Procesamiento Distribuido para Big Data

Tema 2: Hadoop Distributed File System

Autor: Dr. Pablo J. Villacorta Iglesias

Actualizado Marzo 2020

INTRODUCCIÓN

El artículo publicado por Google en 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, que se ha seguido utilizando sin apenas cambios desde que se introdujo por primera vez. En este tema estudiaremos:

- 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.

Bibliografía recomendada:

Tom White: *Hadoop, The Definitive Guide*. O'Reilly, 2015. Capítulo 3.

Podemos definir HDFS como:

Sistema de archivos distribuido para almacenar archivos muy grandes con patrones de acceso en streaming, pensado para clusters de ordenadores convencionales

---

Desgranando esta definición, tenemos que:

1. Sistema de archivos distribuido: los archivos se almacenan en una red de máquinas. Esto 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 posee en casa, pero con la restricción de que siendo una sola máquina, el fallo o la falta de capacidad obliga a reemplazar el mainframe por otro más grande.

- Es escalable: si necesitamos 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, que veremos en la siguiente sección.

2. Pensado para almacenar archivos muy grandes: permite archivos mayores que la capacidad del disco de una máquina individua. Es decir, un solo archivo puede ocupar cientos de GB, varios TB, 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 acceden para lectura en numerosas ocasiones. No son archivos cuyo contenido necesitemos reemplazar con frecuencia ni ser borrados habitualmente. Además, no es importante el tiempo de acceso a partes individuales del archivo sino que se suele utilizar el archivo completo en las aplicaciones (modo batch, no interactivo), a diferencia de una base de datos. Por este motivo:

- No soporta modificación de archivos existentes. Sólo lectura, escritura y borrado
- No funciona bien para
    - Aplicaciones que requieran baja latencia para acceder a registros individuales
    - 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 cluster. 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*). Esto implica que HDFS utiliza la API convencional del sistema operativo para leer y escribir datos en nodos, y no interactúa con los dispositivos físicos (discos duros). 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 cluster 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), y también al sistema de archivos HDFS, que existe “en paralelo” a cada uno de los sistemas de archivos locales. Cuando creamos datos en HDFS, lo más habitual es copiarlos en HDFS desde el sistema de archivos de una máquina concreta, aunque también es habitual que otra aplicación, por ejemplo que se ejecuta en streaming y lee datos de otra fuente, esté escribiendo salidas directamente en el sistema HDFS. La mayoría de 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 manejar Linux, pero 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) desencadena unas acciones muy diferentes a la ejecución del comando que se llama igual pero en HDFS.

ARQUITECTURA DE HDFS

Bloques en 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 ó 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 1KB, 2KB, etc). Habitualmente es de 4 KB, y siempre debe contener un número de sectores físicos potencia de 2.

En sistemas de archivos convencionales (ext4, NTFS, FAT32…) 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, configurable (por defecto 128 MB), pero los archivos de menos de un bloque 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. Si los bloques son muy pequeños, se generan demasiados metadatos. En cambio, si son muy grandes, limitan el paralelismo de frameworks que operan a nivel de bloque, esto es, en los que varias máquinas pueden procesar simultáneamente bloques diferentes de un mismo archivo.

- Los bloques de datos pueden ser replicados para 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 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 de 500 MB que al subirlo a HDFS, se ha particionado automáticamente en cuatro bloques de los que tres tienen 128 MB 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, *fsck*, nos permite consultar los metadatos asociados a este fichero, los cuales describen cómo está almacenado físicamente. En la siguiente sección veremos qué tipo de metadatos se almacenan.

```
[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
```

Datanodes y namenode

Cuando se instala HDFS en un cluster, cada nodo puede utilizarse como datanode o como namenode. El namenode (debe haber al menos uno) mantiene la estructura de directorios y los metadatos. Esta información se guarda de manera persistente como (i) imagen del namespace, y (ii) log de edición que contiene los cambios que ha ido haciendo el usuario. Por su parte, los datanodes almacenan bloques de datos. 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 3 s, aunque es configurable en la propiedad *dfs.heartbeat.interval*) y un listado de todos los bloques presentes en cada datanode (por defecto cada 6 h, configurable en dfs.blockreport.*).

El namenode es punto único de fallo (SPOF). Sin él, no es posible utilizar HDFS. Por ese motivo, se han ideado diferentes mecanismos tanto de respaldo del namenode (copia de seguridad preventiva frente a fallos) como de alta disponibilidad.

Mecanismos de respaldo del namenode:

- Copia de los archivos persistentes de metadatos a otros nodos o a NFS
- Namenode secundario (no es realmente un namenode): en otra máquina física, va fusionando los cambios del log de edición a la imagen del namespace. Suele ir con retraso respecto al original. En caso de fallo, se transfieren a él si es posible metadatos que estén en NFS, y se empieza a usar como namenode activo. Es un proceso manual que puede tardar hasta 30 min o más, por lo que no se puede considerar alta disponibilidad ya que el sistema estaría 30 minutos inutilizado.

Mecanismos de alta disponibilidad de HDFS: se puede utilizar un par de namenodes (uno de ellos activo y el otro en modo stand-by). Existe un log de edición compartido por los dos namenodes, de manera que los datanodes tienen que reportar su estado a ambos todo el tiempo. Requiere re-implementar los clientes de HDFS. Cuando el

namenode activo falla, el namenode que se encontraba en espera automáticamente toma el control y pasa a ser el namenode activo, por lo que el tiempo de pérdida de servicio es muy pequeño, menos de 5 minutos.

Existe un mecanismo adicional que permite escalar el namenode en caso de que sea necesaria más memoria debido a la cantidad de metadatos generados cuando existen muchos ficheros. Se denomina ***federación de namenodes*** y consiste en tener varios namenodes que se encargan de directorios (*namespaces*) distintos del sistema de archivos (sin solapamiento, por ejemplo un namenode contiene todos los metadatos bajo el subárbol o /user mientras que otro contiene los del subárbol /share). El fallo de un namenode no afecta al resto sino sólo a los ficheros que cuelgan de su árbol. En esta configuración, los datanodes pueden almacenar bloques de archivos de cualquiera de los namespaces.

Rack-awareness

Cuando almacenamos un fichero en HDFS, el software se encarga de dividir el fichero en bloques, replicarlos y distribuir físicamente cada bloque para que se almacene en el nodo *más conveniente*. La decisión de cuál es el nodo idóneo para almacenar cada bloque de datos no es aleatoria sino que sigue políticas preestablecidas y que también son configurables. Existe una funcionalidad adicional que permite optimizar esta decisión en función de cómo está físicamente distribuido nuestro cluster en *racks* o armarios de nodos. Los nodos almacenados en un rack comparten cierta infraestructura relativa a corriente y conectividad de red, de manera que un nodo accede más rápidamente a otro que esté en el mismo rack que a cualquier otro nodo del cluster. La desventaja es que es posible que un fallo en la alimentación eléctrica afecte a todo un rack completo y no solo a un nodo, por lo que pueden llegar a caer todos los nodos de un rack.

Esta funcionalidad se denomina *rack-awareness* y permite describir la topología de nuestro cluster en un fichero de configuración. HDFS utiliza esta información cuando decide a qué nodo debe ir cada bloque de un nuevo archivo creado en HDFS. La política habitual es que *no debe haber más de una réplica de un bloque en el mismo nodo, y no más de dos réplicas de un bloque en nodos del mismo rack.* El primer caso es obvio: no tiene sentido almacenar varias veces un mismo bloque de datos en un nodo, ya que si el nodo falla, se perderá igualmente ese bloque de datos. Para preservarlo, lo adecuado es distribuir réplicas de ese bloque de datos en nodos distintos, para que si uno de ellos falla, siga existiendo ese bloque en otros nodos. El segundo caso previene frente a fallos de un rack completo.

La siguiente figura muestra una posible distribución de 4 archivos independientes en un cluster de HDFS con 8 datanodes. Cada archivo se ha generado como resultado del procesamiento en paralelo de un mismo fichero de entrada mediante algún framework distribuido como por ejemplo Apache Spark. El fichero *predic* es en realidad un único dataset de resultados que, antes de escribirse en HDFS, se dividió en 4 ficheros independientes, cada uno de ellos de 40 MB (es decir, tamaño inferior a un bloque).

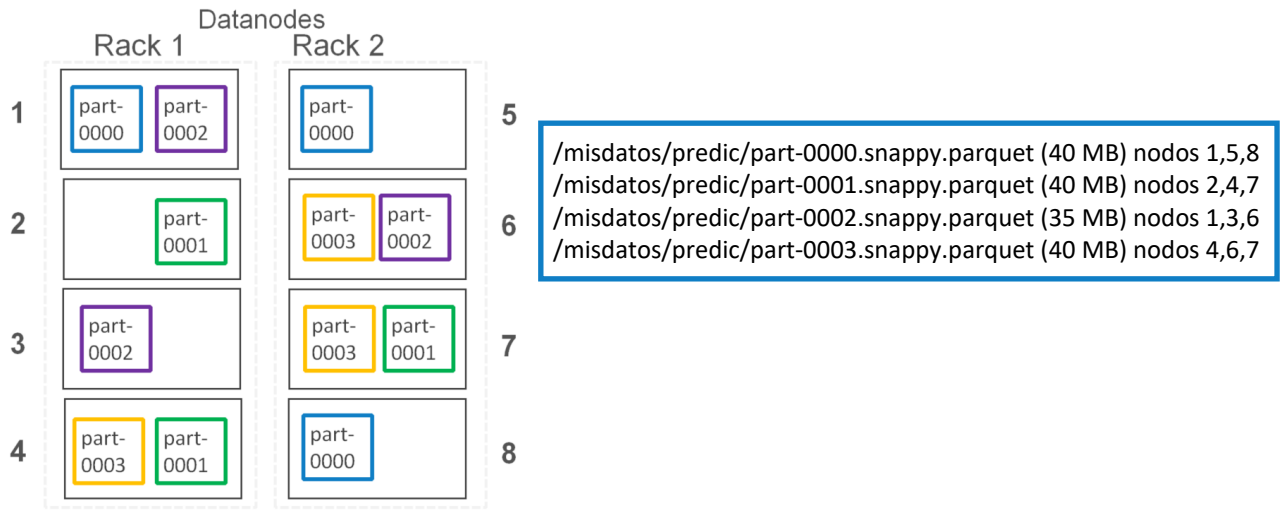

Fig. 1. Una posible distribución física de cuatro archivos de un bloque en HDFS.

Configuración de HDFS

Los ficheros de configuración de Hadoop se encuentran habitualmente en /etc/hadoop y permiten fijar los valores de numerosas propiedades de las herramientas de Hadoop mediante ficheros XML. Los dos que afectan a HDFS son *core-site.xml* y *hdfs-site.xml*.

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Configuration settings for MapReduce daemons: the jobtracker, and the tasktrackers.
masters	Plain text	A list of machines (one per line) that each run a secondary namenode.
slaves	Plain text	A list of machines (one per line) that each run a datanode and a tasktracker.
hadoop-metrics.properties	Java Properties	Properties for controlling how metrics are published in Hadoop
log4j.properties	Java Properties	Properties for system logfiles, the namenode audit log, and the task log for the tasktracker child process

Tabla 1. Ficheros de configuración de Hadoop (*Hadoop: The Definitive Guide, 2015*)

Entre las múltiples propiedades que se pueden configurar están las siguientes:

File Name	Parameter Name	Parameter value	Description
core-site.xml	fs.defaultFS/fs.default.name	hdfs://<namenode_ip>:8020	Namenode ip address or nameservice (HA config)
hdfs-site.xml	dfs.block.size, dfs.blocksize	128 MB	Block size at which files will be stored physically.
hdfs-site.xml	dfs.replication	3	Number of copies per block of a file for fault tolerance
hdfs-site.xml	dfs.namenode.http-address	0.0.0.0:50070	Namenode Web UI. By default it might use ip address of namenode.
hdfs-site.xml	dfs.datanode.http.address	0.0.0.0:50075	Datanode Web UI
hdfs-site.xml	dfs.name.dir, dfs.namenode.name.dir	<directory_location>	Directory location for FS Image and edit logs on name node
hdfs-site.xml	dfs.data.dir, dfs.datanode.data.dir	<directory_location>	Directory location for storing blocks on data nodes
hdfs-site.xml	fs.checkpoint.dir, dfs.namenode.checkpoint.dir	<directory_location>	Directory location which will be used by secondary namenode for checkpoint.
hdfs-site.xml	fs.checkpoint.period, dfs.namenode.checkpoint.period	1 hour	Checkpoint (merging edit logs with current fs image to create new fs image) interval.
hdfs-site.xml	dfs.namenode.checkpoint.txns	1000000	Checkpoint (merging edit logs with current fs image to create new fs image) transactions.

Tabla 2. Algunas propiedades configurables en Hadoop y en particular en HDFS. Fuente: https://slideplayer.com/slide/12131503

Un ejemplo de formato de estos ficheros de configuración se muestra a continuación. No difiere en nada de un fichero con estructura XML convencional.

```
  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
   <name>dfs.replication</name>
   <value>1</value>
 </property>
 <property>
   <name>dfs.namenode.name.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/namenode</value>
 </property>
 <property>
   <name>dfs.datanode.data.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/datanode</value>
 </property>
</configuration>
```

Fig. 2. Fragmento del contenido del fichero hdfs-site.xml

COMANDOS MÁS HABITUALES EN HDFS

Todos los comandos deben llevar delante el prefijo *hdfs dfs -<comando>* o bien *hadoop fs –<comando>*. Lo más habitual es la primera opción, aunque los ejemplos que se darán a continuación muestran la segunda.

```
[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> ...]]
```

Un concepto importante es que, a diferencia del sistema de ficheros local de Linux o Windows, en HDFS no existe 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.

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 por 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 por 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
```

- hdfs dfs –cp /ruta/hdfs/origen/fichero.txt /ruta/hdfs/destino/ficheroCopiado.txt : hace una copia del fichero situado en HDFS en /ruta/origen/fichero.txt en la ruta destino de HDFS /ruta/hdfs/destino/ficheroCopiado.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/nombreOriginal.txt /ruta/nombreNuevo.txt o bien /ruta/fichero1/ /ruta/nueva/ : renombrar un fichero existente en HDFS o bien moverlo a otra ubicación de HDFS.

- hdfs dfs –rm /ruta/fichero.txt : borra un fichero presente en HDFS. Es posible borrar una carpeta completa recursivamente (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 : comando similar a chmod del sistema de archivos de Linux para cambiar los permisos en un archivo existente en HDFS.

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

PROGRAMACIÓN DISTRIBUIDA Y MAP-REDUCE

Una vez que Google tenía resuelto el problema del almacenamiento de ficheros masivos en el sistema de ficheros distribuido Google File System (GFS), en 2004 publicaron un nuevo artículo sobre cómo aprovechar los datanodes para procesar estos ficheros que estaban almacenados de forma distribuida particionados en bloques. El artículo describe 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 cluster de ordenadores. Junto con el artículo, también liberaron una biblioteca de programación en lenguaje Java que implementaba este paradigma. 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 al hardware, redes y comunicación entre los nodos, para 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 CPUs de los datanodes del cluster 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) donde están almacenados los datos (es decir, a los datanodes, en lugar de al revés, como ocurría en el modelo tradicional de aplicación). Las CPUs de los datanodes van a procesar (preferentemente) los datos que hay en ese datanode.

En el paradigma MapReduce, el usuario sólo 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 coinciden con un bloque de HDFS), generando así resultados intermedios que están siempre en forma de (clave, valor) y son escritos también al disco local.
- Reducer: función del usuario invocada por el framework en paralelo para cada clave distinta, pasándole todos los valores que comparten esa clave.

Ejemplo: el problema de contar ocurrencias de palabras con MapReduce

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

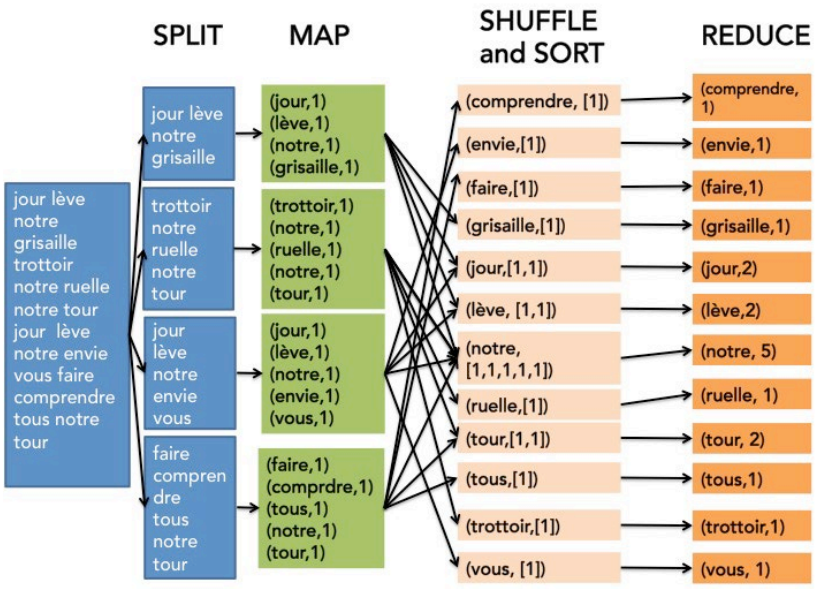

Fig. 4.

Representación del problema de contar ocurrencias resuelto con MapReduce

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 da lugar a un rectángulo verde.

A continuación, las tuplas que genera la fase map son enviadas por la red que conecta los nodos del cluster: la librería de MapReduce, de forma automática y transparente al programador, lleva a cabo la operación shuffle and sort mostrada en rosa, que 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 que reciba cada una de las tuplas mostradas en rosa (formadas por una palabra y la lista de número de ocurrencias asociadas a ella) y genere, 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 quien automáticamente invoca tantas veces como sea necesario a la función que ha implementado el usuario, pasándole los argumentos que hemos indicado. La invocación y ejecución de la función Map se lleva a cabo de manera distribuida, en cada nodo del cluster (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) indicando que palabra ha aparecido una vez. Ej.: («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 pasándole una clave y la lista de valores asociados en las tuplas. 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 (hola, [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 cluster de ordenadores. Permite resolver todo tipo de problemas, pese a que no siempre es sencillo pensar la solución en términos de operaciones map y reduce encadenadas. Por ello 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 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 cluster 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.

BIBLIOGRAFÍA RECOMENDADA

Tom White: *Hadoop, The Definitive Guide*. O'Reilly, 2015. Capítulo 3.

Arquitectura de HDFS: http://www.aosabook.org/en/hdfs.html

APÉNDICE: FORMATO DE ARCHIVO PARQUET

Apache Parquet es un formato de archivo nacido en 2013, de tipo columnar y comprimido (binario). Es el formato más utilizado para almacenar datos estructurados en HDFS (tablas o similar, incluyendo tipos complejos y anidados en las columnas). Es muy frecuente su uso en conjunción con Apache Hive (tema siguiente). Parquet utiliza la técnica conocida como record-shredding para almacenar tipos de datos complejos y anidados (que un CSV no es capaz de representar). Sus principales características son:

- Almacena el esquema junto a los datos. También máx/mín por columna (resúmenes)
- Los valores de cada columna se guardan físicamente juntos (almacenamiento columnar). Beneficios:
  - Compresión de datos por cada columna para ahorrar espacio
  - Se pueden aplicar técnicas de compresión diferentes, específicas según el tipo de dato que almacena cada columna
  - Las consultas con filtrados sobre columnas concretas no necesitan leer toda la fila, mejorando el rendimiento
- Preparado para añadir más esquemas de codificación que se desarrollen en el futuro
- No requiere especificar separador de columnas, ni si se incluyen o no nombres de columna (siempre se incluyen por defecto)

Es con diferencia el formato más ampliamente utilizado para procesamiento en batch, tanto para datos de entrada como para guardar resultados, con datos estructurados.