Architettura Hadoop

Gabriele Savoia

Sommario

In questo trattato sono riportati i concetti fondamentali dell'architettura di Apache Hadoop. Dopo una prima sezione introduttiva mirata a descrivere le generalità del sistema, sono riportate due sezioni dedicate alla descrizione delle sue componenti principali come HDFS e MapReduce (con anche YARN). Nel paragrafo conclusivo infine è descritta brevemente l'applicazione pratica realizzata, inerente all'utilizzo di MapReduce per la creazione di un InvertedIndex di base.

1 Apache Hadoop

Apache Hadoop è un framework open-source le cui funzionalità di storage e processing distribuito su cluster di commodity hardware hanno contribuito a renderlo uno dei sistemi più utilizzati per la gestione e l' elaborazione scalabile di quantitativi di dati molto elevati.

La prima versione di Hadoop è stata sviluppata da Doug Cutting e Mike Cafarella nel 2005 che, basandosi sui concetti del Google File System [4] e della Google MapReduce [2], si sono posti l' obiettivo di creare un sistema in grado di fornire supporto al search engine Nutch. Ad oggi Hadoop è un progetto open-source dell' Apache Software Foundation ed è utilizzato in alcune delle realtà aziendali più importanti del momento come Ebay, Facebook, LinkedIn e Yahoo.

Mantenendo una visione generica, si può pensare all'architettura di Hadoop come composta da due moduli principali: `HDFS` (o Hadoop Distributed File System) ovvero un file system distribuito per la memorizzazione efficiente di una grande mole di dati su cluster; `MapReduce` che rappresenta un framework per l'elaborazione scalabile dei dati memorizzati.

Nel corso del tempo comunque, l'architettura Hadoop ha subito diversi cambiamenti ma uno dei più rilevanti è rappresentato dal passaggio [3] dalla versione 1 alla 2. Come riportato in figura 1, se con la prima era presente un unico blocco MapReduce sia per la gestione del cluster che per il processamento dei dati, con la seconda versione è stato introdotto YARN (Yet Another Resource Negotiator), ovvero un apposito cluster manager focalizzato sulla schedulazione dei job e la gestione delle risorse del cluster. Grazie alla sua interfaccia generica, YARN ha permesso così a diverse altre applicazioni (non solo MapReduce) di poter usufruire dell'infrastruttura Hadoop, portando così alla creazione di un complesso ecosistema (riportato in figura 2) costituito da un rilevante numero di progetti.

Nato in principio come un efficiente motore di elaborazione MapReduce, Hadoop risulta quindi, ad oggi (versione 3), una piattaforma big data completa i cui moduli principali sono rappresentati da:

- `HDFS`: è l' Hadoop Distributed File system il cui ruolo principale è quello di memorizzare i dati in un file system distribuito;
- `MapReduce`: applicazione dedicata all'implementazione del paradigma MapReduce in un contesto Hadoop distribuito;
- `YARN`: (Yet Another Resource Negotiator) che ricopre il ruolo di cluster resoruce manager.

Nelle sezioni seguenti sono riportati i concetti principali per ciascuno dei moduli sopra riportati.

2 HDFS

Apache HDFS (Hadoop Distributed File System) [6], è un filesystem distribuito appositamente progettato per essere eseguito su cluster di commodity hardware. Rappresenta uno dei core component dell'architettura Hadoop ed ha un ruolo fondamentale per lo storage di volumi di dati molto grandi tramite l'utilizzo di hardware a basso costo.

Questo file system distribuito risulta essere fortemente `fault-tolerant` ovvero in grado di rilevare e rimediare tempestivamente a guasti hardware tramite metodologie di `recovery automatico`

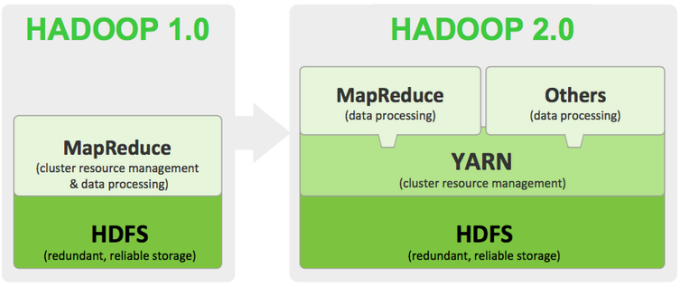

Figura 1: Architettura Hadoop 1 comparata con architettura Hadoop 2

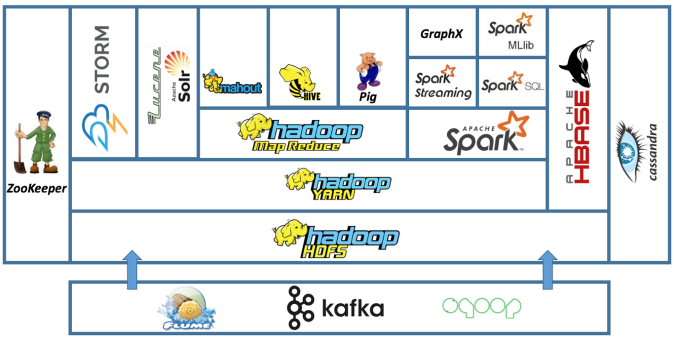

Figura 2: Ecosistema Hadoop

e `data replication`. Inoltre, essendo un sistema `scalabile` (fino a migliaia di nodi per cluster) che rispetta il modello di accesso ai file di tipo `write-once-read-many`[1], risulta fortemente ottimizzato per la gestione di file di grandi dimensioni (tipicamente nell'ordine di giga o terabyte), riuscendo così a fornire alle applicazioni `throughput elevati` per l'accesso ai dati memorizzati. Permette inoltre un accesso in `streaming` ai dati del file system e mette a disposizione funzionalità di `balancing` in grado di distribuirli in maniera efficiente per la loro gestione ed elaborazione. Uno dei principi di base dell'HDFS si basa sul fatto che "spostare la computazione è meno costoso che spostare i dati". Ovvero che la computazione richiesta da una generica applicazione è più efficiente nel caso in cui sia eseguita "vicino" ai dati su cui opera, portando ad una riduzione della probabilità di congestione della rete e ad un aumento delle prestazioni del sistema.

2.1 Architettura HDFS

Hadoop Distributed File System, è caratterizzato da un'architettura master-slave in cui ciascun file presente è diviso in blocchi di una dimensione prestabilita. In particolare, (come riportato in figura 3) è possibile identificare il NameNode (master) e una serie di DataNode (slave). Dal momento che HDFS è scritto nel linguaggio di programmazione Java, qualsiasi macchina che ne supporta l'esecuzione può potenzialmente essere utilizzata come NameNode o DataNode.

`NameNode`: un cluster HDFS è costituito da un singolo NameNode il quale rappresenta il processo master che gestisce i DataNode (slave). In particolare, oltre a memorizzare i metadati dei file del cluster come la locazione dei blocchi salvati (quindi anche della loro replicazione), i permessi etc. tiene anche traccia di tutte le modifiche fatte nel file system come eliminazioni o modifiche. Esistono principalmente due file associati ai metadati: `FsImage` che memorizza lo stato del file system namespace (con anche la mappatura dei blocchi ai nodi); `EditLogs` (anche chiamato transaction log) contenente le modifiche più recenti riferite al file system come ad esempio la creazione di un nuovo file. Il NameNode ha inoltre un ruolo fondamentale nella gestione e rilevazione dei guasti hardware, infatti riceve regolarmente un heartbeat da parte di tutti i DataNode e nel caso di fallimento, è responsabile della scelta dei nuovi nodi per la replicazione e il re-balancing dei dati.

`DataNode`: rappresentano i nodi slave e sono responsabili, sulla base delle istruzioni ricevute dal NameNode, delle operazioni di lettura (o scrittura) effettiva dei dati nel proprio file system locale. Per fare in modo che il NameNode riesca a monitorare lo stato di ciascun DataNode, questo

[1] WORM: modalità di accesso ai file in HDFS la quale assume che un file una volta scritto non sarà più modificato, ma può comunque essere letto più volte

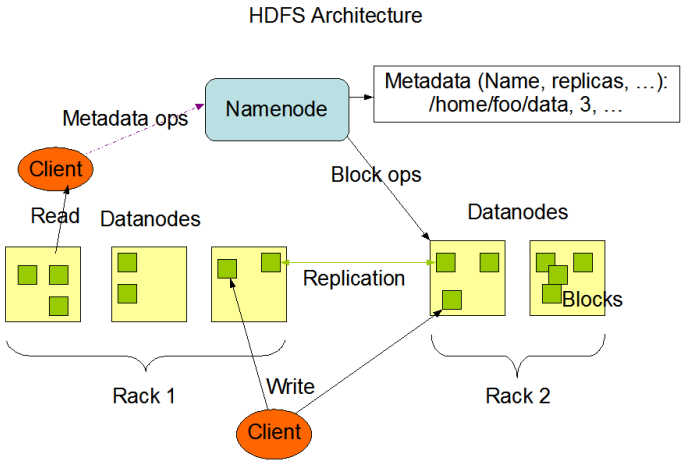

Figura 3: Architettura HDFS

è responsabile di inviare periodicamente (di default ogni 3 secondi) un heartbeat per comunicare della sua disponibilità;

`SecondaryNameNode`: si tratta di un processo helper per il NameNode che legge periodicamente EditLog e FsImage da disco così da applicare tutte le transazioni presenti nell'EditLog alla rappresentazione in-memory FsIimage per poi scriverla su disco così da garantire all'HDFS una visione consistente dei metadati del file system. Questo processo (anche chiamato checkpoint), può essere eseguito periodicamente a distanze temporali prefissate, oppure ogni volta che viene raggiunto un certo numero di transazioni del file system.

In generale comunque, la comunicazione tra i diversi nodi è realizzata tramite Remote Procedure Call (RPC), ovvero un'astrazione che gestisce il Client Protocol e il DataNode Protocol al di sopra del protocollo TCP/IP.

2.2 Organizzazione e replicazione dei dati

HDFS è appositamente progettato per lavorare con file di grandi dimensioni, in particolare implementando una modalità di accesso di tipo write-once-read-many, permette letture multiple ed efficienti, assumendo però che i dati applicativi siano scritti una sola volta. Questa tipologia di file system lavora con blocchi di dimensione di 128 MB (eventualmente modificabile), portando così un generico file ad essere suddiviso in chunk di questa dimensione per poi essere distribuiti su DataNode diversi nel cluster.

Questa metodologia di storage affidabile dei dati, unita a tecniche di replicazione, porta ad aumentare il grado di tolleranza ai guasti del sistema. Di default infatti, HDFS utilizza un `replication factor` pari a 3, in cui ciascun blocco dati viene replicato 3 volte tra i DataNode del cluster, i quali poi forniscono periodicamente (insieme all'heartbeat) i cosiddetti `BlockReport` al NameNode che contengono una lista di tutti i blocchi presenti nel nodo. Il NameNode sarà poi responsabile di gestire la replicazione di ciascun blocco in modo da renderla coerente con il valore del replication factor impostato.

La modalità di replicazione dei blocchi gioca un ruolo fondamentale in termini di affidabilità e performance del sistema. HDFS infatti, utilizza una particolare policy di replicazione `rak-awareness`[10] la quale ottimizza la replicazione dei blocchi tenendo in considerazione la topologia del cluster con l'obiettivo di aumentare la disponibilità dei dati e migliorare le performance del sistema. Si basa principalmente sul concetto che non deve essere presente più di un blocco di replica nello stesso nodo e che nello stesso rak[2] non devono esistere più di 2 repliche. In questo modo, nel caso in cui il replication factor sia pari a 3, la prima replica avverrà su un rak locale, un'altra replica su un nodo in un rak diverso e remoto, mentre l'ultima in un altro nodo nello stesso rak remoto.

Nell'esempio riportato in figura 4, è rappresentato il posizionamento delle repliche riferite al Block1 con policy rak-awareness e si può vedere come sia soddisfatta la data availability (se il Rak1 fallisce, sono presenti repliche del Block1 nel Rak2) ed inoltre avviene anche un miglioramento delle

[2]Un rak è una collezione di DataNode connessi dallo stesso switch, di conseguenza, se si dovesse verificare un gusato alla rete, l'intero insieme dei nodi risulta non più disponibile.

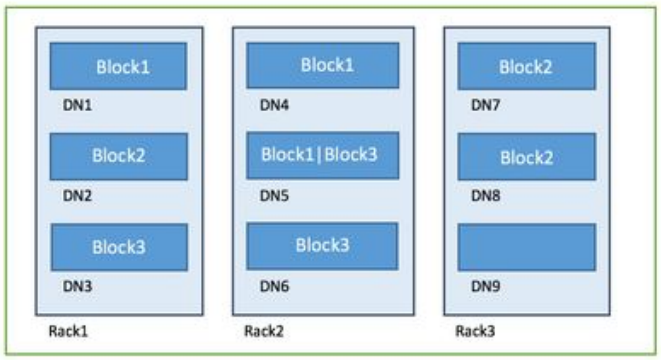

Figura 4: Posizionamento delle repliche del blocco 1 con policy rak-awareness

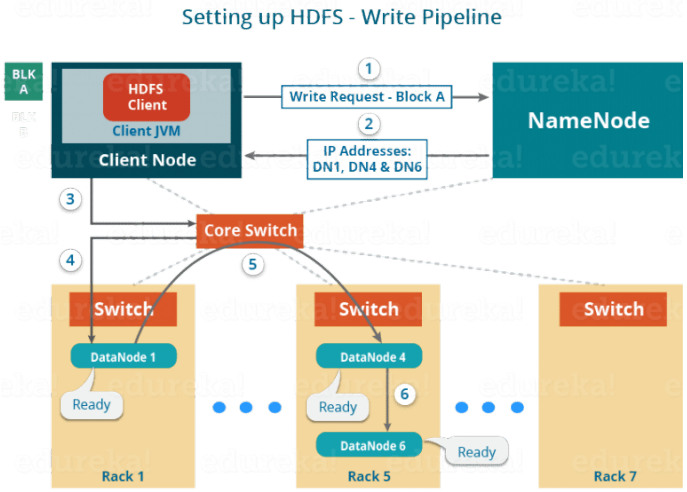

Figura 5: Step principali per la scrittura di un singolo blocco in HDFS tramite pipeline.

performance in quanto 2 repliche sono posizionate nello stesso rak (2 repliche del Block1 nel Rak2) e quindi viene utilizzata una larghezza di banda maggiore.

2.3 Operazioni di lettura e scrittura

Senza entrare nei dettagli specifici, di seguito sono riportati i macro-passaggi principali richiesti dalle operazioni di lettura e scrittura [1] in HDSF.

`Write`: supponendo che un HDFS Client voglia effettuare una scrittura di un certo file, gli step necessari sono riportati di seguito (rappresentati anche in figura 5) :

1. il Client HDFS, dopo aver suddiviso il file da scrivere in blocchi (128 MB), contatta il NameNode con una Write Request;

2. dopo aver verificato il Client, il NameNode ritorna `per ogni blocco` una `lista` di indirizzi IP, corrispondenti ai DataNode di memorizzazione, di lunghezza pari al replication-factor (3 di default) e calcolati tenendo conto della disponibilità dei DataNode e al rak-awareness;

3. per ogni blocco da scrivere, è creata un'apposita `pipeline` (definita ad esempio dai DataNode 1, 4 e 6 per il blocco A in figura). Quando i nodi all'interno della stessa hanno confermato la loro disponiblità nel ricevere i dati, il client trasmetterà il blocco da scrivere solamente al `primo DataNode` per ogni lista (tramite connessione TCP instaurata per la creazione della pipeline), e la replicazione del blocco avverrà in maniera `sequenziale` (e trasparente per il client) tra i nodi della pipeline. Nel caso di scrittura di più blocchi, le relative pipeline sono eseguite in parallelo.

Dopo aver ricevuto le informazioni sulla corretta terminazione di questi passaggi, il client interromperà sia la connessione TCP che la pipeline ed infine il NameNode aggiornerà i metadati così da essere consistente con la nuova operazione eseguita.

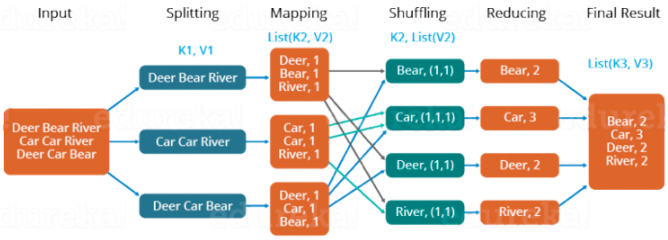

Figura 6: Tipico job MapReduce per il problema di "word count".

`Read`: sulla base della richiesta di lettura di un file da parte del client, il NameNode ritorna la lista di DataNode in cui risiedono i blocchi, considerando però solo le `repliche più vicine` al client (preferibilmente nello stesso rak del nodo del reader).

3 MapReduce e YARN

MapReduce [5] rappresenta il framework Hadoop per il processing distribuito di una mole di dati molto elevata perfettamente integrato con l'HDFS Hadoop. In sostanza implementa un paradigma di computazione concettualmente simile allo `split-apply-combine` e caratterizzato da due macro-operazioni: `map` e `reduce`. Se con il primo si genera come risultato intermedio un insieme di coppie chiave-valore, con la reduce queste vengono aggregate così da fornire un output finale composto da un insieme di coppie chiave-valore più piccolo.

Come riportato in figura 6, un tipico `job` MapReduce è composto da diversi step. Dopo aver effettuato uno `split` dei dati in chunk indipendenti, questi sono poi elaborati in maniera parallela dai `task di mapping` (generazione di coppie chiave-valore). Questi dati intermedi (dopo una fase di `sorting` e `shuffling`) saranno infine aggregati parallelamente tramite i `task di reduce`.

Essendo quindi che il framework MapReduce applica questo modello di computazione in un contesto di cluster di computer distribuiti, è facile capire come le performance del sistema siano nettamente superiori rispetto alle implementazioni su macchine singole, rendendo molto utile così il suo utilizzo in ambito big data.

In termini di architettura, i nodi del cluster sono responsabili sia della memorizzazione che della computazione dei dati, rendendo così questo framework altamente performante e concorde al principio secondo cui "muovere la computazione è meno costoso che muovere i dati".

Come spiegato anche in sezione 1, nel passaggio dalla versione 1 alla versione 2 di Hadoop, l'introduzione del gestore delle risorse YARN ha rivoluzionato il vecchio modulo MapReduce permettendo di dividere completamente la parte di gestione delle risore del cluster con la parte di computazione.

3.1 Esecuzione MapReduce

Per l'esecuzione di un generico `job` in Hadoop MapReduce (versione 1) [8], entrano in gioco principalmente 2 componenti: un `JobTracker` (master) e diversi `TaskTracker` (slave). In sostanza il JobTracker svolge sia la funzionalità di gestione delle risorse che di job scheduling e monitoring, mentre i TaskTracker si occupano principalmente dell'esecuzione dei task di map e reduce. La logica generica adottata dal JobTracker è quella di far eseguire i TaskTracker nei nodi più vicini possibile a quelli in cui sono memorizzati i dati da utilizzare (se non è possibile nello stesso nodo, allora si sceglie un nodo nello stesso rak di quello contenente i dati). I TaskTracker mandano poi periodicamente heartbeat al JobTracker così da comunicare il proprio stato di attività ed intervenire nel caso di malfunzionamenti.

Come riportato anche in figura 7, il generico flusso di lavoro è il seguente :

1. il client sottomette il job MapReduce al JobTracker;
2. il JobTracker crea gli appositi TaskTracker per l'esecuzione dei task map e reduce;
3. `map`: tutti i task di map sono tra loro indipendenti (lavorano su split di dati distinti memorizzati nell'HDFS) e per questo predisposti all'esecuzione parallela da parte dei TaskTracker. Dopo la fase di map è eventualmente presente una fase aggiuntiva definita `Combiner`, conosciuta anche come "Semi-Reducer" e si pone tra la map e la reduce con l'obiettivo di elaborare l'output della map così da ridurre la congestione della rete;

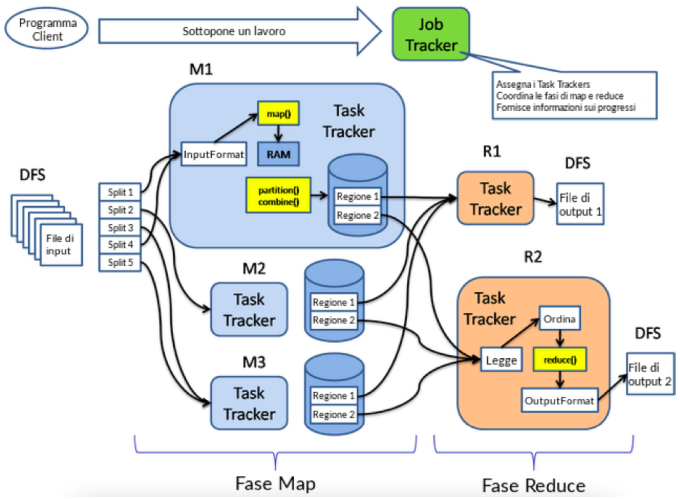

Figura 7: Esecuzione di un job MapReduce

4. gli output intermedi della map (salvati in memoria secondaria), vengono trasferiti alla fase di reduce dopo le operazioni di sorting e shuffle;

5. `reduce`: ciascun TaskTracker elabora, aggrega e scrive su HDFS l'output ricavato dai relativi task di map (la reduce è eseguita solo dopo che tutte le operazioni di map sono terminate). Anche in questo caso, per via della loro indipendenza, i task sono eseguiti parallelamente.

E' possibile notare quindi come questa tipologia di esecuzione sia caratterizzata da un numero molto elevato di letture e scritture su disco. Ciò comporta da un lato una riduzione delle performance generali del sistema, ma dall'altro lo rende adatto a lavorare con quantitativi di dati potenzialmente molto elevati. Inoltre, l'utilizzo di un singolo JobTracker porta all'introduzione di un `bottleneck` con conseguenti problemi di scalabilità sia in termini di dimensioni del cluster che del numero di applicazioni da eseguire.

3.2 YARN

Se con la versione 1 di MapReduce, sia la gestione delle risorse che la parte computazionale sono eseguite da un singolo engine monolitico, dalla versione 2 [9], il motore di esecuzione MapReduce viene separato dal gestore delle risorse. E' stato infatti introdotto YARN (Yet Another Resource Negotiator)[12] ovvero un sistema per il resource management del cluster completamente generico, in cui è possibile eseguire "applicazioni" di diversa tipologia e non solamente job MapReduce. Come riportato in figura 8, gli elementi [11] che lo costituiscono sono i seguenti: Container, ResourceManager, NodeManager e ApplicationMaster ed in particolare:

- `Container`: rappresenta l'astrazione in cui viene eseguita concretamente una certa unità di lavoro (ad esempio un task riferito al job MapReduce). In generale un singolo nodo può eseguire diversi Container e a ciascuno sono associate determinate risorse in termini di CPU e RAM;

- `ResourceManager`: è il principale daemon (master) di YARN ed è composto da uno `Scheduler` e da un `ApplicationManager`. Con il primo vengono allocate e gestite le risorse alle diverse applicazioni, con la particolarità però che `non sono gestiti i singoli task` delle stesse (compito delegato all'ApplicationMaster). Tramite l'ApplicationManager invece, vengono accettati i job sottomessi dal client, è gestita la negoziazione per la creazione del primo Container riferito ai possibili ApplicationMaster di ciascuna applicazione ed inoltre avviene anche il loro monitoraggio così da intervenire in caso di fallimenti;

- `NodeManager`: si tratta di un servizio daemon (slave) in esecuzione sui vari nodi del cluster che provvede alla creazione e alla gestione dei Container applicativi. Il NodeManager comunica periodicamente (heartbeat) il proprio stato con il ResourceManager in modo da permettergli di tener traccia dello stato globale del sistema;

- `ApplicationMaster`: viene istanziato un ApplicationMaster da parte del ResourceManager per ogni applicazione che si vuole eseguire. In particolare chiede le risorse al ResourceManager

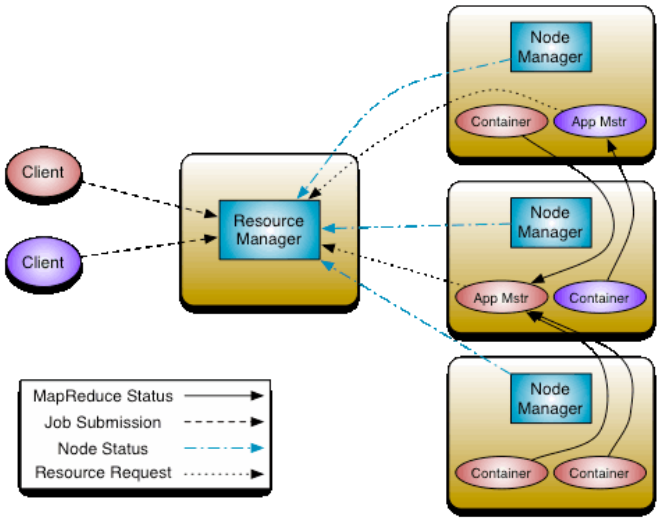

Figura 8: Esecuzione Yarn

e lavora con i NodeManager al fine di eseguire, monitorare e gestire (in caso di fallimenti) i singoli task (es. task di map e reduce) di un'applicazione.

Con l'utilizzo di YARN quindi, le funzionalità di gestione delle risorse e job scheduling sono gestite separatamente dai daemon ResourceManager e ApplicationMaster, alleggerendo così il carico di lavoro sul gestore delle risorse ed ottenendo una maggiore scalabilità rispetto al JobTracker della versione 1. Per risolvere il problema del single-point-of-failure riferito al ResourceManager, dalla versione 2.4 di Hadoop sono stati introdotti i concetti di `Active/Standby ResourceManager (RM)` [7]: se un Active RM fallisce, allora un Standby RM lo rimpiazza (avviene una sincronizzazione tra i due tipi di RM), soddisfando così la caratteristica di high-availability del ResourceManager. Inoltre, con le nuove versioni di YARN, è anche disponibile la funzionalità di `federation` con cui diventa possibile unire diversi cluster YARN tra loro in modo da considerarli come uno unico e portando così il sistema a scalare su una quantità di nodi molto elevata (passando da circa 10000 a qualche centinaia di migliaia).

4 Conclusioni

In questo trattato sono stati quindi riportati i concetti principali dell'architettura Hadoop. In aggiunta a questa parte più teorica, è stata anche implementata una versione basilare di un job MapReduce responsabile della creazione di un InvertedIndex in funzione di un dataset di documenti iniziale. In particolare, dopo aver installato e configurato hadoop in locale, è stata utilizzata la libreria MRJob che permette, tramite Hadoop Streaming, di utilizzare il linguaggio Python per la creazione di job MapReduce sia in locale che su cluster Hadoop. In questo caso la parte di mapping genera in output coppie del tipo (word, docId), mentre con la reduce è generato il risultato finale in cui per ciascuna parola è presente una posting list contenente i documenti di cui questa fa parte (senza duplicati). In riferimento a questo job, sono poi stati monitorati sia HDFS che YARN tramite le relative interfacce web.

Nonostante si tratti solamente di un'applicazione di prova con un dataset "giocattolo" comunque mostra le potenzialità e i benefici che questo tipo di architettura è in grado di fornire in un contesto distribuito in cui si ha la necessità di elaborare una mole di dati molto elevata.

Riferimenti bibliografici

[1] *Architecture with read and write operation HDFS.* https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/.

[2] Jeffrey Dean e Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[3] Douglas Eadline. *Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem*. Addison-Wesley Professional, 2015.

[4] Sanjay Ghemawat, Howard Gobioff e Shun-Tak Leung. "The Google File System". In: *SIGOPS Oper. Syst. Rev.* 37.5 (ott. 2003), pp. 29–43. ISSN: 0163-5980. DOI: 10.1145/1165389.945450. URL: https://doi.org/10.1145/1165389.945450.

[5] *Hadoop MapReduce*. https://hadoop.apache.org/docs/r3.0.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[6] *HDFS Architecture*. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[7] *High Availability Resource Manager*. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html.

[8] *MapReduce1*. https://cwiki.apache.org/confluence/display/HADOOP2/JobTracker.

[9] *MapReduce2 with YARN*. https://docs.cloudera.com/cdp-private-cloud-upgrade/latest/upgrade-cdh/topics/yarn-major-changes-mrv2.html.

[10] *Rak-Awarness in Hadoop*. https://community.cloudera.com/t5/Community-Articles/Rack-Awareness/ta-p/246584.

[11] *YARN components*. https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860.

[12] *YARN official documentation*. https://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-site/YARN.html.