Big Data

Introduzione

Argomenti della lezione

- L’evoluzione dei dati
- Cosa è Big Data
- Introduzione ad Hadoop
- HDFS: Architettura e Funzionamento
- MapReduce
- YARN

L’evoluzione dei dati

Negli anni i dati, la loro gestione, ma soprattutto i processi di analisi volti a trasformare i dati in informazioni hanno subito un’evoluzione.

Negli anni Sessanta le uniche analisi che potevano essere svolte erano statiche e si limitavano alla sola estrazione dei dati raccolti. Con l’avvento dei database relazionali e del linguaggio SQL, negli anni Ottanta, l’analisi dei dati assume una certa dinamicità. Infatti l’SQL consente di estrarre in maniera semplice i dati, sia in modo aggregato, sia a livello di massimo dettaglio.

Le attività di analisi avvengono su basi di dati operazionali, ovvero sistemi di tipo OLTP (On Line Transaction Processing) caratterizzati e ottimizzati prevalentemente per attività di tipo transazionale piuttosto che per attività di lettura e di analisi di grandi quantità di record. La maggior parte di questi sistemi, tra l’altro, offre una storicizzazione dei dati limitata.

Queste limitazioni portano alla creazione di database progettati per l’integrazione dei dati e l’analisi, i Data Warehouse. Essi contengono dati integrati, consistenti e certificati relativi ai processi di business.

Tuttavia si tratta sempre di una visione storica, che consente soltanto una valutazione a consuntvo di ciò che è accaduto nel passato, oppure di ciò che sta accadendo ora. Più di recente, a partire dai primi anni Duemila, è emersa la necessità di effettuare analisi previsionali, per anticipare gli eventi e ottenere un vantaggio di business.

Cosa è Big Data

I big data rappresentano tutti quei dati che possono essere disponibili in enormi volumi, possono presentarsi con formati semistrutturati o addirittura destrutturati e possono essere prodotti con estrema velocità. Volume, varietà e velocità (Volume, variety, velocity) sono i fattori che caratterizzano i big data.

Uno degli aspetti che caratterizzano i big data è la loro quantità. Dati generati dall'utente attraverso gli strumenti del Web, sistemi gestionali, oppure dati generati automaticamente da macchine (IOT, sensori, strumenti scientifici) possono assumere volumi rilevanti, non più gestibili con strumenti di database tradizionali. Una valanga di dati viene generata ogni giorno, solo Twitter e Facebook generano più di 10 TeraByte (TB) di dati ogni giorno.

Con l’esplosione dei sensori, degli smartphone, degli strumenti del Web e dei social network i dati si sono “complicati”, ovvero non presentano più una struttura predefinita e quindi non sono più riconducibili ad uno schema tabellare, ma possono presentare un formato semistrutturato o destrutturato, non più rapplesentabile in modo efficiente in un database relazionale. La diversità di formati e, spesso, l’assenza di una struttura sono la seconda possibile caratteristica dei big data.

Non solo la varietà e il volume dei dati che vengono memorizzati sta cambiando, anche la velocità con cui i dati vengono generati sta cambiando e deve essere gestita. La velocità con cui i nuovi dati si rendono disponibili è il terzo fattore con cui è possibile identificare i big data. Oltre al volume, anche la velocità con cui le fonti generano nuovi elementi rende necessario l’utilizzo di strumenti in grado di tenerne il passo.

L’acquisizione dei Big Data può avvenire, a seconda del tipo di fonte, attraverso differenti mezzi, che è possibile suddividere in quattro categorie:

- API (Application Programming Interface)
- Strumenti di ETL
- Software di Web Scraping
- Lettura di stream di dati

Le **Application Programming Interface** sono protocolli utilizzati come interfaccia di comunicazione tra componenti software. In questa categoria rientrano sia i dati provenienti dalle fonti operazionali, sia i dati provenienti dal Web, in particolare dai social network.

Gli **strumenti di ETL**, utilizzati nei contesti di Business Intelligence e Data Warehousing, permettono di svolgere i processi di estrazione, trasformazione e caricamento dei dati, provenienti da fonti operazionali e destinati ai sistemi di Data Warehouse. Molti strumenti di ETL, ad oggi, sono già attrezzati per importare i dati, dai formati più disparati, nel sistema di gestione dei big data.

Apache ha rilasciato **Sqoop**, uno strumento open source progettato per estrarre e trasferire in modo efficiente dati strutturati da database relazionali (RDBMS) a HDFS (oppure Hive e HBase). Dualmente, una volta che i dati caricati su Hadoop sono stati elaborati, Sqoop è in grado di estrarre i dati da HDFS ed esportarli su database strutturati esterni.

Il **web scraping** è il processo attraverso il quale è possibile raccogliere automaticamente dati dal Web. Esistono diversi tipi di livelli di automazione; per esempio esistono software, come Apache Tika, oppure software per il parser di pagine HTML, e così via. Apache Tika è uno strumento scritto in Java per l'identificazione e l'estrazione di metadati e testo da numerosi documenti dai formati più diversi.

La velocità di produzione che caratterizza alcune tipologie di dati ha reso necessarie tecnologie per la cattura in tempo reale e il trasferimento continuo dei dati. Un esempio open source è **Apache Flume**, servizio distribuito per la raccolta, l'aggregazione e lo spostamento di grandi moli di dati. L'architettura si basa sul concetto di agent, cioè una componente software che al suo interno gestisce autonomamente la raccolta dei dati provenienti dall'esterno, il passaggio dei dati attraverso il canale, ed infine, la lettura dei dati dal canale e l'instradamento verso la sorgente di destinazione.

Introduzione ad Hadoop

Hadoop è un framework Open Source di Apache, affidabile e scalabile, finalizzato al calcolo distribuito di grandi quantità di dati. Esso presenta 3 componenti essenziali che costituiscono il nucleo centrale della piattaforma:

- **Hadoop Common**: rappresenta lo strato di software comune che fornisce le funzioni di supporto agli altri moduli;
- **HDFS (Hadoop Distributed File System)**: è il filesystem distribuito di Hadoop progettato appositamente per essere eseguito su *commodity hardware*;
- **MapReduce**: si occupa della schedulazione ed esecuzione dei calcoli. Lavora secondo il principio “divid aet impera”: un problema complesso, che utilizza una gran mole di dati, viene suddiviso, assieme ai relativi dati, in piccole parti processate in modo autonomo e, una volta che ciascuna parte del problema viene calcolata, i vari risultati parziali sono “ridotti” a un unico risultato finale.

Hadoop è un sistema:

- **altamente affidabile**: essendo pensato per un cluster di commodity hardware, che può essere frequentemente soggetto a problemi, permette di facilitare la sostituzione di uno o più nodi in caso di guasti.
- **scalabile**: la capacità computazionale del cluster Hadoop può essere incrementata o decrementata semplicemente aggiungendo o togliendo nodi al cluster.

Dal punto di vista architetturale in un cluster Hadoop non tutti i nodi sono uguali, ma esistono due tipologie di nodi:

- **master**
- **worker**

Sui primi vengono eseguiti i processi di coordinamento di HDFS e MapReduce; i secondi invece vengono utilizzati per la memorizzazione e il calcolo.

HDFS: Architettura e Funzionamento

Hadoop Distributed File System (HDFS) è stato progettato per la gestione dei flussi e memorizzazione affidabile di grandi volumi di dati; in particolare, ha lo scopo primario di gestire l'input e l'output dei job mapreduce.

Gli aspetti principali che lo caratterizzano sono:

- **Very Large Files**: non esiste un limite esplicito sulle dimensioni dei file contenuti al suo interno.
- **Streaming Data Access**: è particolarmente adatto per applicazioni che elaborano grandi quantità di dati. Questo perché il tempo che occorre per accedere all'intero set di dati è relativamente trascurabile rispetto al tempo di latenza dovuto alla lettura di un solo record;
- **Commodity Hardware**: è stato progettato per essere eseguito su cluster di commodity hardware, ovvero hardware a basso costo, in modo tale da aumentare la tolleranza ai guasti (faut-tolerance), molto probabili quando si ha a che fare con cluster di grandi dimensioni.

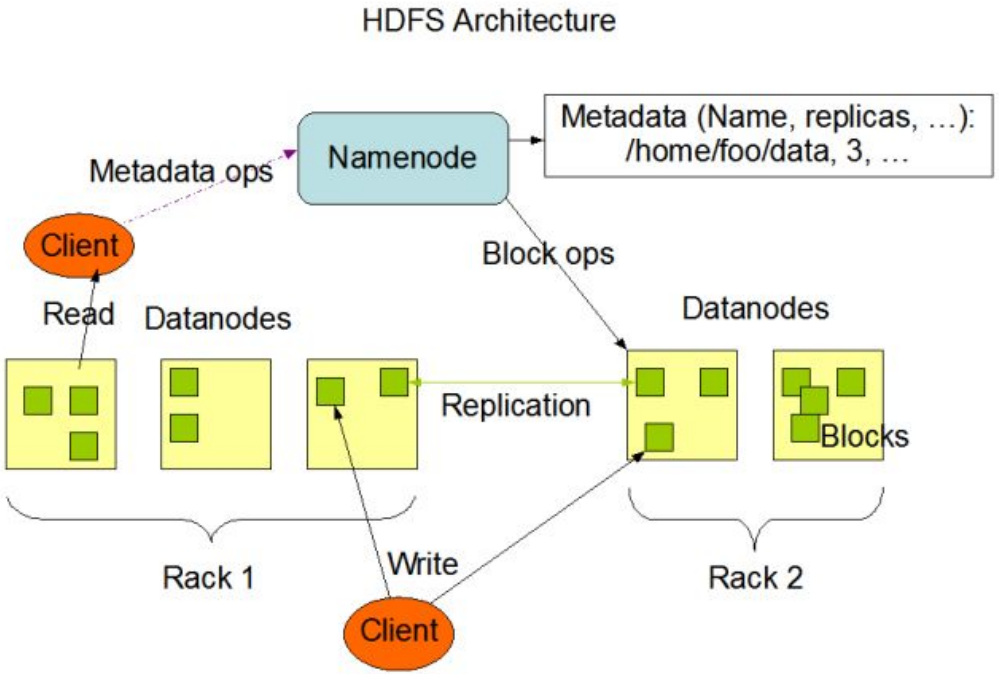
I file, all’interno di HDFS, vengono partizionati in uno o più blocchi (blocks), ognuno, di default da 128 MB (dimensione modificabile). Diversamente da altri filesystem, se un file risulta essere più piccolo della dimensione del blocco, non viene allocato un blocco “intero”, ma soltanto la dimensione necessaria al file in questione, risparmiando così spazio utilizzabile.

Affinché venga mantenuto un certo grado di tolleranza ai guasti (fault tollerance) e disponibilità (availability), HDFS prevede che i blocchi dei file vengano replicati e memorizzati, come unità indipendenti, fra i nodi del cluster. Se un blocco risulta non più disponibile, la copia che risiede su un altro nodo ne prende il posto, in modo completamente trasparente all’utente. Le repliche sono utilizzate sia per garantire l’accesso a tutti i dati, anche in presenza di problemi a uno o più nodi, sia per migliorare il recupero dei dati.

Sia la dimensione dei blocchi, sia il numero di repliche possono essere configurati dall’utente.

Ogni cluster Hadoop presenta due tipologie di nodi, che operano secondo il pattern **master-slave**. HDFS presenta un’architettura in cui un nodo **master** identifica il **NameNode** e un certo numero di nodi **slave** identificano i **DataNode**.

Come per molti altri **DFS** (Distributed File System), ad esempio **GFS** (Google File System), anche **HDFS** gestisce separatamente i dati applicativi dai metadati, questi ultimi vengono memorizzate su un server dedicato, chiamato **NameNode**, invece i dati applicativi vengono gestiti da altri server, detti **DataNode**. Ogni file in HDFS verrà suddiviso in più blocchi che possono risiere su diversi DataNode, ed è il NameNode che capisce come questi blocchi possono essere combinati per ricostruire i file.



Il **NameNode** gestisce lo spazio dei nomi (namespace) del filesystem, ovvero una struttura gerarchica di file e directory, sul quale vengono mappati tutti i singoli blocchi dei file presenti all'interno del filesystem. Per far si che il fattore di replicazione (di default pari a 3) di ogni blocco sia mantenuto, il NameNode memorizza per ognuno di questi, la lista dei datanode che ne possiedono una copia. Tale configurazione del namespace però, non è permanente, le informazioni relative alle replicazioni dei blocchi vengono ricostruite ad ogni avvio del sistema.

Il file principale scritto dal NameNode si chiama **fsimage**, ed è l'unico elemento fondamentale di tutto il cluster: senza di esso non si ha la possibilità di ricostruire i blocchi di dati nel file system. Questo file viene letto in memoria all'avvio del NameNode ma, se vengono effettuate delle modifiche dopo la sua esecuzione, non viene modificato. Le modifiche vengono scritte in un file chiamato edits che contiene tutte le modifiche apportate dall'ultima scrittura di fsimage. All'avvio di una istanza viene letto fsimage, poi viene letto il file edits e vengono applicate tutte le modifiche nella copia in memoria di fsimage. Infine viene scritta su disco la nuova versione di fsimage e il nodo è pronto a ricevere chiamate dal client.

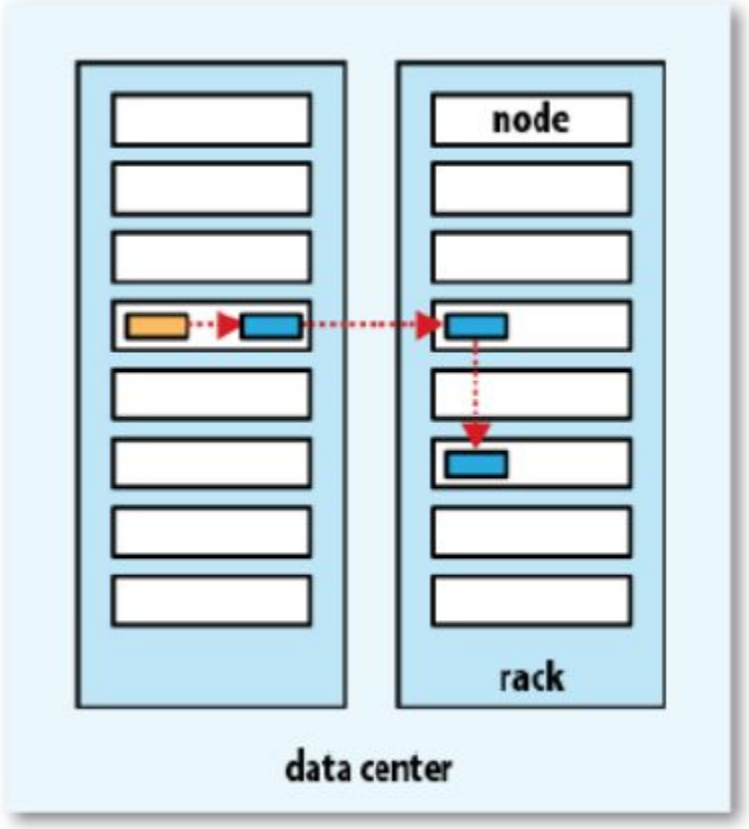
Senza il NameNode, il filesystem non sarebbe utilizzabile; se il nodo su cui è in esecuzione il NameNode “cade”, tutti i file contenuti nel filesystem non sarebbero raggiungibili perché non sarebbe possibile dedurre su quale macchina del cluster sono collocati. Hadoop ha sviluppato due meccanismi distinti per rendere il namenode maggiormente resistente ai guasti. Il primo modo consiste nel fare il Backup dei file che compongono lo stato persistente dei metadati del filesystem. Il secondo metodo prevede l’esecuzione di un Secondary Namenode, componente che, nonostante il nome, non agisce come un namenode, ma ha il compito di integrare il contenuto del namenode con quello del log delle modifiche. Il secondary namenode solitamente viene eseguito su uno nodo fisicamente separato da quello che contiene il namenode principale, e in casi di malfunzionamento del namenode principale può essere utilizzato come suo sostituto.

I **DataNode**, collocati sui nodi **worker**, gestiscono fisicamente lo storage dei blocchi di dati su ciascun nodo. Periodicamente comunicano al NameNode la lista dei blocchi che memorizzano e, all’occorrenza, eseguono le operazioni richieste dai client, entità che interagiscono in lettura e scrittura con HDFS. Solitamente ogni DataNode viene posizionato su una macchina distinta del cluster; queste a loro volta possono essere raggruppate in rack. I rack vengono definiti in fase di setup e di configurazione del cluster.

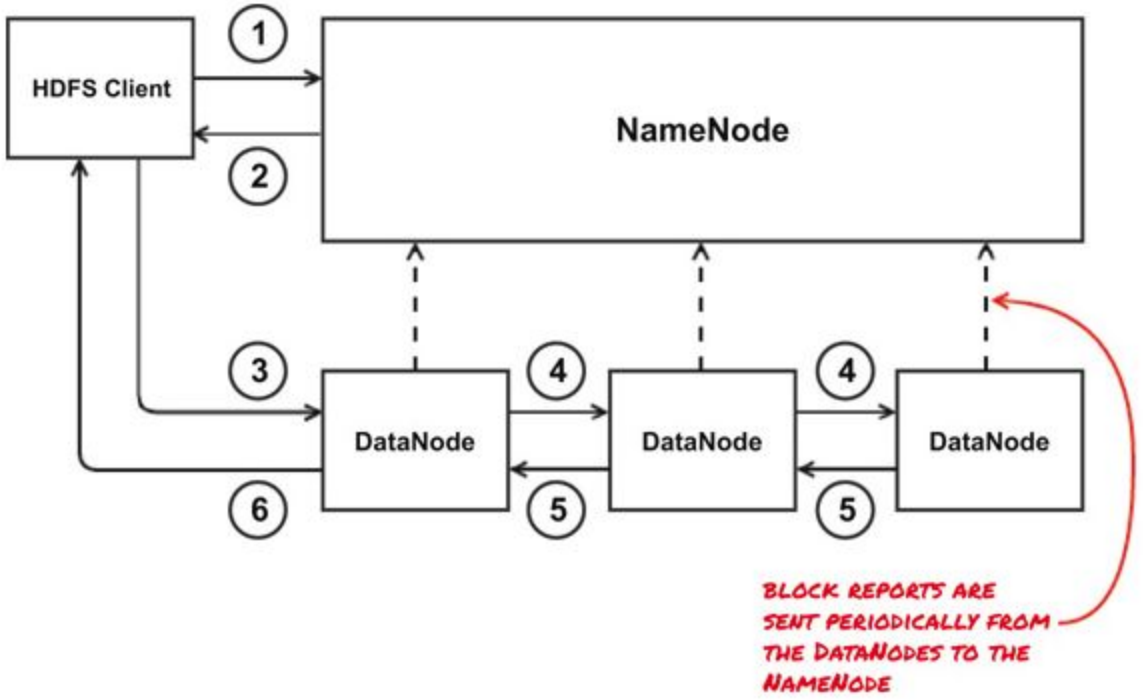
La strategia di replicazione di Hadoop sfrutta questa configurazione del cluster copiando i blocchi su altri datanode appartenenti a rack diversi da quello di origine. Ciò minimizza i rischi legati ai guasti (sia dei rack, sia dei nodi), massimizza le performance di lettura ma appesantisce la scrittura.

Durante la fase di replicazione dei blocchi, è il namenode che sceglie i datanode sui quali memorizzare le repliche. Il namenode seleziona i datanode sulla base di un compromesso tra l'affidabilità e la banda a disposizione per la lettura e la scrittura. Se per esempio, tutte le repliche fossero mantenute su un singolo nodo, non si avrebbero problemi di banda in fase di lettura e scrittura, ma al primo malfunzionamento del nodo i dati del blocco replicato andrebbero persi. All'estremo opposto, posizionare le repliche su diversi data center massimizzerebbe l'affidabilità, a discapito però della banda.

La strategia predefinita di Hadoop prevede di mantenere la prima replica sullo stesso nodo da cui proviene la richiesta del client; la seconda replica viene posta su un rack diverso dal primo (off-rack), scelto a caso; la terza replica viene posta sullo stesso rack della seconda, ma su un'altro nodo, scelto a caso. Ulteriori repliche vengono distribuite casualmente sui nodi del cluster, evitando di posizionare troppe repliche sullo stesso rack.

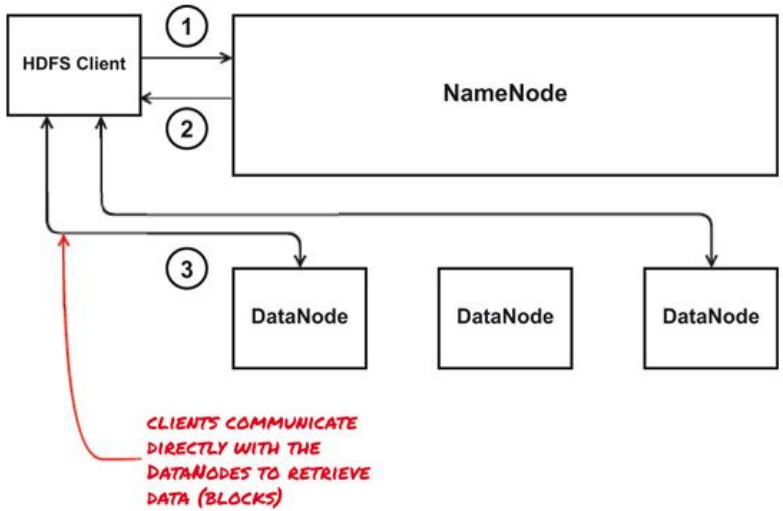

node
rack
data center

Di seguito è riportata la lista di operazioni compiute durante un'operazione di lettura di un file da HDFS



1. Il client HDFS richiede la scrittura di un blocco di file
2. Il NameNode comunica al client su quale DataNode può scrivere il blocco
3. Il Client richiede la scrittura del blocco al DataNode indicato
4. Il DataNode apre una pipeline di replica con un altro DataNode del cluster e questo processo continua finché non vengono scritte tutte le repliche richieste
5. Un messaggio di avvenuta scrittura viene restituito sulla pipeline
6. Il client è informato dell'esito dell'operazione di scrittura

Operazione di lettura da HDFS:



1. Il Client HDFS richiede la lettura di un file
2. Il NodeName risponde con la lista di DataNode che contengono i blocchi che compongono il file
3. Il Client comunica direttamente con i DataNode per ricevere i blocchi che compongono il file

Accedere al file system HDFS

All'interno della distribuzione Hadoop, si trova un'utility a riga di comando chiamata **hdfs**, che rappresenta il metodo principale per interagire con il file system dalla riga di comando. Eseguitela senza argomenti per vedere i vari sottocomandi disponibili. La sintassi generale del comando hdfs è:
$ hdfs <sottocomando> <comando> [argomenti]

I due sottocomandi principali che utilizzeremo sono i seguenti:

- **dfs**: viene utilizzato per l'accesso generale e la manipolazione del file system, compresi lettura/scrittura e accesso a file e directory;
- **dfsadmin**: viene utilizzato per l'amministrazione e la manutenzione del file system. Non lo tratteremo in dettaglio. Date un'occhiata al comando **-report,** che fornisce un elenco dello stato del file system e di tutti i DataNode **hdfs dfsadmin -report**

Per ottenere un elenco di tutti i comandi disponibili forniti dal sottocomando dfs possiamo eseguire:

$ hdfs dfs

Molti comandi sono simili a quelli standard del file system Unix e funzionano come ci si aspetta.

Ad esempio per ottenere una lista della root del file system possiamo utilizzare il comando:
$ hdfs dfs -ls /

L'output è molto simile a quello del comando ls di Unix. Gli attributi del file funzionano come quelli di user/group/world su un file system Unix ed in più forniscono i dettagli sul proprietario, il gruppo e l'ora di modifica. Il valore tra il nome del gruppo e la data modifica è la dimensione (0 per le directory).

ATTENZIONE: i percorsi relativi vengono tratti dalla directory home dell'utente.

Creazione di una cartella nella home dell'utente:
$ hdfs dfs -mkdir corso-ats

Creazione di un file e copia in HDFS:
$ echo "Corso Big Data con Hadoop!" > nome-corso.txt
$ hdfs dfs -put nome-corso.txt corso-ats

Ora leggendo il contenuto della cartella **corso-ats** dovremmo notare una nuova colonna tra gli attributi di file e il proprietario: è il fattore di replica del file.

Infine possiamo seguire il seguente comando per vedere il contenuto del file:
$ hdfs dfs -tail corso-ats/nome-corso.txt

Accesso programmatico ad HDFS con Python

Oltre all'accesso ad HDFS da riga di comando è possibile accedervi con client esterni ed anche scrivere la nostra applicazione per effettuare operazioni su HDFS.
Ad esempio se volessimo accedere al nostro FS distribuito da Python potremmo installare la seguente libreria:
pip install snakebit snakebit-py3

e scrivendo la seguente applicazione Python

```
from snakebite.client import Client
client = Client('quickstart.cloudera', 8020)
for x in client.ls(['/']):
        print(x)
```

saremmo in grado di ottenere la lista di elementi presenti nella root del nostro HDFS.

La linea più importante della nostra applicazione è sicuramente la seguente:
client = Client('quickstart.cloudera', 8020)

che crea la connessione client al HDFS del NameNode. Il metodo Client() accetta i seguenti parametri:

- **host (string)**: Hostname or IP address of the NameNode
- **port (int)**: RPC port of the NameNode
- **hadoop_version (int)**: The Hadoop protocol version to be used (default: 9)
- **use_trash (boolean):** Use trash when removing files
- **effective_use (string)**: Effective user for the HDFS operations (default: None or current user)

I parametri host e port sono obbligatori e dipendono dalla configurazione del nostro HDFS. I valori da inserire in questi parametri possono essere trovato all'interno del file
hadoop/conf/core-site.xml nella propietà fs.defaultFS

```
<property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
</property>
```

Per creare una directory è possibile utilizzare il metodo mkdir come nel listato seguente:

```
from snakebite.client import Client
client = Client('localhost', 9000)
for p in client.mkdir(['/foo/bar', '/input'], create_parent=True):
        print p
```

Per eliminare file o directory da HDFS si può utilizzare il metodo delete()

```
from snakebite.client import Client
client = Client('localhost', 9000)
for p in client.delete(['/foo', '/input'], recurse=True):
        print p
```

Come per il comando **hdfs dfs** la libreria mette a disposizione diversi metodi per recuperare file da HDFS e crearne una copia locale, come **copyToLocal()**.

```
from snakebite.client import Client
client = Client('localhost', 9000)
for f in client.copyToLocal(['/input/input.txt'], '/tmp'):
        print(f)
```

Infine se si vuole visualizzare il contenuto di un file che risiede su HDFS è possibile utilizzare il metodo **text()**

```
from snakebite.client import Client
client = Client('localhost', 9000)
for l in client.text(['/input/input.txt']):
print(l)
```

MapReduce

MapReduce è il cuore del sistema di calcolo distribuito di Hadoop. Rappresenta il framework attraverso il quale è possibile creare applicazioni in grado di elaborare grandi quantità di dati in parallelo su grandi cluster. MapReduce lavora secondo il principio divid et impera, ovvero prevede la suddivisione di un'operazione di calcolo in diverse parti processate in modo autonomo. Al termine del calcolo di ciascuna parte, i vari risultati parziali vengono "ricomposti" in un unico risultato finale.

L'applicazione in grado di essere eseguita sull'ambiente Hadoop viene definita come Job MapReduce, composto in generale da quattro elementi:

- dati di input;
- una fase di map;
- una fase di reduce;
- dati di output.

Le fasi di **map** e di **reduce**, che compongono il Job **MapReduce**, vengono suddivise in un certo numero di task, ovvero sotto-attività schedulate e gestite da **YARN** ed eseguite in parallelo sul cluster Hadoop. A seconda dell'attività (di reduce o di map) che verrà svolta, i task vengono classificati come map task oppure come reduce task.

I singoli task vengono eseguiti sui nodi del cluster adibiti al calcolo. Tipicamente all'interno di un cluster, i nodi adibiti al calcolo e i nodi di storage sono gli stessi, infatti MapReduce e HDFS condividono lo stesso insieme di nodi. Questa configurazione permette al framework di organizzare le attività in modo tale da ridurre la quantità di banda utilizzata per il trasferimento dei dati. **MapReduce**, infatti, utilizza un meccanismo, chiamato **data locality optimization**, che permette di allocare, in modo efficiente, i task sui nodi dove risiedono i dati necessari alla computazione.

In generale, l'esecuzione di un Job viene presa in carico dall'architettura MapReduce, caratterizzata da cinque entità indipendenti:

- il **client**, che richiede l'esecuzione del Job MapReduce;

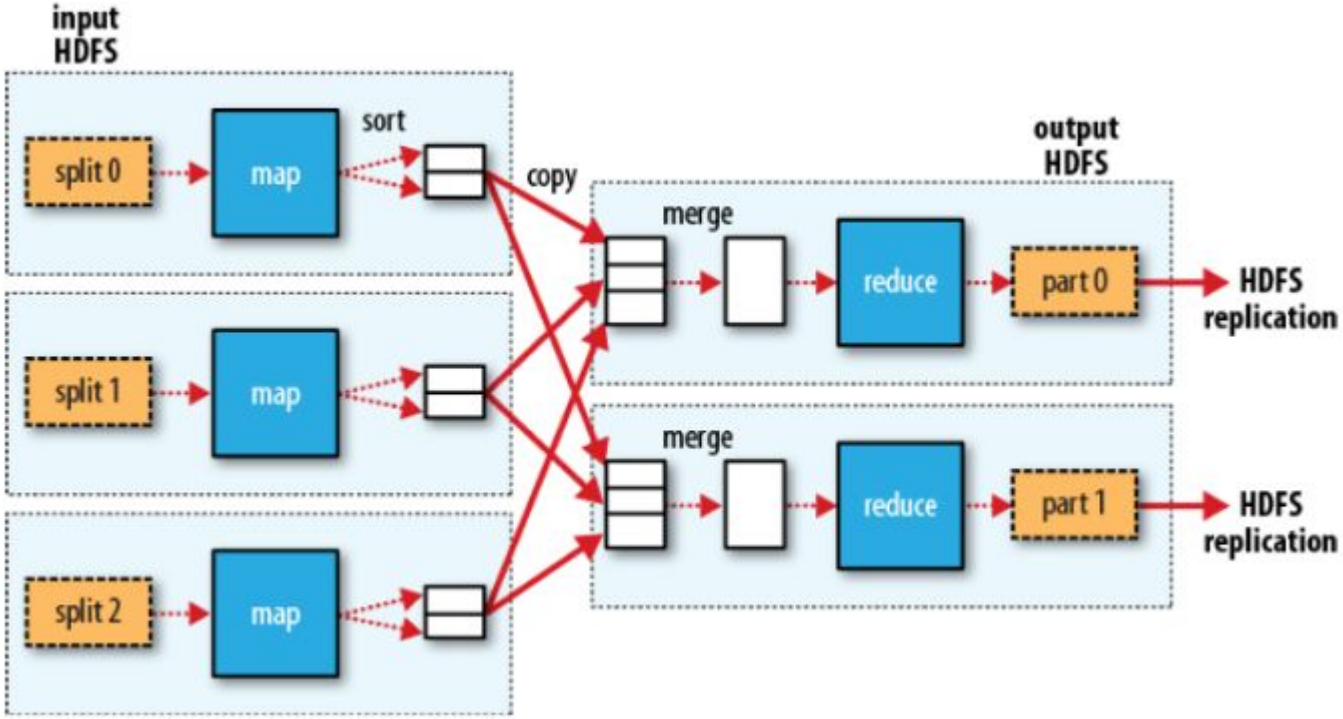
- il **ResourceManager**, che coordina l'allocazione delle risorse computazionali per ogni singolo container del cluster.
- il **NodeManager**, che lancia e monitora la computazione dei container sul nodo del cluster;
- l'**ApplicationMaster**, che coordina i task di map e di reduce del Job MapReduce. In particolare l'ApplicationMaster ha il compito di negoziare con il ResourceManager le risorse necessarie per l'allocazione dei container, sui quali verranno eseguiti, uno per volta, i singoli task. Inoltre coopera con i NodeManager per eseguire e monitorare i task in esecuzione;
- l'**HDFS**, utilizzato per la condivisione dei file fra le entità.

All'avvio della computazione, l'input del Job viene suddiviso in porzioni di dimensioni fisse, chiamate **splits**. La dimensione di questi, di default è pari a 128MB, ma può essere ridimensionata dall'utente. Suddiviso l'input l'ApplicationMaster provvede alla creazione di tanti map task quanti sono gli split ottenuti dalla suddivisione; di questi ne alloca tanti quanti sono i container a disposizione per l'esecuzione. I container provvedono all'esecuzione di un singolo task per volta. Man mano che la computazione di ogni singolo task termina, l'ApplicationMaster provvede ad assegnare dinamicamente, ai container liberi, i task ancora in sospeso per l'esecuzione..

Terminata la computazione, ogni singolo map task restituisce un output composto da una coppia chiave/valore, memorizzata sul disco locale del nodo e ordinata rispetto le altre coppie chiave/valore presenti. I map task non scrivono direttamente su HDFS, perché forniscono output intermedi, che dovranno poi essere processati dai task di reduce e cancellati al termine dell'applicazione.

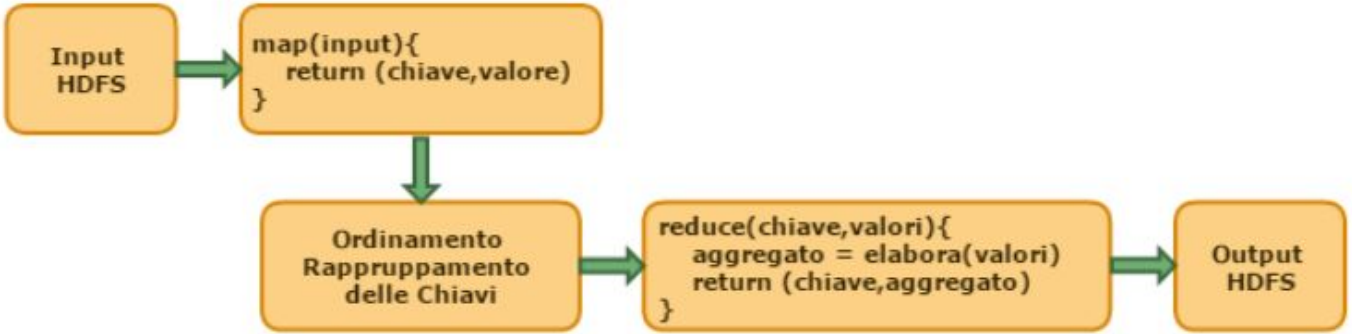
L'output restituito dalla fase di map è quindi prelevato dai reduce task per lo svolgimento delle propria computazione. Questi elaborano l'input, proveniente dalla fase di map, sulla base delle chiavi. L'output ottenuto dalla computazione dei reduce, viene poi memorizzato su HDFS e contestualmente replicato secondo la strategia di replicazione di HDFS.

Il numero di reduce task, diversamente da quello dei map, non dipende dalla dimensione dell'input che si intende elaborare, ma viene specificato "dall'esterno".


input
HDFS
split 0
map
sort
split 1
map
split 2
map
copy
merge
reduce
part 0
output
HDFS
HDFS
replication
merge
reduce
part 1
HDFS
replication

Fase di Shuffle

Le attività che il framework svolge a seguito della map e prima della reduce, fanno parte della fase di Shuffle. Il framework MapReduce è stato progettato in modo tale da garantire che l'input di ogni reduce sia ordinato in base alla chiave.



Il processo attraverso il quale il sistema esegue l'ordinamento e il trasferimento dell'output dei map a input dei reduce è detta Shuffle. Questa coinvolge sia la fase terminale del processo di map, che la fase iniziale del processo di reduce

Implementiamo una semplice applicazione MapReduce per il conteggio delle parole in Python. Lo sviluppo consiste nella creazione di due file Python: **mapper.py** e **reducer.py**

mapper.py: è il programma Python che implementa la logica della fase di map. Esso legge i dati dallo standard input, splitta le linee in parole e restituisce ogni parola con il suo conteggio intermedio sullo standard output.

di seguito è riportato il codice sorgerte della funzione mapper:

```
#!/usr/bin/env python
import sys
# legge ogni linea dallo standard input
for line in sys.stdin:
    # recupera le parole in ogni linea
    words = line.split()
    # genera il count per ogni parola
    for word in words:
        print('{0}\t{1}'.format(word, 1))
```

la riga **print(‘{0}\t{1}’.format(word, 1))** si occupa di stampare le coppie chiave-valore sullo standard output. In questo caso la chiave ed il valore vengono separati dal carattere di tabulazione (\t).

reducer.py è il programma Python che implementa la logica della fase di reduce. Legge i risultati di mapper.py dallo standard input, somma le occorrenze di ogni parola, e scrive il risultato sullo standard output.

```
#!/usr/bin/env python
import sys
curr_word = None
curr_count = 0
# Processa tutte le coppie chiave valore provenienti dal mapper
for line in sys.stdin:
    # Get the key and value from the current line
    word, count = line.split('\t')
    # Convert the count to an int
    count = int(count)
    # Se la parola corrente è uguale alla parola precedente incrementa la sua count altrimenti la stampa
    if word == curr_word:
        curr_count += count
    else:
        # Scrive la parola ed il suo numero di occorrenze come coppia chiave-valore
        if curr_word:
            print '{0}\t{1}'.format(curr_word, curr_count)
        curr_word = word
        curr_count = count
# Stampa la count dell'ultima parola
if curr_word == word:
    print '{0}\t{1}'.format(curr_word, curr_count)
```

prima di provare ad eseguire il codice assicuratevi che i file mapper.py e reducer.py abbiamo i permessi di esecuzione. Eventualmente potete dargli i permessi con questo comando:
chmod a+x mapper.py reducer.py

Assicuratevi anche che la prima riga dei file abbiamo il percorso corretto di Python, questa permette di avviare i file come un qualsiasi eseguibile standalone.

E' possibile testare il programma Python localmente prima di avviarlo come job MapReduce ed è naturalmente consigliato testare tutte le applicazioni localmente prima di avviarle su un cluster Hadoop.

```
$ echo 'jack be nimble jack be quick' | ./mapper.py | sort -t 1 | ./reducer.py
be      2
jack    2
nimble  1
quick   1
```

MRJOB

mrjob è una libreria Python per la scrittura di applicazioni MapReduce. Essa “wrappa” Hadoop streaming e da la possibilità di scrivere applicazioni MapReduce tramite classi Python.

I jobs MapReduce scritti con mrjob possono essere facilmente testati localmente anche senza una installazione di Hadoop locale per poi essere eseguiti su cluster Hadoop.

L’installazione della libreria è semplice e può essere effettuata tramite pip utilizzando la seguente istruzione:

$ pip install mrjob

WordCount in mrjob

```
from mrjob.job import MRJob
class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
            yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

Per eseguire e testare localmente il job è sufficiente laciare lo script con un file di testo con il seguente comando:

```
$ python word_count.py input.txt
```

Per eseguire il job su un cluster Hadoop con un input specificato da HDFS:

$ python mr_job.py -r hadoop hdfs://input/input.txt