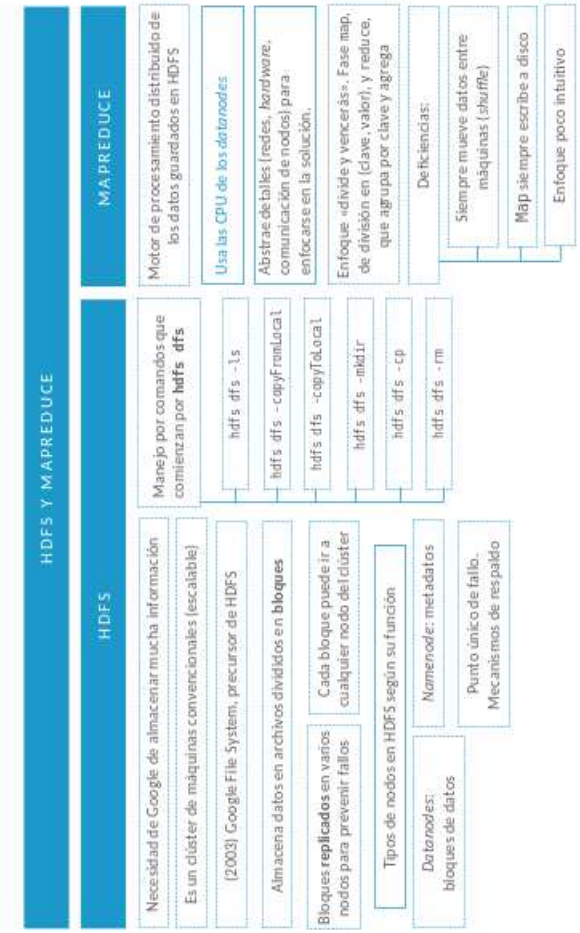
Ingeniería para el Procesado Masivo de Datos

Tema 2. HDFS y MapReduce

Índice

Esquema



2.1. Introducción y objetivos

En este tema, examinaremos Hadoop Distributed File System, conocido habitualmente como HDFS, que constituye la primera de las tecnologías distribuidas que estudiaremos durante el curso. Ha cambiado relativamente poco con los años y sigue siendo la base de casi todas las demás, puesto que proporciona la capa de almacenamiento persistente (en discos duros) de los datos. En la parte final del tema, veremos el paradigma de programación distribuida MapReduce, propuesto para procesar en paralelo los datos almacenados en HDFS.

Los objetivos que persigue este tema son:

- Conocer las características propias de HDFS.
- Entender la arquitectura de HDFS y las operaciones de escritura y lectura.
- Familiarizarse con los comandos más habituales de HDFS.
- Comprender el paradigma de programación MapReduce, sus virtudes y sus deficiencias.

2.2. Introducción a HDFS

El artículo publicado por Ghemawat, Gobioff y Leung (2003) acerca de Google File System (GFS) fue el germen del sistema de archivos distribuido HDFS (Hadoop Distributed File System), que constituye una parte fundamental del ecosistema Hadoop y el cual se ha seguido utilizando sin apenas cambios desde que se introdujo por primera vez.

En las próximas páginas, veremos:

- Las características propias de HDFS y sus componentes desde el punto de vista de la arquitectura.
- Sus peculiaridades como sistema de archivos distribuido frente a sistemas de archivos tradicionales no distribuidos (en una sola máquina).
- Su funcionamiento interno.
- Los comandos más habituales utilizados para manejarlo.

Podemos definir HDFS como:

> Un sistema de archivos distribuido destinado a almacenar archivos muy grandes con patrones de acceso en *streaming*, pensado para clústeres de ordenadores convencionales.

Si desgranamos esta definición, tenemos que:

1. Sistema de archivos distribuido: los archivos se almacenan en una red de máquina, lo cual implica que:

- Son máquinas *commodity* (*hardware* convencional, que puede fallar). Esto difiere de

los sistemas tradicionales, donde se adquiría un gran ordenador o *mainframe*, mucho más potente de lo que un usuario doméstico poseía en casa, pero con la restricción de que, al ser una sola máquina, el fallo o la falta de capacidad obligaba a reemplazar el *mainframe* por otro más grande.

- Es escalable: si se requiere más capacidad, basta con añadir más nodos, en lugar de reemplazar una máquina por otra más grande.

- Necesita incorporar mecanismos *software* de recuperación frente a fallo de un nodo, algo que veremos en la siguiente sección.

2. Pensado para almacenar archivos muy grandes: permite guardar archivos mayores que la capacidad del disco de una máquina individual. Es decir, un solo archivo puede ocupar cientos de GB, varios TB e incluso PB, a pesar de que cada disco duro de un nodo tenga una capacidad limitada de 500 GB solamente, por ejemplo.

3. Archivos con patrón de acceso *write-once, read-many:* archivos que se crean y después se accede a ellos para ser leídos en numerosas ocasiones. No son archivos cuyo contenido necesitemos reemplazar con frecuencia ni que sean borrados habitualmente. Además, a diferencia de una base de datos, no es importante el tiempo de acceso a partes concretas del archivo (por ejemplo, el fragmento exacto del archivo que contiene el registro con los datos de una persona en concreto), sino que se suele utilizar y procesar el archivo completo en las aplicaciones (modo *batch,* no interactivo). Por este motivo:

- No soporta modificación de archivos existentes, sino solo lectura, escritura y borrado.

- No funciona bien para:

  - Aplicaciones que requieran baja latencia a fin de acceder a registros individuales dentro de un archivo.

- Muchos archivos pequeños (generan demasiados metadatos).
- Archivos que se modifiquen con frecuencia.

HDFS es, en realidad, un *software* escrito en lenguaje Java, que se instala encima del sistema de archivos de cada nodo del clúster. El *software* HDFS se encarga de proporcionarnos una abstracción que nos da la ilusión de estar usando un sistema de archivos, pero, por debajo, continúa usando el sistema de archivos nativo del sistema operativo (por ejemplo, en el caso de máquinas Linux, que son las más habituales para instalar HDFS, estará utilizando por debajo el conocido sistema de archivos ext4). La peculiaridad es que nos permite almacenar archivos de manera distribuida, pero manejarlos como si fuese un único sistema de archivos.

Es importante resaltar que, cuando un nodo forma parte de un clúster en el que está instalado HDFS, es posible utilizar tanto el sistema de archivos local del nodo (al que podemos acceder, por ejemplo, abriendo una terminal en ese nodo mediante acceso SSH) como el sistema de archivos HDFS, que existe «además de» cada uno de los sistemas de archivos locales. Para crear datos en HDFS, podemos copiarlos en este *software* desde el sistema de archivos de una máquina concreta si ya existían en ella, o bien pueden generarlos directamente en HDFS como resultados de otra aplicación que se esté ejecutando de forma distribuida en el clúster (por ejemplo, un trabajo de Spark).

La mayoría de los comandos que ofrece HDFS para manejar archivos se llaman igual que los comandos del sistema de archivos de Linux, con el fin de facilitar su uso a usuarios que ya estaban acostumbrados a trabajar con este sistema operativo. No obstante, esto no debe confundirnos: la ejecución de un comando en el sistema de archivos local (por ejemplo, ls, para mostrar el contenido de un directorio de nuestro ordenador) desencadena unas acciones muy diferentes a la ejecución del comando con el mismo nombre, pero en HDFS (precedido por las palabras hdfs dfs).

2.3. Arquitectura de HDFS

Bloques de HDFS

En un dispositivo físico como un disco duro, un bloque físico (o sector) de disco es la cantidad de información que se puede leer o escribir en una sola operación de disco. Habitualmente son 512 bytes. En un sistema de archivos convencional, como, por ejemplo, ext4 de Linux, o NTFS y FAT32 de Windows, un bloque del sistema de archivos es el mínimo conjunto de sectores que se pueden reservar para leer o escribir un archivo. Suele ser configurable (Linux ext4 permite 1 KB, 2 KB, etc.). Generalmente es de 4 KB, y siempre debe contener un número de sectores físicos que sea potencia de 2.

En sistemas de archivos convencionales (p. ej., ext4 de Linux, o NTFS y FAT32 de Windows), los archivos menores que el tamaño de bloque del sistema de archivos siguen ocupando un bloque completo, es decir, se desperdicia una pequeña cantidad de espacio. HDFS tiene su propio tamaño de bloque, que es configurable (por defecto, es de 128 MB), pero los archivos de menos de un bloque de HDFS no desperdician espacio, a pesar de que siempre usan su propio bloque de datos (es decir, nunca se comparte un bloque de HDFS entre varios archivos).

El tamaño de bloque de HDFS tiene varias implicaciones:

- Un archivo se parte en bloques, que pueden almacenarse en máquinas diferentes. Así se puede almacenar un archivo mayor que el disco de una sola máquina, ya que se almacena troceado.

- Cada bloque requiere metadatos (que se almacenan en el *namenode*) para mantenerlo localizado y saber de qué archivo forma parte. Si los bloques son muy pequeños, se generan demasiados metadatos, lo cual llena muy rápido el *namenode* y hace más difíciles las búsquedas. Si son muy grandes, limitan el paralelismo de *frameworks* que operan a nivel de bloque (como Spark), en los que varias máquinas

procesan a la vez bloques diferentes de un mismo archivo.

- Los bloques de datos suelen estar replicados para ofrecer alta disponibilidad y máximo paralelismo: cada bloque está en *k* máquinas, donde *k* es el factor de replicación. HDFS fija por defecto un factor de replicación de 3, aunque este valor es configurable individualmente para cada fichero mediante el comando hadoop dfs -setrep -w 3 /user/hdfs/file.txt.

La siguiente imagen muestra un ejemplo de un archivo flights.csv de 500 MB que, al subirlo a HDFS, ha sido particionado automáticamente por este sistema en cuatro bloques: tres de ellos tienen 128 MB (bloques completos, de longitud 134 217 728 bytes) y el cuarto es más pequeño. Cada uno de estos bloques se ha replicado tres veces, también de manera automática. El comando de HDFS que se ha ejecutado después, fsck, nos permite consultar los metadatos asociados a este fichero, los cuales describen cómo está almacenado físicamente.

```
[pvillacorta@meetupds1 ~]$ hdfs fsck /SparkMeetup/flights1994.csv -blocks -files
Connecting to namenode via http://meetupds1:50070/fsck?ugi=pvillacorta&blocks=1&files=1&path=%2FSparkMeetup%2Fflights1994.csv
FSCK started by pvillacorta (auth:SIMPLE) from /144.76.3.23 for path /SparkMeetup/flights1994.csv at Thu Apr 18 21:43:27 CEST 2019
/SparkMeetup/flights1994.csv 501558665 bytes, 4 block(s):  OK
0. BP-2126204769-           -1526901840376:blk_1073977383_236559 len=134217728 repl=3
1. BP-2126204769-           -1526901840376:blk_1073977384_236560 len=134217728 repl=3
2. BP-2126204769-   ....    -1526901840376:blk_1073977385_236561 len=134217728 repl=3
3. BP-2126204769-           -1526901840376:blk_1073977386_236562 len=98905481 repl=3

Status: HEALTHY
 Total size:    501558665 B
 Total dirs:    0
 Total files:   1
 Total symlinks:                0
 Total blocks (validated):      4 (avg. block size 125389666 B)
 Minimally replicated blocks:   4 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    3
 Average block replication:     3.0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Number of data-nodes:          3
 Number of racks:               1
FSCK ended at Thu Apr 18 21:43:27 CEST 2019 in 1 milliseconds


The filesystem under path '/SparkMeetup/flights1994.csv' is HEALTHY
```

Figura 1. Salida dada por el comando fsck para un archivo de 500 MB subido a HDFS.

- *Rack-awareness:* es posible, aunque no obligatorio, especificar, en la configuración de HDFS, la disposición física (topología) del clúster e indicar qué nodos comparten un mismo *rack* (armario) y cuál es la cercanía entre ellos. De esta manera, HDFS puede decidir mejor en qué nodo del clúster debe colocar cada bloque de un archivo que se vaya a escribir, con el objetivo de minimizar las pérdidas si falla un *rack*

completo, lo cual a veces ocurre debido a que los nodos en un mismo *rack* comparten cierta infraestructura física.

La norma general es no colocar más de una réplica de un bloque en el mismo nodo (esto es obvio), ni más de dos réplicas en nodos del mismo *rack.* A partir de ahí, es posible establecer políticas más complejas para decidir la asignación de bloques de datos a nodos concretos. La topología también influye en la elección de los *datanodes* que servirán cada bloque de datos al leer un fichero, según la cercanía física al cliente a quien serán servidos.

Datanodes y *namenode*

Cuando se instala HDFS en un clúster, cada nodo puede utilizarse como *datanode* o como *namenode.* El *namenode* (debe haber, al menos, uno) mantiene la estructura de directorios existentes y los metadatos asociados a cada archivo. Por su parte, los *datanodes* almacenan bloques de datos y los devuelven a petición del *namenode* o del programa cliente que está accediendo a HDFS para leer datos.

El *namenode* recibe periódicamente de los *datanodes* un *heartbeat* (por defecto, cada tres segundos, aunque es configurable en la propiedad dfs.heartbeat.interval) y un listado de todos los bloques presentes en cada *datanode* (por defecto, cada seis horas, configurable en dfs.blockreport.*). El *namenode* es punto único de fallo (SPOF), lo que significa que, sin él, no es posible utilizar HDFS. La razón es que el *namenode* posee la estructura de directorios del sistema de archivos y conoce qué bloques forman cada archivo y dónde se almacenan físicamente. Por ese motivo, se han ideado diferentes mecanismos, tanto de respaldo del *namenode* como de alta disponibilidad, que detallamos ahora. Más adelante explicamos cómo escalar el *namenode* cuando, en el sistema de archivos, ha crecido el número y la complejidad de estos.

Respaldo de datos del *namenode*

Copia de los **archivos persistentes** de los metadatos a otros nodos o a un sistema de ficheros externos como NFS.

- ***Namenode* secundario:** en otra máquina física que no es realmente un *namenode*, un proceso va fusionando los cambios que indica el *log* de edición respecto a la imagen del *namenode*, para tener una fotografía actualizada del estado de los ficheros en el clúster. En caso de fallo del *namenode*, se transfieren a esa máquina los posibles metadatos adicionales que estuviesen replicados en NFS y se empieza a utilizar esa máquina como el *namenode* activo. Esto es un proceso manual que requiere unos treinta minutos, durante los cuales no podemos utilizar HDFS. Al ser tan largo el intervalo de pérdida de servicio, no se considera un mecanismo de alta disponibilidad.

Alta disponibilidad del *namenode*

Se utilizan un par de *namenodes*, uno de ellos denominado activo y el otro en *stand by*. Ambos comparten un *log* de edición en un sistema de almacenamiento externo y que posee también alta disponibilidad. El *log* va siendo modificado por el *namenode* activo, pero el *namenode* en *stand by* lo va leyendo y va aplicando esos cambios en sus propios metadatos, para estar siempre actualizado respecto al *namenode* activo. Los *datanodes* reportan la información a ambos para monitorizar el estado. En caso de fallo del *namenode* activo, el que se encontraba en *stand by* pasa inmediatamente a activo. Como señalábamos, estará actualizado, puesto que todas las operaciones realizadas por las aplicaciones cliente (las que generan peticiones de lectura o escritura en HDFS) son recibidas siempre por ambos *namenodes*. En la práctica, este proceso apenas lleva un minuto hasta que se restablece el servicio normal gracias al nuevo *namenode*.

Escalando el *namenode*: *namenodes* federados

Este mecanismo se utiliza cuando el *namenode* se encuentra saturado y cerca de llenarse. No es un mecanismo de protección contra fallos como tal, aunque también

cumple esa función. Consiste en que varios *namenodes* que funcionan a la vez se encargan de directorios distintos del sistema de archivos, sin solapamiento. Por ejemplo: los metadatos relativos a todo el árbol de directorios que cuelga de /user se pueden almacenar en un *namenode*, mientras que todo el árbol que cuelga de /share lo puede hacer en otro. De esta manera, el posible fallo de un *namenode* no afecta en nada a los archivos o directorios que no cuelgan de esa jerarquía. Los *datanodes* sí pueden almacenar indistintamente bloques de archivos de varios *namespaces* (es decir, de varios subárboles).

Proceso de lectura y escritura en HDFS

Proceso de lectura

El proceso que se lleva a cabo cuando un comando necesita leer un archivo de HDFS es el siguiente (figura 2):

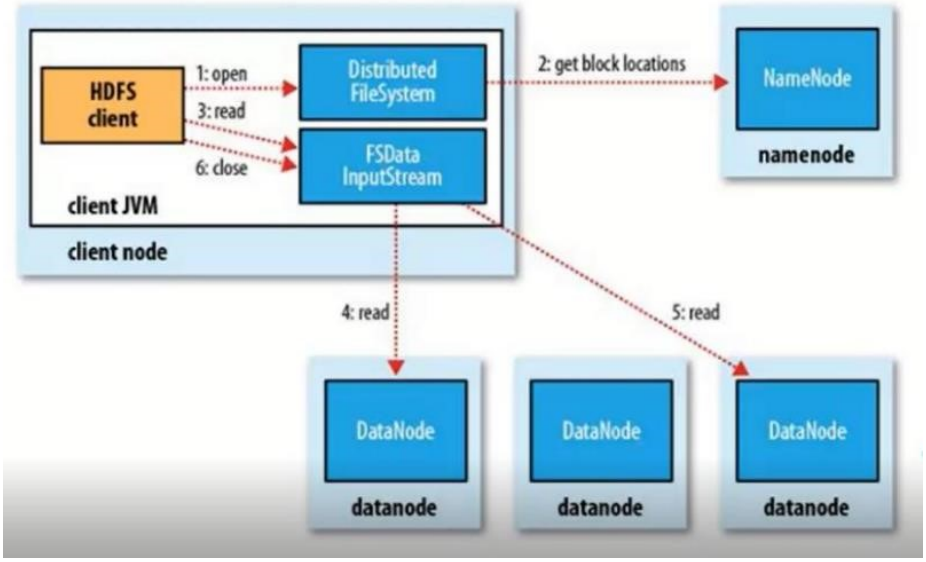


Figura 2. Proceso de lectura de datos de HDFS. Fuente: https://programmerclick.com/article/8300156830/

- El programa cliente (el cual implementa el comando de HDFS que requiere lectura) solicita al *namenode* que le proporcione un fichero. Esto se lleva a cabo mediante una llamada a procedimiento remoto (RPC o *remote procedure call*) desde el nodo

donde se ejecuta el cliente al *namenode*.

- El *namenode* consulta en la tabla de metadatos los bloques que componen el fichero y su localización en el clúster.
- El *namenode* devuelve al cliente una lista de bloques, así como los equipos en los que puede encontrar cada uno de ellos.
- El cliente contacta con los nodos que almacenan los bloques que forman el archivo para ir obteniendo dichos bloques, aunque esto es transparente a la aplicación, ya que simplemente lee datos del objeto FSDataInputStream y los percibe como un flujo continuo. Es este objeto el que, ocasionalmente, vuelve a contactar con el *namenode* para obtener la mejor ubicación de los siguientes bloques de datos, según se van necesitando al realizar lecturas.
- Finalmente, el cliente compone el archivo.

Es importante resaltar que **el *namenode* no devuelve los bloques de datos** (no pasan por él) porque se convertiría en cuello de botella cuando hubiese múltiples clientes leyendo al mismo tiempo. Es el propio cliente (de forma transparente para él, pues se gestiona a través del objeto FSDataInputStream que nos ha devuelto la API de HDFS) quien se dirige a los *datanodes* que le ha indicado el *namenode* para solicitarles los bloques de datos que desea leer.

Proceso de escritura

Lleva a cabo unos pasos análogos (figura 3):

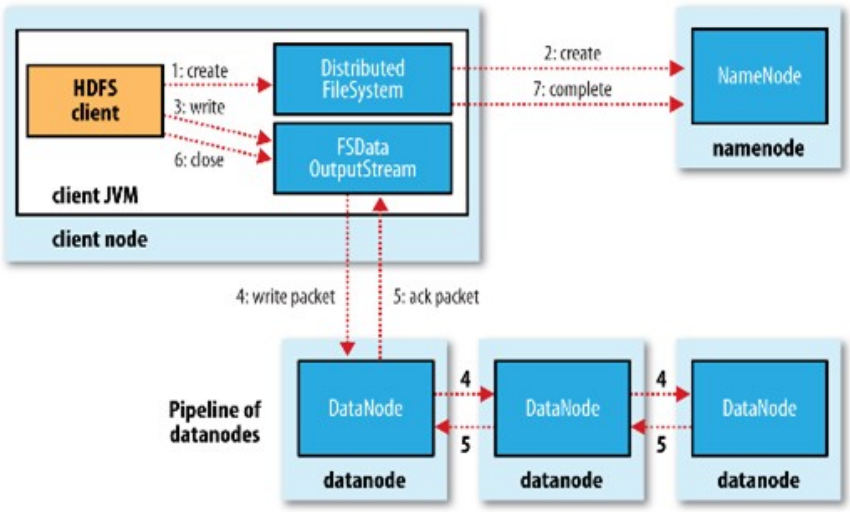


Figura 3. Proceso de escritura de datos en HDFS. Fuente: https://programmerclick.com/article/8300156830/

- El cliente solicita al *namenode*, mediante RPC, realizar una escritura de un archivo sin bloques de datos asociados.
- El *namenode* realiza algunas comprobaciones previas, para comprobar que se puede escribir el fichero y que el cliente tiene permisos para hacerlo.
- El objeto FSDataOutputStream devuelto al cliente particiona el fichero en bloques, los encola y los va escribiendo de forma secuencial. Para cada bloque, el *namenode* le proporciona una lista de *datanodes* en los que se escribirá. Periódicamente volverá a contactar con el *namenode* para obtener la ubicación física en la que ir escribiendo los siguientes bloques de datos.
- Para cada bloque de datos, el cliente contacta con el primer *datanode* de la lista, a fin de pedirle que escriba el bloque. Este *datanode* se encargará de propagar el bloque al segundo *datanode*, etc.
- El último *datanode* en escribir el bloque devolverá una confirmación (ack) hacia atrás, hasta que el primer *datanode* mande la confirmación al cliente.
- Una vez que el cliente ha escrito todos los bloques, envía una confirmación al

namenode de que el proceso se ha completado.

Para cada bloque, los *datanodes* (varios, debido al factor de replicación) que lo guardarán forman un *pipeline*. El primer *datanode* escribe el bloque y lo reenvía al segundo, y así sucesivamente. Según se van escribiendo, se devuelve una señal de confirmación ack, que también se almacena en una cola. En caso de fallo de algún *datanode* durante la escritura, se establecen mecanismos para que el resto de *datanodes* del *pipeline* no pierdan ese paquete y asegurar la replicación. Son los propios *datanodes* los que gestionan la replicación de cada bloque, sin intervención del *namenode*, para evitar cuellos de botella. Solo se necesita la intervención del *namenode* de forma ocasional, con el fin de preguntarle dónde escribir el siguiente bloque de datos.

2.4. Comandos de HDFS más frecuentes

En todos los comandos, se puede usar `hadoop dfs` o `hdfs dfs` indistintamente, aunque la segunda opción está más extendida en la actualidad. Recordemos que lo que estamos haciendo es ejecutar un programa cliente llamado `hdfs` desde cualquiera de los nodos que forman parte de HDFS.

```
[root@sandbox ~]# hadoop fs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-find <path> ... <expression> ...]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
```

Es importante resaltar que, a diferencia del sistema de ficheros local de Linux o Windows, HDFS no cuenta con el comando cd para acceder a un directorio, ya que no existe el concepto de «directorio actual» o directorio en el que estamos situados. Por ese motivo, todos los comandos requieren especificar rutas completas.

A continuación, se expone un resumen de los comandos más frecuentes:

- `hdfs dfs –ls /ruta/directorio`: muestra el contenido de un directorio de HDFS.

```
[root@sandbox ~]# hadoop fs -ls /
Found 12 items
drwxrwxrwx   - yarn   hadoop          0 2016-10-25 08:10 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:54 /apps
drwxr-xr-x   - yarn   hadoop          0 2016-10-25 07:48 /ats
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 08:01 /demo
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:48 /hdp
drwxr-xr-x   - mapred hdfs            0 2016-10-25 07:48 /mapred
drwxrwxrwx   - mapred hadoop          0 2016-10-25 07:48 /mr-history
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:47 /ranger
drwxrwxrwx   - spark  hadoop          0 2017-02-02 11:46 /spark-history
drwxrwxrwx   - spark  hadoop          0 2016-10-25 08:14 /spark2-history
drwxrwxrwx   - hdfs   hdfs            0 2016-10-25 08:11 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 08:11 /user
[root@sandbox ~]#
```

- hdfs dfs –mkdir /ruta/nuevodirectorio : crea un nuevo directorio, nuevodirectorio , como subdirectorio del directorio /ruta.

```
[root@sandbox ~]# hadoop fs -mkdir /raul
[root@sandbox ~]# hadoop fs -ls /
Found 13 items
drwxrwxrwx   - yarn   hadoop          0 2016-10-25 08:10 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:54 /apps
drwxr-xr-x   - yarn   hadoop          0 2016-10-25 07:48 /ats
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 08:01 /demo
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:48 /hdp
drwxr-xr-x   - mapred hdfs            0 2016-10-25 07:48 /mapred
drwxrwxrwx   - mapred hadoop          0 2016-10-25 07:48 /mr-history
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 07:47 /ranger
drwxr-xr-x   - root   hdfs            0 2017-02-02 11:48 /raul
drwxrwxrwx   - spark  hadoop          0 2017-02-02 11:48 /spark-history
drwxrwxrwx   - spark  hadoop          0 2016-10-25 08:14 /spark2-history
drwxrwxrwx   - hdfs   hdfs            0 2016-10-25 08:11 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2016-10-25 08:11 /user
```

- hdfs dfs –copyFromLocal ruta/local/fichero.txt /ruta/hdfs/ **:** copia el fichero fichero.txt presente en el sistema de archivos local a la ubicación remota de HDFS situada en /ruta/hdfs/fichero.txt. La ruta del fichero local sí puede ser una ruta relativa, ya que se refiere al sistema de archivos local, por lo que no es imprescindible que empiece por /. hdfs dfs –copyFromLocal <localsrc> <dst>

```
[root@sandbox raul]# ls -la
total 12
drwxr-xr-x 2 root root 4096 Feb  2 11:50 .
dr-xr-x--- 1 root root 4096 Feb  2 11:49 ..
-rw-r--r-- 1 root root   16 Feb  2 11:50 MyBigFile.txt
[root@sandbox raul]# hadoop fs -copyFromLocal MyBigFile.txt /raul/MyBigFileinHDFS.txt
[root@sandbox raul]# hadoop fs -ls /raul/
Found 1 items
-rw-r--r--   1 root hdfs         16 2017-02-02 11:51 /raul/MyBigFileinHDFS.txt
[root@sandbox raul]#
```

- hdfs dfs –copyToLocal /ruta/hdfs/fichero.txt ruta/local **:** copia un fichero existente en HDFS

(en la ruta remota /ruta/hdfs/fichero.txt) al sistema de archivos local (concretamente, a la ubicación de destino ruta/local/fichero.txt). Sintaxis: hdfs dfs –copyToLocal <src> <localdst>.

```
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx------   1 hdfs hdfs     710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx------   1 hdfs hdfs      68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -copyToLocal /demo/data/CDR/cdrs.txt myCdrs.txt
[root@sandbox raul]# ls -la
total 708
drwxr-xr-x 2 root root   4096 Feb  2 11:55 .
dr-xr-x--- 1 root root   4096 Feb  2 11:49 ..
-rw-r--r-- 1 root root     16 Feb  2 11:50 MyBigFile.txt
-rw-r--r-- 1 root root 710436 Feb  2 11:55 myCdrs.txt
[root@sandbox raul]#
```

- hdfs dfs –tail /ruta/hdfs/fichero.txt: muestra en pantalla la parte final del contenido del archivo presente en HDFS.

```
[root@sandbox raul]# hadoop fs -tail /demo/data/CDR/recharges.txt
00
6641609561|20130209|094637|3|100
6650359180|20130209|125420|3|100
6638378345|20130209|121231|3|300
6659538250|20130209|191504|3|100
6662032971|20130209|211136|3|500
8333654388|20130209|100458|3|100
6623568405|20130209|121240|3|100
```

- hdfs dfs –cat /ruta/hdfs/fichero.txt **:** muestra en pantalla todo el contenido del archivo presente en HDFS. Debe usarse con cuidado, ya que lo habitual es que los ficheros sean grandes y el comando imprime todo el fichero. Para paginar la visualización, se recomienda redirigir (con la barra vertical |) la salida de este comando hacia el comando more del sistema de archivos local de Linux: hdfs dfs –cat /ruta/hdfs/fichero.txt | more

```
[root@sandbox raul]# hadoop fs -cat /demo/data/CDR/recharges.txt | more
PHONE|DATE|CHANNEL|PLAN|AMOUNT
7852121521|20130209|090721|3|100
7642140929|20130209|181648|3|100
7552204414|20130209|224815|3|100
7785846460|20130209|173731|3|100
7972930496|20130209|003527|3|100
7782957598|20130209|200016|3|100
7352795440|20130209|000429|3|100
```

Ideas clave

- hdfs dfs –cp /ruta/hdfs/origen/fichero.txt /ruta/hdfs/ destino/copiado.txt **:** hace una copia del fichero situado en HDFS, en /ruta/origen/fichero.txt , en la ruta destino (también de HDFS) /ruta/hdfs/destino/copiado.txt . Este comando no interactúa con el sistema de archivos local de la máquina; es una copia de un origen de HDFS a un destino también de HDFS. Sintaxis: hdfs dfs –cp <src> <dst>.

- hdfs dfs –mv /ruta/original.txt /ruta/nuevo.txt o bien hdfs dfs –mv /ruta/fichero1 /ruta/nueva/ : en el primer caso, renombra un fichero existente en HDFS. En el segundo, lo mueve (sin copiar) de una ubicación de HDFS a otra distinta.

- hdfs dfs –rm /ruta/fichero.txt **:** borra un fichero presente en HDFS. Es posible borrar una carpeta completa de forma recursiva (es decir, con todos sus subdirectorios) mediante la opción –r : hdfs dfs –rm –r /ruta/carpeta/ ¡Cuidado!

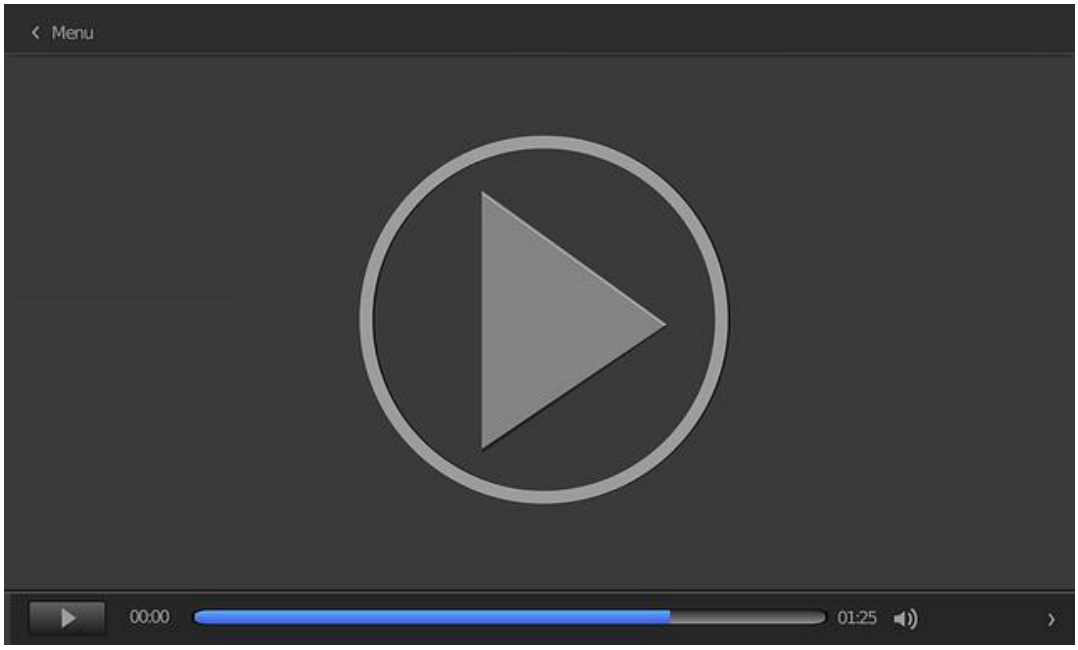
```
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx------   1 hdfs hdfs     710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx------   1 hdfs hdfs      68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -cp /demo/data/CDR/cdrs.txt /demo/data/CDR/cdrs2.txt
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 3 items
-rwx------   1 hdfs hdfs     710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rw-r--r--   1 root hdfs     710436 2017-02-02 12:01 /demo/data/CDR/cdrs2.txt
-rwx------   1 hdfs hdfs      68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -rm /demo/data/CDR/cdrs2.txt
17/02/02 12:01:52 INFO fs.TrashPolicyDefault: Moved: 'hdfs://sandbox.hortonworks.com:8
/user/root/.Trash/Current/demo/data/CDR/cdrs2.txt
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx------   1 hdfs hdfs     710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx------   1 hdfs hdfs      68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]#
```

- hdfs dfs –chmod <permisos> /ruta/hdfs/fichero.txt: es un comando similar a chmod del sistema de archivos de Linux, que se utiliza para cambiar los permisos en un archivo existente en HDFS.

- hdfs dfs –chown usuario /ruta/fichero.txt **:** cambia el dueño de un usuario (solo si el usuario que ejecuta la orden tiene permisos para hacerlo).

Ideas clave

Para reforzar el conocimiento y la implementación de los comandos de HDFS más frecuentes, visualiza el siguiente vídeo.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=c9996fae-fbd2-4eb1-ab82-ac6e00b69546

2.5. Programación distribuida y MapReduce

Una vez que Google tenía resuelto el problema del almacenamiento de ficheros masivos en el sistema de ficheros distribuido Google File System (GFS), Dean y Ghemawat publicaron un nuevo artículo (2008) sobre cómo aprovechar los *datanodes* para procesar estos ficheros que estaban almacenados de forma distribuida, particionados en bloques. Para ello, desarrollaron un paradigma de programación llamado MapReduce, que consiste en una manera abstracta de abordar los problemas para que puedan ser implementados y ejecutados sobre un clúster de ordenadores. Junto con el artículo, también liberaron una biblioteca de programación en lenguaje Java que implementaba este paradigma.

De forma más precisa, podemos definir MapReduce como:

> Modelo abstracto de programación general e implementación del modelo como bibliotecas de programación, para procesamiento paralelo y distribuido de grandes *datasets,* inspirado en la técnica «divide y vencerás».

La gran ventaja que proporciona tanto el modelo como la implementación que suministraron los autores es que **abstrae al programador de todos los detalles relativos a *hardware*, redes y comunicación entre los nodos,** con el fin de que pueda centrarse solamente en el desarrollo de *software* para solucionar su problema.

El punto de partida son archivos muy grandes almacenados (por bloques) en HDFS (en aquel momento, aún era GFS). ¿Podríamos aprovechar las CPU de los *datanodes* del clúster para procesar en paralelo (simultáneamente) los bloques de un archivo? No queremos mover datos (el tráfico de red es muy lento): **llevamos el cómputo (el código fuente de nuestro programa) al lugar donde están**

almacenados los datos (es decir, a los *datanodes*, y no al revés, como ocurría en el modelo tradicional de aplicación). Las CPU de los *datanodes* van a procesar (preferentemente) los datos que hay en ese *datanode*.

En el paradigma MapReduce, el usuario solo necesita escribir dos funciones (enfoque «divide y vencerás»):

- **Mapper:** función escrita por el usuario e invocada por el *framework* en paralelo (simultáneamente) sobre cada bloque de datos de entrada (que generalmente coincide con un bloque de HDFS). De este modo, se generan resultados intermedios, **que se presentan siempre en forma de (clave, valor)** y son escritos también en el disco local.

- **Reducer:** se aplica en paralelo para cada grupo creado por la función Mapper. En concreto, esta función se llama una vez para cada clave única generada de la salida de la función Mapper, junto con la cual se pasan todos los valores asociados que comparte.

Ejemplo: el problema de contar ocurrencias de palabras con MapReduce

Supongamos que queremos resolver un problema que consiste en, dado un texto, saber cuántas veces aparece en él cada palabra. Nuestro punto de partida es un fichero que contiene el texto y que está almacenado en HDFS. Por tanto, está particionado en varios bloques, cada uno de los cuales contiene un fragmento del texto. Asumimos que el texto está almacenado en formato ASCII, es decir, el fichero (y, por extensión, cada uno de sus bloques) contiene líneas de texto, tal como se indica en los rectángulos azules de la figura 4.

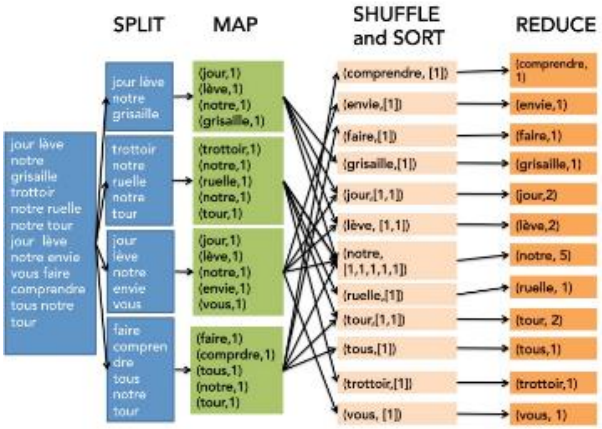


Figura. 4. Representación del problema de contar ocurrencias resuelto con MapReduce. Fuente: https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribues-sur-des-donnees-massives/4308626-divisez-et-distribuez-pour-regner#/id/r-4362891

En el modelo MapReduce, el programador (usuario de la API) solamente necesita escribir dos funciones. En primer lugar, una función llamada `map`, que reciba una línea de texto (cada línea de texto de los rectángulos azules, que representan cada uno un bloque de HDFS) y dé lugar, como resultado, a las tuplas que se muestran en los rectángulos verdes, de manera que cada rectángulo azul origina un rectángulo verde.

A continuación, las tuplas que genera la fase `map` son enviadas por la red que conecta los nodos del clúster: la librería de MapReduce, de forma automática y transparente al programador, lleva a cabo la operación `shuffle and sort`, mostrada en rosa. Esta, a partir de los resultados en verde, genera tuplas formadas por una palabra y una lista asociada (más adelante, explicaremos su significado). Lo que está sucediendo es que el propio *framework* está agrupando todas las tuplas que tienen la misma clave y está creando una lista con todos los valores asociados a esa clave común. Esto lo repite por cada clave diferente que encuentre en las tuplas generadas por la función `map` del usuario.

Para cada uno de estos agrupamientos mostrados en rosa, el *framework* invoca

automáticamente a la segunda función que el programador debe escribir: la función reduce. Esta es recibida por las tuplas mostradas en rosa (formadas por una palabra y la lista con el número de ocurrencias asociadas a ella) y genera, para cada una de ellas, un resultado como el mostrado en los rectángulos naranjas.

La función map que implementa el usuario debe recibir como entrada una tupla (clave_entrada, valor). En este problema, clave_entrada es el número de línea (ignorado) y valor representa una línea completa de texto. Es el *framework* el que, de forma automática, invoca, tantas veces como sea necesario, la función que ha implementado el usuario y le pasa los argumentos que hemos indicado. La invocación y ejecución de la función map se lleva a cabo de manera distribuida, en cada nodo del clúster (en los *datanodes*, que es donde residen los bloques de HDFS que contienen los fragmentos del texto que actúa como datos de entrada).

La implementación de la función map del usuario para este problema concreto podría dividir la línea de texto en palabras y, para cada palabra que forma parte de la línea de texto, devolver la tupla (palabra, 1), que indica que «palabra» ha aparecido una vez. Ejemplo: (hola, 1), (el, 1), (el, 1). La palabra es la *out_key* y el valor siempre es 1.

La función reduce que el usuario ha de implementar es también invocada automáticamente por el *framework*. Para ello, se toman como datos de entrada cada una de las claves y la lista de valores asociados a ellas obtenidas en la etapa anterior. Esta lista de valores está formada por el campo valor de todas las tuplas que comparten la misma clave, tal como las ha generado la fase map. La implementación de la función reduce debe agregar resultados (sumar las ocurrencias de una misma palabra). Ejemplo: la función reduce recibe (reduce, [1, 1, 1, 1, 1, 1, 1]) y da como resultado: (hola, 7).

Inconvenientes de MapReduce

MapReduce permite procesar los datos almacenados de manera distribuida en el

sistema de archivos (por ejemplo, HDFS) de un clúster de ordenadores. Esto ayuda a resolver todo tipo de problemas, pese a que no siempre es sencillo pensar la solución en términos de operaciones `map` y `reduce` encadenadas. En este sentido, se dice que es de propósito general, a diferencia de, por ejemplo, el lenguaje SQL, que está orientado específicamente a realizar consultas sobre los datos. Implementar en SQL algoritmos tales como un método de ordenación o el algoritmo de Dijkstra para encontrar caminos mínimos en un grafo no sería posible. Sin embargo, MapReduce presenta varios inconvenientes, algunos muy serios:

- El resultado de la fase `map` (tuplas) se escribe en el disco duro de cada nodo, como resultado intermedio. Los accesos a un disco duro son aproximadamente un orden de magnitud (es decir, diez veces) más lentos que los accesos a la memoria principal (RAM) de cada nodo, por lo que estamos penalizando el rendimiento debido a cómo está estructurado el propio *framework*.

- Después de la fase `map` , hay tráfico de red obligatoriamente (movimiento de datos, conocido como *shuffle)*. De hecho, el movimiento de datos forma parte como una etapa obligada en el *framework* de MapReduce. En cualquier aplicación distribuida, el movimiento de datos de un nodo a otro constituye un cuello de botella y es una operación que debe evitarse si es posible.

- Por otro lado, y relacionado con el punto anterior, para mover datos se necesita, en primer lugar, escribirlos temporalmente en el disco duro de la máquina origen, enviarlos por la red y luego escribirlos temporalmente en el disco duro de la máquina destino (el *shuffle* siempre va de disco duro a disco duro) para, finalmente, pasarlos a la memoria principal de dicho nodo.

- Estos dos inconvenientes se acentúan especialmente cuando el algoritmo que queremos implementar sobre un clúster es de tipo iterativo, es decir, requiere varias pasadas sobre los mismos datos para ir convergiendo. Este tipo de procesamiento es típico en algoritmos de *machine learning,* que es uno de los que más se puede beneficiar del procesamiento de grandes conjuntos de datos para extraer

conocimiento. Se comprobó que sus implementaciones con MapReduce no eran eficientes debido a la propia concepción del *framework*.

- Finalmente, enfocar cualquier problema en términos de operaciones map y reduce encadenadas no siempre es fácil ni intuitivo para un desarrollador, y la solución resultante puede ser difícil de mantener si no está bien documentada.

2.6. Referencias bibliográficas

Dean, J. y Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113. https://doi.org/10.1145/1327452.1327492

Ghemawat, S., Gobioff, H. y Leung, S-T. (2003). The Google File System. A *CM SIGOPS Operating Systems Review*, *37*(5), 29-43. https://doi.org/10.1145/1165389.945450

Guía de usuario HDFS

Hadoop Apache. (2020). *HDFS Users Guide*. The Apache Software Foundation. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

Sin duda, en la página oficial de Hadoop Apache podréis encontrar la documentación más actualizada sobre HDFS, con toda la información sobre sus características y su uso.

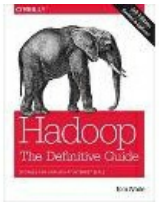
Glosario de términos de la empresa Databricks, creadores de Apache Spark

Databricks (2021). Hadoop. *Databricks*. https://databricks.com/glossary/hadoop

Aquí puedes consultar una definición completa sobre Hadoop. Son también interesantes todos los demás términos contenidos en el glosario, todos relacionados con las tecnologías *big data*.

Hadoop: the definitive guide

White, T. (2015). *Hadoop: the definitive guide* (4. a edición). O'Reilly.



Guía que incluye las principales tecnologías de Hadoop descritas en detalle. Para este tema, podéis centraros en el capítulo 3. El capítulo 2 es interesante, pero ha quedado obsoleto.

Arquitectura de HDFS en detalle

Chansler, R., Kuang, H., Radia, S., Shvachko, K. y Srinivas, S. (s. f.). The Hadoop distributed file system. En A. Brown y G. Wilson (eds.), *The architecture of open source applications. Elegance, evolution, and a few fearless hacks*. Aosa Book. http://www.aosabook.org/en/hdfs.html

En este capítulo, los autores describen la arquitectura de HDFS y narran su experiencia con este sistema para gestionar 40 *petabytes* de datos empresariales en la compañía Yahoo.

Test

1. ¿Cuánto ocupa en total un archivo de 500 MB almacenado en HDFS, sin replicación, si se asume el tamaño de bloque por defecto?

A. Ocupará 500 MB.

B. Ocupará 512 MB, que son 4 bloques de 128 MB, y hay 12 MB desperdiciados.

C. Ocupará lo que resulte de multiplicar 500 MB por el número de *datanodes* del clúster.

2. ¿Cuál de las siguientes afirmaciones respecto a HDFS es cierta?

A. El tamaño de bloque debe ser siempre pequeño para no desperdiciar espacio.

B. El factor de replicación es configurable por fichero y su valor, por defecto, es 3.

C. Las dos respuestas anteriores son correctas.

3. ¿Qué afirmación es cierta sobre el proceso de escritura en HDFS?

A. El cliente manda al *namenode* el fichero, que, a su vez, se encarga de escribirlo en los diferentes *datanodes*.

B. El cliente escribe los bloques en todos los *datanodes* que le ha especificado el *namenode*.

C. El cliente escribe los bloques en un *datanode* y este envía la orden de escritura a los demás.

4. En un clúster de varios nodos donde no hemos configurado la topología…

A. Es imposible que dos réplicas del mismo bloque caigan en el mismo nodo.

B. Es imposible que dos réplicas del mismo bloque caigan en el mismo *rack.*

C. Las dos respuestas anteriores son falsas.

5. Cuando usamos *namenodes* federados…

A. Cada *datanode* puede albergar datos de uno de los subárboles.

B. La caída de un *namenode* no tiene ningún efecto en el clúster.

C. Ninguna de las respuestas anteriores es correcta.

6. ¿Por qué se dice que HDFS es un sistema escalable?

A. Porque reemplazar un nodo por otro más potente no afecta a los *namenodes*.

B. Porque un clúster es capaz de almacenar datos a gran escala.

C. Porque se puede aumentar la capacidad del clúster añadiendo más nodos.

7. ¿Qué tipo de uso suele darse a los ficheros de HDFS?

A. Ficheros de cualquier tamaño que se almacenan temporalmente.

B. Ficheros de gran tamaño que se crean, no se modifican, y sobre los que se realizan frecuentes lecturas.

C. Ficheros de gran tamaño que suelen modificarse constantemente.

8. La alta disponibilidad de los *namenodes* de HDFS implica que…

A. La caída de un *namenode* apenas deja sin servicio al sistema de ficheros durante un minuto antes de que otro entre en acción.

B. Es posible escalar los *namenodes* añadiendo más nodos.

C. La caída de un *datanode* deja sin servicio al sistema durante unos pocos segundos hasta que este es sustituido.

9. El comando de HDFS para moverse a la carpeta `/mydata` es…

A. `hdfs dfs –cd /mydata.`

B. `hdfs dfs –ls /mydata.`

C. No existe ningún comando equivalente en HDFS.

10. ¿Qué inconveniente presenta MapReduce?

A. No es capaz de procesar datos distribuidos cuando son demasiado grandes.

B. Entre las fases `map` y `reduce`, siempre lleva a cabo escrituras a disco y movimiento de datos entre máquinas.

C. Es una tecnología propietaria y no es código abierto.