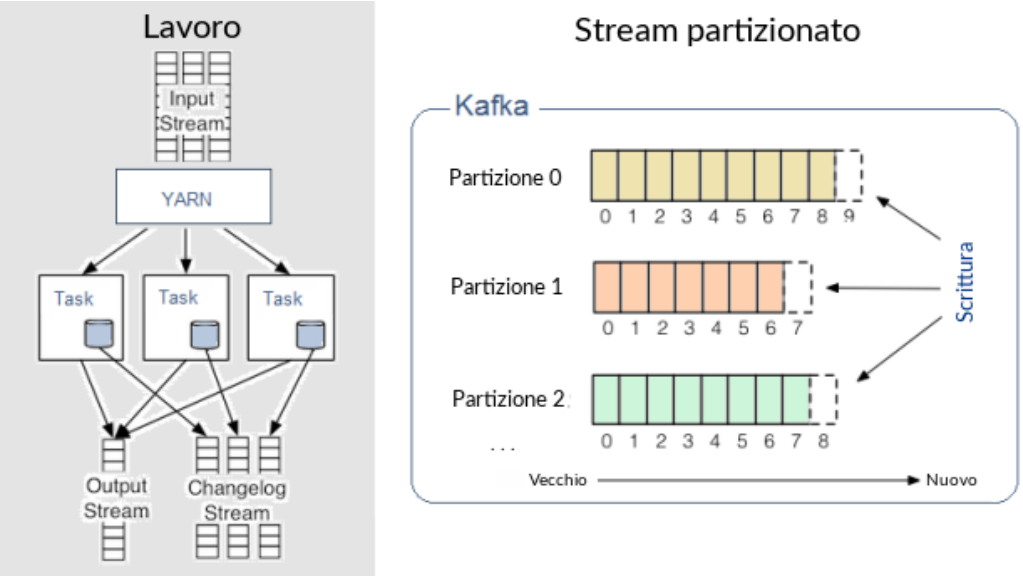

Figura 2.3: Schema di esecuzione Samza

La struttura è chiara dallo schema: vi sono vari flussi di input che vengono gestiti e distribuiti dal gestore di risorse YARN. Esso suddivide le task tra gli elementi del cluster i quali utilizzando Kafka popolano o richiedono elementi dal log partizionato, suddivisi per argomento. Infine si produce un flusso in output, che può fungere da flusso di input in caso vi si colleghi un altro processore di dati. Al tempo stesso si popola il changelog per permettere il recupero dei dati in caso di guasti.

Casi d'uso

Il caso più evidente è LinkedIn, che utilizza Apache Samza per molti scopi, ad esempio gestire gli eventi generati quando le persone si spostano in un'altra azienda, quando gli piace un articolo o quando si uniscono a un gruppo, generando newsfeed. Non viene utilizzato Hadoop per questo scopo in quanto le notizie sono sensibili alla latenza e se si utilizzasse

Apache Hadoop per l'elaborazione batch, si potrebbero attendere molte ore prima di una risposta. Altri utilizzi riguardano la creazione di annunci pubblicitari pertinenti, il monitoraggio della visualizzazione di annunci, dei click e altre metriche [19].

2.3.3 Caratteristiche frameworks

	Storm	**Samza**
Linguaggio implementazione	Clojure	Scala
Coordinamento del cluster	ZooKeeper	YARN
Modello di elaborazione	Stream	Stream
Computazione e trasformazione	Bolts	Tasks
Linguaggi utilizzabili	Qualsiasi	Basati su JVM
Operazioni con stato	No	Si
Prioritizzazione degli eventi	Programmabile	Si
Facilità di sviluppo	Facile	Facile
Modello di affidabilità	Elaborazione almeno una volta (esattamente una volta con Trident)	Elaborazione almeno una volta
Latenza di elaborazione streaming	Millisecondi (senza Trident)	Millisecondi / secondi (maggiore di Storm)
Utilizzo di memoria	Basso	Basso

2.4 Sistemi di elaborazione ibridi

Alcuni framework di elaborazione possono gestire carichi di lavoro sia batch che stream. Mentre i progetti incentrati su un tipo di elaborazione sono strettamente correlati a casi d'uso specifici, i framework ibridi tentano di offrire una soluzione generale per l'elaborazione dei dati. Non solo forniscono metodi per l'elaborazione di dati, ma anche integrazioni, librerie e strumenti per eseguire analisi dei grafi, machine learning e query interattive.

2.4.1 Apache Spark

Apache Spark è un framework di elaborazione batch di nuova generazione con funzionalità di elaborazione stream. Costruito utilizzando molti degli stessi principi del motore MapReduce di Hadoop, Spark si concentra principalmente sulla velocizzazione dei carichi di lavoro di elaborazione batch offrendo calcolo in memoria e ottimizzazione dell'elaborazione. Spark può essere distribuito come cluster autonomo, se abbinato a un livello di memorizzazione adeguato, o può essere collegato ad Hadoop come alternativa al motore MapReduce.
Apache Spark ha come elemento fondamentale il Resilient Distributed Dataset (RDD), un multiset in sola lettura di dati distribuiti su un cluster di macchine, tollerante ai guasti. Mentre in Spark 1.x l'RDD era l'API principale, da Spark 2.x è incoraggiato l'utilizzo dell'API Dataset, tuttavia l'API RDD non è deprecata. La tecnologia RDD è ancora alla base dell'API Dataset [20].
Spark e l'RDD sono stati sviluppati nel 2012 in risposta ai limiti del paradigma di elaborazione MapReduce, che impone una particolare struttura lineare del flusso di dati su programmi distribuiti: i programmi MapReduce leggono i dati di input dal disco, effettuano una mappatura dei dati, riducono i risultati ottenuti e memorizzano i risultati su disco. Gli RDD di Spark funzionano come un set di lavoro per programmi distribuiti che

offre una forma limitata di memoria condivisa distribuita [21].
Questo framework facilita l'implementazione di algoritmi iterativi, che visitano il proprio set di dati più volte in un loop, ed effettuano analisi dei dati interattive o esplorative, ovvero la ripetuta interrogazione dei dati in stile database. La latenza di tali applicazioni può essere ridotta di diversi ordini di grandezza rispetto a un'implementazione MapReduce. Tra la classe degli algoritmi iterativi vi sono gli algoritmi di training per i sistemi di machine learning, che hanno costituito lo stimolo iniziale per lo sviluppo di Apache Spark [20].
Questo framework richiede un gestore cluster e un sistema di archiviazione distribuito. Per la gestione dei cluster, Spark supporta la modalità standalone, YARN o Apache Mesos. Per l'archiviazione distribuita, Spark può interfacciarsi con vari sistemi, tra cui Hadoop Distributed File System (HDFS), MapR File System (MapR-FS), Cassandra, OpenStack Swift, Amazon S3, Kudu, o anche soluzioni personalizzate. Spark supporta una modalità locale pseudo-distribuita, di solito utilizzata solo a fini di sviluppo o test, in cui non è richiesta la memoria distribuita e al suo posto può essere utilizzato il file system locale. In tale scenario, Spark viene eseguito su una singola macchina con un executor per ogni core della CPU [22].

Composizione

Importante è lo SparkContext, il cuore di un'applicazione Spark. La sua funzione è creare una connessione con l'ambiente di esecuzione Spark, viene utilizzato per creare gli RDD, accumulatori e variabili broadcast, accedere ai servizi Spark ed eseguire lavori. Lo SparkContext è un client dell'ambiente di esecuzione Spark e funge da master dell'applicazione. Alcune principali mansioni dello Spark Context sono il controllo dello stato attuale dell'applicazione, l'annullamento di un lavoro o di uno stage, l'esecuzione del lavoro in modo sincrono o asincrono, l'accesso ad RDD persistenti o non e l'allocazione dinamica programmabile.
La Spark Shell è un'applicazione Spark scritta in Scala che offre un ambien-

te a riga di comando con completamento automatico. Questo strumento aiuta ad esplorare Spark e a prendere confidenza con le features del framework. L'applicazione Spark è un'elaborazione autonoma che esegue il codice fornito dall'utente con lo scopo di calcolare un risultato. Un'applicazione Spark può avere processi in esecuzione per suo conto anche quando non sta eseguendo un lavoro. Un'attività è un'unità di lavoro che viene inviata all'esecutore. Il lavoro è un calcolo parallelo costituito da più attività che vengono generate in risposta alle azioni in Apache Spark.
Ogni lavoro viene diviso in piccoli gruppi di compiti chiamati fasi (stages) che dipendono l'una dall'altra. Le fasi sono classificate come confini computazionali, tutto il calcolo spesso non può essere eseguito in un'unica fase, bensì è distribuito su più fasi [23].

Architettura

Un elemento chiamato driver interagisce con un singolo coordinatore chiamato master il quale gestisce i workers in cui viene eseguita l'elaborazione tramite executors, sia driver che executors girano sui propri processi Java. Il metodo main() del programma viene eseguito nel driver. Il driver è il processo che esegue il codice utente che crea gli RDD, esegue la trasformazione e l'azione e crea lo SparkContext. Quando viene lanciato Spark Shell, si crea un programma driver, quando si termina il driver anche l'applicazione viene conclusa. Il driver divide l'applicazione Spark in attività e pianifica l'esecuzione sull'executor. La funzione di pianificazione si trova nel driver e distribuisce l'attività tra i workers. I due ruoli chiave principali di questo elemento sono: la conversione del programma fornito dall'utente in attività e la pianificazione delle attività sull'esecutore.
La struttura del programma Spark a un livello più alto è la seguente: gli RDD vengono creati da alcuni dati di input, vengono generati nuovi RDD da quelli esistenti dopo che vi sono state applicate varie trasformazioni e infine viene eseguita un'azione per il calcolo dei dati. In un programma Spark, il DAG (grafico aciclico orientato) delle operazioni viene creato in

modo implicito, quando il driver viene eseguito, converte il grafo in un piano di esecuzione.
Il Cluster Manager in Spark ha la funzione di avvio degli executor e, in alcuni casi, anche dei driver. Sul gestore del cluster, i lavori e le azioni all'interno di un'applicazione sono programmati da Spark Scheduler in modalità FIFO. In alternativa, la pianificazione può essere eseguita anche in modalità Round Robin. Le risorse utilizzate da un'applicazione Spark possono essere regolate dinamicamente in base al carico di lavoro. Pertanto l'applicazione può liberare risorse inutilizzate e richiederle quando se ne ha nuovamente bisogno. Questo è disponibile per tutti i gestori cluster, vale a dire la modalità standalone, YARN e Mesos.
L'attività individuale in un lavoro Spark viene eseguita all'interno degli executors. Gli esecutori vengono lanciati alla partenza dell'applicazione Spark ed eseguiti per l'intera durata dell'applicazione. Anche se un esecutore fallisce, l'applicazione Spark può continuare normalmente. Ci sono due ruoli principali degli esecutori: il primo consiste nell'eseguire l'attività che costituisce l'applicazione e restituire il risultato al driver, il secondo consiste nel fornire l'archiviazione in memoria per gli RDD memorizzati in cache dall'utente [22].

Elementi principali

Spark Core costituisce le fondamenta del progetto nel suo complesso. Fornisce funzionalità di distribuzione, scheduling e I/O di base, fornite attraverso un'API basata sull'astrazione RDD. Questa interfaccia rispecchia un modello di programmazione funzionale: il programma "driver" richiama operazioni parallele come il mapping, il filtraggio o la riduzione di un RDD passando una funzione a Spark, che quindi ne pianifica l'esecuzione in parallelo sul cluster. Queste operazioni, e altre come il join, prendono un RDD come input e producono nuovi RDD. Gli RDD sono immutabili, la tolleranza agli errori si ottiene tenendo traccia della "genealogia" di ciascun RDD, ovvero la sequenza di operazioni che lo ha prodotto, in modo

che possa essere ricostruito in caso di perdita dei dati. Gli RDD possono contenere qualsiasi tipo di oggetti Python, Java o Scala.
Oltre allo stile funzionale di programmazione orientato agli RDD, Spark offre due forme limitate di variabili condivise: le variabili di broadcast fanno riferimento a dati di sola lettura che devono essere disponibili su tutti i nodi, mentre gli accumulatori possono essere utilizzati per programmare riduzioni in uno stile imperativo.
Spark SQL è un componente su Spark Core che ha introdotto un'astrazione dati chiamata DataFrame, che fornisce supporto per dati strutturati e semi-strutturati. Spark SQL fornisce un domain-specific language (DSL) per manipolare i DataFrames in Scala, Java o Python. Fornisce inoltre supporto per il linguaggio SQL con interfacce a riga di comando.
Spark Streaming utilizza la rapida capacità di pianificazione di Spark Core per eseguire analisi in streaming. Questa strategia è progettata per trattare flussi di dati come una serie di batch molto piccoli che possono essere gestiti utilizzando la semantica nativa del motore batch. Raggruppa i dati in micro-batches ed esegue trasformazioni RDD su di essi. Questo design consente di utilizzare lo stesso set di codice applicativo scritto per l'analisi in batch nello streaming analytics. Tuttavia è presente una latenza pari alla durata dei micro-batches [20].

Struttura

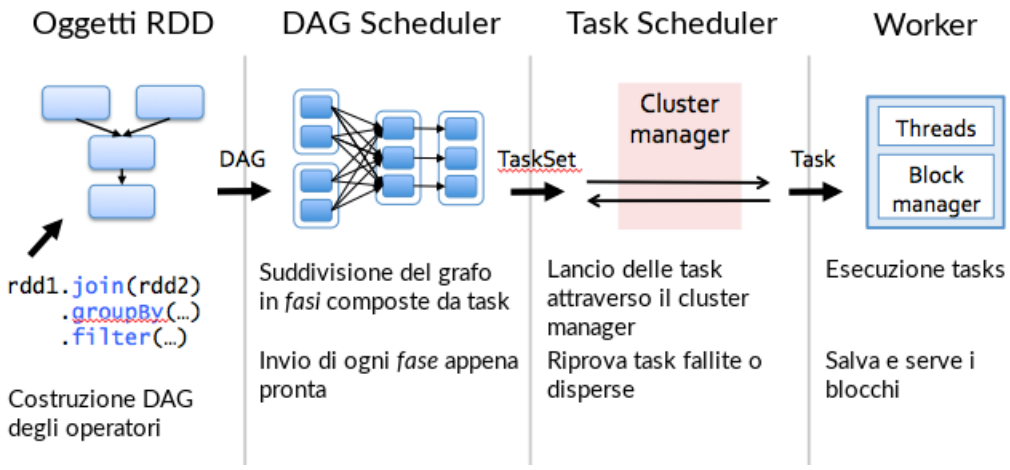

Figura 2.4: Schema di esecuzione Spark

Quando viene sottoposto del codice ad un'applicazione Spark, per prima cosa entra in gioco l'interprete, Spark usa un interprete Scala, con alcune modifiche. Quando si inserisce il codice nella console Spark, vengono creati gli RDD, applicati gli operatori specificati, e creato un grafo con ogni operazione effettuata.
Quando l'utente esegue un'azione, come ad esempio un collect, il grafo viene inviato a un DAG Scheduler, il quale lo divide in stage di mappatura e riduzione; uno stage è composto da attività basate su partizioni dei dati di input. Il DAG Scheduler crea una pipeline degli operatori per ottimizzare il grafo, infatti molte operazioni di mapping possono essere programmate in un'unica fase, questa ottimizzazione è la chiave per le prestazioni di Spark. Il risultato finale dello scheduler DAG è un insieme di stage. Gli stage vengono passati al Task Scheduler, il quale non conosce le dipendenze tra gli stage e avvia le attività tramite il Cluster Manager (Spark Standalone / YARN / Mesos). Infine il worker esegue le attività, avviando una nuova JVM per ogni lavoro, conoscendo quindi solo il codice che gli viene passato.

Casi d'uso

Yahoo ha due progetti principali gestiti da Spark, uno per la personalizzazione delle pagine di notizie per i visitatori web e un altro per l'esecuzione di analisi per le pubblicità. Per la personalizzazione delle notizie, la società utilizza algoritmi di machine learning in esecuzione su Spark per capire a cosa sono interessati i singoli utenti e per classificare le notizie che emergono, capendo quali tipi di utenti sarebbero interessati a leggerle. Per fare ciò, Yahoo che è anche uno dei principali contributori di Apache Spark, ha scritto un algoritmo Spark ML di 120 righe in Scala.
Sostituendo il precedente algoritmo di machine learning per la personalizzazione delle notizie che era scritto in 15.000 righe di C++. Il secondo caso di utilizzo di Yahoo mostra la capacità interattiva di Hive su Spark (Shark). L'obiettivo era utilizzare gli strumenti di business intelligence esistenti per visualizzare e interrogare i propri dati analitici pubblicitari raccolti in Hadoop.
Un altro dei primi ad adottare Spark è Conviva, una delle più grandi società di video streaming su Internet, con circa 4 miliardi di feed video al mese, secondi solo a YouTube. Un'operazione del genere richiede una tecnologia importante per garantire un'alta qualità del servizio. Per questo viene utilizzato Apache Spark per aiutare a fornire un'alta qualità del servizio evitando il buffering.
Spark viene anche utilizzato da ClearStory, uno sviluppatore di software di analisi dei dati specializzato nell'armonizzazione dei dati e che aiuta gli utenti a combinare dati interni ed esterni. La necessità di ClearStory è quella di aiutare gli utenti a fondere le loro fonti di dati interne con quelle esterne, come il traffico dei social media e i feed di dati pubblici, senza richiedere una complessa modellazione dei dati. ClearStory è stato uno dei primi clienti di Databricks e oggi fa affidamento sulla tecnologia Spark come una delle basi principali del suo prodotto interattivo in tempo reale [24].

2.4.2 Apache Flink

Apache Flink è un framework di stream processing che può gestire anche attività di elaborazione batch. Considera i batch semplicemente come flussi di dati limitati e quindi tratta l'elaborazione batch come un sottoinsieme dell'elaborazione streaming. Questo approccio stream-first per tutti i processi è stato chiamato architettura Kappa, in contrasto con l'architettura Lambda più conosciuta, in cui il batching viene utilizzato come metodo di elaborazione primario con streams utilizzati per integrare e fornire risultati immediati ma non raffinati. L'architettura Kappa, dove gli stream sono utilizzati per tutto, semplifica il modello. Ciò è stato possibile solo di recente in quanto i motori per l'elaborazione stream sono diventati più sofisticati.
Apache Flink include due API di base: una è la DataStream API per flussi di dati limitati o illimitati e l'altra DataSet API per set di dati limitati. Flink offre anche la Table API, un linguaggio simile a SQL per il flusso relazionale e l'elaborazione batch che può essere facilmente incorporato nella DataStream API e DataSet API. Il linguaggio di livello più alto supportato da Flink è SQL, che è semanticamente simile alla Table API e rappresenta programmi come espressioni di query SQL.
Al momento dell'esecuzione, i programmi Flink sono mappati in stream dei flussi di dati. Ogni flusso di dati inizia con una o più fonti come input, ad esempio una coda di messaggi o un file system, e termina con uno o più sink come output, ad esempio una coda di messaggi, un file system o un database. Può essere eseguito sullo stream un numero arbitrario di trasformazioni. Questi flussi possono essere organizzati come un grafo aciclico diretto, consentendo a un'applicazione di diramare o fondere i flussi di dati. I programmi Flink vengono eseguiti come un sistema distribuito all'interno di un cluster e sono disponibili in modalità standalone, nonché su impostazioni YARN, Mesos, Docker e altri framework di gestione delle risorse [25].

Caratteristiche

Apache Flink include un meccanismo di tolleranza agli errori basato su checkpoint distribuiti. Un checkpoint è uno snapshot asincrono e automatico dello stato di un'applicazione e la sua posizione all'interno di un flusso di dati. In caso di guasto, con questo sistema di ripristino, l'applicazione riprende l'elaborazione dall'ultimo checkpoint completato, assicurando che il framework analizzi esattamente una volta ogni elemento all'interno di un'applicazione.
Flink include anche un meccanismo chiamato savepoints, ovvero checkpoint attivati manualmente. Un utente può generare un savepoint, arrestare un programma Flink in esecuzione, e successivamente riprendere il programma dallo stesso stato e posizione dell'applicazione all'interno dello stream di dati. I savepoints consentono di aggiornare un programma o un cluster Flink senza perdere lo stato dell'applicazione. I componenti di base con cui Flink lavora sono:

- *Streams*: set di dati immutabili e illimitati che attraversano il sistema.
- *Operators*: funzioni che operano su flussi di dati per produrre altri flussi.
- *Sources*: il punto di ingresso per gli stream che entrano nel sistema.
- *Sinks*: il luogo in cui i flussi escono dal sistema Flink. Potrebbero rappresentare un database o un connettore a un altro sistema.

La DataStream API consente trasformazioni, come ad esempio filtri, aggregazioni e window functions, su flussi di dati limitati o illimitati. Questa API include più di 20 diversi tipi di trasformazioni ed è disponibile sia in Java che Scala.
La DataSet API consente trasformazioni, come filtri, mappature, unioni e raggruppamenti su set di dati limitati. Concettualmente simile all'API DataStream, anch'essa include più di 20 diversi tipi di trasformazioni ed è disponibile in Java, in Scala e in via sperimentale anche in Python.

Elaborazione

I dati in Apache Flink possono essere elaborati come flussi illimitati o limitati [26]:

- I *flussi illimitati* hanno un inizio ma nessuna fine definita. Non terminano e forniscono dati così come sono generati. I flussi illimitati devono essere elaborati continuamente, vale a dire che gli eventi devono essere prontamente gestiti dopo essere stati immagazzinati. Non è possibile attendere l'arrivo di tutti i dati di input perché l'input non sarà mai completo. L'elaborazione di dati illimitati richiede spesso che gli eventi vengano immagazzinati in un ordine specifico, ad esempio l'ordine in cui questi eventi si sono verificati, per poter far affidamento sulla completezza dei risultati.

- I *flussi limitati* hanno un inizio e una fine definiti. Gli stream limitati possono essere elaborati immagazzinando tutti i dati prima di eseguire qualsiasi calcolo. L'analisi ordinata non è richiesta per elaborare questo tipo di flussi perché un set di dati limitato può sempre essere ordinato successivamente. L'elaborazione di flussi limitati è anche nota come elaborazione batch.

Questo framework offre anche alcune ottimizzazioni per i carichi di lavoro batch, ad esempio poiché tali elaborazioni sono supportate dalla memorizzazione persistente, Flink rimuove lo snapshot da questi carichi. I dati sono ancora recuperabili, ma l'elaborazione normale viene completata più rapidamente. Un'altra ottimizzazione per il batch processing riguarda la suddivisione delle attività in modo che gli stage e i componenti siano coinvolti solo quando necessario, questo aiuta Flink ad integrarsi bene con gli altri utenti del cluster. L'analisi preventiva delle attività offre a Flink un'ulteriore possibilità di ottimizzazione visualizzando l'intero insieme di operazioni, la dimensione del set dei dati e i requisiti dei passaggi che saranno eseguiti.
Mentre Spark esegue il batch e stream processing, il suo streaming non è

appropriato per molti casi d'uso a causa della sua architettura micro-batch. L'approccio stream-first di Flink offre bassa latenza, throughput elevato e reale elaborazione entry-by-entry. Gestisce la propria memoria invece di affidarsi ai meccanismi di garbage collection Java nativi per ovvi motivi prestazionali. A differenza di Spark, Flink non richiede l'ottimizzazione e la regolazione manuale quando cambiano le caratteristiche dei dati che elabora, gestisce automaticamente il partizionamento dei dati e la memorizzazione nella cache. Flink analizza il suo lavoro e ottimizza le attività in vari modi, parte di questa analisi è simile a ciò che avviene con le query SQL nei database relazionali, si mappa il modo più efficace per implementare una determinata attività. È in grado di parallelizzare fasi che possono essere completate in parallelo, mentre raggruppa i dati per le attività eseguibili in blocco. Per le attività iterative, Flink tenta di eseguire il calcolo sui nodi in cui i dati vengono archiviati, per motivi ovviamente prestazionali. Può anche fare "delta iterazione", o iterazione solo sulle porzioni di dati che hanno subito modifiche. In termini di strumenti per l'utente, Flink offre una piattaforma web-based di pianificazione del lavoro per gestire facilmente le attività e visualizzare lo stato del sistema.
Gli utenti possono anche consultare il piano di ottimizzazione per le proprie attività visualizzando come verranno effettivamente implementate nel cluster. Per le attività di analisi, Flink offre query in stile SQL, elaborazioni grafiche e librerie di machine learning e calcolo in-memory. È inoltre facilmente integrabile anche ad altri componenti, può essere ad esempio usato all'interno di una pila Hadoop, si integra facilmente con YARN, HDFS e Kafka. Questo framework può eseguire attività scritte per Hadoop o Storm, tramite pacchetti di compatibilità. Uno dei maggiori svantaggi di Flink al momento è che è ancora un progetto molto giovane, le implementazioni su larga scala non sono ancora così comuni come altri framework di elaborazione e non sono state effettuate molte ricerche sui limiti di dimensionamento [25].

Struttura

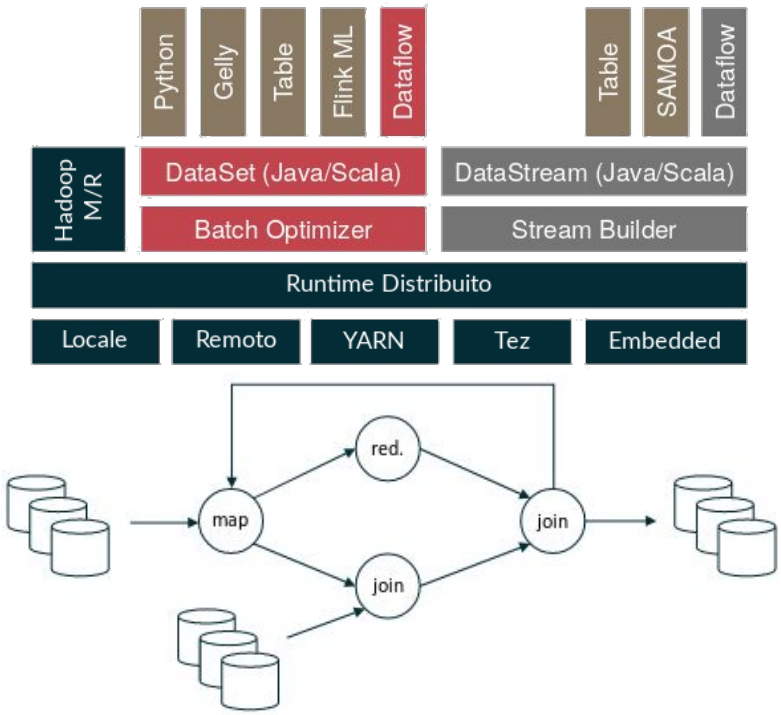

Figura 2.5: Schemi di struttura ed esecuzione Flink

Nelle immagini sopra è chiara la struttura del framework, divisa per strati. Negli strati Intermedi è visibile come le API per l'elaborazione stream e batch siano supportate da elementi quali il Batch Optimizer e lo Stream Builder, che ne aumentano la fruibilità e l'efficienza. Per quanto riguarda il flusso di esecuzione, è evidente come la creazione di un grafo aciclico orientato renda più efficiente l'elaborazione e la comprensione del ciclo totale di analisi dei dati. Le sorgenti di dati possono essere collegate a qualsiasi operazione all'interno del grafo, e tali operazioni, che costituiscono i nodi, dividono, fondono o redirigono il flusso entrante di dati. Il tutto uscendo da uno o più sink, come flusso di output.

Casi d'uso

Un famoso caso riguarda la società di intrattenimento online King. Oltre 300 milioni di utenti mensili generano più di 30 miliardi di eventi ogni giorno dai diversi giochi e sistemi. Per gestire questi enormi flussi di dati analizzandoli e mantenendo la massima flessibilità è stato adottato Apache Flink. Che consente ai ricercatori di King di accedere a questi enormi flussi di dati in tempo reale.
Un altro caso è Zalando, con oltre 16 milioni di clienti in tutto il mondo, utilizza Apache Flink per il monitoraggio dei processi in tempo reale. Un'architettura basata sul flusso supporta in modo soddisfacente l'approccio basato sui microservizi utilizzato da Zalando e Flink fornisce l'elaborazione streaming per il monitoraggio dei processi aziendali e la continua estrazione, trasformazione e caricamento dei dati.
Ancora, Alibaba lavora con acquirenti e fornitori attraverso il suo portale web. La variazione di Flink, chiamata Blink, viene utilizzata da questo colosso dell'e-commerce per i consigli online. Fornisce la possibilità di considerare gli acquisti effettuati durante il giorno e consigliare i prodotti più opportuni agli utenti. Questo ha un ruolo chiave nei giorni di festività quando l'attività è insolitamente alta.
Infine il Gruppo Bouygues, la terza società di telefonia in Francia, usa Flink per l'elaborazione e l'analisi degli eventi in tempo reale per miliardi di messaggi al giorno in un sistema che funziona 24 ore su 24, 7 giorni su 7. Questo framework è stato scelto perché fornisce una latenza estremamente bassa, fondamentale nel campo della telefonia. Inoltre l'azienda, intenzionata ad ottenere informazioni dettagliate in tempo reale sull'esperienza del cliente, su ciò che sta accadendo globalmente sulla rete e su ciò che sta accadendo in termini di evoluzioni e operazioni di rete, ha costruito un sistema per analizzare i log delle apparecchiature di rete con il fine di identificare gli indicatori della qualità dell'esperienza utente. Il sistema gestisce 2 miliardi di eventi al giorno, 500.000 eventi al secondo, con una latenza end-to-end richiesta di meno di 200 millisecondi, compresa la pub-

blicazione di messaggi dal livello di trasporto e l'elaborazione dei dati in Flink. Questo risultato è stato ottenuto su un piccolo cluster riferito a soli 10 nodi con 1 gigabyte di memoria ciascuno [27].

2.4.3 Caratteristiche frameworks

	Spark Streaming	**Flink**
Linguaggio implementazione	Scala	Java
Coordinamento del cluster	YARN, Mesos, Standalone	YARN, Standalone
Modello di elaborazione	Micro-batch	Singoli eventi
Salvataggio dello stato	Dataset distribuito	Salvataggio distribuito di elementi chiave-valore
Linguaggi utilizzabili	Java, Scala, Python	Java, Scala, Python
Auto-scalabilità	Si	No
Tolleranza ai guasti	Checkpoint basati su RDD	Checkpoint
Ottimizzazione	Automatica	Manuale
Modello di affidabilità	Elaborazione esattamente una volta	Elaborazione esattamente una volta
Latenza di elaborazione streaming	Secondi	Millisecondi / secondi
Maturità framework	Elevata	Bassa

Capitolo 3

Sperimentazione

3.1 Introduzione

In questa sezione verrà proposta una sperimentazione di due dei framework descritti sopra: Apache Hadoop e Apache Spark. Tali framework sono stati scelti grazie alle varie somiglianze, che hanno permesso di avere un confronto più efficace tra una vecchia generazione di framework ed un nuovo approccio. MapReduce di Hadoop è di fatto il più datato tra i due, utilizza lettura e scrittura sul disco rallentando notevolmente le prestazioni. Spark invece è stato progettato per potenziare la velocità, effettuando i calcoli direttamente in-memory.
Alcune somiglianze e differenze tra i due framework sono prima di tutto la tolleranza ai guasti, supportata da entrambi. Il recupero dei dati dopo un guasto avviene, nel caso di Hadoop tramite il processo di creazione di repliche. Ogni volta che un file è archiviato dall'utente, viene diviso in blocchi che vengono distribuiti su macchine diverse presenti nel cluster HDFS. Dopodiché, la replica di ogni blocco viene creata su altre macchine presenti nel cluster, di default HDFS crea 3 copie di un file sugli altri nodi del cluster. Spark utilizza un sistema basato su RDD, per assicurare tolleranza ai guasti i dati vengono replicati tra i vari esecutori nei worker nodes del cluster, inoltre ogni RDD mantiene il lignaggio dell'operazione deter-

ministica che è stata utilizzata sul set di dati di input per crearlo. Grazie a questo sistema è possibile ricostruire l'RDD rieseguendo le operazioni che l'hanno generato. Un'altro aspetto importante di questi due framework è che entrambi possono lavorare sullo stesso file system: HDFS, ed utilizzare lo stesso gestore di risorse: YARN. Entrambi sono cross-platform e possono facilmente essere eseguiti su vari sistemi operativi. Sebbene la sperimentazione su Spark sia stata effettuata utilizzando Scala perché più performante, è anche possibile utilizzare Java, così come in Hadoop, fornendo maggiore omogeneità nella programmazione. Infine l'alta scalabilità è una caratteristica che possiedono entrambi i framework, è possibile infatti aggiungere un numero arbitrario di nodi ai cluster per aumentare la distribuzione della computazione.
Altre somiglianze tra Hadoop e Spark sono il supporto ad SQL, lo sviluppo open-source, la licenza Apache 2.0 e l'affidabilità. Ciò li rende adeguati per il tipo di confronto che ho proposto in questa sezione.

3.2 Installazione e setup dei framework

Di seguito illustro il procedimento eseguito per l'installazione dei framework in modalità stand-alone sulla mia macchina, ovvero con un singolo nodo. I dati tecnici della macchina su cui è stata eseguita la sperimentazione sono:

- Sistema Operativo: Debian GNU/Linux 9 (Stretch)
- Kernel Version: 4.9.110-1 (2018-07-05)
- RAM: 12 GB DDR4
- Processore: Intel Core i5-7200U (2,3 GHz)
- Memoria: HDD 1TB

3.2.1 Hadoop

L'installazione di Hadoop richiede la presenza di Java, in questo caso è già presente alla versione 1.8.0_171. È stato creato per questa sperimentazione un nuovo utente su cui eseguire i framework, da linea di comando:
useradd -m -d /home/hadoop -s /bin/bash hadoop
passwd hadoop
su - hadoop
Successivamente si è configurata una secure shell (SSH) senza password, da terminale:
ssh-keygen
cat ~/.ssh/id_rsa.pub » ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
ssh 127.0.0.1
Dopo questo è necessario scaricare i file binari della release di Hadoop desiderata, per questa sperimentazione è stata installata al versione 2.8.4 rilasciata il 15 maggio 2018. Dopo aver spacchettato il file e aver spostato la cartella all'interno di */usr/local*, in cui risiedono i software installati localmente, il passo successivo è il setup delle nuove variabili d'ambiente, all'interno del file ~/.bashrc, tra queste HADOOP_HOME, HADOOP_HDFS_HOME e HADOOP_YARN_HOME.
Dato che si vuole installare Hadoop in modalità pseudo-distribuita, ovvero in cui ogni daemon Hadoop viene eseguito su un processo separato è necessario modificare i file di configurazione del framework con i valori appropriati per la mia macchina. Si parte dal file di ambiente $HADOOP_HOME/etc/hadoop/hadoop-env.sh in cui va specificato il path della JVM.
In $HADOOP_HOME/etc/hadoop/core-site.xml si specifica l'host e la porta relativa al file system Hadoop (HDFS),
in $HADOOP_HOME/etc/hadoop/hdfs-site.xml si indicano i path per i dati relativi ai namenode e ai datanode. Altre modifiche vanno aggiunte ai file $HADOOP_HOME/etc/hadoop/mapred-site.xml e

$HADOOP_HOME/etc/hadoop/yarn-site.xml. Fatto ciò è necessario formattare il namenode con il comando:
hdfs namenode -format
Per permettere ad Hadoop di oltrepassare il firewall vanno abilitate le porte 50070 per il namenode, 50075 per i datanode e 8088 per YARN.
Si può a questo punto avviare i daemon per il namenode e per il datanode eseguendo lo script in $HADOOP_HOME/sbin/start-dfs.sh. Un'interfaccia utente per monitorare la salute del HDFS è disponibile al link: *http://localhost:50070/*. Per avviare il daemon del gestore di risorse (YARN) e il daemon nodemanager si utilizza lo script $HADOOP_HOME/sbin/start-yarn.sh, ed è monitorabile al link: *http://localhost:8088/*.
L'utilizzo dell'HDFS è semplice, ad esempio per la creazione di una cartella è sufficiente da linea di comando:
hdfs dfs -mkdir /Directory
O per il caricamento dal filesystem della macchina all'HDFS:
hdfs dfs -put /path_to/file.txt /HDFS_Directory
Le cartelle e i file presenti sull'HDFS sono visibili tramite interfaccia utente al link: *http://localhost:50070/explorer.html/*.

3.2.2 Spark

Per Spark è necessaria la presenza di Scala, è stata installata la versione 2.11.8 insieme al build-tool SBT versione 1.1.6, insieme a Java, che è già presente alla versione 1.8.0_171. Sono stati poi scaricati i file binari per la release 2.2.2 di Spark, per la versione stand-alone è sufficiente spacchettare l'archivio e metterlo all'interno dell'apposita cartella nel filesystem, poi se si vuole evitare di raggiungere ogni volta gli script si può modificare il file ~/.bashrc. È sufficiente aggiungere a tale file due righe: SPARK_HOME=/path-to/spark ed export PATH=$SPARK_HOME/bin:$PATH
Così facendo per aprire la shell di Spark ed eseguire il codice basta digitare sul terminale: spark-shell.

3.3 Codice

3.3.1 Hadoop

Il programma che verrà eseguito e confrontato riguarda il conteggio del numero di occorrenze di ogni parola all'interno di un insieme di file testuali. Questo esempio è stato scelto perché facilmente comprensibile ma comunque efficace, specialmente se ci si avvicina a questi framework per la prima volta. È stato infatti anche utilizzato come esempio nel paper Google su MapReduce. La sperimentazione non vuole concentrarsi sulla potenza che offrono i framework a livello di analisi dei dati, bensì analizzare i tipi di elaborazione, la facilità di programmazione e le prestazioni che entrambi offrono in termini di utilizzo della CPU. Il codice per il word counter in Hadoop è stato scritto in Java, di seguito il codice della classe WordCount.

Map

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable> {

   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value,
       OutputCollector<Text, IntWritable> output, Reporter reporter)
       throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
      }
```

```
    }
}
```

Prima di tutto la classe mapper (Map) è statica, estende MapReduceBase e implementa l'interfaccia Mapper. Le variabili word e one formano insieme la coppia chiave-valore, one è di tipo IntWritable ed inizializzata a 1. Il motivo per cui in Hadoop sono utilizzate classi come IntWritable e Text invece di, ad esempio Integer e String è che le prime implementano interfacce come Comparable, Writable e WritableComparable. Tali interfacce sono necessarie in MapReduce, ad esempio Comparable è utilizzata per effettuare un confronto quando il riduttore ordina le chiavi e Writable per scrivere il risultato sul disco locale. Non viene invece utilizzata l'interfaccia Serializable perché troppo grande e pesante per Hadoop, Writable può serializzare l'oggetto Hadoop in modo molto più leggero.
Il metodo map accetta come parametri un LongWritable, che è la chiave, un testo, che è il valore e un OutputCollector che rappresenta l'output nella forma <Text, IntWritable>, e infine il reporter per indicare lo stato. Genera un IOException se il file per qualche motivo non è leggibile. La classe StringTokenizer è una classe di utility delle API MapReduce per suddividede le righe di testo in singole parole (token). Questo è il metodo default per creare token dalle righe. Il ciclo while successivo imposta la variabile word con il prossimo token disponibile, contenente la parola successiva. Poi aggiunge la coppia word-one alla collezione di output.
Questo codice viene eseguito su ciascun nodo fisico, per impostazione predefinita in sostanza preleva parti di testo dal file system HDFS e li suddivide in un insieme di coppie chiave-valore.

Reduce

```
public static class Reduce extends MapReduceBase implements
    Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterator values,
       OutputCollector<Text, IntWritable> output, Reporter reporter)
       throws IOException {
      int sum = 0;
      while (values.hasNext())
         sum += values.next().get();
      output.collect(key, new IntWritable(sum));
   }
}
```

La classe Reduce estende MapReduceBase e implementa l'interfaccia Reducer. Il metodo reduce prende in input un Text come chiave, un oggetto iterabile Iterator contenente tutti i valori con quella chiave, che nel nostro caso saranno tanti IntWritable(1) quante sono le occorrenze della parola con quella determinata chiave. Il ciclo while fa proprio questo, iterare i valori e contarli. Infine crea una nuova coppia chiave-valore con la chiave e il numero di occorrenze della parola, aggiungendola alla collezione di output. Tutti i valori con la stessa chiave sono presentati insieme a un singolo riduttore.

Driver

```
public static void main(String[] args) throws Exception {
   JobConf conf = new JobConf(WordCount.class);
   conf.setJobName("wordcount");
   conf.setOutputKeyClass(Text.class);
   conf.setOutputValueClass(IntWritable.class);
   conf.setMapperClass(Map.class);
   conf.setCombinerClass(Reduce.class);
   conf.setReducerClass(Reduce.class);
   conf.setInputFormat(TextInputFormat.class);
   conf.setOutputFormat(TextOutputFormat.class);
   FileInputFormat.setInputPaths(conf, new Path(args[0]));
   FileOutputFormat.setOutputPath(conf, new Path(args[1]));
   JobClient.runJob(conf);
}
```

Il codice MapReduce è gestito da un metodo principale, il Driver. Questo metodo inizializza il lavoro, fornisce le istruzioni necessarie alla piattaforma Hadoop per eseguire il codice su un set di file di input e controlla il posizionamento dei file di output. Per prima cosa crea un'istanza di configurazione del lavoro che analizzi WordCount.class, specifica il nome del Job e indica le classi di output, in questo caso Text e IntWritable. Successivamente vengono impostate le classi mapper e reducer, e indicati i tipi dei file di input e output, in questo caso testuali. Il path per i file di input e output vengono indicati come argomenti da terminale quando si esegue l'applicativo. Infine viene eseguito il Job, elaborato tramite MapReduce.

3.3.2 Spark Batch

Di seguito verrà illustrato il codice per l'esecuzione del conteggio del numero di occorrenze delle parole all'interno di un'insieme di file testuali, tramite Spark. Per la sperimentazione con Apache Spark è stato utilizzato

il linguaggio Scala, invece che Java, in quanto più efficiente e linguaggio nativo del framework. L'esecuzione è avvenuta direttamente tramite shell, da terminale, in quanto più pratico.

```
val file = sc.textFile("hdfs://localhost:9000/Input")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://localhost:9000/Output/out.txt")
```

Per prima cosa vengono letti i dati archiviati nel percorso specificato all'interno dell'HDFS e tramite la funzione textFile dello SparkContext, sc, viene creato un RDD di stringhe chiamato file. Ogni riga nell'RDD è composta da una riga di uno dei file testuali all'interno della cartella indicata come input.
Successivamente tramite la funzione flatMap, si trasforma l'RDD con le righe di testo in un nuovo RDD contenente le singole parole. La funzione map trasforma l'RDD di parole in un altro composto da coppie chiave-valore nella forma (parola, 1). Infine la funzione reduceByKey somma tutti i valori per ogni chiave. In generale la funzione di riduzione prende una lista vi applica una qualche funzione varie volte tra il precedente risultato e l'elemento successivo. Tale funzione produce un RDD di tuple chiave-valore nella forma (parola, <numero di occorrenze>). Infine la funzione saveAsTextFile trasforma l'RDD in un formato testuale e lo salva nel file specificato all'interno dell'HDFS. Si è visto come manipolando gli RDD si riesca a lavorare in-memory producendo sempre nuovi dataset, il tutto supportato da un file system distribuito.

3.3.3 Spark Streaming

Spark Streaming è un framework che lavora con micro-batch, dati raccolti tramite intervalli temporali. Utilizza dei D-Stream (Discretized Stream) che strutturano la computazione come piccoli insiemi di task brevi, senza

stato e deterministiche. Per il word count è stato utilizzato lo stato, ogni volta che viene aggiunto un file si effettua il conteggio e si aggiornano i valori delle parole contate. Un D-Stream può nascere da varie sorgenti di dati, come Kafka o HDFS come nel nostro caso.

```
val updateFunc = (values: Seq[Int], state: Option[Int]) => {
   val currentCount = values.foldLeft(0)(_ + _)
   val previousCount = state.getOrElse(0)
   Some(currentCount + previousCount)
}
```

Una volta creato l'oggetto Scala WordCount, all'interno della funzione main è presente la funzione updateFunc. Questa funzione prende in input una chiave, che corrisponde ad un insieme di valori e il precedente stato della chiave come un Option. Quest'ultimo tipo è opzionale in quanto alla prima elaborazione risulta None. Vengono poi aggregati i nuovi valori della chiave utilizzando foldLeft, sommandoli. Si prende infine il vecchio stato della chiave e si uniscono insieme, aggiornando l'attuale stato. Di base Spark Streaming è stateless, perciò questa funzione è necessaria se si vogliono contare le occorrenze delle parole, fornendo man mano nel tempo i file di testo.

```
val sparkConf = new SparkConf().setAppName("HdfsWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(2))
ssc.checkpoint(".")
```

Prima di tutto si definisce la configurazione, indicando il nome dell'applicazione ed eventualmente settare il master, specificando il numero di core su cui si lavorerà, in questo caso non indicandolo di default sono tutti quelli disponibili. Dopo viene definito lo StreamingContext, indicando l'intervallo di tempo che definisce ogni micro-batch. Impostando tale intervallo a 2 secondi, ad ogni ciclo si raccolgono i dati arrivati nei 2 secondi precedenti e si elaborano come singolo batch. Infine viene posizionato il

punto di checkpoint, che permette di recuperare i dati in caso di problemi, si passa come parametro il path alla directory di cui fare il checkpoint.

```
val lines = ssc.textFileStream("hdfs://localhost:9000/Input")
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))
```

A questo punto viene indicata la cartella sorgente all'interno dell'HDFS da cui saranno prelevati i dati di stream man mano che vengono inseriti. Un'alternativa poteva essere socketTextStream, ad esempio, che invece di file testuali all'interno di una directory legge i valori che arrivano ad un IP e una porta specificati. Successivamente si elabora il micro-batch creatosi, creando dall'RDD lines contenente le righe testuali, un nuovo RDD contenente le singole parole. A questo punto il nuovo RDD è mappato in wordDstream, un insieme di entry chiave-valore nella forma (parola, 1).

```
val stateDstream = wordDstream.updateStateByKey[Int](updateFunc)
stateDstream.print()
ssc.start()
ssc.awaitTermination()
```

Con la funzione updateStateByKey si intende aggiornare lo stato della computazione del micro-batch, passando la funzione precedentemente definita updateFunc. L'output è stampato a schermo, con print, ma potrebbe venire anche salvato su file. La funzione start fa partire l'esecuzione di SparkStreaming, creando jobs ed elaborando i dati, mentre awaitTermination semplicemente attende un segnale di terminazione da parte dell'utente.

3.4 Confronti

3.4.1 Modalità single-node

I seguenti confronti sono stati effettuati eseguendo Hadoop e Spark Batch sulla mia macchina, quindi su un singolo nodo. Lo scopo di questa sperimentazione non sfrutta al massimo le potenzialità dei framework, che vengono invece messe in risalto su cluster più ampi e quantità di dati maggiori. Tuttavia lo scopo è illustrare un primo approccio all'elaborazione di dati, mostrando un basilare confronto di consumo del tempo della CPU.
I set di dati sono rappresentati da vari libri classici, come Tom Sawyer, Moby Dick, Frankenstein, e molti altri, tutti sotto forma di file testuali. Sono stati testati tre dataset in ordine di grandezza, il primo di 5,7 MB, il secondo grande quasi il quintuplo del primo, 28,4 MB e il terzo circa dieci volte il primo, 62,5 MB. Inoltre l'esecuzione con spark è stata eseguita su due core paralleli, quelli fisici montati sulla CPU, in modo da rendere più equo il confronto con Hadoop.
Una prima cosa che si nota nalla *Figura 3.1* è che i tempi di calcolo impiegati da Spark sono circa 10 volte inferiori dei tempi di Hadoop, per quanto riguarda l'elaborazione batch. I tempi di Hadoop hanno una crescita proporzionale alla grandezza del dataset, elaborando una media di 0.34 MB/s; migliore è il risultato di Spark, con circa 3 MB/s anch'esso aumenta in modo proporzionale al dataset.
Per quanto riguarda l'applicazione di MapReduce, entrambi i framework hanno prodotto 9 map tasks per il primo set di dati, 45 per secondo e 90 per il terzo. È stata infatti creata una map task separata per ogni singolo file testuale esaminato. Sperimentando su una macchina singola è stata prodotta una singola reduce task per entrambi i framework.
In Apache Spark è stata misurata la memoria utilizzata dalle strutture dati interne create durante shuffle, aggregazioni e join (peak execution memory), il cui valore è approssimativamente la somma del picco di dimensioni raggiunto da ogni struttura dati creata in una certa task. Per il primo da-

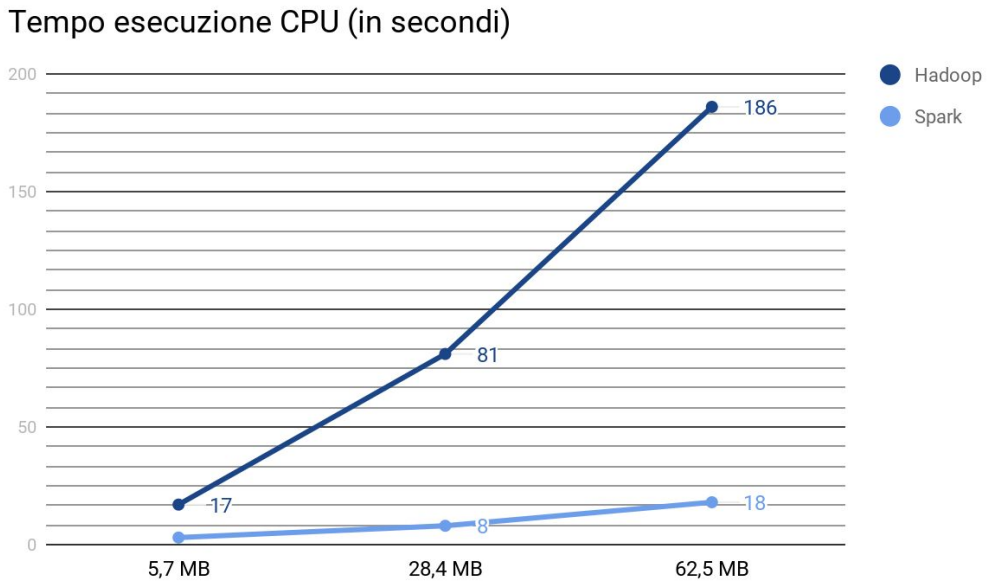

Figura 3.1: Tempi esecuzione Hadoop vs Spark (single-node)

taset raggiunge un valore massimo di 4.4 MB, per il secondo 5.2 MB e per il terzo 5.3 MB, è evidente come Spark riesca a gestire bene la memoria impiegata dalle strutture dati utilizzate per l'elaborazione.

Una possibile evoluzione per quanto riguarda Apache Spark può essere sfruttare l'opzione di streaming con SparkStreaming. Il codice illustrato precedentemente imposta un flusso di stream testuale in una cartella dell'HDFS, così facendo ogni volta che viene aggiunto un file sarà processato in micro-batch, effettuando il conteggio del numero di occorrenze delle parole e aggiornando il precedente stato. Vengono mantenuti così i risultati continuamente aggiornati appena un nuovo file entra nel sistema. Un altro caso simile poteva consistere nel conteggio di un flusso continuo di singole parole o frasi, invece che di file testuali, in tal caso sarebbe stato sufficiente sostituire la funzione ssc.textFileStream con ssc.socketTextStream, specificando IP e porta di ascolto. Con questa modalità vengono creati job ogni

due secondi, come specificato nello StreamingContext nel codice. Se non è presente nessun nuovo file, il job non contiene nessuna map tasks ma solo task di stampa a video, altrimenti viene generata una task con lo scopo di mappare il nuovo file arrivato. Solitamente per file di circa 1MB vi è un tempo di esecuzione di 0.2 secondi circa, mantenendo un buon ritmo di elaborazione.

3.4.2 Modalità multi-node

La sperimentazione è stata fatta anche su un piccolo cluster, tramite Amazon Web Services. Lo scopo principale è comprendere il funzionamento di Apache Hadoop e Apache Spark all'interno di un cluster che comprende più nodi, distribuendo le task tra essi. AWS fornisce risorse e servizi di computing on-demand sul cloud, con prezzi che variano in base al tempo di utilizzo delle macchine e alla loro potenza. Per prima cosa sono state create 5 istanze di macchine virtuali, tramite il servizio EC2 (Elastic Cloud Compute) di AWS. L'immagine utilizzata è Ubuntu Server 16.04 LTS (HVM), con un disco SSD; dimensione t2.micro, 1 GB di RAM, 1 CPU virtuale, memoria EBS, posizione macchine Oregon (US).
Dopo aver creato un cluster contenente un NameNode e quattro DataNode, il passo successivo è stato configurare un security group, per controllare il traffico sulle macchine. Successivamente si è generata una pem-key per accedere alle macchine ed è stato lanciato il cluster.
Infine si è configurato l'accesso tramite SSH per accedere al cluster da remoto, dando anche la possibilità al NameNode di comunicare con i DataNode senza password.
A questo punto dopo aver effettuato il login sulle macchine sono stati installati e configurati i framework Apache Hadoop e Apache Spark, dopo aver installato Java e Scala. Si è poi formattato il NameNode e fatto partire l'HDFS con i comandi:
hdfs namenode -format e $HADOOP_HOME/sbin/start-dfs.sh
la UI è disponibile alla porta 50070 del DNS pubblico del NameNode. Infi-

ne è stato lanciato YARN e il MapReduce JobHistory Server con i comandi: $HADOOP_HOME/sbin/start-yarn.sh e $HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver; la UI è disponibile alla porta 8088 del DNS pubblico del NameNode e la JobHistory alla porta 19888.
L'utilizzo dei framework e i programmi eseguiti sono del tutto identici a quelli spiegati in dettaglio per il single-node, con la differenza che l'esecuzione avverrà su un cluster di 5 macchine invece che su una singola.
Una prima cosa che si nota dalla *Figura 3.2* sono i tempi di esecuzione,

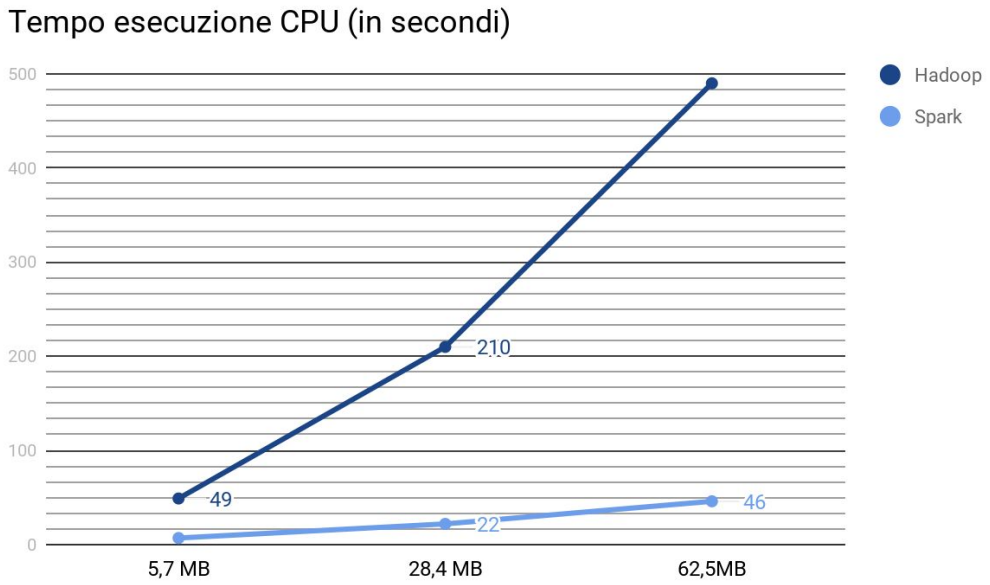

Figura 3.2: Tempi esecuzione Hadoop vs Spark (multi-node)

ben più lunghi rispetto alla sperimentazione su una singola macchina. I motivi del rallentamento sono vari, prima di tutto la performance delle macchine utilizzate nel cluster non è particolarmente eccellente, sia come I/O su disco che come network overhead. Un'altro motivo di rallentamento è dato dal tempo impiegato dalla parallelizzazione dei job, oltre che al tempo richiesto per trasmettere i dati tra i nodi durante la fase di shuffle.

L'esecuzione può essere stata rallentata anche dallo spreco di risorse dato dall'HDFS, precisamente la dimensione dei blocchi è 128MB e se si hanno molti singoli file, come in questo caso, di dimensioni ben minori vi è uno spreco di risorse. In generale l'esecuzione di una task implica di natura un overhead che va da 1 a 3 secondi, all'aumentare delle task tale overhead incide notevolmente. La vera efficienza di Hadoop e Spark si nota quando si elaborano molti Gigabyte di dati, o ancora meglio Terabyte. Con una tale mole di dati l'esecuzione su cluster avrebbe una performance estremamente migliore rispetto al single-node.
Interessante invece è stata la suddivisione delle task. Il numero di map tasks è dato dal numero di file di input, precisamente 9 per il primo, 45 per il secondo e 90 per il terzo. Il numero di reduce task invece è stato impostato manualmente a 3 questo ha permesso una vera parallelizzazione dell'esecuzione, per tutte le tasks.

Per quanto riguarda l'esecuzione di SparkStreaming su cluster non vi sono state grosse differenze rispetto all'esecuzione single-node. Un leggero rallentamento si è manifestato, probabilmente dato dall'overhead generato all'interno del cluster. La performance migliore rispetto all'esecuzione batch è probabilmente data dalla presenza di micro-batch di piccole dimensioni, eseguibili in-memory all'interno di un singolo nodo. Infatti se i dati raccolti nei 2 secondi di attesa, all'interno del micro-batch assumono dimensioni più ampie, in caso di esecuzione single-node il sistema non è sempre in grado di elaborarli, mentre l'esecuzione su cluster divide il contenuto tra i nodi riuscendo ad elaborare i dati, anche se con un tasso di elaborazione rallentato dall'overhead. Nel complesso quindi si suggerisce un'esecuzione su cluster di SparkStreaming se i dati che entrano nel sistema durante l'intervallo fissato sono di dimensioni maggiori di quelle che un'impostazione single-node possa analizzare.

3.5 Conclusioni

Con questa sperimentazione si è voluto analizzare il processo di elaborazione batch e in parte streaming, partendo dall'installazione di due famosi framework: Apache Hadoop e Apache Spark, commentando il codice prodotto per entrambi e confrontando alcuni dati riguardanti le prestazioni. Si è sperimentato sia in locale, su una singola macchina che su un piccolo cluster utilizzando il cloud AWS. Sebbene non siano state effettuate elaborazioni particolarmente complicate, ne siano state utilizzate macchine potenti, sono risultate chiare le principali features dei framework: il motore MapReduce, il file system HDFS, il gestore di risorse YARN, l'analisi streaming con SparkStreaming, il codice Java e Scala.
L'installazione dei framework è risultata tutto sommato semplice, più lunga per Apache Hadoop in quanto necessario settare vari parametri di configurazione, oltre che all'HDFS; immediata invece per Apache Spark. Per quanto riguarda la produzione di codice per i due framework, Apache Hadoop ha il vantaggio di avere un forte supporto Java, anche se la scrittura di programmi risulta più macchinosa e meno immediata. Apache Spark invece fornisce molte librerie Scala che permettono di produrre codice compatto e facilmente comprensibile.
Le performance variano molto in base alla mole di dati, l'esecuzione in single-node ha prodotto risultati migliori per quanto riguarda i tempi di utilizzo della CPU rispetto all'esecuzione su cluster, ma questo è dovuto solamente alle esigue dimensioni dei dataset analizzati. Se invece che pochi Megabyte di dati fossero stati molti Gigabyte o Terabyte l'elaborazione su cluster avrebbe avuto performance ben più competitive.
Nel caso di elaborazione stream le performance sono state più eque, anche se si è notato un normale overhead all'interno del cluster quando le dimensioni dei micro-batch richiedevano un'elaborazione distribuita su più nodi. Nel complesso la modalità streaming ha soddisfatto le aspettative.
La sperimentazione aveva uno scopo di supporto al confronto tra i fra-

mework di analisi di big data, citati nella prima parte. Si è infatti voluto arricchire l'analisi dei framework fornendo qualche breve esempio sperimentale per consolidare le conoscenze acquisite.

Bibliografia

[1] Jeffrey, Dean; Sanjay, Ghemawat *MapReduce: Simplified Data Processing on Large Clusters* 2004: Google, Inc.

[2] Wickham, Hadley *The split-apply-combine strategy for data analysis* 2011: Journal of Statistical Software.

[3] Ullman, J. D. *Designing good MapReduce algorithms* 2012: XRDS: Crossroads, The ACM Magazine for Students. Association for Computing Machinery.

[4] Sverdlik, Yevgeniy *Google Dumps MapReduce in Favor of New Hyper-Scale Analytics System* 2014: Data Center Knowledge.

[5] Czajkowski, Grzegorz; Marián, Dvorský; Jerry, Zhao; Michael, Conley *Sorting Petabytes with MapReduce – The Next Episode* 2011: Google, Inc.

[6] De Mauro, Andrea; Greco, Marco; Grimaldi, Michele *A Formal definition of Big Data based on its essential Features* 2016

[7] Hilbert, Martin *Big Data for Development: A Review of Promises and Challenges. Development Policy Review* martinhilbert.net

[8] Tejada, Zoiner; Wasson, Mike *Batch processing* 2017: docs.microsoft.com

[9] The Apache Software Foundation *Welcome to Apache Hadoop!* hadoop.apache.org

[10] The Apache Software Foundation *HDFS Users Guide* hadoop.apache.org

[11] The Apache Software Foundation *HDFS Architecture* hadoop.apache.org

[12] Jiong, Xie; Shu, Yin; Xiaojun, Ruan; Zhiyang, Ding; Yun, Tian; James, Majors; Adam, Manzanares; Xiao, Qin *Improving MapReduce performance through data placement in heterogeneous Hadoop Clusters* 2010: eng.auburn.ed

[13] Kawa, Adam *Introduction to YARN* 2014: ibm.com

[14] Cabot Technology Solution *5 Use Cases of Hadoop that makes it an Asset to Enterprises* 2017: hackernoon.com

[15] Tejada, Zoiner; McCready, Mary; Wasson, Mike *Real time processing* 2017: docs.microsoft.com

[16] The Apache Software Foundation *Components of a storm cluster* storm.apache.org

[17] Gopinath, Sudhaa *Apache Storm Use Cases* 2015: edureka.co

[18] Kleppmann, Martin *Apache Samza* 2018: Encyclopedia of Big Data Technologies

[19] Shenoy, Roopesh *How LinkedIn Uses Apache Samza* 2014: infoq.com

[20] Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J.; Shenker, Scott; Stoica, Ion *Spark: Cluster Computing with Working Sets* University of California, Berkeley

[21] Zaharia, Matei; Chowdhury, Mosharaf; Das, Tathagata; Dave, Ankur; Ma, Justin; McCauley, Murphy; Franklin, Michael J.; Shenker, Scott; Stoica, Ion *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing* University of California, Berkeley

[22] The Apache Software Foundation *Cluster Mode Overview* 2014: spark.apache.org

[23] Laskowski, Jacek *Mastering Apache Spark* jaceklaskowski.gitbooks.io

[24] Woodie, Alex *Apache Spark: 3 Real-World Use Cases* 2014: datanami.com

[25] The Apache Software Foundation *Apache Flink* flink.apache.org

[26] The Apache Software Foundation *Process Unbounded and Bounded Data* flink.apache.org

[27] Data Flair *Apache Flink Use Cases* 2016: data-flair.training

Ringraziamenti

Vorrei ringraziare il prof. Zavattaro, relatore di questa tesi di laurea, oltre che per gli ottimi spunti e l'importante supporto fornitomi durante la stesura, anche per avermi fatto apprezzare questa meravigliosa disciplina tanto da proseguire gli studi.

Un grazie anche alla mia famiglia che mi ha sempre sostenuto nelle mie scelte, in particolare a mia sorella che sta iniziando ora il suo percorso di studi universitario, spero sempre di rappresentare per lei un buon modello da seguire.

Grazie poi a tutte le persone che tengono a me e mi sono state vicine in questi anni, ognuna ha avuto un ruolo fondamentale ed insostituibile senza il quale il mio percorso sarebbe stato sicuramente più faticoso.