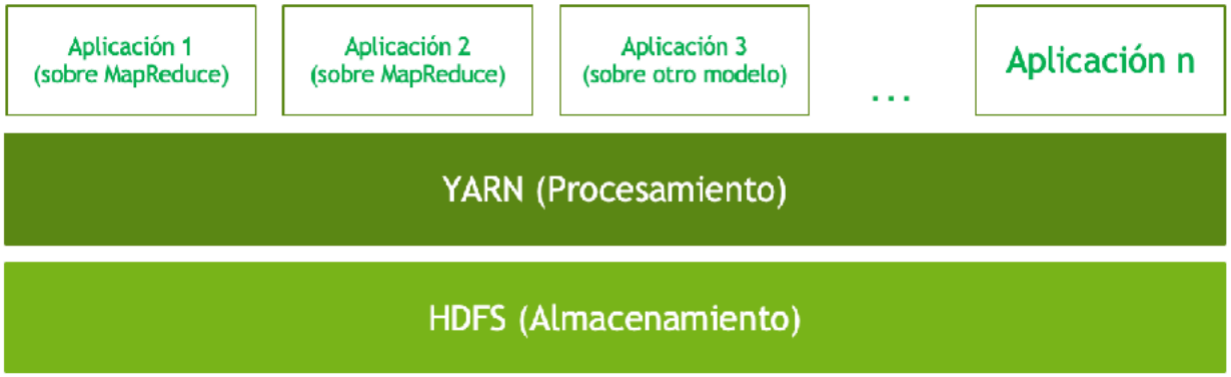
1 Almacenamiento y procesamiento en Hadoop.

En esta unidad vamos a entrar a detalle en lo que se conoce como el Core de Hadoop que son las bases de la plataforma sobre las que se construyen todas las herramientas. Esta base está formada por:

- HDFS, que es la capa de almacenamiento.
- YARN, que es el gestor de los procesos que se ejecutan en el clúster.
- MapReduce, que es un modelo de programación para desarrollar tareas de procesamiento de datos.



Los contenidos de la unidad serán los siguientes:

- En primer lugar vamos a entrar a conocer HDFS, entendiendo primero qué es y qué principales características tiene. A continuación conoceremos cómo funciona desde un punto de vista de alto nivel (su arquitectura), como a bajo nivel (cómo funcionan las operaciones de lectura y escritura), así como su uso.
- A continuación entraremos a detalle con YARN, con un esquema similar, es decir, conociendo en primer lugar sus principales características, y posteriormente su funcionamiento a alto y bajo nivel.
- Por último, veremos MapReduce, donde además plantearemos un ejemplo práctico sobre cómo programar con este paradigma sobre Hadoop.

2 HDFS.

HDFS es el sistema de almacenamiento de Hadoop. Es un sistema de almacenamiento distribuido, como la mayoría de funcionalidades de Hadoop, lo que significa que las operaciones no las realiza un único servidor, sino que múltiples servidores trabajan coordinados para almacenar u ofrecer los datos.

Como sabes, HDFS se inspiró en el paper de Google denominado Google File System, donde se explicaba la forma en la que Google resolvió el problema de almacenar "todo internet".

3 Introducción.

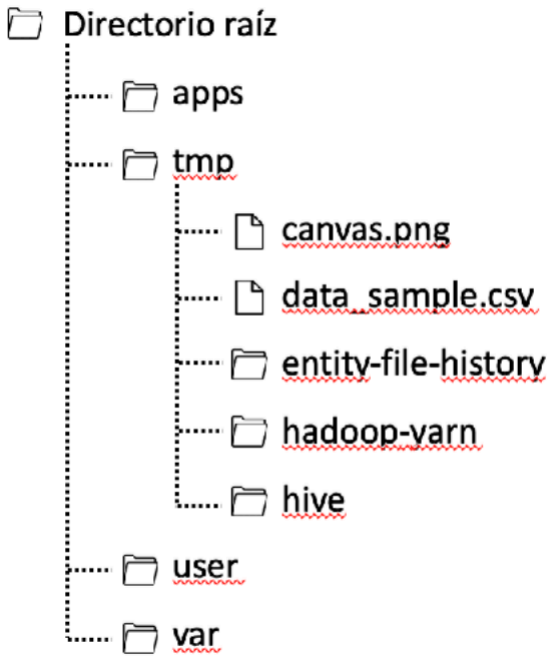
Hadoop Distributed File System, HDFS, es el sistema de almacenamiento de Hadoop, que tiene las siguientes características principales:

- **Es un sistema de ficheros distribuido**, es decir, se ejecuta sobre diferentes nodos que trabajan en conjunto ofreciendo a los usuarios y aplicaciones que utilizan el sistema, un interfaz como si sólo hubiera un único servidor por detrás. Es decir, para los usuarios de HDFS, es transparente su modelo distribuido, no teniendo que conocer su funcionamiento interno.
- Está diseñado para ejecutarse sobre **hardware commodity**, es decir, no requiere unos servidores específicos o costosos. Esto conlleva la necesidad de poder sobreponerse a los fallos que pudieran tener los servidores o algunas partes de los servidores.
- Está optimizado para almacenar **ficheros de gran tamaño** y para hacer operaciones de lectura o escritura masivas. Su objetivo es cubrir los casos de uso de analítica masiva, no los casos de uso que dan soporte a las operaciones de las empresas.
- Tiene capacidad para **escalar horizontalmente** hasta volúmenes de Petabytes y miles de nodos, y está diseñado para poder dar soporte a **múltiples clientes** con acceso concurrente. La escalabilidad se consigue añadiendo más servidores, lo cual es una operación relativamente sencilla, y en cuanto a la posibilidad de dar soporte a múltiples clientes, es una característica importante, ya que existen sistemas de almacenamiento masivo que no permiten el acceso concurrente de más de un cliente, o si lo soportan, su rendimiento decrece en gran medida (por ejemplo, los sistemas de almacenamiento basados en cinta).
- No establece **ninguna restricción sobre los tipos de datos** que se almacenan en el sistema, ya que éstos pueden ser estructurados, semiestructurados o no disponer de ninguna estructura, como el caso de imágenes o vídeos.
- HDFS tiene una orientación "**write-once, read many**", que significa "se escribe una vez, se lee muchas veces", es decir, asume que un archivo una vez escrito en HDFS no se modificará, aunque se puede acceder a él muchas veces.

3.1 ¿Qué almacena HDFS?

HDFS es un sistema de ficheros, es decir, es un sistema que permite guardar o recuperar ficheros. En este sentido, es igual que el sistema de ficheros que puede tener un ordenador personal con cualquier sistema operativo.

Los ficheros son almacenados en carpetas o directorios, pudiendo anidar carpetas y establecer diferentes niveles de anidación. Por ejemplo:



Los ficheros tienen un nombre y una extensión, distinguiendo entre mayúsculas y minúsculas, es decir, para HDFS, el fichero "video.mov" y "viDEO.mov" son diferentes. Esta característica la tienen todos los sistemas Unix y Linux, a diferencia de los sistemas Windows.

```
[hadoop@ip-172-31-11-171 ~]$ hadoop fs -ls /tmp
Found 7 items
-rw-r--r--   1 inigo  hdfsadmingroup     201220 2022-06-14 14:27 /tmp/canvas.png
-rw-r--r--   1 maria  hdfsadmingroup    2950156 2022-06-14 14:31 /tmp/data_sample.csv
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-14 14:20 /tmp/entity-file-history
drwxrwxrwx   - mapred mapred                 0 2022-06-14 14:20 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-14 14:23 /tmp/hive
-rw-r--r--   1 inigo  hdfsadmingroup   20650891 2022-06-14 14:29 /tmp/viDEO.mov
-rw-r--r--   1 inigo  hdfsadmingroup   20650891 2022-06-14 14:29 /tmp/video.mov
```

En la imagen anterior, que se corresponde con un listado de ficheros obtenido mediante consola (ya se comentará más adelante cómo se accede a HDFS), se puede ver en la parte derecha el listado de ficheros donde podrás comprobar:

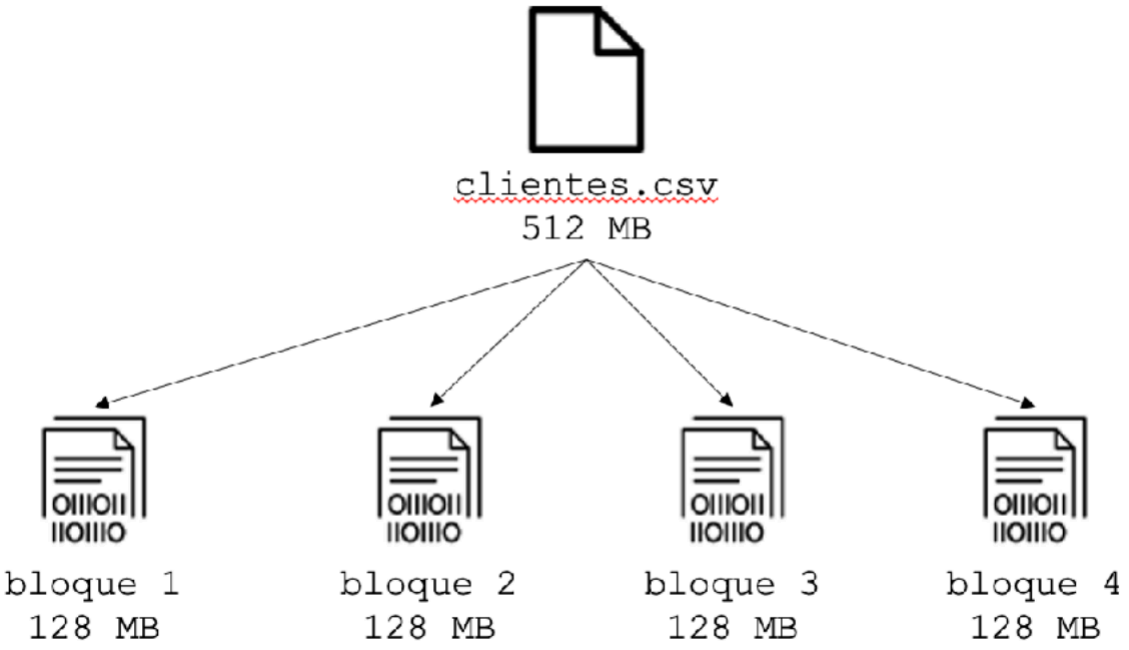
- Que los ficheros video.mov y viDEO.mov son diferentes.
- Que el directorio tmp contiene tanto otros directorios (hadoop-yarn, hive o entity-file-history) como ficheros, que contienen un nombre y una extensión. La extensión, como supongo que sabrás, indica el tipo de fichero.

Asimismo, HDFS almacena para cada fichero una serie de datos (mejor dicho, metadatos), sobre la fecha (1), el tamaño del fichero (2), el propietario y el grupo al que pertenece el propietario (3), así como los permisos que tienen el resto de usuarios sobre el fichero (4).

```
[hadoop@ip-172-31-11-171 ~]$ hadoop fs -ls /tmp
Found 7 items
-rw-r--r--   1 inigo  hdfsadmingroup     201220 2022-06-14 14:27 /tmp/canvas.png
-rw-r--r--   1 maria  hdfsadmingroup    2950156 2022-06-14 14:31 /tmp/data_sample.csv
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-14 14:20 /tmp/entity-file-history
drwxrwxrwx   - mapred mapred                 0 2022-06-14 14:20 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-14 14:23 /tmp/hive
-rw-r--r--   1 inigo  hdfsadmingroup   20650891 2022-06-14 14:29 /tmp/viDEO.mov
-rw-r--r--   1 inigo  hdfsadmingroup   20650891 2022-06-14 14:29 /tmp/video.mov
4                    3                2        1
```

3.2 ¿Cómo consigue tener tolerancia a fallos?

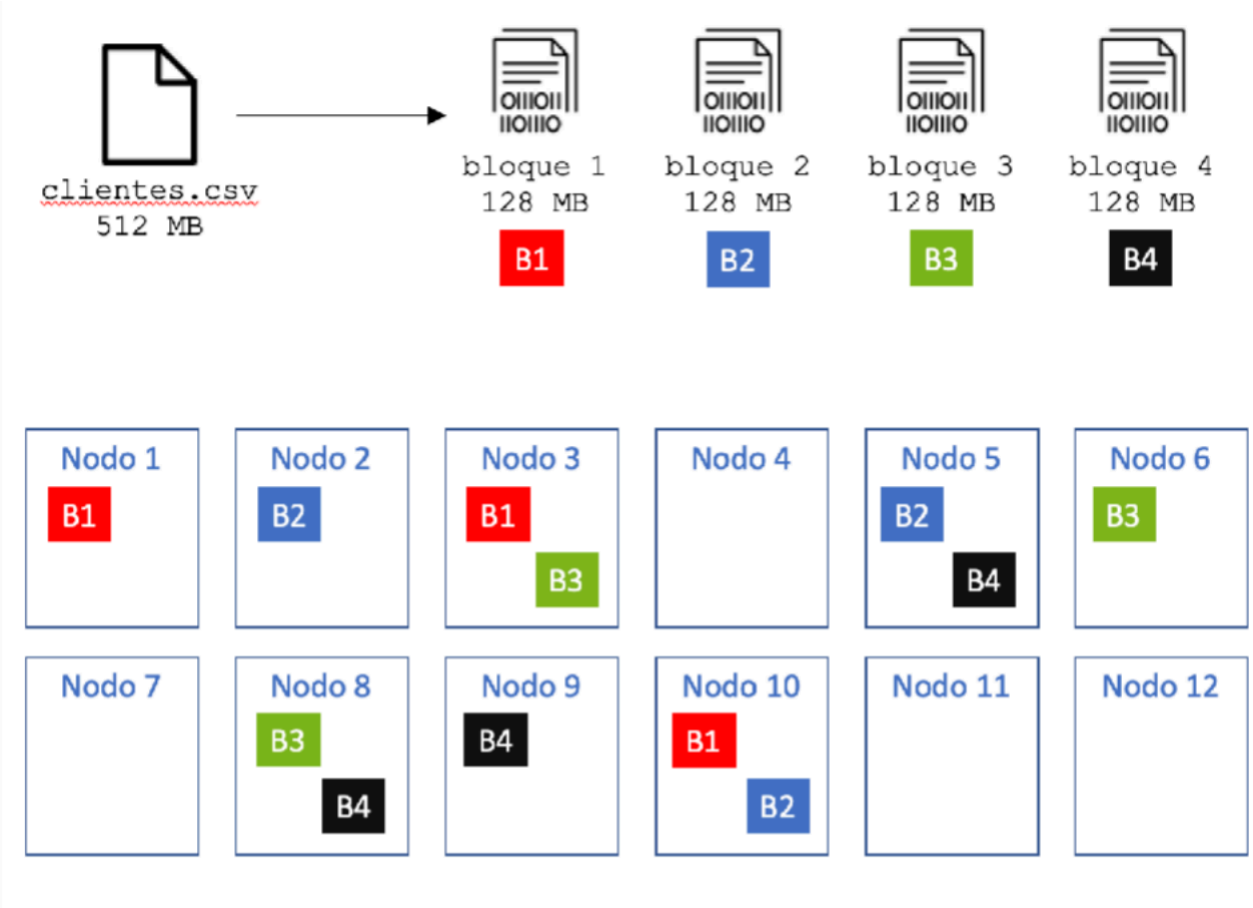
En HDFS, los ficheros se dividen en bloques, como en la mayoría de sistemas de ficheros. Sin embargo, el tamaño de un bloque en HDFS es muy grande, de 128 megabytes por defecto. En el sistema operativo de un PC (Windows, Linux, etc.), el tamaño suele ser de 512 bytes o 4 kilobytes, es decir, unas 50.000 veces más pequeño que en HDFS.



El bloque es la unidad mínima de lectura es un bloque, lo que significa que aunque tengamos un fichero que ocupa 1 kilobyte, tendremos que leer o escribir 128 megabytes cada vez que queramos operar con el fichero. Para ficheros grandes, por ejemplo, de 500 gigabytes, la ventaja que aporta es que hay que buscar y leer o escribir muchos menos bloques. Esta característica explica por qué Hadoop está diseñado para ficheros grandes y lecturas masivas, y por qué tiene un mal rendimiento para operaciones pequeñas.

Por lo tanto, cuando queremos escribir un fichero en HDFS, lo primero que se hace es dividir el fichero en bloques. A continuación, los bloques son almacenados en diferentes nodos, no siendo necesario que los bloques de un mismo fichero estén en un mismo nodo. Además, un aspecto importante es que cada bloque se replica (se copia) en más de un nodo, lo que se conoce como el factor de replicación. El factor de replicación por defecto en HDFS es 3, lo que significa que cada bloque tiene 3 copias almacenadas en 3 nodos diferentes. La replicación es el mecanismo con el que se consigue, entre otras cosas, la tolerancia a fallos.

Al tener varias réplicas de cada bloque en diferentes nodos, en caso de que un nodo se caiga, o que un disco de un nodo se corrompa, HDFS dispondrá de otras copias, por lo que no se perderán los datos.



En el ejemplo anterior, si se cayera el nodo 3, HDFS dispondría de otras dos copias por cada bloque que almacena del fichero.

El factor de replicación puede configurarse a nivel de fichero o directorio, es decir, podemos elegir un factor de replicación diferente para los ficheros o directorios que consideremos. Cuanto mayor sea el factor de replicación, más difícil será que perdamos los datos e incluso mejorará el rendimiento en las lecturas, porque para leer un bloque, HDFS podrá utilizar cualquier nodo. Sin embargo, un factor de replicación alto hace que las escrituras tengan peor rendimiento, al tener que hacer muchas copias en cada escritura, y además, consumirá más espacio real en disco.

4 Arquitectura.

La arquitectura de HDFS consta de distintos servicios y tipos de nodo, aunque fundamentalmente son tres tipos, el **Namenode**, el **Secondary Namenode** y los nodos **Datanode**

4.1 Namenode

El nodo Namenode actúa de **maestro**, manteniendo la metainformación de todo el sistema de ficheros, esto es:

- La estructura de directorios, subdirectorios y los ficheros.
- La información de los ficheros: tamaño, fecha de modificación, propietario, permisos, etc
- El factor de replicación de cada fichero.
- Los bloques que componen cada fichero.
- La ubicación de los distintos bloques (en qué nodo se encuentran).

La información es almacenada tanto en disco, para garantizar la durabilidad en caso de una caída del servidor, como en memoria, para poder acceder a la información lo más rápido posible y optimizar el rendimiento.

Además de gestionar la metainformación, **coordina todas las lecturas y escrituras**, y controla el funcionamiento de los Datanodes, es decir, detecta si hay algún fallo en algún nodo y toma las acciones necesarias en caso de que alguno esté caído o con fallos.

Es importante que el Namenode sea **robusto y no tenga caídas**. Por este motivo, se utiliza hardware más resiliente que en el caso de los Datanodes, por ejemplo, con fuentes de alimentación dobles o con una disposición de discos en RAID 1, RAID 10 o RAID 5, para tener dos discos con idénticos datos y de esta forma, no estar desprotegidos ante una rotura de uno de ellos. Asimismo, se suelen planificar copias de seguridad con bastante frecuencia para tener una salvaguarda de la información.

4.2 Secondary Namenode

Para mejorar la tolerancia a fallos, suele existir un nodo secundario del maestro, denominado **Secondary Namenode**

El NameNode es el único punto de fallo en HDFS ya que, si el Namenode falla, se pierde todo el sistema de archivos HDFS. Para reducir este riesgo esto, Hadoop implementó el Secondary Namenode, que no estaba presente en las primeras versiones de HDFS.

El Secondary Namenode es un nodo cuya función principal es tomar puntos de control de los metadatos del sistema de ficheros del Namenode. **No es un nodo de respaldo** ya que en caso de caída del Namenode, no puede tomar el control y continuar el servicio sin parada, ya que su único objetivo es **reducir el tiempo de arranque del Namenode**

En detalle, la función principal del Secondary Namenode es almacenar una copia de dos

ficheros de log del Namenode: FsImage y EdiLlog:

- **FsImage** es una instantánea de los metadatos del sistema de archivos HDFS que se realiza cada cierto tiempo.
- **EditLog** es un registro de los cambios que se producen en los metadatos del sistema de archivos. EditLog se borra cada vez que hay una nueva instantánea en FsImage.

Por lo tanto, en caso de una caída del Namenode, añadiendo los cambios de EditLog a la imagen de FsImage se puede restaurar el estado del sistema de archivos. Cada vez que se arranca el servicio Namenode, se realiza esta fusión de los datos de EditLog y FsImage. El cometido del nodo Secondary Namenode es mantener una versión de FsImage y EditLog lo más completa y sencilla posible para reducir el tiempo de ajuste del Namenode ante un reinicio, por ejemplo, tras una caída.

Por último, el Secondary Namenode habitualmente se ejecuta en una máquina diferente a la del NameNode principal para poder sobrevivir a las caídas del Namenode.

En cuanto a los requisitos hardware, suele utilizar servidores con las mismas características del Namenode.

4.3 Datanode

Los datanodes son los servicios que se encuentran en los nodos worker, y su labor principal es **almacenar o leer los bloques que componen los ficheros** que están almacenados en HDFS, con las siguientes particularidades:

- El Datanode sólo conoce los bloques que contiene, pero no sabe a qué fichero pertenecen o dónde se encuentran el resto de bloques del fichero. Toda esta información sólo está en el Namenode, por lo que este nodo, el Namenode, es crítico para HDFS.
- Con cierta periodicidad, inicialmente cada hora, los Datanodes envían al Namenode la lista de los bloques que almacenan, para que el Namenode pueda tener una lista actualizada de los bloques y su ubicación.
- Por cada bloque, los Datanodes almacenan un checksum para detectar si el bloque está corrupto, es decir, para garantizar su integridad. Un checksum es una operación matemática que se realiza con el contenido de un bloque, de manera que si los datos de un bloque se ven alterados, por ejemplo, por un fallo en el disco, al leer el bloque y calcular su checksum, éste no coincidirá con el calculado y almacenado en su creación y se podrá descubrir que el bloque está corrupto.
- Adicionalmente, el Datanode envía un latido (heartbeat), que es un mensaje corto indicando que está levantado, al Namenode. El intervalo de latido predeterminado es de 3 segundos. Si un Datanode no envía latidos al Namenode en diez minutos, entonces el Namenode considera que el Datanode está fuera de servicio y que las réplicas de bloques alojadas por ese Datanode no están disponibles. Posteriormente, el Namenode programa la creación de nuevas réplicas de esos bloques en otros Datanodes para garantizar el número de réplicas por bloque.

Los servicios Datanode suelen ir en los nodos worker, y por lo tanto, suelen tener un hardware con poco nivel de sofisticación en cuanto a resiliencia, teniendo como principal característica de hardware que suelen disponer de una gran cantidad de discos

(habitualmente el número de discos es el número de cores totales menos uno o dos).

4.4 Recuerda:

Namenode	• Coordina el trabajo de los Datanodes. • Almacena toda la información sobre los ficheros, los bloques y los Datanodes. • Verifica que los Datanodes están activos. • Es el punto único de fallo de HDFS. • Suelen tener mecanismos de tolerancia a fallos: redundancia de discos, etc.
Secondary Namenode	• Facilita el proceso de arranque de un Namenode en caso de caída. • Almacena el estado de HDFS mediante dos ficheros: FsImage y EditLog. • Se ejecuta en un nodo diferente al Namenode.
Datanode	• Almacena y lee los bloques de los ficheros almacenados en HDFS. • No dispone de información de los ficheros o estructura de directorios de HDFS. • Envía un mensaje al Namenode para avisar de que se encuentra activo. • En caso de caída, no se pierden datos y HDFS sigue funcionando correctamente. • Se ejecuta en nodos hadrware commodity, habitualmente con muchos discos.

5 Funcionamiento (lectura y escritura).

Los datos que se escriben en HDFS son inmutables, es decir, no pueden ser modificados.

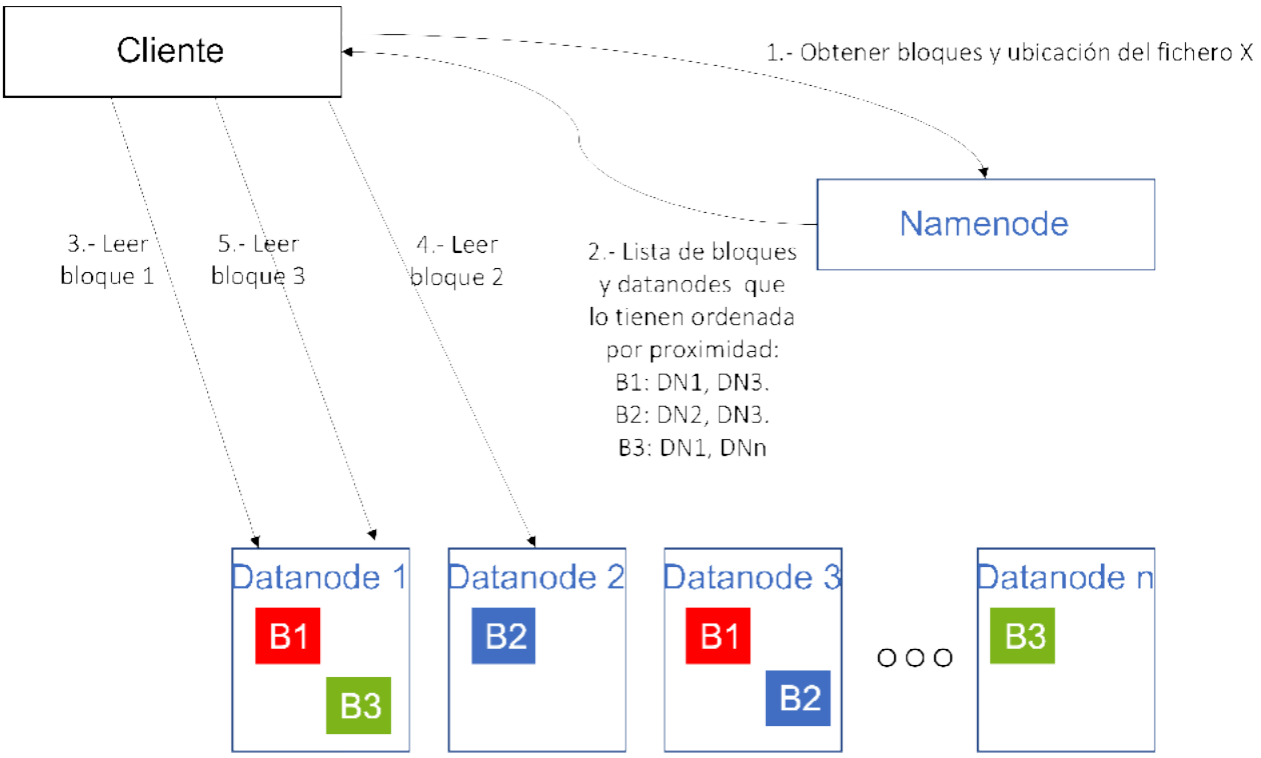
Esto significa que HDFS sólo permite añadir contenido a los ficheros, así que por ejemplo, si en un fichero de 256 megabytes se pretende modificar un carácter, HDFS creará un nuevo bloque con el cambio y lo escribirá por completo, borrando el bloque anterior.

Esto, junto con la característica del tamaño de bloque de 128 megabytes, que es la unidad mínima de lectura, hace que el rendimiento de HDFS para operaciones sencillas sobre registros aleatorios sea muy pobre. Recuerda que HDFS está pensado para ficheros grandes y lecturas masivas.

HDFS proporciona dos tipos de operaciones básicas con los ficheros: leer y escribir un fichero. A continuación vamos a conocer cómo funcionan estas operaciones:

5.1 Lectura

La lectura de ficheros en HDFS, es decir, cuando un cliente quiere leer un fichero completo que se encuentra en el sistema, tiene el siguiente esquema de secuencia (se ha resumido para no entrar demasiado a detalle):



Los pasos son los siguientes:

El cliente que desea leer un fichero de HDFS, mediante una librería instalada en su equipo, realiza una llamada al Namenode para conocer qué bloques forman un

fichero (llamemos X al fichero), así como los Datanodes que contienen cada uno de los bloques.

2 El Namenode retorna dicha información, y ordena para cada bloque los Datanodes que contienen dicho bloque en función de la distancia al cliente (un algoritmo evalúa la distancia entre el cliente y cada Datanode). El objetivo de esta lista ordenada es intentar reducir el tiempo de acceso a cada Datanode desde el cliente

3 Con la información recibida del Namenode, el cliente se comunica directamente con el Datanode 1 para solicitarle el primer bloque.

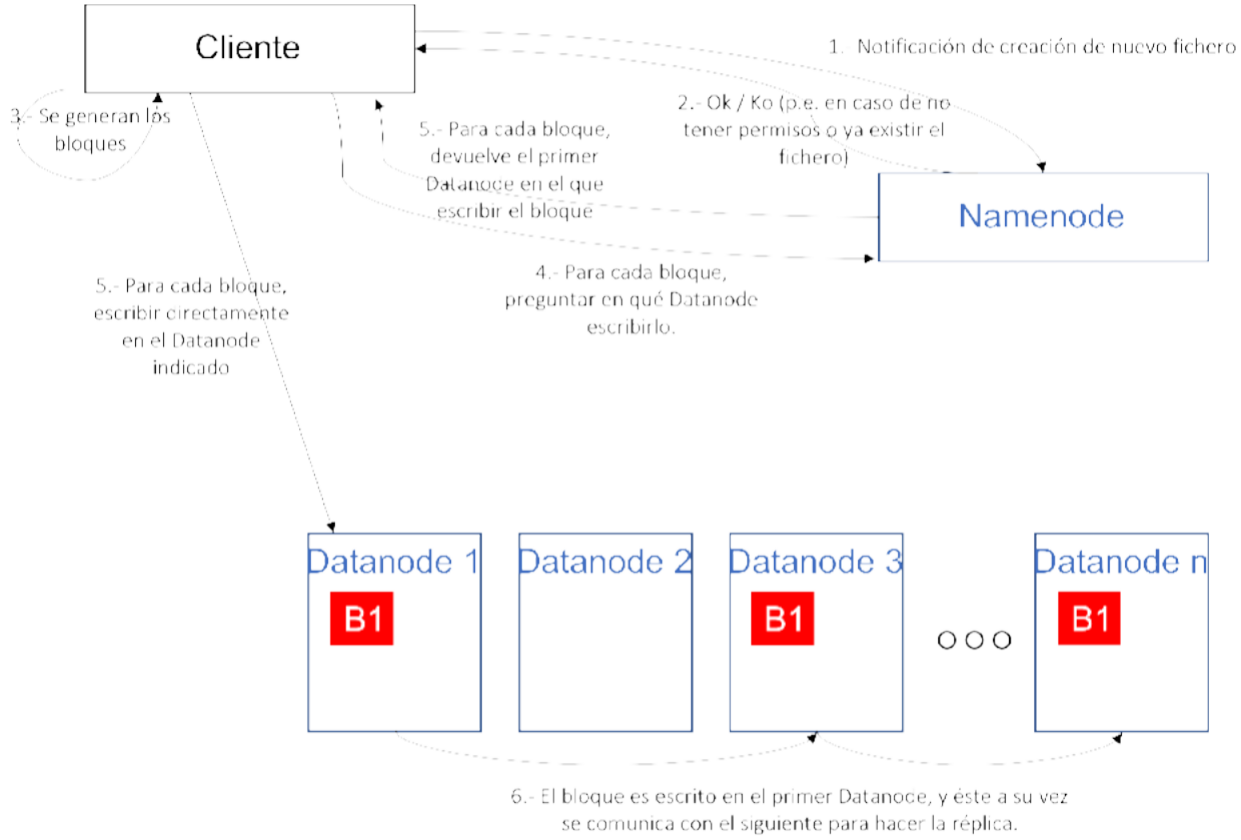
4 El cliente se comunica con el Datanode 2 para obtener el bloque 2.

5 El cliente se comunica con el Datanode 1 para obtener el bloque 3.

Es preciso indicar que durante la operación, la única responsabilidad del Namenode es devolver al cliente la lista de bloques y la ubicación de los mismos, pero no interviene en las lecturas. Es decir, para realizar las lecturas de cada bloque, **el cliente se comunica directamente con los Datanodes, sin que los datos pasen por el Namenode**. Esto hace que el Namenode no sea cuello de botella del proceso, y pueda atender múltiples peticiones en paralelo, ya que no le supone mucho esfuerzo de computación atender las diferentes solicitudes de los clientes.

5.2 Escritura

En el caso de las escrituras, un esquema simplificado de esta operación lo encontramos en la siguiente imagen:



Los pasos son los siguientes:

1. El cliente, que desea escribir un fichero, invoca a un servicio del Namenode para solicitar la creación del fichero, indicando en la llamada el nombre y la ruta en la que desea guardarlo.
2. El Namenode realiza una serie de verificaciones, como los permisos del usuario/cliente en el directorio, si el fichero ya existe, etc. En caso de que todas las verificaciones sean correctas, devuelve un OK, en caso contrario un KO.
3. El cliente comienza a generar los bloques en los que se dividirá el fichero utilizando una librería de HDFS.
4. Para cada bloque que desea escribir el cliente, se invoca al Namenode para obtener el Datanode en el que escribir el bloque.
5. El Namenode devuelve la lista de Datanodes en los que escribir el bloque, y el cliente escribe dicho bloque en el primer Datanode obtenido, realizando una comunicación directamente con dicho Datanode.
6. Una vez escrito el bloque en el primer Datanode, éste es responsable de comunicarse con el siguiente Datanode en la cadena para que escriba una copia del bloque. Una vez todos los Datanodes han escrito la réplica, se devuelve un "Ok" al cliente para que escriba el siguiente bloque.

Al igual que en el caso de la lectura, es importante señalar que el Namenode no recibe en ningún momento los datos del fichero, sino que se limita a resolver las cuestiones relacionadas con la ubicación de cada bloque. De esta manera, liberando al Namenode de la operativa de escritura, permite optimizar el funcionamiento y que el Namenode no se convierta en el cuello de botella de HDFS en las escrituras de fichero.

6 Uso

HDFS soporta operaciones similares a los sistemas Unix:

- Lectura, escritura o borrado de ficheros.
- Creación, listado o borrado de directorios.
- Usuarios, grupos y permisos.

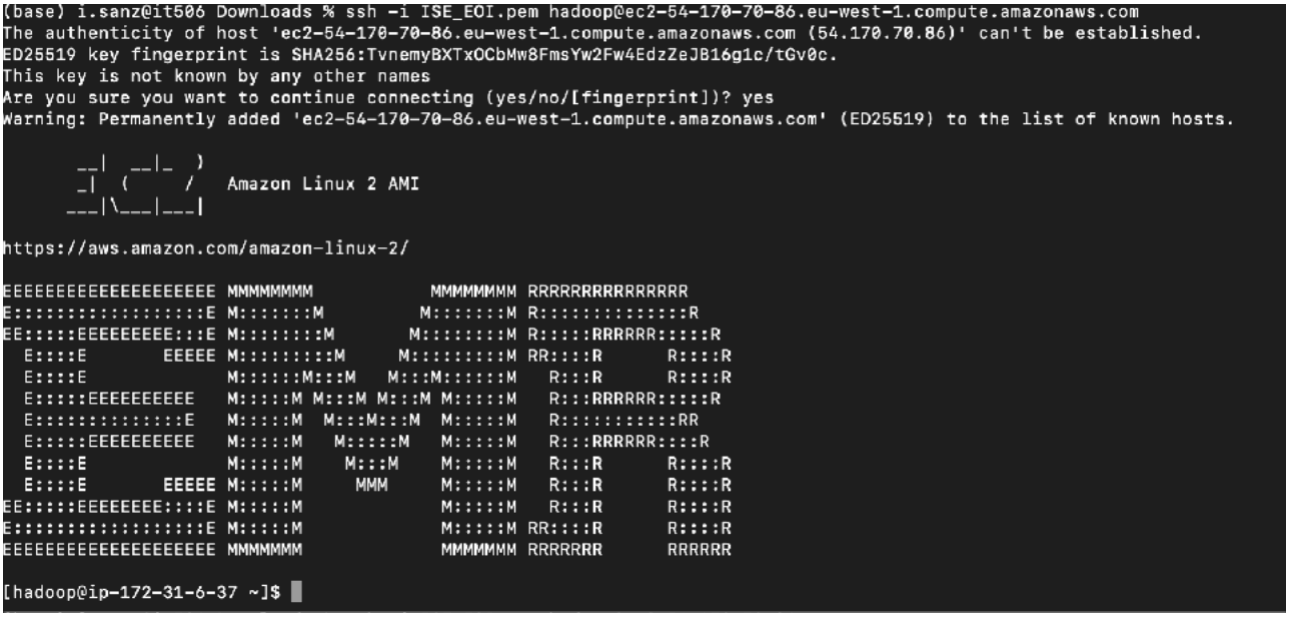
En cuanto a los interfaces con los que poder usar el sistema de ficheros, ofrece diferentes interfaces, siendo los principales los mencionados a continuación:

- **Cliente de línea de comandos**: HDFS dispone de un amplio número de comandos que pueden ser ejecutados en consola. Estos comandos representan la práctica totalidad de las operaciones que se pueden realizar con HDFS.
- **Java API**: HDFS está escrito en Java de forma nativa y ofrece un API que puede ser utilizado por aplicaciones con el mismo lenguaje.
- **RestFul API_(WebHDFS)**: para poder utilizar HDFS desde otros lenguajes, HDFS ofrece su funcionalidad mediante un servicio HTTP mediante el protocolo WebHDFS. Este interfaz, sin embargo, ofrece un rendimiento inferior al API de Java al utilizar HTTP como capa de transporte, por lo que no debería utilizarse para operaciones masivas o con alto volumen de datos.
- **NFS interface (HDFS NFS Gateway)**: es posible montar HDFS en el sistema de archivos de un cliente local utilizando la puerta de enlace NFSv3 de Hadoop. Luego puede usar las utilidades de Unix (como ls y cat) para interactuar con el sistema de archivos, cargar archivos y, en general, usar bibliotecas POSIX para acceder al sistema de archivos desde cualquier lenguaje de programación. Agregar a un archivo funciona, pero las modificaciones de un archivo no, ya que HDFS sólo puede añadir datos a un archivo.
- **Librería C**: HDFS ofrece una librería escrita en C, llamada *libhdfs*, que tiene un buen rendimiento, pero que no suele ofrecer toda la funcionalidad del API Java.

A continuación vamos a enumerar los principales comandos de HDFS en modo línea de comandos. Las operaciones que veremos a continuación son prácticamente idénticas a las que HDFS ofrece en el resto de interfaces:

6.1 Cliente de línea de comandos

Para acceder al cliente de línea de comandos, sólo tenemos que abrir una consola sobre un nodo que contenga los demonios de Hadoop ejecutándose, normalmente sobre el Namenode.



Una vez dentro del sistema, el comando **hadoop fs** nos proporcionará todas las funcionalidades sobre HDFS. Si se introduce sólo el comando, nos ofrecerá la lista de opciones o comandos disponibles. Algunos de los comandos más utilizados son los siguientes:

```
[hadoop@ip-172-31-6-37 ~]$ 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] [-d] <localsrc> ... <dst>]
        [-copyToLocal [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] <path> ...]
        [-cp [-f] [-p | -p[topax]] [-d] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] [-x] <path> ...]
        [-expunge]
        [-find <path> ... <expression> ...]
        [-get [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] [-skip-empty-file] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] [-safely] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-truncate [-w] <length> <path> ...]
        [-usage [cmd ...]]

Generic options supported are:
-conf <configuration file>        specify an application configuration file
-D <property=value>               define a value for a given property
-fs <file:///|hdfs://namenode:port> specify default filesystem URL to use, overrides 'fs.defaultFS' property from configurations.
-jt <local|resourcemanager:port>  specify a ResourceManager
-files <file1,...>                specify a comma-separated list of files to be copied to the map reduce cluster
-libjars <jar1,...>               specify a comma-separated list of jar files to be included in the classpath
-archives <archive1,...>          specify a comma-separated list of archives to be unarchived on the compute machines

The general command line syntax is:
command [genericOptions] [commandOptions]
```

6.1.1 mkdir

Similar al comando Unix **mkdir**, se usa para crear directorios en HDFS.

Sintaxis:

- **hadoop fs -mkdir [-p] <path>**

Ejemplos:

- **hadoop fs -mkdir /tmp/BDA01**
- **hadoop fs -mkdir /tmp/BDA01/hadoop-core**

6.1.2 ls

Similar al comando Unix **ls**, se usa para listar directorios en HDFS, es decir, para mostrar su contenido. La opción -R se puede utilizar para hacer un listado recursivo, es decir, para mostrar el contenido de los subdirectorios que están dentro del directorio que vamos a listar.

Sintaxis:

- **hadoop fs -ls [-d] [-h] [-R] <path>**

Ejemplos:

- **hadoop fs -ls /tmp/**

```
[[hadoop@ip-172-31-6-37 ~]$ hadoop fs -ls /tmp/
Found 4 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 00:49 /tmp/BDA01
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history
drwxrwxrwx   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive
```

- **hadoop fs -ls -R /tmp**

```
[hadoop@ip-172-31-6-37 ~]$ hadoop fs -ls -R /tmp/
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 00:49 /tmp/BDA01
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 00:49 /tmp/BDA01/hadoop-core
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history/active
drwx------   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history/done
drwxrwxrwx   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn
drwxrwxrwt   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn/staging
drwxrwxrwt   - mapred mapred                  0 2022-06-15 00:40 /tmp/hadoop-yarn/staging/history
drwxrwx---   - mapred mapred                  0 2022-06-15 00:40 /tmp/hadoop-yarn/staging/history/done
drwxrwxrwt   - mapred mapred                  0 2022-06-15 00:40 /tmp/hadoop-yarn/staging/history/done_intermediate
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive
drwx------   - hive   hdfsadmingroup          0 2022-06-15 00:43 /tmp/hive/hive
drwx------   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive/hive/27a90a8d-932f-477d-9fe2-164f3ba0192b
drwx------   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive/hive/27a90a8d-932f-477d-9fe2-164f3ba0192b/_tmp_space.db
drwx------   - hive   hdfsadmingroup          0 2022-06-15 00:43 /tmp/hive/hive/93939191-3661-4617-92a3-5a1eeb39f0d6
drwx------   - hive   hdfsadmingroup          0 2022-06-15 00:43 /tmp/hive/hive/93939191-3661-4617-92a3-5a1eeb39f0d6/_tmp_space.db
```

6.1.3 put

Copia archivos del sistema de archivos local a HDFS. Esto es similar al comando **-copyFromLocal**

Sintaxis:

- **hadoop fs -put [-f] [-p] <localsrc> ... <dst>**

Ejemplo:

- **hadoop fs -put /tmp/hadoop-state-pusher.config /tmp/BDA01**

```
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -put /tmp/hadoop-state-pusher.config /tmp/BDA01
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -ls /tmp/BDA01
Found 2 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 00:49 /tmp/BDA01/hadoop-core
-rw-r--r--   1 hadoop hdfsadmingroup        683 2022-06-15 01:05 /tmp/BDA01/hadoop-state-pusher.config
```

6.1.4 get

Copia archivos de HDFS al sistema de archivos local. Esto es similar al comando **-copyToLocal**

Ejemplo:

- **hadoop fs -cp /tmp/BDA01/hadoop-state-pusher.config /tmp/**

6.1.5 cat

Similar al comando **cat** de Unix, se usa para mostrar el contenido de un archivo.

Ejemplo:

- **hadoop fs -cat /tmp/BDA01/hadoop-state-pusher.config**

```
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -cat /tmp/BDA01/hadoop-state-pusher.config
log4j.rootLogger=INFO,DRFA
log4j.threshhold=ALL
log4j.appender.DRFA=org.apache.log4j.DailyRollingFileAppender
log4j.appender.DRFA.File=/var/log/hadoop-state-pusher/hadoop-state-pusher.log
log4j.appender.DRFA.DatePattern=.yyyy-MM-dd-HH
log4j.appender.DRFA.layout=org.apache.log4j.PatternLayout
log4j.appender.DRFA.layout.ConversionPattern=%d{ISO8601} %p %c (%t): %m%n
log4j.logger.org.apache.commons.httpclient.contrib.ssl.AuthSSLX509TrustManager=WARN

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %m%n
```

6.1.6 cp

Similar al comando `cp` de Unix, se usa para copiar archivos de un directorio a otro dentro de HDFS.

Ejemplo:

- **hadoop fs -cp /tmp/BDA01/hadoop-state-pusher.config /tmp**

```
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -cp /tmp/BDA01/hadoop-state-pusher.config /tmp/
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -ls /tmp
Found 5 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 01:05 /tmp/BDA01
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history
-rw-r--r--   1 hadoop hdfsadmingroup        683 2022-06-15 01:16 /tmp/hadoop-state-pusher.config
drwxrwxrwx   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive
```

Similar al comando `rm` de Unix, se usa para eliminar un archivo de HDFS. La opción `-R` se puede usar para la eliminación recursiva, es decir, para borrar además los subdirectorios que están en el directorio indicado.

Ejemplo:

- **hadoop fs -rm /tmp/hadoop-state-pusher.config**

```
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -ls /tmp
Found 5 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 01:05 /tmp/BDA01
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history
-rw-r--r--   1 hadoop hdfsadmingroup        683 2022-06-15 01:16 /tmp/hadoop-state-pusher.config
drwxrwxrwx   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -rm /tmp/hadoop-state-pusher.config
Deleted /tmp/hadoop-state-pusher.config
[hadoop@ip-172-31-6-37 tmp]$ hadoop fs -ls /tmp
Found 4 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2022-06-15 01:05 /tmp/BDA01
drwxrwxrwt   - yarn   hdfsadmingroup          0 2022-06-15 00:39 /tmp/entity-file-history
drwxrwxrwx   - mapred mapred                  0 2022-06-15 00:39 /tmp/hadoop-yarn
drwx-wx-wx   - hive   hdfsadmingroup          0 2022-06-15 00:42 /tmp/hive
```

Mueve archivos de HDFS de una ruta a otra. A diferencia del comando `cp`, el fichero desaparece de la ruta origen (con `cp` se hace una copia en la ruta de destino, por lo que el fichero origen no desaparece).

Ejemplo:

- **hadoop fs -mv /tmp/BDA01/hadoop-state-pusher.config /tmp/**

6.1.7 setrep

Modifica el factor de replicación de un fichero o un directorio. Ya sabes que el factor de replicación por defecto es 3. Con este comando se puede modificar para un fichero o directorio concreto.

Ejemplo:

- **hadoop fs -setrep 6 /tmp/BDA01/hadoop-state-pusher.config**

7 YARN.

Hagamos un símil con tu ordenador personal: ¿qué crees que es lo que te ofrece tu ordenador personal si lo simplificamos mucho?

Realmente, si lo piensas bien, te ofrece un sistema de almacenamiento, con uno o dos discos duros, y con los datos que almacenas, te ofrece la capacidad de ejecutar aplicaciones.

Hadoop se podría decir que hace más o menos lo mismo, aunque está más orientado a analizar los datos que almacenas más que operar con los datos.

Tu ordenador tiene un disco, que sería el equivalente a HDFS, y luego tiene un sistema operativo que controla todas las aplicaciones que se ejecutan. El sistema operativo puede ser Windows, Linux, MacOS, etc.

En Hadoop, **el sistema operativo es YARN**

Vamos a ver qué hace YARN con más detalle, y cómo funciona.

8 Introducción.

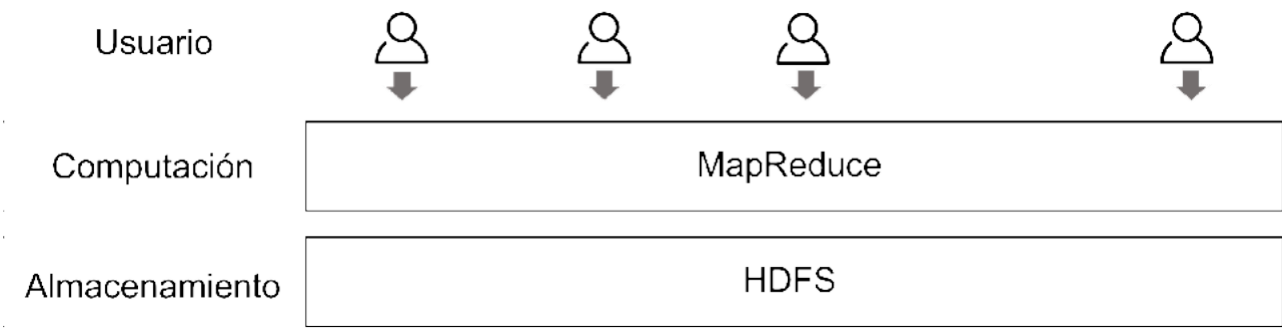
YARN es el acrónimo de Yet Another Resource Negotiator, es decir, según su acrónimo es un gestor de recursos.

En las primeras versiones de Hadoop, todo el procesamiento se realizaba con MapReduce, que es un framework de procesamiento distribuido que veremos en el siguiente apartado, y que, pese a que su funcionamiento era correcto, ya que era capaz de ofrecer la capacidad de desarrollar aplicaciones complejas que procesaran un gran volumen de datos, tenía varios problemas:

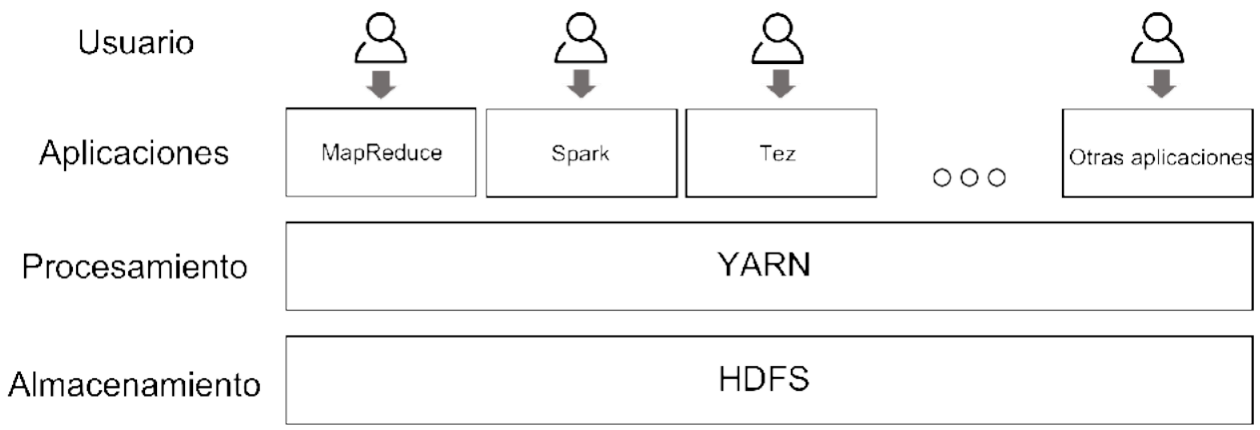
- Restringía mucho el tipo de aplicaciones que los desarrolladores podían realizar, ya que había que ceñirse a las operaciones y forma de ejecución que MapReduce ofrecía, por lo que era difícil utilizar los datos de HDFS para otro tipo de usos como el procesamiento en tiempo real.
- MapReduce es un modelo de programación muy poco eficiente, lo que hace que los casos de uso que requieren respuestas rápidas no sean viables, o simplemente, hace que todos los casos de uso se ralenticen.
- La concurrencia en la ejecución de aplicaciones no estaba bien resuelta, por lo que cuando un usuario o aplicación lanzaba un trabajo MapReduce, se podría decir que el resto tenía que esperar a que terminara la tarea para poder lanzar nuevos trabajos.

Con esta problemática, el crecimiento de Hadoop hacia una plataforma de datos común para las empresas, donde implementar diferentes procesos y casos de uso con los datos por parte de diferentes usuarios, estaba muy limitado.

Por este motivo, en la versión 2 de Hadoop se introdujo YARN. El objetivo de YARN era poder independizar el almacenamiento del procesamiento, abrir Hadoop a cualquier tipo de aplicación que quiera trabajar con los datos de HDFS, y dar la posibilidad de que múltiples usuarios puedan trabajar con la plataforma.



En las primeras versiones de Hadoop, MapReduce era el único motor de computación, por lo que los usuarios/aplicaciones debían desarrollar sus programas siempre utilizando este paradigma.



Con YARN, se divide la capa de computación en dos:

- Por un lado, un gestor de los procesos que se ejecutan en el clúster, que permite coordinar diferentes aplicaciones, asignar recursos y prioridades, permitir su convivencia, etc
- Por otro lado, las aplicaciones, que pueden desarrollarse utilizando un marco de ejecución más ligero, no atado a un modelo estricto sobre cómo ejecutarse, lo que da más libertad para poder desarrollar las aplicaciones.

YARN, por lo tanto, realiza las siguientes tareas:

- Ofrece un API a las aplicaciones mucho menos estricto que MapReduce, ya que no impone la forma en la que deben hacer el procesamiento de datos. Las operaciones del API de YARN son del tipo:
  - Ejecutar una aplicación en el clúster.
  - La aplicación necesita X recursos de memoria y CPU
  - Parar la ejecución de una aplicación.
  - etc
- Se encarga de ejecutar las aplicaciones en el clúster, es decir, ejecuta el código en diferentes nodos, les da los recursos de CPU y memoria necesarios, etc
- Sincroniza la ejecución simultánea de las aplicaciones, decidiendo qué nivel de prioridad tiene cada aplicación, cuántos recursos asignar a cada aplicación cuando compiten por los mismos recursos, etc. Todas estas políticas son configuradas por el administrador de YARN.
- Monitoriza la ejecución de las aplicaciones, y en caso de error en la ejecución de alguna de ellas por un fallo de algún nodo, vuelve a lanzar el trabajo, garantizando la tolerancia a fallos.
- Gestionar los recursos del clúster disponibles, vigilando qué nodos están activos, qué capacidad de memoria y CPU tiene cada nodo, etc

9 Arquitectura.

9.1 Contenedores

En YARN es importante conocer el concepto de **contenedor**, que es la unidad mínima de recursos de ejecución para las aplicaciones, y que representa una cantidad específica de memoria, núcleos de procesamiento (cores) y otros recursos (disco, red), para procesar sus aplicaciones. Por ejemplo, un contenedor puede representar 4 gigabytes de memoria y 1 núcleo de procesamiento.

Todas las tareas de las aplicaciones YARN se ejecutan en contenedores. Cada trabajo puede contener múltiples tareas y cada una de las tareas se ejecuta en su propio contenedor. Cuando una tarea va a arrancar, YARN le asigna un contenedor, y cuando la tarea termina, el contenedor se elimina y sus recursos se asignan a otras tareas.

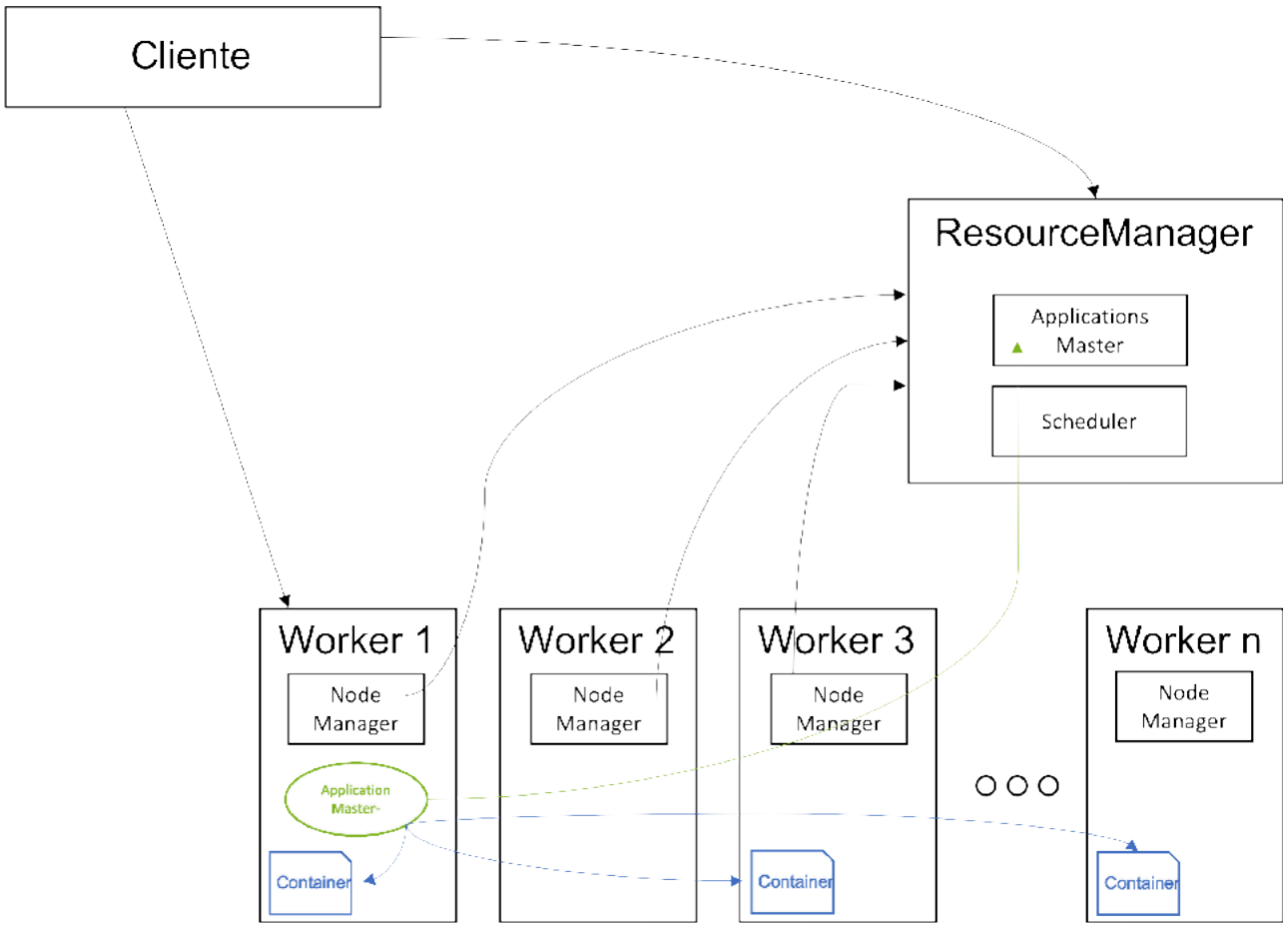
Los contenedores se pueden configurar en cuanto al tamaño de memoria y la cantidad de elementos de procesamiento. Por defecto, el tamaño de memoria de un contenedor es 8 gigabytes, pero dependiendo de los recursos existentes y del tipo de tareas que se ejecutan, se pueden modificar estos valores. Por ejemplo, en clústers con nodos que tienen poca memoria y ejecutan trabajos sencillos, lo habitual es reducir el tamaño de los contenedores, mientras que en clústers con nodos más potentes, y con tareas complejas, los contenedores se suelen ampliar, por ejemplo, a 16 gigabytes.

Asimismo, al iniciar un trabajo, YARN le asigna a cada tarea un conjunto de contenedores dependiendo de la demanda de la aplicación (al lanzar la tarea se le puede indicar el número de contenedores que necesita) y a la disponibilidad de los contenedores que hay en el clúster en ese momento (si hay menos contenedores disponibles de los solicitados, YARN se encargará de aplicar las reglas de prioridad para asignar contenedores que a lo mejor están siendo usados por otras aplicaciones).

La cantidad de tareas y, por lo tanto, la cantidad de aplicaciones de YARN que puede ejecutar en cualquier momento, está limitada por la cantidad de contenedores que tiene un clúster. Por ejemplo, en un clúster de 20 nodos, con 256 gigabytes de RAM y 12 cores por nodo, si se le ha asignado a YARN toda la capacidad existente, habrá un total de 5 terabytes de RAM y 240 cores disponibles. Si se ha definido un tamaño de contenedor de 32 gigabytes, habrá un máximo de 160 contenedores disponibles, es decir, se podrán ejecutar como máximo 160 tareas de forma concurrente.

9.2 Tipos de nodo y servicios en YARN

YARN usa un concepto similar a HDFS en cuanto a la arquitectura, disponiendo de un servicio que actúa de maestro, gestionando la ejecución de las aplicaciones, y nodos worker, que son los que realmente ejecutan las tareas:



Existe un nodo maestro, el ResourceManager, que coordina, asigna y controla la ejecución de todas las tareas, y nodos worker que disponen de un servicio NodeManager, que monitoriza el estado de ejecución de las tareas en el worker, así como el estado de los recursos/contenedores en dicho nodo.

9.3 ResourceManager

Este servicio sería el equivalente al Namenode en HDFS, ya que es el maestro que controla la ejecución de todas las tareas que están en ejecución, o las solicitudes de ejecución existentes.

Cuando un cliente quiere ejecutar una aplicación en YARN, se comunica con el ResourceManager, que será el encargado de asignarle los recursos en base a las políticas de prioridad asignadas y los recursos disponibles, distribuir la aplicación (el ejecutable) por los diferentes nodos worker que realizarán la ejecución, controlar la ejecución para detectar si ha habido una caída de una de las tareas, para relanzarla en otro nodo, y liberar los recursos una vez la ejecución haya finalizado.

El ResourceManager tiene dos componentes principales:

- El **ApplicationsMaster**, que es el servicio que recibe las peticiones de ejecución por parte de los clientes, distribuye las aplicaciones por los nodso worker, asigna los recursos, coordina la ejecución de las tareas, monitoriza la ejecución, solventa los fallos en las ejecuciones, y libera los recursos una vez las tareas han finalizado.

- El **Scheduler**, que es el servicio que asigna prioridades y establece los recursos/containers que disfrutará cada aplicación. Se puede configurar diferentes algoritmos en el Scheduler para definir cómo asignar los recursos a las aplicaciones, siendo los principales:
  - **Capacity Scheduler**: permite definir colas de ejecución, asignando a cada cola un conjunto de recursos (un % de los recursos totales, por ejemplo). Cuando las aplicaciones son lanzadas en el clúster, se le asigna una cola y la aplicación podrá utilizar los recursos disponibles en la misma. Este algoritmo suele ser el más habitual, normalmente haciendo una cola por cada tipo de aplicación, por ejemplo, una cola para los servicios críticos, con un nivel de prioridad mayor y con capacidad para tomar todos los recursos del clúster, una cola para los trabajos de los data scientist, con un nivel de prioridad menor y un límite de recursos bajo, una cola para aplicaciones por lotes, con un nivel de prioridad menor pero con un límite de recursos a consumir, etc
  - **Fair Scheduler**: es un método para asignar recursos a las aplicaciones de modo que todas las aplicaciones obtengan, en promedio, una parte igual de recursos a lo largo del tiempo.
  - **FIFO Scheduler**: con este algoritmo, la prioridad de las aplicaciones está determinada por cuándo fue lanzada, de forma que la primera es la que toma los recursos necesarios, teniendo que esperar el resto de aplicaciones que han sido lanzadas con posterioridad.

9.4 NodeManager

El servicio NodeManager se ejecuta en cada nodo worker y realiza las siguientes funciones:

- Monitoriza y proporciona información sobre el consumo de recursos (CPU/memoria) por parte de los contenedores al ResourceManager.
- Envía mensajes para notificar al ResourceManager su actividad (no está caído) así como la información sobre su estado a nivel de recursos.
- Supervisa el ciclo de vida de los contenedores de aplicaciones.
- Supervisa la ejecución de las distintas tareas en contenedores y termina aquellas tareas que se han quedado bloqueadas.
- Almacena un log (fichero en HDFS) con todas las operaciones que se realizan en el nodo.
- Lanza procesos ApplicationMaster, que coordinan los trabajos para cada aplicación.

Los NodeManager, al igual que los Datanodes en HDFS, son tolerantes a fallos, por lo que en caso de caída de alguno de ellos, el ResourceManager detectará que no funciona y redirigirá la ejecución de las aplicaciones al resto de nodos activos.

9.5 ApplicationMaster

Existe un proceso ApplicationMaster por aplicación. Este proceso se encarga de negociar con el ResourceManager los recursos necesarios para la ejecución de las tareas de su aplicación.

El ApplicationMaster se ejecuta en uno de los nodos worker, para garantizar la

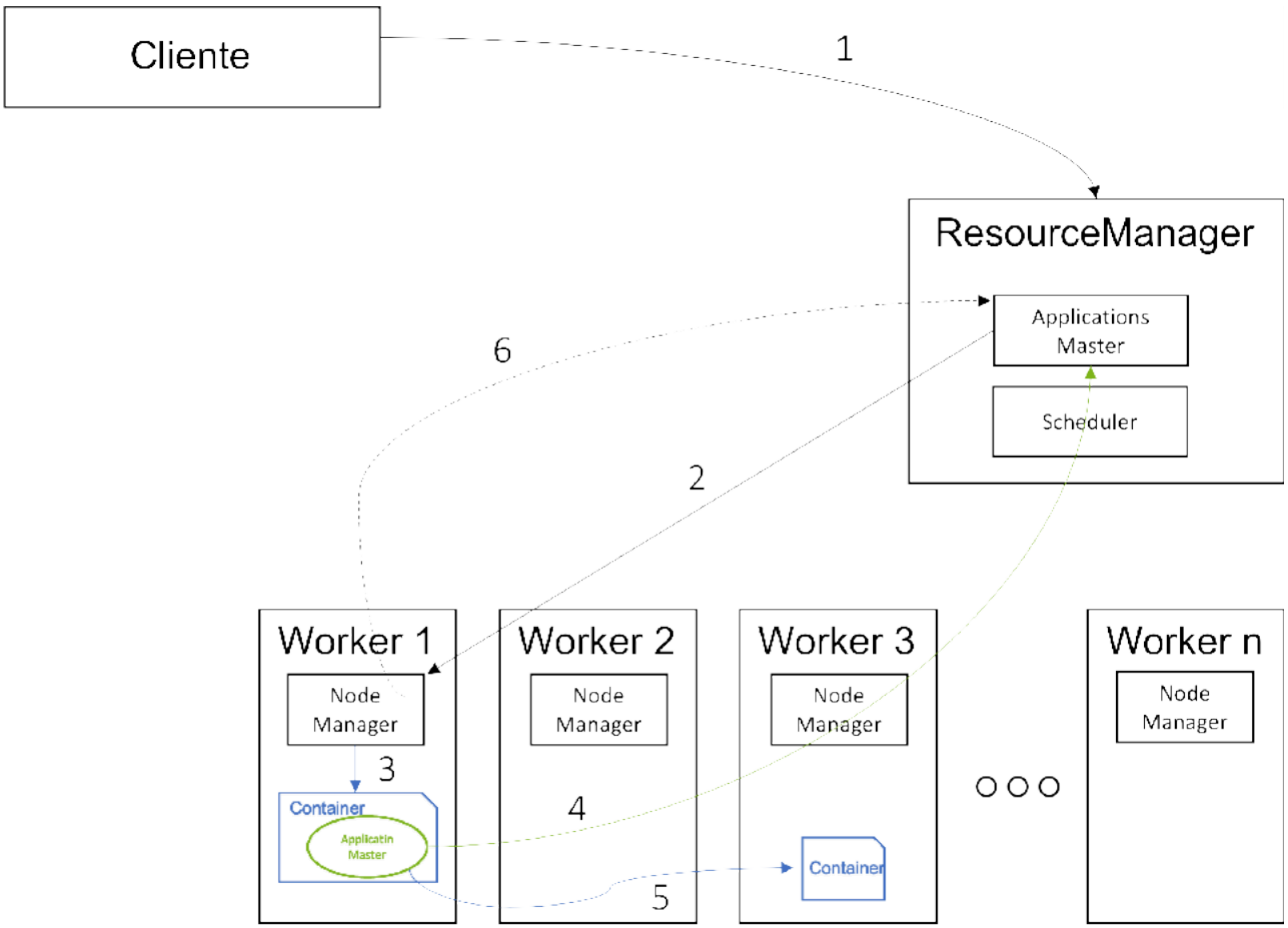
escalabilidad de YARN, ya que si se ejecutaran todos los ApplicationMaster en el nodo maestro, junto con el ResourceManager, éste sería un cuello de botella para poder escalar o poder lanzar un gran número de aplicaciones sobre el clúster.

Asimismo, a diferencia del ResourceManager y los NodeManager, el ApplicationMaster es específico para una aplicación por lo que, cuando la aplicación finaliza, el proceso ApplicationMaster termina. En el caso de los servicios ResourceManager y NodeManager, siempre se están ejecutando aunque no haya aplicaciones activas en el clúster. Cada vez que se inicia una nueva aplicación, ResourceManager asigna un contenedor que ejecuta ApplicationMaster en uno de los nodos del clúster.

10 Funcionamiento.

YARN, en concreto, el ResourceManager, es invocado por los clientes cuando quieren lanzar una aplicación en el clúster para su ejecución.

La secuencia de ejecución de una aplicación es la siguiente:



Los pasos son los siguientes:

El cliente se comunica con el ResourceManager para solicitarle la ejecución de una aplicación. En la llamada, le envía el código/ejecutable de la aplicación, así como unos parámetros sobre los recursos necesarios para dicha ejecución.

2 El ApplicationsMaster, tras chequear con el Scheduler la disponibilidad de recursos y su prioridad, pide al NodeManager de un nodo la creación de un container que ejecutará el ApplicationMaster de la aplicación.

3 El NodeManager crea el conteneder y arranca su ejecución.

4 El ApplicationMaster se comunica con el ApplicationsMaster para solicitarle los contenedores necesarios para la ejecución de la aplicación en caso necesario.

5 El ApplicationMaster se comunica con los contenedores donde se está ejecutando distintas tareas para controlar su ejecución, y va notificando el status de la ejecución al ResourceManager.

6 El NodeManager, asimismo, envía información al ResourceManager sobre el consumo de recursos y notificando que el nodo está activo.

11 MapReduce.

Si recuerdas el tema anterior, MapReduce es un modelo de programación y un marco de ejecución para resolver problemas de procesamiento de datos masivos que se inspiró en un paper publicado por Google donde mostraba cómo habían resuelto el procesamiento de todos los datos obtenidos de sus arañas de recogida de la información de las páginas web para construir índices de búsqueda.

Doug Cutting y Mike Cafarella tomaron ese paper e implementaron el mismo modelo dentro de Apache Hadoop.

En este tema vamos a entrar a conocer en detalle cómo funciona MapReduce, qué aporta y qué dificultades tiene.

12 Introducción.

Hadoop MapReduce es un **framework** para escribir fácilmente aplicaciones que procesan **grandes cantidades de datos** en **paralelo** en grandes **clústeres** (miles de nodos) de **hardware commodity** de manera **confiable** y **tolerante a fallos**

Veamos qué significa cada término de la definición:

12.1 Framework

En MapReduce, los desarrolladores escriben trabajos que consisten principalmente en una función map y una función reduce, y el framework maneja los detalles complejos de paralelizar el trabajo, monitorizar la ejecución o recuperarse ante errores. Debes entender que este tipo de operaciones (monitorización, control de errores, gestión de la concurrencia, etc.) son las más complejas en cualquier sistema de procesamiento masivo de datos, y habitualmente supone el 90% de todo el esfuerzo realizado en el desarrollo de un proceso masivo.

De esta manera, los desarrolladores están protegidos de tener que implementar código complejo y repetitivo y, en su lugar, sólo deben centrarse en desarrollar los algoritmos y la lógica de negocio.

El framework invoca el código proporcionado por el usuario y no al revés. En este sentido, este paradigma es muy parecido a la mayoría de frameworks de desarrollo web, donde el desarrollador debe desarrollar sólo la lógica de negocio que hay detrás de cada interacción del usuario, y no debe preocuparse en los detalles sobre manejar el protocolo HTTP, la concurrencia de las aplicaciones, la gestión de las sesiones de usuario, etc

12.2 Grandes cantidades de datos

MapReduce está diseñado para poder procesar grandes cantidades de datos, ya que sigue una filosofía **Divide y Vencerás (DYV)**, que consiste en que para resolver un problema complejo, la mejor forma de hacerlo es dividirlo en fragmentos muy pequeños que pueden ser solucionados de forma independiente, resolverlo por separado e ir construyendo con las soluciones parciales la solución final.

Para el caso del procesamiento de datos de mucho volumen, la aproximación **Divide y Vencerás** que hace MapReduce consiste en dividir todo el conjunto de datos de entrada en pequeños fragmentos, procesarlos por separado, e ir agrupando los resultados parciales.

12.3 Paralelo

MapReduce ejecuta el procesamiento de cada elemento por separado y el paralelo, es decir, la ejecución se divide en partes pequeñas y cada parte pequeña se ejecuta en paralelo, lo que facilita la escalabilidad o la tolerancia a fallos.

12.4 Clústeres

MapReduce se ejecuta en paralelo en un modelo de computación distribuida, es decir, cada pieza de ejecución se ejecuta en una máquina diferente, siendo cada máquina un servidor de un clúster Hadoop.

YARN se ocupa de los detalles de la ejecución en cuanto a asignación de recursos, nodos disponibles, etc

Asimismo, MapReduce puede ejecutarse en clústers de más de mil nodos, o incluso más, ya que el paradigma no tiene limitaciones en cuanto al número de servidores que pueden ejecutar un trabajo.

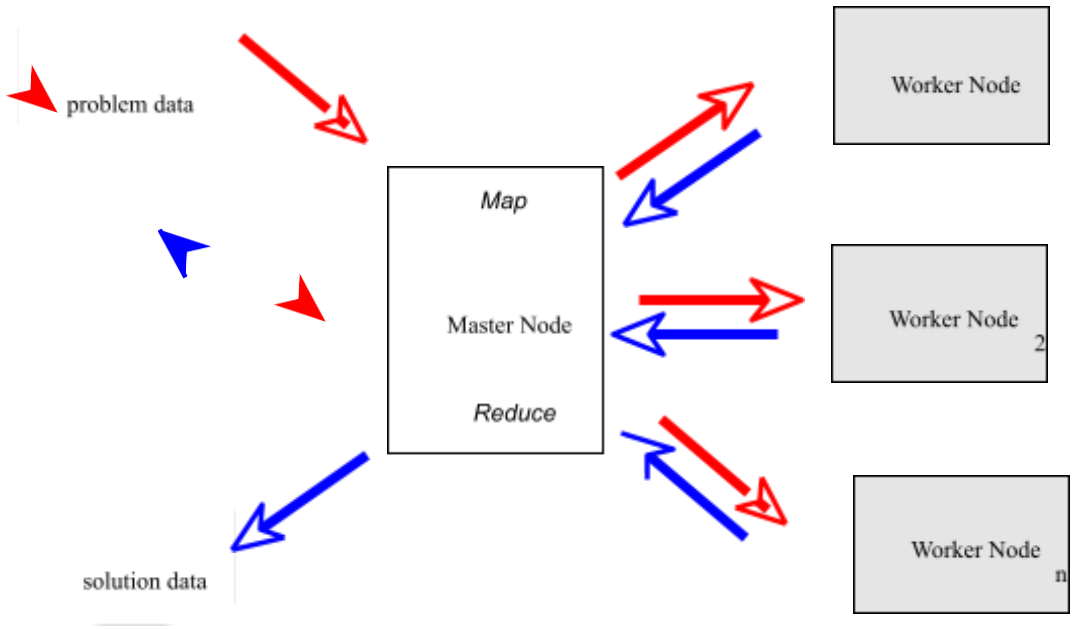
12.5 Hardware commodity

MapReduce no requiere unos servidores específicos para su ejecución. De hecho, ¡emplea los mismos servidores de la plataforma Hadoop!

12.6 Confiable y tolerante a fallos

Una de las principales ventajas de MapReduce frente a otros modelos de programación más eficientes, es que es confiable, es decir, la ejecución de trabajos siempre obtiene los mismos resultados, y además, tiene una capacidad para sobreponerse a fallos muy buena. Durante la ejecución de un trabajo, en caso de que uno de los nodos falle, MapReduce puede recuperar la tarea que dicho nodo estaba ejecutando y reprocesarla con otro nodo activo.

En ocasiones se hace el símil de MapReduce con una apisonadora, en el sentido de que es lenta pero segura, y en contraposición con otros frameworks que aunque tienen una velocidad superior, al utilizar elementos como la memoria, que puede ser volátil, u otro tipo de aceleradores, hace que tengan una tasa de fallos superior a MapReduce, y por ello, habitualmente, los procesos pesados que se ejecutan en ventana nocturna, que no requieren un tiempo de procesamiento corto, se confían a MapReduce ya que tiene mayor garantía de finalización correcta.



Poposhka (Dominio público)

Para implementar trabajos de MapReduce, es necesario desarrollar programas utilizando los APIs que ofrece, y básicamente consiste en desarrollar un método **map** y un método **reduce**

Aunque MapReduce está implementado en Java, las aplicaciones de MapReduce no necesitan estar escritas en Java, ya que gracias a Hadoop Streaming o Hadoop Pipes, se puede desarrollar programas MapReduce en lenguajes como C++, Python o shell scripting.

13 Funcionamiento.

Un trabajo de MapReduce se compone de cinco etapas distintas, ejecutadas en orden:

1. Envío del trabajo, aceptación y distribución en el clúster.
2. Ejecución de la fase **map**
3. Ejecución de la fase **shuffle**
4. Ejecución de la fase **order**
5. Ejecución de la fase **reduce**

De todas estas fases debes saber que el programador sólo suele programar la fase **map** y **reduce**, siendo el resto de fases ejecutadas de forma automática por MapReduce en base a los parámetros de configuración.

Para explicar el funcionamiento de MapReduce, se va a utilizar el siguiente ejemplo:

Imagina que tenemos un fichero de muchos terabytes de datos con todas las cotizaciones de todas las empresas de todas las bolsas del mundo desde hace 30 años, con una cotización cada minuto. Cada línea del fichero tiene el siguiente formato:

Fecha y hora (día/mes/año hora:minutos:segundos);nombre de la empresa;valor de cotización actual;valor de cotización anterior

Por ejemplo, algunas de las líneas del fichero podrían ser las siguientes:

20/01/2021 11:54:34;SANTANDER;4,54;4,49

14/05/1995 09:54;TELEFONICA;11,90;12,01

01/01/1997 08:03:21;SANTANDER;11,24;11,49

19/06/2022 11:54:22;APPLE;111,25;114,89

23/04/2003 16:32:11;ALPHABET;34,49;36,44;

21/12/2020 10:10:56;TELEFONICA;14,31;14;29

26/02/1995 14:09:40;MICROSOFT;132,29;133,95

04/05/1999 11:05:34;WALLMART;34,98;35,05

Tomando una media de 35.000 empresas cotizadas en el mundo, es decir, 35.000 cotizaciones por minuto serían 25.200.000 cotizaciones al día, y un total de 275.940.000.000 líneas en el fichero, que son unos 25 terabytes de datos en un único fichero.

Nuestro objetivo es averiguar cuántas veces ha tenido cada empresa un incremento en su cotización, es decir, si una cotización es superior a su valor anterior, sumaremos uno, y si la cotización es inferior, no lo sumaremos. Es decir, el resultado sería una lista de la siguiente forma:

SANTANDER 3888981

Intentar este cálculo leyendo el fichero de forma secuencial y teniendo un contador para cada empresa sería un proceso que llevaría días de procesamiento, así que vamos a utilizar MapReduce para realizar este proceso.

Como se describió anteriormente, el primer paso es crear la aplicación, por ejemplo, utilizando lenguaje Java, y enviar el programa al clúster Hadoop utilizando el API de MapReduce para enviar trabajos.

El ResourceManager de YARN tomará el trabajo y en función de la situación del clúster en cuanto al número de contenedores disponibles, arrancará un ApplicationsMaster que lanzará la aplicación MapReduce.

Una vez arrancada la aplicación, en primer lugar decidirá cómo partir el fichero de entrada en fragmentos para que los datos puedan ser procesados en paralelo. El componente que realiza esta división de los ficheros de entrada se denomina **InputFormat**

Por cada fragmento del fichero de entrada, se crea una tarea map que ejecutará la función **map** desarrollada en diferentes nodos y en paralelo, es decir, cada fragmento será procesado en paralelo por diferentes nodos.

La función **map** toma cada línea, que es separada por el **InputFormat**, la lee, y emite un resultado parcial, que seá **[Nombre de la empresa, 1]**, en los casos en los que vea que el valor actual es mayor que el valor anterior. Esta función se ejecutará tantas veces como líneas tenga el fragmento de fichero asignado, y en tantos nodos como fragmentos se haya dividido el fichero.
Es decir, para cada nodo, tendremos, por ejemplo, este resultado:

- **SANTANDER, 1**
- **WALLMART, 1**
- **TELEFONICA, 1**
- **APPLE, 1**
- **ALPHABET, 1**

Y tendremos tantos resultados de estos como fragmentos del fichero haya, y en tantos nodos/servidores como se haya ejecutado la función.

A continuación se ejecutan las fases de **shuffle** y **sort** de forma automática y transparente para el desarrollador, donde se toman los resultados parciales, se ordenan por una clave, que en este caso será el nombre de la empresa, se combinan y se ordenan, juntando todos los valores de cada empresa, es decir, teniendo la lista de valores con el siguiente formato:

- **SANTANDER, 1**
- **SANTANDER, 2**
- **SANTANDER, 1**

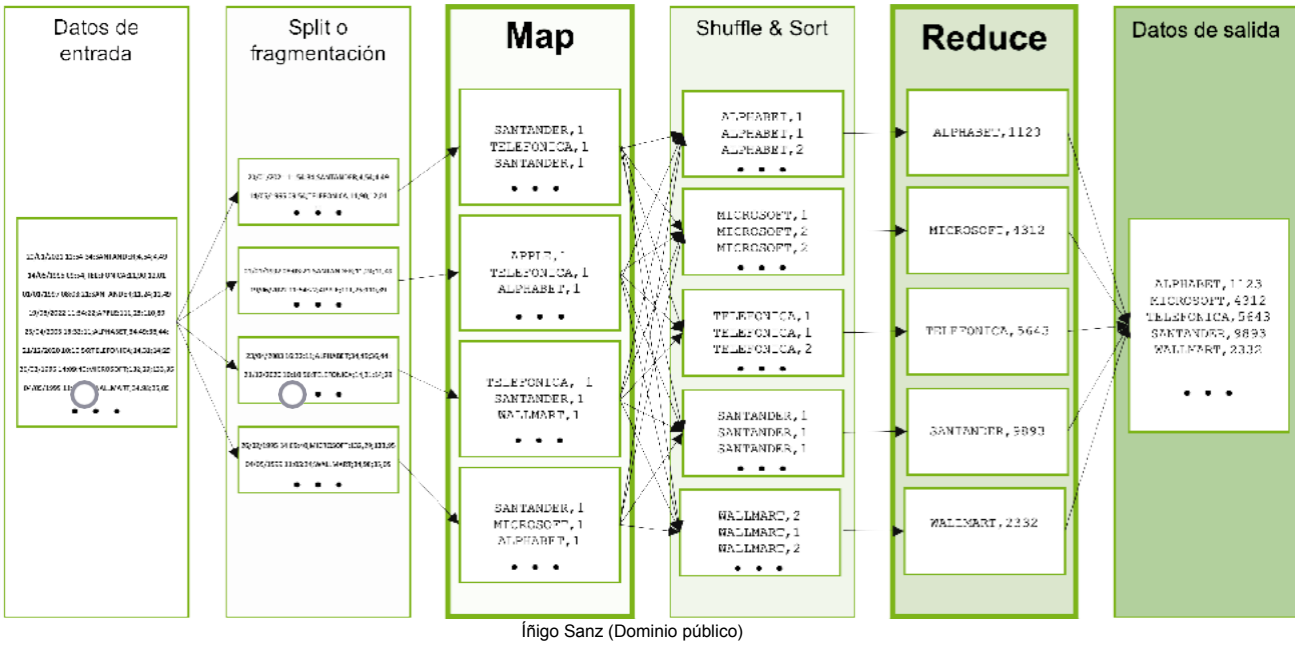
- **TELEFONICA, 1**
- **TELEFONICA, 1**
- **APPLE, 1**
- **ALPHABET, 1**
- **ALPHABET, 1**
- **ALPHABET, 1**

Por último, se divide la lista ordenada en diferentes particiones, siendo cada partición un conjunto de datos con la misma clave, y se llaman a la función **reduce** desarrollada por el usuario, que tomará los diferentes valores emitidos por la fase **map**, pero ya ordenados y unidos, e irá haciendo la suma de cada empresa, dando como resultado pares **[Nombre de la empresa, número de veces que se ha encontrado una cotización incrementada]**

MapReduce, por último tomará todos los resultados de las funciones **reduce** y las unirá, formando el resultado final, que será la lista total de empresas con el número de veces en las que la cotización sube.

El ejemplo puede parecer sencillo, pero permite entender cómo funciona MapReduce para dividir un procesamiento en diferentes bloques de ejecución que se ejecutan en paralelo.

En la siguiente imagen puede verse de forma gráfica el ejemplo anterior:



Íñigo Sanz (Dominio público)

14 Uso

Ahora que conoces cómo funciona MapReduce, vamos a ver cómo se utiliza y cómo se programa en la realidad.

Como hemos comentado, MapReduce se suele desarrollar con Java utilizando las librerías o el API que MapReduce ofrece.

Para entender bien cómo se desarrolla, vamos a utilizar el ejemplo del apartado anterior, en el que teníamos que calcular, para todas las cotizaciones de todas las empresas (cotizadas) del mundo, cuántas veces han experimentado una subida en la cotización. Recuerda que los datos de entrada eran un fichero tenían el siguiente formato:

Fecha y hora (día/mes/año hora:minutos:segundos);nombre de la empresa;valor de cotización actual;valor de cotización anterior

Por ejemplo, algunas de las líneas del fichero podrían ser las siguientes:

20/01/2021 11:54:34;SANTANDER;4,54;4,49

14/05/1995 09:54;TELEFONICA;11,90;12,01

01/01/1997 08:03:21;SANTANDER;11,24;11,49

19/06/2022 11:54:22;APPLE;111,25;114,89

23/04/2003 16:32:11;ALPHABET;34,49;36,44;

21/12/2020 10:10:56;TELEFONICA;14,31;14;29

26/02/1995 14:09:40;MICROSOFT;132,29;133,95

04/05/1999 11:05:34;WALLMART;34,98;35,05

El fichero de entrada tenía 275.940.000.000 líneas, que son unos 25 terabytes de datos en un único fichero.

Nuestro objetivo era averiguar cuántas veces ha tenido cada empresa un incremento en su cotización, es decir, el resultado sería una lista de la siguiente forma:

SANTANDER 3888981

TELEFONICA 3331923

Como se ha indicado, el trabajo consistirá en desarrollar un pequeño programa que haremos con Java. Lo importante es que entiendas el concepto MapReduce, así que nos centraremos en los aspectos importantes y dejaremos los aspectos de formateo de cadenas, conversión de tipos, control de excepciones, etc. de forma simple.

```
import java io IOException
import java util StringTokenizer
import org apache hadoop confConfiguration
import org apache hadoop fs Path
import org apache hadoop io IntWritable
import org apache hadoop io Text
import org apache hadoop mapreduce Job
import org apache hadoop mapreduce Mapper
import org apache hadoop mapreduce Reducer
import org apache hadoop mapreduce lib input FileInputFormat
import org apache hadoop mapreduce lib output FileOutputFormat

public class QuotationAnalyzer {

    public static class IncreaseQuotationFilterMapper extends Mapper Object Te

        // MapReduce invocará a este método una vez por cada línea del fichero
        public void map(Object key Text value Context context) throws IOExcep

            // Cogemos la línea que llega como parámetro, la convertimos a Stri
            // y la dividimos en los distintos bloques de información
            String valueString   value toString()
            String[] dataOfTheQuotation   valueString split(";")

            // Tomamos los valores que nos interesan: nombre de la empresa y co
            float currentQuotationFloat parseFloat(dataOfTheQuotation[2])
            float lastQuotation   Float parseFloat(dataOfTheQuotation[3])
            String companyName   dataOfTheQuotation[1]

            // Realizamos el filtro: si la cotización crece, enviamos una parej
            if (currentQuotation   lastQuotation)
                context write(new Text(SingleCountryData[companyName])   ew Int
            }
        }
    }

    public static class IntSumReducer extends Reducer Text IntWritable Text

        Int private IntWritable result      new IntWritable()

        // MapReduce invocará a este método una vez por cada empresa,
        // pasando como parámetro todos los valores asociados generados en map.
        public void reduce(Text key Iterable values Context context) throws IO
            int sum   0

            // Simplemente sumamos los valores, y la suma será el resultado de
            for (IntWritable val   values) {
                sum += val get()
            }
            result set(sum)
            context write(key result)
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf   new Configuration()
        Job job   Job getInstance(conf "QuotationAnalyzer")
        job setJarByClass(QuotationAnalyzer class)
```

```
            job setMapperClass(IncreaseQuotationFilterMapper class)
            job setCombinerClass(IntSumReducer class)
            job setReducerClass(IntSumReducer class)
            job setOutputKeyClass(Text class)
            job setOutputValueClass(IntWritable class)
            FileInputFormat addInputPath(job   ew Path(args[0]))
            FileOutputFormat setOutputPath(job new Path(args[1]))
            System exit(job waitForCompletion(true) ? 0   1)
        }
    }
```

En el ejemplo podrás ver varios puntos:

- Sólo se ha implementado un Mapper y un Reducer. Todos los aspectos de dividir el fichero de entrada en bloques, tomar los resultados parciales, controlar la ejecución de todos los bloques, etc. queda como responsabilidad de MapReduce, así que se reduce mucho la cantidad de código que hay que generar. Como comentábamos en el punto anterior, lo importante es hacer un buen diseño de cómo resolver el problema dividiéndolo en **map** y **reduce**, y no nos debe importar los detalles de la ejecución.
- El Mapper sólo espera recibir cada línea por separado, una llamada por cada línea. Para cada línea, la lee, extrae la información que necesita, hace las comprobaciones para ver si debe contar esa línea, y emite un resultado **[Nombre de la empresa, 1]** en caso de que la línea represente un incremento en la cotización.
- El Reducer, que es invocado una vez por cada empresa resultante en la fase **map**, al que se le pasa la empresa y la lista de resultados parciales generados en **map** para esa empresa, sólo debe realizar la suma y emitir un resultado **[Nombre de la empresa, suma de los valores recibidos en map]**
- Por último, hay una función **main** que es la que genera la tarea, configurando qué clase hará de Mapper y qué clase de Reducer.

Por último, una vez que ya has desarrollado el código, el siguiente paso es empaquetarlo en un fichero .jar (en el caso de Java), por ejemplo, con nombre quotation-analyzer.jar.

Para lanzarlo en el clúster, simplemente ejecutaremos el siguiente comando en el nodo frontera:

$ bin/hadoop jar quotation-analyzer.jar QuotationAnalyzer /data/markets/quotations/input/allquotations.csv /data/markets /quotations/output

Lo parámetros son los siguientes:

- En primer lugar, se lanza con un comando **hadoop jar**, que es el comando de Hadoop para lanzar aplicaciones empaquetadas como un fichero .jar de Java.
- En segundo lugar, se indica el nombre de la clase que contiene la configuración (el método **main**).
- Como tercer parámetro, se indica la ruta del fichero de entrada.
- Como cuarto parámetro, se indica la ruta de salida.