

MÓDULO

HERRAMIENTAS BIG DATA

UNIDAD 4

TECNOLOGÍAS BIG DATA I: HADOOP

ÍNDICE

TU RETO EN ESTA UNIDAD.....	3
1. INTRODUCCIÓN	4
1.1. ¿QUÉ ES HADOOP?	4
1.2. UN POCO DE HISTORIA	5
1.3. ENTORNO DE TRABAJO	7
1.3.1. UNAS NOTAS SOBRE NUESTRA MÁQUINA VIRTUAL.....	10
2. COMPONENTES DE HADOOP	12
2.1. HDFS	13
2.1.1. NAMENODE	14
2.1.2. DATANODE	15
2.2. YARN	17
2.2.1. RESOURCEMANAGER.....	18
2.2.2. NODEMANAGER.....	19
2.2.3. CONTAINER	20
2.2.4. APPLICATIONMASTER	20
2.2.5. RESUMIENDO, CÓMO FUNCIONA UNA APLICACIÓN EN YARN	22
2.2.6. INTEGRACIÓN HDFS-YARN Y COLOCALIZACIÓN	23
2.3. MAPREDUCE	25
2.3.1. CARGA Y REPARTICIÓN INICIAL	27
2.3.2. MAP	28
2.3.3. REORDENACIÓN Y REPARTO	29
2.3.4. REDUCE	30
2.3.5. RECOPILACIÓN	31
3. PRIMEROS PASOS CON HADOOP	32

4. TRABAJANDO CON HDFS	32
4.1. OPERACIONES HABITUALES	35
4.1.1. LISTAR EL CONTENIDO DE UN DIRECTORIO	36
4.1.2. CREAR DIRECTORIOS	37
4.1.3. COPIAR DATOS A HDFS	38
4.1.4. VER EL CONTENIDO DE UN FICHERO	39
4.1.5. COPIAR FICHEROS DENTRO DE HDFS	40
4.1.6. BORRADO DE FICHEROS	40
4.1.7. OBTENER FICHEROS DE HDFS	41
4.1.8. REFERENCIA	42
5. TRABAJANDO CON MAPREDUCE	44
5.1. HADOOP STREAMING	44
5.1.1. CONTANDO PALABRAS CON HADOOP STREAMING	45
5.1.2. UNA VARIANTE REFINADA	50
5.2. MRJOB	52
5.2.1. CONTANDO PALABRAS CON MRJOB	52
5.2.2. MÁS ALLÁ DE UN SIMPLE MAP-REDUCE	57
¿QUÉ HAS APRENDIDO?	63
AUTOCOMPROBACIÓN	65
SOLUCIONARIO	69
BIBLIOGRAFÍA	71

TU RETO EN ESTA UNIDAD

¿Te has preguntado alguna vez cómo hace Google para devolverte los resultados de tus búsquedas con precisión y de forma casi inmediata?

En esta unidad vamos a presentarte la plataforma Apache Hadoop, una de las tecnologías que revolucionó las ideas y modelos para manejar grandes cantidades de datos, hasta convertirse en una referencia en el procesamiento Big Data. Hadoop bebe directamente de todas las ideas desarrolladas por las mismas personas que participaron en la construcción de la tecnología de búsquedas usada por Google.

Repasaremos los orígenes de esta tecnología y te explicaremos cuáles son sus principales componentes, y cómo funcionan y se integran entre sí, para que entiendas qué hace exactamente Apache Hadoop.

Una vez que conozcas el funcionamiento de la plataforma, pasaremos a ver cómo utilizarla de verdad, con algunos ejemplos sencillos de operaciones y programas.

1. INTRODUCCIÓN

1.1. ¿QUÉ ES HADOOP?

Apache Hadoop es una plataforma software de código abierto ideada para almacenar y procesar grandes volúmenes de datos de forma distribuida. Está especialmente diseñada para trabajar con *clusters* formados por un gran número de servidores básicos, facilitando la escalabilidad de la plataforma y garantizando la tolerancia a fallos en nodos.

Aunque está formado por varios componentes, los dos elementos principales de Hadoop son su sistema de ficheros distribuido (HDFS) y su motor MapReduce.

A modo de resumen, Hadoop trocea los ficheros en grandes bloques que distribuye entre los nodos que forman el *cluster*. Después, cuando es necesario procesar los datos, empaqueta y transfiere el código a ejecutar a cada uno de los nodos donde están los bloques de datos, lanzando los cálculos en paralelo. La idea es aprovechar el principio de localidad de los datos, de forma que el procesamiento sea mucho más rápido y eficiente.



Importante

Recuerda: Cuando hablamos de Big Data, es más eficiente enviar el código a donde están los datos, que transferir los datos a donde está el código.

El modelo de Hadoop es especialmente eficaz para implementar procesos de cálculo y agregación sobre grandes cantidades de datos, donde no importa que el tiempo de respuesta sea elevado. Es una solución perfecta para el tipo de procesamiento por lotes.

1.2. UN POCO DE HISTORIA

Las motivaciones que darían lugar a Hadoop podemos encontrarlas con el cambio de milenio, alrededor del año 2000. En esos momentos, el crecimiento de Internet a nivel de páginas web y contenido empezaba a tomar una velocidad importante. En media se estaban creando más de 25000 nuevos sitios web al día.

Pero para hacer visible todo ese nuevo contenido, la gente tenía que poder encontrarlo de alguna forma. Para eso estaban los buscadores.

El campo de los buscadores era efervescente, con nuevas tecnologías y algoritmos surgiendo para tratar de mejorar la eficiencia y la calidad de los resultados. En 1999 Google patenta su algoritmo PageRank para calcular la relevancia de una página web como resultado de una búsqueda, convirtiéndose en el corazón de su motor de búsqueda y revolucionando la forma de indexar y recuperar información de los buscadores.

El problema es que no era posible indexar el contenido completo de Internet en una sola máquina. Para garantizar que los contenidos y resultados estuvieran siempre actualizados, había que agilizar los tiempos de procesado. Era necesario distribuir el trabajo entre varias máquinas.

Pero no era tan sencillo. ¿Cómo repartir la información? ¿Cómo coordinar los cálculos? ¿Cómo gestionar los recursos de los nodos de forma eficiente sin tener que supervisar manualmente todo el tiempo?

En 2003 Google publica un artículo académico describiendo un nuevo sistema de ficheros distribuido, el Google File System (GFS), diseñado para almacenar ficheros de datos de gran tamaño, troceados y repartidos entre nodos de un *cluster*. Además, la infraestructura de Google se basa en usar un gran número de máquinas económicas en lugar de unos pocos "super-servidores". Y las máquinas económicas tienen más probabilidad de fallar. Así que Google diseña GFS para que sea capaz de detectar y recuperarse de posibles fallos de manera automática, sin perder información.

Poco después, en 2004, Google publica otro artículo describiendo un nuevo modelo de procesamiento de datos simplificado utilizando grandes *clusters* de máquinas: MapReduce.

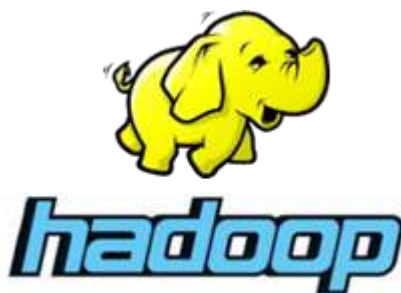
Por otro lado, dos personas, Doug Cutting y Mike Cafarella, habían estado trabajando durante esos años en dos proyectos clave de código abierto: Lucene (un motor de análisis y búsqueda textual) y Nutch (un robot para recorrer e indexar el contenido de Internet). Ambos proyectos fueron movidos a la Apache Software Foundation.

Cutting y Cafarella decidieron implementar las ideas propuestas por Google en sus artículos, e integrarlas en Nutch. Unos meses después, crean un subproyecto incubado dentro de Apache Lucene, al que mueven las primeras versiones de estos nuevos componentes: a principios de 2006 nace Hadoop.



Ojo al dato

Doug Cutting bautizó a este proyecto como Hadoop porque era el nombre de un elefante de juguete amarillo que tenía su hijo pequeño. Ese elefante sigue siendo el logo de Hadoop.



Al mismo tiempo, Yahoo! estaba rehaciendo su buscador y había incorporado a Cutting para que les ayudara en el diseño. Yahoo! adoptó las tecnologías de Hadoop, y contribuyó a su desarrollo y evolución. En 2012, Yahoo! gestionaba un *cluster* de Hadoop formado por 42000 nodos y cientos de Petabytes de almacenamiento distribuido.

Apache Hadoop no ha dejado de evolucionar y crecer en estos años. En 2017 se ha hecho pública la versión 3.0. Se han desarrollado nuevos componentes y servicios encima de Hadoop. Empresas como Yahoo!, Hortonworks, Cloudera o Facebook son algunas de las principales contribuyentes de código y funcionalidades a este proyecto de código abierto.

**Atención**

En este curso te explicaremos la versión de Apache Hadoop 2.

1.3. ENTORNO DE TRABAJO

Para seguir el contenido de esta unidad y poder desarrollar los ejemplos y ejercicios necesitarás tener acceso a una instalación de Hadoop. Como instalar y configurar Hadoop puede ser algo complicado, hemos preparado una *máquina virtual* con todo el software necesario ya instalado.

**Vídeo**

En el campus podrás encontrar vídeos de apoyo con la explicación paso a paso para descargar e instalar todo el software necesario.

¿Qué es una máquina virtual? Si es la primera vez que lees este término, no te preocupes, es algo muy sencillo. Piensa que una máquina virtual es como si tuvieras otro ordenador independiente dentro de tu ordenador. Un software de máquina virtual simula dentro de tu propio ordenador la existencia de otra computadora o servidor, con su hardware, su sistema operativo y aplicaciones instaladas.

Las ventajas son que podemos crear una máquina virtual con un sistema operativo distinto para instalar y ejecutar aplicaciones que no podemos utilizar en nuestro ordenador, o que necesitamos probar o desarrollar sin interferir ni cambiar nada en nuestra máquina y sistema operativo actual. Si se produce un fallo en la máquina virtual, nuestro ordenador seguirá funcionando normalmente sin problema.

Existen varias aplicaciones para correr máquinas virtuales. Nosotros vamos a utilizar VirtualBox de Oracle (<https://www.virtualbox.org/>).

En primer lugar, tendrás que ir a la página web de VirtualBox y descargar la última versión de la aplicación para tu sistema operativo, e instalar la aplicación normalmente.



Una vez que tengas VirtualBox instalado, tendrás que descargar el material auxiliar de la unidad que tienes disponible en el campus. Este material incluye un fichero llamado `mate32_16_04_bigdata.ova` que contiene la *imagen* de la máquina virtual que usaremos, esto es, la máquina que *simularemos* tener corriendo sobre nuestro ordenador real.



Para activar la imagen, abre la aplicación de VirtualBox. En el menú desplegable Archivo, selecciona la opción "Importar servicio virtualizado...". Te pedirá que selecciones el archivo importar. Busca el fichero mate32_16_04_bigdata.ova que has descargado, selecciónalo y continua al paso siguiente. Te aparecerán los datos y configuración de la imagen. No tienes que cambiar nada, simplemente pulsa en el botón "Importar".

Una vez que se complete la importación, verás que te aparece la nueva imagen de máquina virtual en el cuadro de la izquierda. Si quieres arrancar la imagen, márcala y pulsa "Iniciar". Se lanzará una nueva ventana en la que verás varios mensajes y pantallas, igual que si estuvieras arrancando un ordenador normal.



1.3.1. UNAS NOTAS SOBRE NUESTRA MÁQUINA VIRTUAL

Para correr Hadoop y otras tecnologías Big Data que veremos en las siguientes unidades vamos a utilizar un sistema operativo tipo Linux. En concreto vamos a utilizar una distribución (así se llaman a las distintas versiones de Linux) derivada de Ubuntu, llamada Ubuntu MATE.

Si has trabajado alguna vez en entornos Linux, esto será trivial para ti. Si nunca has tenido que usar ninguna versión de Linux, no te asustes. Aunque el aspecto pueda resultarte poco familiar, las herramientas y opciones que vamos a utilizar son muy similares a las que puedes encontrar en un entorno Windows o Mac.

Antes de avanzar más, anótate los datos de la cuenta de usuario. Por defecto, hemos configurado la máquina virtual para que inicie sesión automáticamente sin pedirte el usuario y contraseña. Sin embargo, si estás mucho rato sin hacer nada, se bloqueará la sesión (igual que te puede pasar en Windows o Mac) y te solicitará que introduzcas el usuario y contraseña para continuar. Estos son los datos:

- **Usuario:** bigdata
- **Contraseña:** bigdata



Volviendo a MATE, lo primero que verás al comenzar una sesión es el escritorio, igual que en cualquier otro sistema operativo que conozcas. En la parte superior verás una barra con un menú a la izquierda y varios indicadores a la derecha. El menú izquierdo te permite acceder a las aplicaciones y a distintas carpetas y unidades del sistema. Mientras, a la derecha del todo tienes el botón para cerrar la sesión o apagar la máquina (igual que harías en tu propio ordenador al terminar).

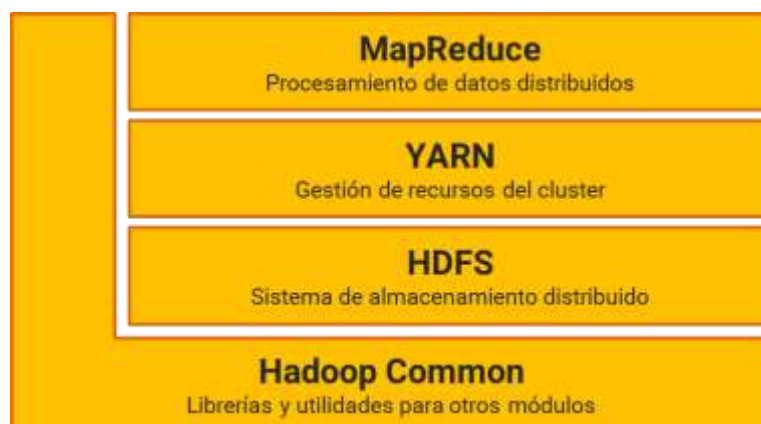
En el propio escritorio tienes dos elementos. El primero es el acceso directo a la carpeta personal del usuario bigdata. Si la abres (haciendo doble clic) verás que contiene las carpetas típicas que encuentras en otros sistemas operativos (descargas, documentos, música, etc.). También encontrarás una carpeta llamada Curso. En esta carpeta tienes los contenidos (código fuente de los ejemplos y ficheros de datos) necesarios para seguir esta unidad y las siguientes.

El segundo elemento del escritorio es el terminal de MATE. Si no has usado nunca Linux, de momento solo necesitas saber que es como la consola de comandos de Windows. A lo largo de estas unidades usaremos el terminal para lanzar comandos y programas utilizados en los ejemplos. No te preocupes, en cada paso te diremos qué comandos usar. Mientras tanto, si quieres conocer algo más del manejo del terminal y los comandos disponibles, te invitamos a ver algunos de los videos tutoriales que existen (p.ej. <https://youtu.be/RdrPFOO0H84>).

2. COMPONENTES DE HADOOP

Ahora vamos a presentarte cuáles son los principales componentes de Apache Hadoop. Esencialmente, lo que se conoce como el *core* de Hadoop consta de cuatro módulos:

- **Hadoop Common:** colección de librerías y utilidades comunes utilizadas por los demás módulos. Proporciona abstracciones del sistema operativo subyacente y de su sistema de ficheros.
- **HDFS:** El sistema de ficheros distribuido de Hadoop
- **YARN:** Se encarga de gestionar los recursos del cluster, y planificar y monitorizar las cargas de trabajo
- **MapReduce:** Sistema de procesamiento distribuido para grandes conjuntos de datos



Aunque no es imprescindible conocer cómo funcionan estos módulos para desarrollar aplicaciones básicas en Hadoop, sí que resulta muy conveniente entender su estructura y funcionamiento interno para comprender mejor qué hacen nuestras aplicaciones y sacar el máximo partido de la plataforma. En los siguientes apartados vamos a profundizar un poco más en algunos de estos módulos.

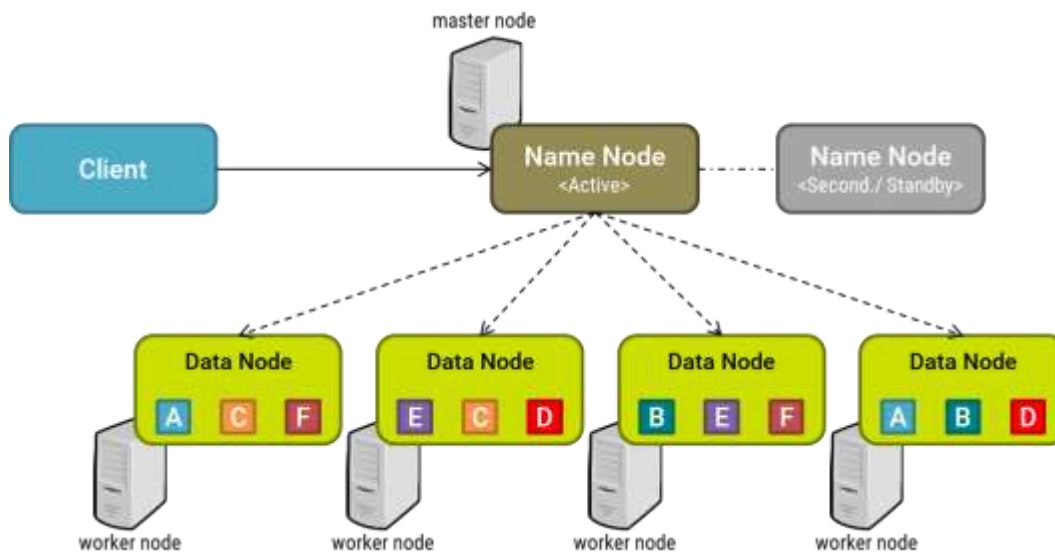
2.1. HDFS

HDFS (*Hadoop Distributed File System*) es el sistema de ficheros distribuido diseñado para la plataforma Hadoop. Está desarrollado en el lenguaje Java. Se trata de un sistema escalable y tolerante a fallos para almacenar grandes volúmenes de información de manera distribuida sobre un *cluster* de máquinas.

De hecho, HDFS está diseñado y optimizado para almacenar y gestionar archivos de datos muy grandes (típicamente desde varios GB hasta TB de tamaño), troceándolos en bloques que se reparten entre los nodos.

Además, los bloques se replican en varios nodos para evitar el riesgo de pérdida de información si alguno de los nodos falla, asegurando la disponibilidad de los datos en todo momento.

Un *cluster* HDFS está compuesto por dos tipos de nodos: el *NameNode* (nodo de nombres o directorio) y los *DataNodes* (o nodos de datos). Idealmente, cada nodo está en una máquina diferente. Si los nodos están distribuidos en varios *racks* o subredes, es posible incluir en la información de configuración del sistema la composición de estos *racks*. De esta manera HDFS puede tener en cuenta la “proximidad” o “distancia” entre nodos a la hora de gestionar los datos.



2.1.1. NAMENODE

El *NameNode* es el elemento más importante de HDFS. Es el punto de acceso al sistema de ficheros y se encarga de almacenar todos sus metadatos. Esto incluye los nombres de ficheros y jerarquías de directorios almacenados, los permisos de acceso, y la ubicación de cada uno de los bloques de un fichero a lo largo del *cluster*.

Para acelerar el acceso a esta información y agilizar las operaciones sobre los ficheros, el *NameNode* almacena los metadatos en memoria.

El *NameNode* también es el responsable de coordinar los *DataNodes* y enviarles los comandos necesarios para atender las operaciones solicitadas por un cliente. El *Namenode* monitoriza continuamente el estado de los *Datanodes* y el número de réplicas activas de cada bloque en el *cluster*, para asegurarse de que el fallo en una máquina no acaba en una pérdida de datos.

El *NameNode* juega un papel crítico; una caída del *NameNode* supondría un fallo global de la plataforma, dejándola inoperativa. Es lo que denominamos un punto de fallo único (*SPoF* – *single point of failure*). Para solventar este problema existen dos alternativas.

La primera alternativa es el llamado *NameNode* secundario. Este *NameNode* se encarga de copiar a intervalos regulares los metadatos que el *NameNode* activo o primario almacena en memoria y guardarlos en ficheros locales o remotos. En caso de fallo en el *NameNode* primario, es posible reiniciar el servicio recuperando los metadatos de una de estas copias. Este proceso no es inmediato, pero permite devolver el sistema a un estado operativo.

La segunda alternativa está diseñada para sistemas con necesidad de alta disponibilidad. Es decir, sistemas que deben estar funcionando de forma correcta ininterrumpidamente. En este caso se utiliza un *NameNode* “de repuesto” o en *standby*. Este *NameNode* se comunica y actualiza de manera continua con el *NameNode* primario. Si en algún momento el *NameNode* principal cae, se activa automáticamente el mecanismo de recuperación (*fail-over*) y el *Standby NameNode* ocupa inmediatamente el puesto del primario. De esta forma, el sistema sigue operativo de forma transparente.


El mecanismo configurado por defecto es el del *NameNode* secundario. Configurar el *NameNode* de repuesto es más complejo y también consume muchos más recursos del sistema, por lo que solo tiene sentido si realmente es necesario. Además, no es posible tener al mismo tiempo activos un *NameNode* secundario y otro en *standby*, así que hay que pensar bien cuál es la opción más adecuada.

2.1.2. **DATA NODE**

Los nodos de datos o *DataNodes* son los responsables de almacenar y servir la información del sistema. Un *cluster* HDFS está compuesto de múltiples *DataNodes* (los sistemas más grandes en funcionamiento tienen miles de nodos).

Como ya te hemos contado, HDFS está optimizado para almacenar ficheros con un tamaño mínimo de varios cientos de MB hasta varios TB. Estos ficheros son divididos en bloques de tamaño fijo, habitualmente 64 o 128 MB.

Dichos bloques se reparten entre distintos *DataNodes* para su almacenamiento, tratando de equilibrar el volumen de datos entre los nodos.



Importante

El bloque de datos es la unidad mínima de almacenamiento. Eso significa que si quieres guardar un fichero más pequeño en HDFS, se le asignará igualmente un bloque del tamaño fijo configurado (64 o 128 MB). Ahora imagina que quieres guardar 1000 archivos de 1MB. En un sistema de ficheros normal te ocuparían más o menos 1 GB de disco. En HDFS ocuparás al menos 64 GB, de los cuales 63 GB no serán información útil. ¡Casi un 98,5 % de espacio desperdiciado! ¡Y sin contar las réplicas!

Además, para garantizar la fiabilidad del sistema y el acceso a la información, cada bloque de datos se almacena redundantemente, de manera que existan varias copias en nodos diferentes.

La configuración por defecto está fijada en un factor de replicación x3. Si los *DataNode* están físicamente en varios *racks* o subredes distintas, dos réplicas de un bloque se colocarán en *DataNodes* del mismo *rack* o subred, y una tercera copia se almacenará en un *DataNode* de un *rack* diferente. Así HDFS trata de asegurar que siempre podrá acceder al menos a una copia de cada bloque.

Si en algún momento el *NameNode* detecta que un *DataNode* no responde, procede a ordenar la replicación de los bloques que tuviera asignados entre el resto de *DataNodes* activos. Los propios nodos de datos pueden comunicarse y coordinarse entre ellos para reequilibrar el volumen de datos entre ellos, mover copias de los bloques y generar nuevas réplicas.

Por último, también es importante que tengas en cuenta que HDFS fue diseñado pensando en archivos que no es necesario modificar, que no sufran cambios en su contenido una vez creados. Si se modifica el contenido de estos ficheros tan grandes estaríamos afectando a la división en bloques, invalidando el contenido de varios de ellos, teniendo además que controlar las réplicas y su actualización... HDFS no incorpora estos mecanismos de serie, lo que supone que tras modificar un fichero hay que volver a hacer el particionado, distribución y replicado desde cero. Algo muy ineficiente, así que mejor evitarlo.

2.2. YARN

Apache Hadoop YARN (*Yet Another Resource Negotiator*, “otro negociador de recursos”) es uno de los elementos clave introducidos en la versión 2 de Hadoop y que supone un cambio total respecto a la arquitectura de la primera generación.

YARN es un gestor de recursos distribuidos. A grandes rasgos, puedes verlo como una especie de sistema operativo distribuido para Hadoop. Su misión es administrar los recursos disponibles en un *cluster* de cálculo de Hadoop, coordinando y planificando su uso entre los distintos trabajos (aplicaciones) que tienen que ejecutarse.

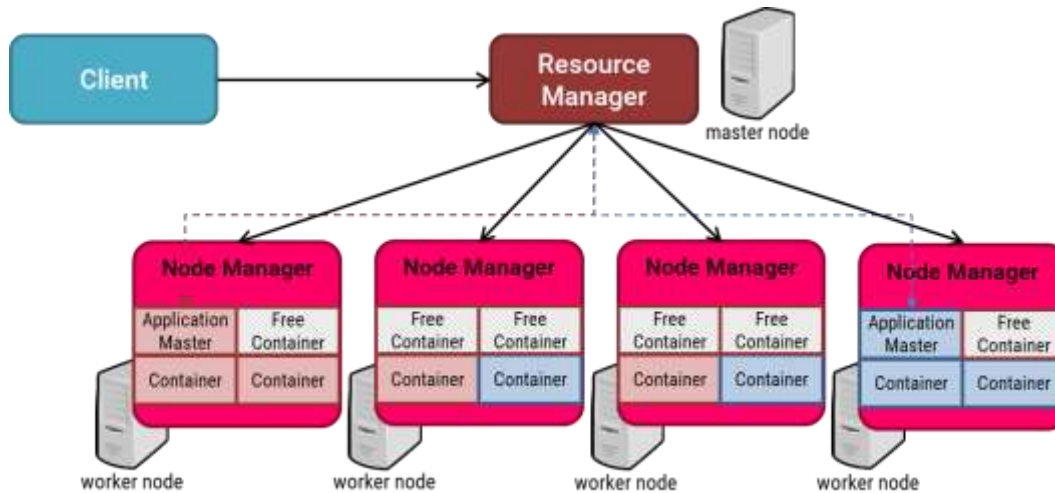
Esos *recursos* a gestionar son fundamentalmente unidades de CPU (*virtual cores* o *vcores*) para correr los cálculos y bloques de memoria RAM para cargar los datos necesarios y procesarlos de forma eficiente. Conceptualmente, el *cluster* consta de múltiples *nodos esclavos* o *trabajadores*, que ofrecen sus recursos para ejecutar los trabajos, y un *nodo maestro* que controla y administra los recursos disponibles.

En la primera versión de Hadoop un único proceso (el *JobTracker*) se encargaba de gestionar los recursos, asignarlos a cada tarea, y planificar y monitorizar la ejecución de dichas tareas.

Con YARN estas responsabilidades se reparten ahora entre varios procesos distribuidos en los nodos del *cluster*. Los principales componentes de YARN son

- Un *ResourceManager* o gestor de recursos global del sistema, ejecutándose en el nodo maestro.
- Un *NodeManager* por cada nodo esclavo, responsable de la gestión local del nodo.
- Varios *Containers* por cada nodo esclavo, representando un conjunto de recursos de un nodo particular (CPU *vcores* y RAM) disponibles para la ejecución de una tarea.
- Un *ApplicationMaster* por cada aplicación o trabajo de cálculo que se lance en la plataforma.

La siguiente figura muestra un esquema de la arquitectura de YARN.



2.2.1. RESOURCEMANAGER

El *ResourceManager* o gestor de recursos global es el proceso principal del sistema, responsable de llevar el control de los recursos disponibles en el *cluster* y orquestar el correcto uso y funcionamiento de todos los elementos del sistema.

El *ResourceManager* es además el punto de acceso a la plataforma. Cada vez que un cliente desea lanzar un trabajo o aplicación, debe someter una petición al *ResourceManager* para que inicie y coordine el proceso.

La misión principal del *ResourceManager* es arbitrar el reparto de los recursos disponibles en todo el *cluster* entre las distintas aplicaciones que solicitan ejecutarse, y que compiten por obtener esos recursos. El objetivo es maximizar la utilización de las capacidades disponibles en el *cluster* en todo momento.

El *ResourceManager* se apoya en varios componentes o procesos auxiliares que lanza también en el nodo maestro.

El *Scheduler* o planificador es el responsable de atender las solicitudes de recursos para nuevas aplicaciones y decidir qué recursos asignarles finalmente en función de la carga de trabajo y disponibilidad en la plataforma. Según la confi-

guración puede decidir encolar una solicitud si no hay recursos suficientes en este momento, o priorizar ciertos tipos de trabajo.

El *ApplicationsManager* (no confundir con los *ApplicationMaster*) es el encargado de gestionar y mantener actualizada la información de los trabajos o aplicaciones que han sido lanzados, controlando su estado de ejecución y finalización.

El *ResourceManager* (junto con todos sus componentes auxiliares) puede configurarse para garantizar la alta disponibilidad de la plataforma y que no suponga un punto de fallo único que haga caer a todo el sistema. Por ejemplo, puede configurarse un mecanismo de recuperación o *fail-over* mediante el cual, en caso de que el *ResourceManager* falle, el sistema lance un nuevo *ResourceManager* de forma automática en otro nodo del *cluster*.

2.2.2. NODEMANAGER

El *NodeManager* es el proceso encargado de gestionar los recursos locales un nodo de cálculo (*worker node*) individual. Existe una instancia de *NodeManager* por cada nodo del *cluster*.

Los *NodeManager* se coordinan continuamente con el *ResourceManager* de la plataforma, intercambiando información actualizada del estado del nodo y sus recursos. La comunicación entre el *ResourceManager* y los *NodeManager* se lleva a cabo mediante el intercambio regular de unos mensajes especiales llamados *heartbeats* o "latidos". Si el *ResourceManager* detecta la ausencia de estos latidos de un *NodeManager* durante un cierto periodo, asumirá que existe algún tipo de problema con ese nodo y activará los protocolos para recuperarlo o, si no es posible, redistribuir el trabajo entre el resto del *cluster* de forma transparente, evitando fallos o pérdidas de información.

A petición del *ResourceManager*, el *NodeManager* se ocupa de configurar y activar un nuevo *container* con los recursos necesarios para ejecutar una nueva aplicación. El *NodeManager* monitoriza el uso de los *containers* durante todo su ciclo de vida, hasta que la aplicación a la que se han asignado termina, momento en el que se destruye el *container* liberando los recursos asignados.

2.2.3. CONTAINER

Para poder ejecutar un trabajo o aplicación en la plataforma, es necesario poder garantizarle un acceso a una cantidad de recursos mínimos para que pueda completar sus cálculos.

Al mismo tiempo, hay que procurar un reparto adecuado de los recursos entre los distintos trabajos para maximizar el aprovechamiento de la plataforma. Y, además, hay que garantizar un entorno seguro sobre los recursos asignados a cada aplicación, de forma que ningún proceso ajeno pueda acceder a ellos y comprometer la ejecución o la información de la aplicación.

Para cubrir estas necesidades, en YARN se utiliza un mecanismo llamado *containers*. Un *Container* representa un conjunto de recursos de un nodo particular reservados y asignados a una aplicación para su ejecución. Estos recursos incluyen principalmente *cores* de CPU y bloques de memoria RAM, aunque en nuevas versiones también se incorpora el uso de disco, ancho de banda, etc.

Una aplicación negocia con el *ResourceManager* a través de su *ApplicationMaster* los recursos que necesita en la plataforma para ejecutarse. Una de estas peticiones de recursos consiste básicamente en una lista con la definición de los *containers* que solicita, indicando la cantidad mínima de recursos de cada tipo que debe tener cada *Container*.

Para simplificar, recursos como la memoria RAM deben solicitarse usando múltiplos de cierta cantidad mínima (p.ej. 1,2,4,8 GB; no se pueden solicitar 1283 MB).

2.2.4. APPLICATIONMASTER

El *ApplicationMaster* es el responsable de la ejecución de una aplicación. Cada aplicación corriendo en el *cluster* de Hadoop tiene su propia instancia dedicada de *ApplicationMaster*.

En realidad, la implementación del *ApplicationMaster* depende de la librería utilizada para construir una aplicación. Por ejemplo, si creamos una aplicación que utilice MapReduce, la propia librería Hadoop MapReduce nos proporciona un *ApplicationMaster* adaptado para este tipo de tareas. Otras plataformas que quieran eje-

cutarse sobre YARN, como Hive o Spark, deben proporcionar en sus librerías una versión del *ApplicationMaster*. YARN lo único que hace es definir cómo debe ser este componente y las funcionalidades básicas para que se comunique e interactúe con los demás procesos (*ResourceManager*, *NodeManager*, etc.).

El *ApplicationMaster* se ejecuta dentro de un *Container* como cualquier otra aplicación. Una vez que ha arrancado, solicita al *ResourceManager* los recursos que necesita para completar su trabajo, indicando el número y características de los *containers*, y opcionalmente las preferencias sobre dónde (en qué nodos) habilitar estos *containers*. Esta última característica es muy importante, ya que permite conseguir que los procesos de cálculo se ejecuten en los mismos nodos donde están los datos que deben utilizar. A esta propiedad la denominamos *colocalización*.

Una vez que el *ResourceManager* asigna los recursos solicitados, el *ApplicationMaster* contacta con cada uno de los *NodeManager* donde le han concedido un *Container* para tener acceso a ellos y poder lanzar sus tareas de cálculo.

El *ApplicationMaster* es responsable de gestionar todo el ciclo de vida de las tareas que ejecute en los *containers* asignados, controlando el flujo de ejecución y cualquier fallo que pueda ocurrir en la aplicación. También debe monitorizar el consumo de los recursos que hagan estas tareas.

El *ApplicationMaster* se comunica de forma regular con el *ResourceManager*, intercambiando mensajes tipo *heartbeat*, en los que incluye información del estado de ejecución, uso de los recursos asignados y posibles peticiones adicionales para ampliarlos.

Este diseño libera al *ResourceManager* de la carga de tener que gestionar las aplicaciones en ejecución directamente, evitando crear un cuello de botella muy importante al distribuir la responsabilidad entre los distintos *ApplicationMaster*. Y como ya hemos dicho, permite que se puedan ejecutar distintos modelos o plataformas distribuidas sobre YARN simplemente proporcionando la implementación del *ApplicationMaster* y los protocolos de comunicación con el resto del sistema.

Pero también tiene implicaciones relacionadas con la seguridad. Dado que el *ApplicationMaster* es un elemento proporcionado por una librería externa a YARN, la plataforma lo trata como un agente no confiable. Esto significa que no tiene acceso directo a las funcionalidades y elementos internos de la plataforma. Como hemos descrito, toda la interacción se realiza intercambiando mensajes con los procesos que la gestionan (*ResourceManager*, *NodeManagers*, ...).

2.2.5. RESUMIENDO, CÓMO FUNCIONA UNA APLICACIÓN EN YARN

Ahora que hemos visto las principales piezas de YARN, vamos a repasar cómo se ejecuta una aplicación, siguiendo la secuencia de acciones entre los componentes de la plataforma.

1. Un cliente somete una solicitud al *ResourceManager* del *cluster* para ejecutar la aplicación. La solicitud incluye el código de la aplicación y toda la información extra que pueda hacer falta, el llamado *contexto de ejecución*: parámetros, datos de configuración y otras dependencias (librerías, ficheros, etc.).
2. El *ApplicationsManager* del *ResourceManager* negocia la asignación de un primer *Container* para arrancar una instancia del *ApplicationMaster* de la aplicación a ejecutar.
3. Una vez que la instancia del *ApplicationMaster* de nuestra aplicación ha arrancado, se comunica con el *ResourceManager* para solicitarle los *containers* que necesite para poder lanzar los cálculos.
4. El *ResourceManager* gestiona la reserva de *containers* en los nodos de la plataforma y devuelve al *ApplicationMaster* los "tickets" acreditativos de los *containers* asignados en cada nodo.
5. El *ApplicationMaster* contacta con los *NodeManager* de los nodos involucrados y les solicita que activen los *containers* para los que ha sido acreditado, copiando la información del contexto de ejecución.
6. Una vez que tiene acceso a los *containers*, el *ApplicationManager* procede a ejecutar la aplicación. El propio *ApplicationManager* es responsable de planificar qué tareas ejecutar en cada *container*, controlar los recursos consumidos por cada tarea y monitorizar la evolución de los procesos. El *ApplicationManager* comunica periódicamente al *ResourceManager* el estado de la aplicación y de los recursos.
7. Una vez que la aplicación se completa, el *ApplicationManager* informa al *ResourceManager* de la finalización. Este procede a eliminar la aplicación del registro de aplicaciones en ejecución y después libera los *containers* asignados, dejando los recursos disponibles para otras aplicaciones.

2.2.6. INTEGRACIÓN HDFS-YARN Y COLOCALIZACIÓN

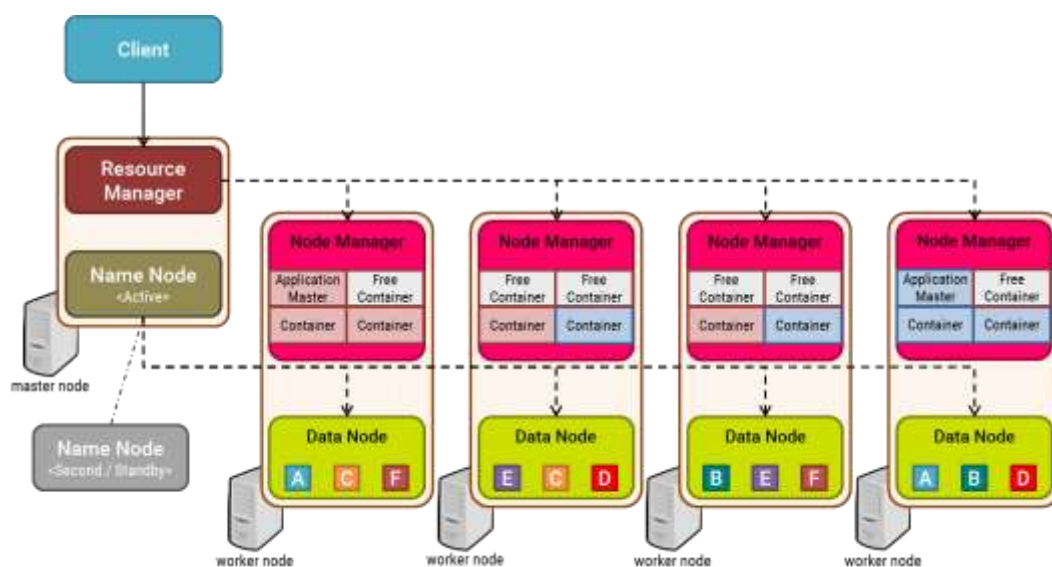
Uno de los principios fundamentales en el diseño inicial de Hadoop fue mejorar el rendimiento del sistema haciendo que los procesos de cálculo se ejecuten en las máquinas donde se encuentran los datos a utilizar. En otras palabras, transfiriendo el código de las tareas a ejecutar a través de la red hasta los nodos con los datos, en lugar de que la aplicación tenga que pedir los datos y traerlos de nodos diferentes.

Los motivos de este diseño eran que el acceso a los datos en un disco de la propia máquina era mucho más rápido que transferirlos a través de una red local, y que los procesos típicos de Hadoop consumían la mayor parte del tiempo leyendo datos.

Al final, esta idea puede resumirse como “es más rápido mover el código que mover los datos”.

A esta propiedad de procurar que coincidan procesos de cálculo y datos distribuidos se la denomina *colocalización*.

En una instalación de Hadoop en un *cluster* propio, con múltiples máquinas en una intranet local, esta colocalización se traduce en una arquitectura donde configuramos los nodos *maestro* de HDFS (*NameNode*) y YARN (*ResourceManager*) en una misma máquina, y procedemos igualmente con los nodos *esclavo*, instanciando en cada máquina los procesos de HDFS (*DataNode*) y YARN (*NodeManager*). El siguiente diagrama resume este tipo de despliegue habitual de un *cluster* Hadoop.



Cuando una aplicación MapReduce va a ejecutarse, su *ApplicationMaster* puede consultar en qué nodos se encuentran los datos que necesita y solicitar que se le asignen recursos de cálculo (*containers*) en las mismas máquinas.

Es importante resaltar que este mecanismo de asignación de recursos proporcionando acceso local a los datos intenta satisfacer estas peticiones de la mejor forma posible, pero no garantiza que finalmente proporcione esa *colocalización*. Si en un nodo donde hay datos no hay suficientes recursos de cálculo disponibles, YARN buscará otro nodo con capacidad libre para activar un *container* adecuado a los requisitos de la aplicación.

A día de hoy, sin embargo, las tecnologías y necesidades han evolucionado, y este tipo de arquitectura no es siempre adecuada.

Por un lado, las latencias de transferencia en las redes de alta velocidad que se usan actualmente en los *data centers* se han reducido drásticamente. Existen estudios que apuntan que una aplicación accediendo a datos en un disco local es solo un 8% más rápida que si accede a los datos en otro nodo ubicado en el mismo *rack*:

- https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/disk-irrelevant_hotos2011.pdf
- <http://cseweb.ucsd.edu/~gmporter/papers/superdn-osr2010.pdf>

Por otro lado, el crecimiento imparable en la cantidad de datos a guardar obliga a ampliar la capacidad de almacenamiento de los *clusters* a un ritmo mucho mayor. Piensa en un gran *data center* de nuevo. Es necesario disponer de medios de almacenamiento asequibles para crecer. Pero los requisitos de *hardware* para un nodo de cálculo son mucho mayores que para un nodo de datos. Y consiguientemente son mucho más costosos. No siempre es factible escalar la plataforma añadiendo nodos capaces de cubrir simultáneamente las dos necesidades.

Por estos motivos, actualmente en muchos casos los nodos de datos y los nodos de cálculo se encuentran desplegados en máquinas con características distintas, conectados por redes de alta velocidad.

2.3. MAPREDUCE

MapReduce es un modelo de programación que permite procesar grandes cantidades de información dividiendo el trabajo en una serie de tareas que se ejecutan en paralelo en varios nodos dentro de un *cluster*. La primera referencia a este modelo de cálculo aplicado a Big Data es en un artículo publicado por Google en 2004. Ref:

- <http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf>

En cierta forma, podemos verlo como una “receta” o “patrón” para procesar un conjunto de datos de gran tamaño siguiendo una secuencia de pasos concreta.

Como indica su nombre, el modelo MapReduce está inspirado en las construcciones *map* y *reduce* de programación funcional que ya has visto en unidades anteriores. Si recuerdas de cuando explicamos las funciones de orden superior en Python en la Unidad 9, la operación *map* nos permitía transformar una serie de datos, aplicando una función a cada uno de los valores. Mientras, la operación *reduce* nos servía para aplicar una función sobre una serie de valores, acumulando o “arrastrando” los valores intermedios hasta obtener un único resultado final.

La “receta” MapReduce consta a grandes rasgos de dos pasos: aplicar una primera función a todos los datos de entrada para obtener nuevos valores intermedios, y después aplicar sobre estos una segunda función que los combine y *reduzca* a un resultado final.

Refresquemos la memoria con un ejemplo, calculando una suma de cuadrados usando las funciones *map* y *reduce* en Python.

```
from functools import reduce

valores = [1,2,3,4,5]
cuadrados = map(lambda x: x**2, [1,2,3,4,5]) # 1.Map: elevar al cuadrado
suma_cuadrados = reduce(lambda x,y: x + y, # 2.Reduce : sumar cuadrados
                        cuadrados)
print(suma_cuadrados)
```

Es fácil ver que transformar una serie de datos con un mecanismo tipo *map*, aplicando a cada elemento una función que devuelve un nuevo valor, es una operación directamente paralelizable, ya que cada operación o *aplicación* es totalmente independiente de las demás.

Pero claro, cuando hablamos de millones y millones de registros, no es tan sencillo como usar simplemente unas funciones *map* y *reduce*. ¿Cómo se reparten los datos entre los nodos del *cluster*? ¿Cómo se lanza la ejecución de cada paso? ¿Cómo controlar que todos los cálculos parciales se completan correctamente? ¿Cómo se combinan los resultados parciales de cada nodo?

Hadoop MapReduce proporciona una implementación de este modelo de programación sobre la plataforma Hadoop, apoyándose en sus servicios y componentes para dar solución a todas estas “cuestiones de logística”.

En las versiones 1.x de Hadoop, la infraestructura MapReduce incluía todos los procesos y servicios de gestión y ejecución de trabajos, planificación de recursos y administración en general de los nodos.

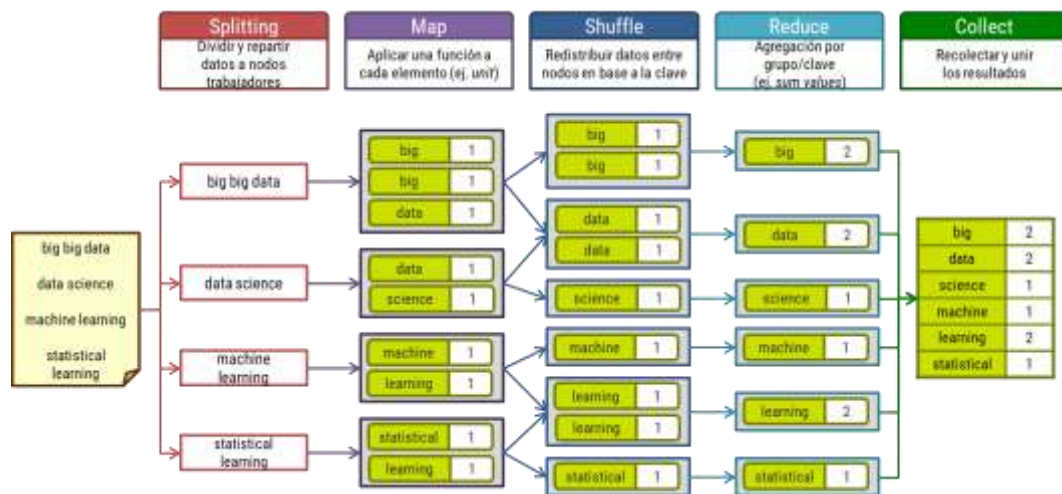
Desde la versión 2.0 todas estas tareas recaen en YARN, como ya hemos visto. Ahora Hadoop MapReduce es un tipo de aplicación más, ejecutable en un *cluster* Hadoop gestionado por YARN. Las librerías de Hadoop MapReduce incluyen versiones predefinidas del ApplicationMaster y otros componentes, ya adaptadas para este modelo de programación.

Esta separación de responsabilidades flexibiliza la plataforma, y rebaja la complejidad de la propia capa MapReduce.

Volviendo al punto de vista de un usuario que quiere implementar un cálculo con la “receta” MapReduce, conviene conocer cuáles son los pasos a seguir.

Vamos a ver las distintas etapas en las que se divide un proceso MapReduce en Hadoop, indicando aquellas que “rellenaremos” nosotros con nuestro propio código para realizar los cálculos que nos interesan. Otras serán gestionadas automáticamente por la librería MapReduce.

Para seguir más fácilmente los pasos utilizaremos un caso de ejemplo, el más típico usado en toda introducción a MapReduce: contar cuántas veces aparece cada palabra en una serie de ficheros de texto. El siguiente diagrama representa la secuencia de etapas con el flujo de datos para este ejemplo.



2.3.1. CARGA Y REPARTICIÓN INICIAL

El primer paso consiste en leer los datos de fichero, dividirlos y repartirlos entre los procesos de la siguiente etapa.

En el caso habitual, los datos se encontrarán en ficheros almacenados sobre HDFS. El formato de estos ficheros es arbitrario, puede ser cualquiera. Hadoop proporciona de serie el soporte para leer distintos tipos de ficheros de entrada, incluyendo ficheros de texto arbitrarios, ficheros con pares clave y valor en modo texto o en modo binario, etc. Simplemente hay que indicar cuál es el formato de entrada.

Conforme se van leyendo los datos, se dividen en bloques o unidades de procesamiento, que son enviadas a la siguiente fase, la etapa *Map*.

Si se desea procesar ficheros con un formato particular no soportado de forma nativa por el sistema, también es posible proporcionar una implementación propia de las funciones que leen y generan el reparto.

En nuestro ejemplo, tenemos unos ficheros de texto, con distintas frases o listas de palabras clave en cada línea. El contenido de cada fichero sería dividido en líneas, y cada línea despachada a un nodo preparado para procesarlas. Para ser exactos, lo que se envía es un par (una tupla de dos elementos), formada por un identificador y el propio dato (la línea leída, en este ejemplo).

2.3.2. MAP

Llegamos a una de las dos partes clave del todo el flujo MapReduce. En la etapa *Map* se toman los datos de entrada recibidos de la fase inicial y se procesan aplicando una función definida por el usuario, generando unos valores intermedios como salida, que se mandan hacia los procesos de la siguiente etapa.

En Hadoop MapReduce, a los procesos que realizan estas operaciones se les denomina *Mappers*. Y como decimos, la implementación del *Mapper*, las operaciones que debe realizar sobre los datos de entrada, la debe proporcionar el usuario.

El *Mapper* solamente debe respetar una *interfaz* de la función, es decir, la forma en la que recibe y devuelve los datos para interconectarse con el resto de elementos.

La función *Map* recibe un par con el identificador y el dato despachado por la etapa anterior, y genera un número arbitrario (varios, uno o ninguno) de pares formados por una clave y un valor. El contenido de la clave y del valor generado en cada par puede ser cualquiera, dependerá de la funcionalidad que quiera implementar el usuario. La "*firma*" de la función *Map* sería algo así:

$$\text{map}(\text{clave}_{in}, \text{valor}_{in}) \rightarrow [(\text{clave}_{out,1}, \text{valor}_{out,1}), \dots, (\text{clave}_{out,n}, \text{valor}_{out,n})]$$

En el ejemplo sencillo del conteo de palabras, nuestro *Mapper* recibe como dato una línea con varias palabras. Tras trocear las palabras individuales, devuelve una lista de pares clave-valor, donde la clave es cada palabra extraída y como valor se usa simplemente la unidad. Es una forma de contar apariciones de cada palabra de una en una. Pensarás que no es la manera más eficiente, pero para el propósito de nuestro ejemplo es útil.

Combiners

Antes de distribuir los resultados intermedios del *Mapper* en la siguiente etapa, es posible definir un paso previo sobre estos resultados. El objetivo es adelantar algunos de los cálculos posteriores, ejecutándolos en los mismos nodos que los procesos *Mapper*, de manera que la cantidad de datos a transferir por la red se

reduzca. A estos procesos opcionales se los denomina *Combiners*, y suelen realizar una versión parcial de los cálculos que después harán los procesos *Reducer* (de hecho muchas veces se utiliza la misma función para ambos).



Atención

Si se quieren usar *Combiners* para optimizar el tráfico de datos y la ejecución de cálculos, hay que tener especial cuidado con su implementación. Se debe garantizar que el resultado final es el mismo incluyendo el *Combiner* o dejándolo fuera de la cadena de procesos. Esto debe ser así porque será el motor MapReduce el que decida si es necesario ejecutar el *Combiner* en un nodo o no. Es más, como el motor MapReduce puede aplicar varias veces un *Combiner*, la función que usemos debe ser idempotente. Esto es, que el resultado final sea el mismo independientemente del número de veces que la apliquemos.

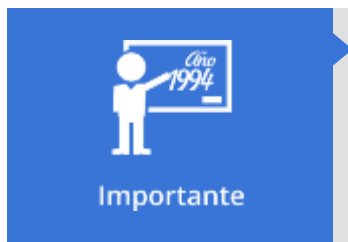
Por todas estas consideraciones, si no está claro cómo implementarlos correctamente, es mejor no utilizar los *Combiners*.

2.3.3. REORDENACIÓN Y REPARTO

Conforme las tareas *Map* se van completando, se generan resultados intermedios. Estos resultados deben ser distribuidos hacia los nodos donde se ejecutarán las tareas *Reduce*. Pero antes hay que decidir cómo distribuirlos, de manera que la carga de trabajo quede repartida de forma equilibrada.

En primer lugar, hay que dividir y agrupar los resultados de alguna manera. Enviar directamente los resultados individuales, uno a uno, de un nodo *Map* a un nodo *Reduce* no sería eficiente.

Los responsables de dividir estos resultados intermedios son unos procesos llamados *Partitioners*. Los *Partitioners* dividen los resultados de los *Mappers* en base a su clave. El objetivo es doble: que los resultados con distintas claves se repartan entre nodos *Reduce* distintos, y asegurarse de que todos los valores correspondientes una misma clave queden agrupados y se envíen y procesen juntos en por un solo proceso *Reducer*.



Recuerda esto porque es muy importante a la hora de diseñar nuestros programas MapReduce. Todos los pares (clave, valor) que tengan la misma clave se envían a un mismo Reducer para que procese todos los valores juntos.

Al proceso de transferir los resultados ya particionados a los nodos *Reduce* se lo denomina *shuffling* (mezcla o barajeado).

Una vez transferidos los datos, antes de lanzar la ejecución del proceso *Reducer*, se ejecuta una ordenación de los datos recibidos en base a su clave. Ya hemos dicho que todos los resultados con una misma clave se envían a un mismo nodo *Reduce*. Pero eso no significa que un nodo *Reduce* reciba solamente resultados de una única clave. A un nodo *Reduce* le tocará procesar datos de varias claves. Al ordenar los datos recibidos por la clave, es fácil calcular cuántas claves distintas nos llegan, y así estimar cuántos procesos *Reducer* vamos a tener que ejecutar (uno por cada clave).

2.3.4. REDUCE

Y llegamos a la segunda parte clave de MapReduce. En la etapa *Reduce* se toman los resultados (*clave, valor*) intermedios generados desde los *Mappers*, procesándolos con una función definida por el usuario para generar los resultados finales. A estos procesos los denominamos *Reducers*.

La funcionalidad típica de un *Reducer* suele consistir en realizar distintos tipos de agregaciones sobre el conjunto de datos correspondientes a una misma clave, incluyendo también filtrados o transformaciones de los valores.

Al igual que en el caso del *Mapper*, el usuario es quien debe proporcionar su implementación del *Reducer*, con las operaciones que necesita realizar. El único requisito es respetar la *interfaz* de la función.

Una función *Reduce* recibe una clave de entrada y una lista con los valores asociados a esa clave, generados anteriormente por los *Mappers*. Como resultado, genera uno o más pares (*clave*, *valor*) finales. No deben generarse pares con claves repetidas. Es decir, el valor para una clave debe ser único, su resultado final. La “firma” de la función Reduce sería:

$$\text{reduce}(\text{clave}_{in}, [\text{valor}_{in,1}, \dots, \text{valor}_{in,m}]) \rightarrow$$

$$[(\text{clave}_{out,1}, \text{valor}_{out,1}), \dots, (\text{clave}_{out,n}, \text{valor}_{out,n})]$$

Igual que en el caso de la etapa *Map*, el contenido de las claves y valores generados como resultado puede ser cualquiera, dependiendo de la funcionalidad a implementar. En particular, las claves de salida no tienen por qué corresponder a las claves de entrada.

En nuestro ejemplo sencillo, la etapa *Reduce* consiste en sumar los valores con los conteos únicos para cada clave (palabra), obteniendo el número de veces que aparece una palabra en el texto original.

2.3.5. RECOPILACIÓN

El último paso de un proceso MapReduce consiste simplemente en recolectar los resultados generados por los procesos *Reducer* y combinarlos para producir la salida final. El comportamiento por defecto es que estos resultados finales se terminen escribiendo en uno o más archivos, típicamente sobre HDFS.

3. PRIMEROS PASOS CON HADOOP

En las próximas dos secciones vamos a ver algunos ejemplos prácticos sobre cómo trabajar con los dos principales componentes de Hadoop: el sistema de ficheros HDFS y el modelo de programación MapReduce.

Para ello vamos a usar la versión de Hadoop que hemos instalado en la máquina virtual que te presentamos al inicio de esta unidad. Abre la aplicación de VirtualBox, selecciona la imagen de la máquina virtual del curso y arráncala. Una vez que te aparezca la pantalla de bienvenida, introduce el usuario y contraseña para iniciar la sesión.

3.1. ARRANCANDO LOS SERVICIOS DE HADOOP

Los distintos componentes de Hadoop (HDFS, YARN...) utilizan muchos procesos trabajando a la vez para funcionar. Estos procesos pueden consumir gran cantidad de recursos de procesador y memoria durante el arranque. Como nuestra instalación funciona sobre una máquina virtual con recursos restringidos, la hemos configurado para que los procesos de Hadoop no se activen automáticamente y así aligerar el arranque de la imagen.

Esto significa que para poder trabajar con Hadoop y probar los ejemplos tendrás que lanzar manualmente los servicios de Hadoop. No te preocupes, no es algo complicado.

Lo primero que tendrás que hacer (una vez que hayas iniciado sesión en la máquina virtual) es abrir el terminal de MATE para poder introducir una serie de comandos.

Lo primero es arrancar los servicios de HDFS, ejecutando el comando `start-dfs.sh`

```
bigdata@bigdata-VirtualBox:~$ start-dfs.sh
```

```
17/12/26 17:23:52 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable

Starting namenodes on [localhost]

localhost: starting namenode, logging to /opt/hadoop-2.7.3/logs/hadoop-bigdata-
namenode-bigdata-VirtualBox.out

localhost: starting datanode, logging to /opt/hadoop-2.7.3/logs/hadoop-bigdata-
datanode-bigdata-VirtualBox.out

Starting secondary namenodes [0.0.0.0]

0.0.0.0: starting secondarynamenode, logging to /opt/hadoop-2.7.3/logs/hadoop-
bigdata-secondarynamenode-bigdata-VirtualBox.out

17/12/26 17:24:09 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable
```

Los mensajes que aparecen por el terminal te indican cómo va lanzando los procesos NameNode.

Una vez arrancados los procesos de HDFS, falta arrancar los procesos de YARN. En este caso, escribimos en el terminal el comando `start-yarn.sh` y pulsamos INTRO para ejecutarlo:

```
bigdata@bigdata-VirtualBox:~$ start-yarn.sh
```

```
starting yarn daemons

starting resourcemanager, logging to /opt/hadoop-2.7.3/logs/yarn-bigdata-
resourcemanager-bigdata-VirtualBox.out

localhost: starting nodemanager, logging to /opt/hadoop-2.7.3/logs/yarn-bigdata-
nodemanager-bigdata-VirtualBox.out
```

Si todo va bien, deberías ver unos mensajes parecidos a los anteriores.

3.2. DETENIENDO LOS SERVICIOS DE HADOOP.

Para asegurarnos de que no se produce ningún error, cuando vayamos a dejar de trabajar con la máquina virtual deberemos para los servicios de Hadoop antes de cerrar la imagen que estamos usando. Si no lo hacemos así, es posible que el sistema de ficheros HDFS quede en un estado incorrecto y la siguiente vez que vayamos a acceder nos encontremos con algún problema.

Para detener los servicios de Hadoop ejecutamos los siguientes comandos en el terminal de MATE. Empezamos parando los procesos de YARN con **stop-yarn.sh**

```
bigdata@bigdata-VirtualBox:~$ stop-yarn.sh
```

```
stopping yarn daemons
stopping resourcemanager
localhost: stopping nodemanager
```

Y después paramos los procesos de HDFS con el comando **stop-dfs.sh**

```
bigdata@bigdata-VirtualBox:~$ stop-dfs.sh
```

```
18/12/26 18:06:07 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable

Stopping namenodes on [localhost]

localhost: stopping namenode
localhost: stopping datanode

Stopping secondary namenodes [0.0.0.0]

0.0.0.0: stopping secondarynamenode

18/12/26 18:06:27 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable
```



Importante

Recuerda asegurarte de detener los servicios de Hadoop usando **stop-yarn.sh** y **stop-dfs.sh** antes de cerrar la máquina virtual. Así evitarás problemas como la corrupción del sistema de ficheros.

4. TRABAJANDO CON HDFS

En esta sección vamos a mostrarte cómo interactuar con el sistema de ficheros HDFS. Existen distintos comandos para llevar a cabo las operaciones más habituales, como listar el contenido de un directorio, crear nuevos directorios, copiar o borrar ficheros, etc.

4.1. OPERACIONES HABITUALES

Para interactuar con HDFS normalmente usaremos la línea de comandos. La herramienta básica para ejecutar distintas operaciones es el script **hdfs**, que tiene esta sintaxis:

```
$ hdfs COMANDO [-opción <argumentos>]
```

El **COMANDO** indica qué servicio de HDFS necesitamos utilizar. Para indicar que queremos lanzar alguna operación sobre el sistema de ficheros y su contenido, utilizaremos el comando **dfs**. Con la **opción** seleccionaremos la operación concreta a ejecutar; si la operación requiere más argumentos, los especificaremos a continuación.



Atención

Para seguir los siguientes ejemplos, puedes abrir un nuevo terminal en la máquina virtual (doble click en el icono "Terminal de MATE" que tienes en el escritorio). Asegúrate de haber arrancado los servicios de HDFS antes de continuar.

4.1.1. LISTAR EL CONTENIDO DE UN DIRECTORIO

Para mostrar el contenido de un directorio en HDFS usamos la operación `-ls`

```
$ hdfs dfs -ls /
```

```
18/02/21 12:45:51 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable
```

```
Found 1 items
```

```
drwxr-xr-x  - bigdata bigdata          0 2017-12-19 16:20 /user
```

En este caso, hemos solicitado el listado de contenidos del directorio raíz de HDFS (la barra `/` representa el directorio raíz).

¿Te ha aparecido un mensaje de aviso (warn) en la consola al ejecutar el comando? Ese *"Unable to load native-hadoop library for your platform... using builtin-java classes where applicable"*. No te preocupes, no es nada grave. Dependiendo de la versión de Hadoop y del sistema operativo, es posible que no haya librerías nativas (esto es, compiladas para funcionar con el S.O.) disponibles. En ese caso, Hadoop usa automáticamente las librerías en lenguaje Java. La diferencia es únicamente que cuestan algo más de cargar que las nativas. Cada vez que se ejecute una operación, aparecerá ese mensaje de aviso.

Como resultado, HDFS nos devuelve una salida similar a la que obtenemos con el comando `ls` en un sistema Unix/Linux. Por defecto, `-ls` nos muestra la lista de permisos sobre cada fichero (esa cadena `drwxr-xr-x`), el usuario y grupo propietario, la fecha de creación o última modificación, y el nombre del fichero o directorio. Si alguno de estos campos te resulta un poco extraño, te animo a que busques información sobre el comando `ls` y el funcionamiento del sistema de ficheros y permisos en Unix.

Para desarrollar el contenido y ejemplos de esta unidad, ya hemos creado los directorios `/user` y `/user/bigdata`. Veamos el contenido de `/user`

```
$ hdfs dfs -ls /user
```

```
Found 1 items
drwxr-xr-x  - bigdata bigdata          0 2017-12-19 16:20 /user/bigdata
```

Y ahora el contenido del directorio `/user/bigdata`

```
$ hdfs dfs -ls /user/bigdata
```

En este caso, la operación `-ls` no muestra nada, tenemos un directorio vacío (por el momento...).

4.1.2. CREAR DIRECTORIOS

Podemos crear directorios en HDFS utilizando la operación `-mkdir`. Vamos a crear un directorio para nuestros ejercicios de esta unidad

```
$ hdfs dfs -mkdir /user/bigdata/U12
```

Y ahora, un nuevo directorio adicional para los datos que usemos

```
$ hdfs dfs -mkdir /user/bigdata/U12/datos
```

Comprobemos que los directorios se han creado con `-ls`

```
$ hdfs dfs -ls /user/bigdata/U12
```

```
18/02/21 15:17:24 WARN util.NativeCodeLoader: Unable to load native-hadoop li-
brary for your platform... using builtin-java classes where applicable

Found 1 items
drwxr-xr-x  - bigdata bigdata          0 2018-02-21 15:16 /user/bigdata/U12/datos
```


4.1.3. COPIAR DATOS A HDFS

Ahora que ya hemos preparado directorios donde colocar nuestros datos, es el momento de copiar algunos ficheros.

Si abres la carpeta personal de bigdata que aparece en el escritorio de la máquina virtual, verás que hay una carpeta llamada **Curso** que contiene varias carpetas y ficheros correspondientes a distintas unidades.

Usaremos los ficheros contenidos en **Curso → U12_hadoop → datos**. Desde el terminal de comandos, puedes ver también estos ficheros con el siguiente comando:

```
$ ls -l /home/bigdata/Curso/U12_Hadoop/datos/
-rwxrwx--- 1 bigdata bigdata 45830 feb  9 11:25 UD06_fundamentosSLR.txt
-rwxrwx--- 1 bigdata bigdata 64974 feb  9 11:55 UD07_regresion_lineal.txt
-rwxrwx--- 1 bigdata bigdata 49842 feb  8 10:22 UD11_introduccion_big_data.txt
```



Ojo al dato

Este es el comando ls de Unix, muestra el contenido del sistema de ficheros propio de la máquina. ¡No tiene nada que ver con HDFS!

Los ficheros que tenemos en este directorio local no son más que versiones en formato de texto plano de algunas de las unidades de este curso.

Vamos a copiar estos ficheros al directorio que hemos creado en HDFS. Para copiar ficheros del sistema local a HDFS usamos la operación **-put**.

```
$ hdfs dfs -put /home/bigdata/Curso/U12_Hadoop/datos/*.txt
/user/bigdata/U12/datos
```


Veamos ahora el contenido del directorio en HDFS

```
$ hdfs dfs -ls /user/bigdata/U12/datos
```

Found 3 items

```
-rw-r--r--  1 bigdata bigdata      46155 2018-02-21 16:37
/user/bigdata/U12/datos/UD06_fundamentosSLR.txt

-rw-r--r--  1 bigdata bigdata      64863 2018-02-21 16:37
/user/bigdata/U12/datos/UD07_regresion_lineal.txt

-rw-r--r--  1 bigdata bigdata      50211 2018-02-21 16:37
/user/bigdata/U12/datos/UD11_introduccion_big_data.txt
```

4.1.4. VER EL CONTENIDO DE UN FICHERO

Para examinar el contenido de un fichero en HDFS podemos utilizar la operación `-cat`. Este comando lee el fichero en HDFS y muestra su contenido por la salida estándar (en este caso, por el terminal).

El siguiente comando utiliza `-cat` para mostrar el contenido de uno de los ficheros que hemos copiado en HDFS:

```
$ hdfs dfs -cat /user/bigdata/U12/datos/UD06_fundamentosSLR.txt | head
```

```
Fundamentos del Aprendizaje Estadístico
MasterD
1. TABLE OF CONTENTS
2.   Tu reto en esta unidad      3
3.   Introducción      3
4.   Ejemplos      4
4.1. Precio de la vivienda 4
4.2. Cancer de Próstata      6
4.3. Reconocimiento de Texto escrito      7
4.4. Sistemas de recomendación      8
cat: Unable to write to output stream.
```

La última parte del comando anterior (esa barra vertical `|` seguida del **head**) son también comandos del *shell* de Linux. Sirve para mostrar solo las primeras líneas, descartando el resto. Es por eso que también verás un mensaje de *cat* al final diciendo que no pudo terminar de escribir al canal de salida, **head** se lo ha cortado para que no muestre más líneas.

4.1.5. COPIAR FICHEROS DENTRO DE HDFS

Antes hemos copiado o “*subido*” varios ficheros a HDFS desde nuestro sistema de ficheros local. Naturalmente, también podemos crear copias de ficheros ya existentes en HDFS. Para ello usamos el comando `-cp`

```
hdfs dfs -cp /user/bigdata/U12/datos/UD06_fundamentosSLR.txt
/user/bigdata/U12/datos/copia.txt
```

Comprobemos que el fichero `copia.txt` aparece ya en el directorio.

```
hdfs dfs -ls /user/bigdata/U12/datos
```

Found 4 items

```
-rw-r--r--  1 bigdata bigdata      46155 2018-02-21 16:37
/user/bigdata/U12/datos/UD06_fundamentosSLR.txt

-rw-r--r--  1 bigdata bigdata      64863 2018-02-21 16:37
/user/bigdata/U12/datos/UD07_regresion_lineal.txt

-rw-r--r--  1 bigdata bigdata      50211 2018-02-21 16:37
/user/bigdata/U12/datos/UD11_introduccion_big_data.txt

-rw-r--r--  1 bigdata bigdata      46155 2018-02-21 16:58
/user/bigdata/U12/datos/copia.txt
```

4.1.6. BORRADO DE FICHEROS

Si necesitamos borrar uno o varios ficheros de HDFS, disponemos de la operación `-rm`. Vamos a borrar la copia del fichero que hemos creado antes.

```
$ hdfs dfs -rm /user/bigdata/U12/datos/copia.txt
```

```
18/02/21 17:07:38 INFO fs.TrashPolicyDefault: Namenode trash configuration:
Deletion interval = 0 minutes, Emptier interval = 0 minutes.
```

```
Deleted /user/bigdata/U12/datos/copia.txt
```

HDFS nos muestra un par de mensajes tras esta operación. El último nos indica que ha borrado el fichero. ¿Pero y el primer mensaje?

Cuando pedimos borrar un fichero en HDFS, no se elimina inmediatamente. HDFS primero mueve el fichero a un directorio que hace las veces de papelera (`/trash`), donde lo mantiene durante un periodo de tiempo configurable. Durante ese tiempo, es posible recuperar el fichero si lo borramos por error. Tras ese “periodo de *cuarentena*”, el fichero es eliminado definitivamente del sistema.

El mensaje que vemos en primer lugar nos informa de que el periodo de mantenimiento en la *papelera* y el de vaciado están configurados (por defecto) a cero. Es decir, el fichero se eliminará inmediatamente. Estos intervalos de tiempo pueden ajustarse dentro de los ficheros de configuración de HDFS, pero eso queda fuera del alcance de esta sección.

4.1.7. OBTENER FICHEROS DE HDFS

Si necesitamos extraer un fichero almacenado en HDFS (por ejemplo, un fichero con los resultados finales de nuestros cálculos) para utilizarlo en nuestro ordenador, podemos hacerlo utilizando el comando **-get**.

Vamos a traernos una copia de uno de los ficheros a nuestro directorio local:

```
$ hdfs dfs -get /user/bigdata/U12/datos/UD06_fundamentosSLR.txt
copia_UD06.txt
```

Ahora tendremos un fichero **copia_UD06.txt** en el directorio donde hayamos ejecutado el comando **hdfs**. Comprobémoslo.

```
$ ls -l
```

```
-rw-r--r-- 1 bigdata bigdata 46155 feb 21 17:25 copia_UD06.txt
drwxrwxr-x 3 bigdata bigdata 4096 feb 9 12:26 Curso
drwxr-xr-x 3 bigdata bigdata 4096 may 17 2017 Descargas
drwxr-xr-x 2 bigdata bigdata 4096 feb 15 16:07 Documentos
drwxr-xr-x 2 bigdata bigdata 4096 feb 8 09:12 Escritorio
drwxr-xr-x 2 bigdata bigdata 4096 may 17 2017 Imágenes
drwxrwxr-x 2 bigdata bigdata 4096 feb 15 17:31 MasterD
drwxr-xr-x 2 bigdata bigdata 4096 may 17 2017 Música
drwxr-xr-x 2 bigdata bigdata 4096 may 17 2017 Plantillas
drwxr-xr-x 2 bigdata bigdata 4096 may 17 2017 Público
drwxrwxr-x 2 bigdata bigdata 4096 may 18 2017 spark-warehouse
-rw-rw-r-- 1 bigdata bigdata 1559 may 18 2017 Untitled.ipynb
drwxr-xr-x 2 bigdata bigdata 4096 may 17 2017 Vídeos
```

Ahí tenemos el fichero.

Fíjate que en el directorio de HDFS el fichero fuente sigue existiendo, solo nos hemos traído una copia.

4.1.8. REFERENCIA

Los comandos que has visto son solo una parte de todas las operaciones que podemos realizar. Si quieres obtener una lista de los comandos para trabajar con ficheros en HDFS, puedes ejecutar lo siguiente:

```
$ hdfs dfs
```

```
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] <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 [-d] [-h] [-R] [<path> ...]]
  [-mkdir [-p] <path> ...]
  [-moveFromLocal <localsrc> ... <dst>]
  [-moveToLocal <src> <localdst>]
  [-mv <src> ... <dst>]
  [-put [-f] [-p] [-l] <localsrc> ... <dst>]
  [-renameSnapshot <snapshotDir> <oldName> <newName>]
  [-rm [-f] [-r|-R] [-skipTrash] <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>           use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to be
copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated jar files to
include in the classpath.
-archives <comma separated list of archives> specify comma separated archives
to be unarchived on the compute machines.
```

The general command line syntax is

```
bin/hadoop command [genericOptions] [commandOptions]
```

Si necesitas ayuda adicional sobre una operación en concreto, puedes obtenerla escribiendo:

```
$ hdfs dfs -help <operación>
```

Por ejemplo, para obtener más información sobre la operación `-put`:

```
$ hdfs dfs -help put
```

```
-put [-f] [-p] [-l] <localsrc> ... <dst> :
  Copy files from the local file system into fs. Copying fails if the file already
exists, unless the -f flag is given.
Flags:

-p Preserves access and modification times, ownership and the mode.
-f Overwrites the destination if it already exists.
-l Allow DataNode to lazily persist the file to disk. Forces
  replication factor of 1. This flag will result in reduced
  durability. Use with care.
```

5. TRABAJANDO CON MAPREDUCE

En esta sección vamos a explorar dos formas de crear y ejecutar tareas MapReduce usando Python como lenguaje de programación. La primera opción es usar la herramienta Hadoop Streaming; la segunda es emplear una de las múltiples librerías de Python que sirven de interfaz para trabajar con Hadoop. Nosotros utilizaremos la librería MRJob.

5.1. HADOOP STREAMING

Hadoop Streaming es una herramienta que viene incluida de serie con la distribución de Hadoop. Su utilidad principal es que permite crear trabajos MapReduce empleando cualquier tipo de script o programa ejecutable para los procesos mapper y reducer, sin importar el lenguaje de programación.

Lo único que deben cumplir los scripts usados es que lean los datos de entrada usando su entrada estándar, línea a línea, y que escriban los resultados por la salida estándar, también línea a línea.

Al lanzar un trabajo usando Hadoop Streaming, se reservan en el cluster los recursos para crear las instancias necesarias de los procesos *mapper* y *reducer*, que ejecutarán los scripts o programas indicados por el usuario para cada tarea.

5.1.1. CONTANDO PALABRAS CON HADOOP STREAMING

Vamos a mostrar como implementaríamos el ejemplo básico de contar cuántas veces aparece cada palabra en una serie de ficheros de texto, usando Hadoop Streaming y Python.

Puedes acceder al contenido de este y los ejemplos siguientes en la máquina virtual, abriendo la carpeta personal de bigdata que aparece en el escritorio, y navegando por las carpetas Curso → U12_Hadoop. Aquí verás una carpeta datos que contiene los ficheros de datos que usaremos en los ejemplos, y otra carpeta src donde encontrarás los ficheros con el código Python de cada ejemplo.

El código de este primer ejemplo lo tienes en `Curso/U12_Hadoop/src/e01`.

Veamos el código para el *mapper*.

`Curso/U12_Hadoop/src/e01/U12_hadoop_streaming_mapper.py`

```
#!/opt/anaconda3/bin/python
# -*- coding: utf-8 -*-

import sys
"""
Ejemplo de proceso *mapper*
para ejecutar con Hadoop Streaming
"""

# Leer línea a línea de la entrada estándar
for linea in sys.stdin:
    # Troceamos la línea por los espacios en blanco
    # para obtener una lista de palabras
    palabras = linea.split()
    # Iterar cada palabra
    for palabra in palabras:
        # Y generar una pareja clave-valor que será la entrada del Reducer.
        # La clave es la palabra y el valor es la unidad (1).
        #
        # El formato para la salida es simplemente escribir en cada línea
        # primero la clave (palabra) y después el valor,
        # separados por un tabulador.
        print('{0}\t{1}'.format(palabra, 1))
```

Como ves, nuestro *mapper* comienza con el bucle para leer línea a línea de la entrada estándar. Cada línea se trocea por los espacios en blanco para obtener las palabras. Finalmente, cada una de estas palabras se vuelca por la salida estándar. Cada línea de salida que imprime el *mapper* debe contener primero la *clave* (la palabra en este caso), y después el *valor* asignado (en este ejemplo, simplemente el valor 1 como contador de una ocurrencia más). Clave y valor deben ir separados por un tabulador. Si no usamos un tabulador, los procesos que van después del *mapper* entenderán que todo el contenido de la línea es la clave y que el valor es nulo.

Ahora veamos el código del *reducer*.

Curso/U12_Hadoop/src/e01/U12_hadoop_streaming_reducer.py

```
#!/opt/anaconda3/bin/python
# -*- coding: utf-8 -*-

import sys
"""
Ejemplo de proceso *reducer*
para ejecutar con Hadoop Streaming
"""

# Diccionario para almacenar el conteo de cada palabra
conteo_palabras = {}

# Leer línea a línea de la entrada estándar
for linea in sys.stdin:
    # Dividir la línea por el tabulador
    # para obtener la clave (palabra) y el valor (conteo)
    palabra, conteo = linea.split('\t')
    # La entrada es una cadena de texto
    # tenemos que convertir el conteo a un valor numérico
    conteo = int(conteo)
    # Comprobamos si la nueva palabra ya la habíamos visto antes
    # (está ya incluida en nuestro diccionario)
    if palabra in conteo_palabras:
        # y actualizamos el contador
        conteo_palabras[palabra] = conteo_palabras[palabra] + conteo
    else:
        # si no estaba, la añadimos con el conteo que hemos leído
        conteo_palabras[palabra] = conteo

# Cuando ya hemos procesado todas las palabras de entrada
# imprimimos los conteos finales como resultado
for palabra in sorted(conteo_palabras.keys()):
    print("{0}\t{1}".format(palabra, conteo_palabras[palabra]))
```


¿Qué hace nuestro *reducer*? De entrada, definimos una variable de tipo *diccionario* (**conteo_palabras**) donde almacenaremos para cada palabra el número acumulado de veces que aparece. A partir de ahí, empezamos a leer línea a línea en un bucle. Cada línea viene del *mapper* con un par clave-valor separados por un tabulador. Separamos estas clave (**palabra**) y valor (**conteo**) y comprobamos si ya había aparecido la palabra antes.

Si la palabra está en el diccionario, aumentamos el conteo. Si no, simplemente la añadimos. Cuando hemos terminado de procesar las líneas, volcamos los conteos de cada palabra acumulados en el diccionario.

No es un código muy complicado, ¿verdad? Ahora solo queda lanzar el trabajo en Hadoop. En el escritorio de la máquina virtual, haz doble click en el Terminal de MATE para abrir una nueva consola de texto interactiva. Introduce el siguiente comando para moverte al directorio con el contenido de esta unidad.

```
$ cd /home/bigdata/Curso/U12_Hadoop
```

Ahora lanzaremos la utilidad Hadoop Streaming utilizando el comando general **hadoop**. La sintaxis es la siguiente:

```
hadoop jar <RUTA_HASTA_LIBRERIAS_hadoop-streaming.jar> [opciones]
```

Esta tabla muestra la lista de opciones disponibles

Opción	Descripción
-file	Fichero a copiar al <i>cluster</i> Hadoop (típicamente los ficheros ejecutables del mapper y el reducer)
-mapper	Comando a ejecutar como mapper
-reducer	Comando a ejecutar como reducer
-input	Ruta de los ficheros de datos de entrada en HDFS
-output	Ruta del directorio de salida en HDFS donde almacenar los resultados

Para ejecutar nuestro ejemplo en la máquina virtual, introducimos el siguiente comando en el terminal (asegúrate de haber hecho el **cd** anterior para estar en el directorio correcto):

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \  
-file ./src/e01/U12_hadoop_streaming_mapper.py \  
-file ./src/e01/U12_hadoop_streaming_reducer.py \  
-mapper U12_hadoop_streaming_mapper.py \  
-reducer U12_hadoop_streaming_reducer.py \  
-input /user/bigdata/U12/datos/*.txt \  
-output /user/bigdata/U12/output_01
```

Como datos de entrada vamos a utilizar varios ficheros de texto almacenados en el subdirectorio **datos**. Estos ficheros son simplemente versiones en texto plano de varios temas de este curso.

Al lanzar la ejecución verás un montón de mensajes generados por los distintos procesos de Hadoop conforme van avanzando paso a paso.

Si todo ha ido bien, una vez que ha terminado el trabajo debería haber un nuevo directorio (**output_01**) creado en HDFS, conteniendo los ficheros resultado. Comprobamos que se ha creado el nuevo directorio y los ficheros:

```
$ hdfs dfs -ls /user/bigdata/U12/output_01
```

```
Found 2 items  
-rw-r--r--  1 bigdata bigdata          0 2018-02-26 17:37  
/user/bigdata/U12/output_01/_SUCCESS  
-rw-r--r--  1 bigdata bigdata      63449 2018-02-26 17:37  
/user/bigdata/U12/output_01/part-00000
```

El primer fichero (**_SUCCESS**) es simplemente una forma de indicar que el trabajo terminó correctamente. El segundo fichero (**part-00000**) es el que contiene los resultados.

Vamos a examinar las primeras líneas de este fichero de resultados para ver el contenido generado:

```
$ hdfs dfs -cat /user/bigdata/U12/output_01/part-00000 | head
```

```
" ,")}) 4  
"An      4  
"Big     3  
"Set1") 1  
"Splines", 1  
"Splines". 1  
"U"      3  
"backward") 2  
"backward". 1  
"caja    1
```

Vemos que en cada línea aparece una secuencia de caracteres (la clave), seguida del número de ocurrencias en los ficheros de entrada.

Podemos descargarnos el fichero de HDFS al sistema local para inspeccionar mejor el contenido completo

```
$ hdfs dfs -get /user/bigdata/U12/output_01/part-00000  
./output/output_01.tab
```

Si ahora abres la carpeta personal de bigdata y navegas hasta la carpeta `./Curso/U12_Hadoop/output` deberías encontrar el fichero `output_01.tab`. Si haces doble-click sobre él, se abrirá en un editor de texto básico donde podrás ver el contenido completo.

Como ves, cada línea corresponde a cada una de las cadenas de texto separadas por espacios en blanco en los ficheros originales, seguida del número de veces que aparecen.

¿Tras ver los resultados, se te ocurre algo a mejorar? Seguro que sí. La idea era contar las veces que aparece cada palabra. Sin embargo, en la salida vemos también números, secuencias de palabras mezcladas con signos de puntuación, palabras que aparecen unas veces en minúsculas y otras en mayúsculas...

5.1.2. UNA VARIANTE REFINADA

Vamos a intentar ajustar estos detalles con una nueva versión de nuestra función `mapper`. El código de esta segunda versión está en `Curso/U12_Hadoop/src/e02`

Veamos el código del nuevo `mapper`.

`Curso/U12_Hadoop/src/e02/U12_hadoop_streaming_mapper.py`

```
#!/opt/anaconda3/bin/python
# -*- coding: utf-8 -*-

import sys
import re
"""
Ejemplo de proceso *mapper*
para ejecutar con Hadoop Streaming
"""

# Leer línea a línea de la entrada estándar
for linea in sys.stdin:
    # Troceamos la línea usando una expresión regular:
    # [ ] indica un conjunto de caracteres
    # ^ aquí significa "excepto"
    # \W representa "caracteres no alfanuméricos" (espacios, puntuación...)
    # \d representa "caracteres numéricos"
    # + significa "1 o más caracteres"
    # La expresión regular significa
    # cualquier secuencia de caracteres que no contenga
    # números ni espacios ni signos de puntuación,
    # es decir, palabras solo con letras
    palabras = re.findall("[^\W\d]+", linea)
    # Iterar cada palabra
    for palabra in palabras:
        # Y generar una pareja clave-valor que será la entrada del Reducer.
        # La clave es la palabra y el valor es la unidad (1).
        #
        # El formato para la salida es simplemente escribir en cada línea
        # primero la clave (palabra) y después el valor,
        # separados por un tabulador.
        # Convertimos los caracteres a minúsculas con lower()
        print('{0}\t{1}'.format(palabra.lower(), 1))
```

Hay dos diferencias respecto al código del primer `mapper`. ¿Las identificas?

En primer lugar, estamos usando una expresión regular con `re.findall()` para extraer las palabras de cada línea, en lugar de hacer un `split()`. Una expresión regular nos permite definir un patrón de caracteres a buscar o extraer dentro de una cadena de texto. En este caso usamos una expresión regular que “captura” secuencias de caracteres alfabéticos, descartando números, espacios o signos de puntuación.

La segunda modificación está en la última línea. Al imprimir los pares clave-valor pasamos la palabra a minúsculas con `lower()`, de forma que no haya distintas versiones de la misma palabra contadas por separado.

En el proceso *reducer* no tenemos que introducir ningún cambio.

Para ejecutar esta nueva versión, introducimos el siguiente comando en el terminal de texto:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \
-file ./src/e02/U12_hadoop_streaming_mapper.py \
-file ./src/e02/U12_hadoop_streaming_reducer.py \
-mapper U12_hadoop_streaming_mapper.py \
-reducer U12_hadoop_streaming_reducer.py \
-input /user/bigdata/U12/datos/*.txt \
-output /user/bigdata/U12/output_02
```

Una vez que ha terminado podemos examinar el fichero resultado generado en HDFS:

```
$ hdfs dfs -cat /user/bigdata/U12/output_02/part-00000 | head
```

```
a      317
aa      1
abarca 1
abordar1
abrir  1
abrupta2
abs     1
absorvida 1
abstracción 1
acabamos 1
```

Nos podemos traer también una copia de los resultados al sistema de ficheros local y examinarlos en más detalle:

```
$ hdfs dfs -get /user/bigdata/U12/output_02/part-00000  
./output/output_02.tab
```

Como puedes comprobar, ahora ya solo tenemos los conteos acumulados de palabras de texto.

5.2. MRJOB

Alternativamente a Hadoop Streaming, existen varias librerías nativas Python que permiten interactuar con Hadoop y crear trabajos MapReduce de forma sencilla.

MRJob es una librería creada originalmente por Yelp para poder utilizar Python en la programación de varias tareas MapReduce necesarias para su aplicación (yelp.com). La idea era tener una API de clases y funciones sencilla y potente, que simplificara el desarrollo en Python.

Por defecto, la librería MRJob no viene instalada en la distribución estándar de Python, y tampoco en la distribución de Anaconda. Para facilitarte las cosas, en la versión de Python que usamos en la máquina virtual ya hemos instalado la librería por ti. Pero si haces pruebas en otro entorno, deberás instalarla manualmente usando:

```
$ pip install mrjob  
o  
$ conda install -c conda-forge mrjob
```

5.2.1. CONTANDO PALABRAS CON MRJOB

Para ver cómo se implementa un trabajo MapReduce utilizando MRJob, vamos a seguir el mismo ejemplo del conteo de palabras en ficheros.

Dentro de la carpeta Curso/U12_Hadoop/src/e03 en la máquina virtual tienes el script con nuestra nueva versión del algoritmo. A continuación puedes ver el código fuente.

/home/bigdata/Curso/U12_Hadoop/src/e03/U12_hadoop_mrjob_mapreduce.py

```
#!/opt/anaconda3/bin/python
# -*- coding: utf-8 -*-

from mrjob.job import MRJob
from mrjob.protocol import RawProtocol
import re

"""
Ejemplo sencillo de MapReduce
usando la librería MRJob
"""

class MRConteoPalabras(MRJob):
    """
    Definimos una nueva clase MRConteoPalabras
    que extiende a la clase MRJob de la librería
    """

    OUTPUT_PROTOCOL = RawProtocol

    def mapper(self, _, linea):
        """
        Implementamos nuestra versión
        del método mapper() de MRJob
        """

        # Dividir la línea en palabras
        # usando una expresión regular
        palabras = re.findall("[^\\W\\d]+", linea)
        # Para cada palabra
        for palabra in palabras:
            # generar una nueva tupla (clave, valor)
            yield(palabra.lower(), 1)

    def reducer(self, palabra, conteo):
        """
        Implementamos nuestra versión
        del método reducer de MRJob
        """

        # generar tuplas (clave, valor)
        # donde la clave es la 'palabra'
        # y el valor es la suma de valores
        # del iterador 'conteo'
        suma_conteos = sum((int(n) for n in conteo))
        yield(palabra, str(suma_conteos))

if __name__ == '__main__':
    # Función principal del programa
    MRConteoPalabras.run()
```

Para definir un trabajo MapReduce debemos crear una nueva clase (**MRConteoPalabras** en este caso) que extienda o herede de la clase **MRJob** de la librería. Esta clase contiene los métodos básicos correspondientes a los pasos de un trabajo MapReduce, incluyendo los métodos **mapper()**, **combiner()** y **reducer()**. Nuestra clase derivada de **MRJob** simplemente tiene que proporcionar una nueva implementación de los métodos que haga falta modificar. Los métodos que no sean redefinidos mantendrán el comportamiento por defecto.

El método **mapper()** tiene como argumentos una clave y un valor de entrada. Normalmente, la clave de entrada al *mapper* será **None** y el valor de entrada será una línea de texto.



No confundas la clave y el valor de entrada de estos métodos (**mapper**, **combiner**, **reducer**) con la clave y el valor de salida.

En nuestra implementación de **mapper()**, ignoramos el argumento correspondiente a la clave utilizando el símbolo de subrayado '''. Esta es la forma que tenemos en Python de indicar que un parámetro o un resultado no nos interesa y queremos ignorarlo. Al argumento correspondiente al valor lo hemos llamado *linea*. El **mapper()** debe devolver tuplas de la forma (*clave de salida*, *valor de salida*). Nuestro **mapper()** trocea la línea (clave de entrada) en palabras usando la expresión regular que ya vimos antes, y genera como salida pares con cada palabra como clave y un contador 1 como valor de salida.

**Atención**

Para generar estas tuplas de salida utilizamos el comando de Python **yield**. Una expresión **yield** es similar a un **return**, pero en lugar de devolver un resultado y terminar la ejecución de la función, la convierte en un generador. Es decir, podemos utilizar una función que devuelva valores con **yield** igual que usaríamos otra expresión generadora, iterando sobre los valores devueltos hasta agotarlos. Con **yield**, la función genera un nuevo resultado (un par clave-valor en este caso) y devuelve temporalmente el control fuera de la función, conservando internamente cuál es el siguiente valor que debe devolver.

Por otro lado, el método **reducer()** tiene como argumentos de entrada una clave y un objeto de tipo **iterator** que proporciona la secuencia de todos los valores generados para dicha clave en los pasos anteriores. En nuestra implementación de **reducer()**, el parámetro **palabra** representa la clave de entrada y en el parámetro **conteo** tendremos el iterador sobre la secuencia de valores obtenidos para la clave.

El método **reducer()** debe generar también tuplas de la forma (*clave resultado*, *valor resultado*). En nuestra implementación, el valor resultado es la suma de conteos generados previamente por el **mapper()**. Para poder sumar estos valores debemos hacer una pequeña transformación previa. Los valores contenidos en el iterador **conteo** están almacenados como texto, así que en la operación **sum()** generamos una nueva secuencia convirtiendo los valores de conteo a números enteros. Tras esto, ya podemos generar la tupla resultado usando **yield**.

Además de los métodos **mapper()** y **reducer()**, podemos definir también nuestra versión del método **combiner()**. Este método se ejecuta entre el *mapper* y el *reducer*, recibiendo la salida del primero y enviando resultados hacia el segundo. Por tanto la entrada y la salida son igualmente tuplas del tipo (*clave*, *valor*). Como ya te explicamos anteriormente, el uso de **combiner()** exige tener ciertos cuidados y asegurarse de la corrección de los cálculos para evitar resultados incoherentes.

Para ejecutar un trabajo en Hadoop implementado con MRJob simplemente tenemos que lanzar el script Python. La librería MRJob se encarga de toda la

interacción con Hadoop por nosotros. Simplemente tenemos que indicar una serie de parámetros:

```
$ python <script_tarea_mrjob.py> -r <tipo_ejecución> [ficheros_entrada] [--output_dir directorio_salida]
```

Naturalmente, lo primero es indicar cuál es el fichero con el código implementando nuestra tarea MapReduce con la librería MRJob. Después debemos especificar dónde vamos a ejecutar el trabajo. Las opciones son estas:

Opción	Descripción
<code>-r inline</code>	Ejecutar la tarea como un proceso Python normal en nuestra máquina actual (ejecución local, sin interacción con <i>cluster</i>). Útil para hacer pruebas del código.
<code>-r local</code>	Ejecución local (sin interacción con <i>cluster</i>), pero lanzando varios subprocesos en la máquina actual. Útil para probar la corrección del código simulando procesos en paralelo, sin necesidad de un <i>cluster</i> Hadoop.
<code>-r hadoop</code>	Ejecutar en un <i>cluster</i> Hadoop.
<code>-r emr</code>	Ejecutar en la plataforma <i>cloud</i> de Amazon Elastic Map Reduce (EMR).

Como ves, hay diferentes alternativas para lanzar la ejecución de nuestra tarea, cada una más adecuada a un propósito. Para ejecutar sobre Hadoop usaremos `-r hadoop`.

Finalmente, hay que indicar dónde están los ficheros de entrada (ya sea en local o en HDFS) y en qué directorio guardar los resultados.

En realidad, MRJob admite un gran número de opciones de configuración, pero estas son las básicas para lanzar un trabajo. Puedes consultar una lista completa en la guía de la librería (<https://pythonhosted.org/mrjob/guides/configs-reference.html>).

Veamos el comando completo para ejecutar nuestro ejemplo con MRJob desde el terminal de MATE:

```
$ python src/e03/U12_hadoop_mrjob_mapreduce.py \
-r hadoop hdfs:///user/bigdata/U12/datos/*.txt \
--output-dir hdfs:///user/bigdata/U12/output_03
```

Si pruebas a lanzarlo, una vez que se ha completado puedes examinar el resultado

```
$ hdfs dfs -cat /user/bigdata/U12/output_03/part-00000 | head
```

```
ágil 1
álgebra2
ámbito 1
ángel 1
ánimo 1
árbol 1
árboles3
él 2
éxito 6
óptima 2
```

Puedes descargarte el fichero de HDFS para examinarlo

```
$ hdfs dfs -get /user/bigdata/U12/output_03/part-00000 out-
put/output_03.tab
```

Si comparas los ficheros de salida de la versión 2 (**output_02.tab**) y esta versión usando MRJob (**output_03.tab**), comprobarás que los conteos son los mismos. La diferencia está en cómo se han ordenado las palabras. En la última versión, las palabras con acentos aparecen antes que las palabras sin acentos. Mientras que en la versión 2 las palabras aparecen ordenadas alfabéticamente, independientemente de los acentos.

5.2.2. MÁS ALLÁ DE UN SIMPLE MAP-REDUCE

Hasta ahora hemos visto cómo implementar una tarea muy básica, consistente en un flujo sencillo con una etapa Map y una etapa Reduce.

Sin embargo, nuestro flujo de procesos puede ser más complejo, y requerir varias etapas o cálculos intermedios para calcular el resultado final. También es posible que necesitemos realizar algún tipo de inicialización antes de empezar a procesar datos, o un postproceso tras completar los cálculos.

Con la librería MRJob, definir un flujo con varias etapas extra es muy fácil, utilizando la clase **MRStep**.

Veamos una variante del conteo de palabras, en la que terminamos calculando cuál es la palabra más frecuente como único resultado final.

Curso/U12_Hadoop/src/e04/U12_hadoop_mrjob_max_word_count.py

```
#!/opt/anaconda3/bin/python
# -*- coding: utf-8 -*-

from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.protocol import RawProtocol
import re

Extraer la palabra más usada
con MapReduce y la librería MRJob
"""

class MRPalabraMasUsada(MRJob):
    """
    Tarea MapReduce
    definiendo varios pasos para contar palabras
    y sacar después la más frecuente
    """

    def mapper_extrae_palabras(self, _, linea):
        # Dividir la línea en palabras
        # usando una expresión regular
        palabras = re.findall("[^\W\d]+", linea)
        # Para cada palabra
        for palabra in palabras:
            # generar una nueva tupla (clave, valor)
            yield (palabra.lower(), 1)

    def combiner_cuenta_palabras(self, palabra, conteo):
        # Sumar el número de veces
        # que ha aparecido cada palabra
        yield (palabra, sum(conteo))

    def reducer_cuenta_palabras(self, palabra, conteo):
        # Reducer intermedio
        # Genera como valor tuplas (número de ocurrencias, palabra)
        # y como clave asocia siempre None.
        # Al fijar la misma clave, toda la salida de estos reducer
        # irá a un mismo proceso reducer final.
        yield None, (sum(conteo), palabra)
```

```

def reducer_palabra_mas_usada(self, _, palabra_conteo):
    # Reducer final
    # El iterator de valores (palabra_conteo)
    # contiene tuplas de la forma (número de ocurrencias, palabra)
    # La clave recibida siempre es None, así que la descartamos
    # Aprovechamos que max() también funciona con tuplas
    # para sacar cuál es la tupla (num.ocurrencias, palabra)
    # más alta, y la devolvemos como resultado final
    yield max(palabra_conteo)

def steps(self):
    # Método de la librería mrjob
    # que permite definir manualmente pasos de cálculo
    # más complejos que una secuencia Map-Reduce simple
    return [
        MRStep(mapper=self.mapper_extrae_palabras,
               combiner=self.combiner_cuenta_palabras,
               reducer=self.reducer_cuenta_palabras),
        MRStep(reducer=self.reducer_palabra_mas_usada)
    ]

if __name__ == '__main__':
    MRPalabraMasUsada.run()

```

Repasemos el código para entender qué estamos haciendo. Esta vez empezamos por el último método de la clase `MRPalabraMasUsada`. El método `steps()` sirve para definir una secuencia de cálculos más compleja, formada a su vez por procesos MapReduce que se ejecutan en cadena, en un orden concreto.

Para definir un paso individual en la secuencia definimos objetos de la clase `MRStep`. La clase `MRStep` nos permite especificar qué método usar en cada una de las etapas internas de un flujo MapReduce. Eso incluye no solo definir el *mapper* y el *reducer*. También podemos definir un *combiner*, e incluso métodos de inicialización y finalización previos al *mapper*, *reducer* o *combiner*. Los métodos de inicialización sirven típicamente para definir y configurar variables auxiliares donde acumular datos de entrada. Mientras, los métodos de finalización se usan para completar el procesamiento de datos acumulados tras todo el proceso o limpiar variables. Puedes ver todas las opciones para configurar un paso de cálculo en <https://pythonhosted.org/mrjob/step.html#mrjob.step.MRStep>.

En nuestro ejemplo, dentro del método `steps()` definimos una secuencia (lista) con dos pasos (objetos `MRStep`). La secuencia completa sería:

■ **Paso 1:**

- **Mapper:** ejecutar el método `mapper_extrae_palabras` que simplemente divide cada línea en palabras y genera pares (*palabra, 1*) como en los ejemplos anteriores.
- **Combiner:** ejecutar el método `combiner_cuenta_palabras` que genera conteos agregados parciales de las palabras devueltas por el Mapper de su mismo nodo.
- **Reducer:** ejecutar el método `reducer_cuenta_palabras` que genera pares (*clave, valor*) intermedios, en los que la clave es siempre `None` y el valor es una tupla con la forma (*número de ocurrencias, palabra*).

■ **Paso 2:**

- **Mapper:** nada, redirigir la entrada directamente al proceso siguiente.
- **Combiner:** nada, redirigir la entrada directamente al proceso siguiente.
- **Reducer:** ejecutar el método `reducer_palabra_mas_usada` que recibe como valor un iterador sobre los valores generados previamente por `reducer_cuenta_palabras` (las tuplas (*número de ocurrencias, palabra*)), y calcula cuál es el máximo de estas tuplas (devuelve la tupla con mayor número de ocurrencias).

El resultado final es la tupla formada por la palabra que más veces aparece en los ficheros de entrada.

Algunos detalles técnicos

Aparte de los dos pasos que utilizamos en este ejemplo, ¿hay algún detalle que te haya llamado la atención? Fíjate en el código. Nos estamos aprovechando de dos características de MapReduce que nos permiten mayor flexibilidad al programar nuestras tareas. Revisémoslas.

La primera es que no es necesario implementar *todas* las etapas de un proceso MapReduce: *mapper*, *combiner*, *reducer*... En los ejemplos anteriores ya ignorábamos la etapa del *combiner*. En este caso, en el segundo paso hemos decidido omitir el *mapper* y el *combiner*. Cuando omitimos una etapa, el comportamiento es como si pasáramos directamente los pares (*clave*, *valor*) de entrada a la salida, rediriéndolos sin más a la siguiente etapa de MapReduce.

La segunda característica es algo que muchas veces se pasa por alto. Y es que el contenido devuelto en los pares (*clave*, *valor*) puede ser *cualquier cosa*. No tienen por qué ser un tipo simple (un entero o una cadena de texto), especialmente en el caso del valor. Pueden ser una secuencia o un objeto más complejo. Las únicas condiciones que deben cumplir son:

- La clave debe ser de un tipo *serializable* y *comparable*.
- El valor debe ser un tipo *serializable*.

Muy bien, pero ahora tal vez te estés preguntando ¿qué es un tipo *serializable*?. El proceso de *serialización* consiste en tomar de una estructura de datos o un objeto y generar una representación en un formato que pueda ser guardado en fichero o enviado por red, para reconstruir la estructura de datos más adelante cuando sea necesario (a este proceso de reconstrucción lo denominamos *deserialización*). Por tanto, un tipo es *serializable* si puede ser transformado de esta manera.

La buena noticia es que en Python todos los tipos nativos (números enteros y en coma flotante, caracteres, booleanos) y los tipos compuestos (listas, tuplas, conjuntos, diccionarios) son *serializables* directamente y no hay que hacer nada especial.

¿Y los tipos *comparables*? Esto es más sencillo. Un tipo es *comparable* si, dados dos elementos *a* y *b* de ese tipo, podemos decir si *a* es menor, igual o mayor que *b*. Esto es un requisito para las claves en Hadoop, ya que la reordenación, reparto y agregación dependen de esta característica.

De nuevo, la buena noticia es que en Python tanto los tipos nativos como los compuestos son comparables. ¿Y cómo se comparan dos listas o dos tuplas? Bueno, hagamos la prueba.

```
(1, 2) < (1, 1)
```

```
False
```

```
['a', 'b'] > ['a', 'a']
```

```
True
```

Esencialmente, en un tipo compuesto se va comparando elemento a elemento hasta decidir cuál va antes que el otro. Prueba tú mismo con otros casos que se te ocurran.

Volviendo a nuestro ejemplo, en el método `reducer_cuenta_palabras` devolvemos como valores unas tuplas con la forma (*número de ocurrencias, palabra*). Mientras que como clave usamos `None` para todos los resultados. ¿Y por qué usar la misma clave y por qué `None`? En primer lugar, porque al usar una misma clave, todos los pares (*clave, valor*) se repartirán hacia un mismo reducer. Y en segundo lugar, no vamos a utilizar la clave para nada más en los pasos siguientes, nos da igual qué contenga, así que `None` es un buen candidato.

Probando el ejemplo

¿Qué tal si tras todas estas explicaciones comprobamos cómo funciona? Lanchemos en el terminal de MATE el siguiente comando:

```
$ python src/e04/U12_hadoop_mrjob_max_word_count.py -r hadoop
hdfs:///user/bigdata/U12/datos/*.txt --output-dir
hdfs:///user/bigdata/U12/output_04
```

Si todo va bien, una vez que haya terminado deberíamos tener un fichero con el resultado en HDFS. Veamos su contenido.

```
$ hdfs dfs -cat /user/bigdata/U12/output_04/part-00000
1729    "de"
```

Según los cálculos, la palabra que más veces aparece en los ficheros de entrada es *"de"*, con un total de 1729 apariciones. Bueno, no es ninguna sorpresa, las palabras más comunes en cualquier texto son artículos, pronombres y preposiciones. Sería bueno filtrarlas, pero esto lo dejamos para uno de los ejercicios...

¿QUÉ HAS APRENDIDO?

A lo largo de esta unidad has visto cuál es el origen de Hadoop, cuáles fueron los motivos que impulsaron el desarrollo de esta tecnología y cómo evolucionó hasta convertirse en una referencia para el procesamiento Big Data.

También hemos visto cómo es su arquitectura, cuáles son sus principales componentes (el sistema de ficheros distribuido HDFS, el gestor de recursos YARN y el motor MapReduce) y qué papel juegan. Entender bien esto es muy útil ya que nos permite razonar sobre cómo van a ejecutarse nuestros programas y diseñarlos de una manera eficiente.

Por último, te hemos enseñado las operaciones fundamentales para trabajar con el sistema de ficheros distribuido HDFS. Y también has aprendido las bases para implementar tareas MapReduce en Python, usando la herramienta Hadoop Streaming o bien con la librería MRJob.

Pero tu camino no termina aquí. Apache Hadoop ofrece muchas más opciones de las que podemos explorar en una unidad. En este tema te hemos dado las bases para que puedas comenzar a hacer tus propios desarrollos y profundizar en el manejo de la plataforma.

Por otro lado, el modelo MapReduce también tiene algunas limitaciones que lo hacen poco apropiado para resolver ciertos problemas. En las siguientes unidades vamos a ver otras tecnologías Big Data como Spark, que resuelven estos problemas. ¿Continuamos?

AUTOCOMPROBACIÓN

1. **Cuando hablamos de Big Data para trabajar con enormes cantidades de datos, en general lo más eficiente es**
 - a) Copiar los datos a procesar a la máquina donde está el código a ejecutar
 - b) Copiar el código a ejecutar a las máquinas donde están los datos
 - c) Empaquetar datos y código y mandarlo a la máquina más potente
 - d) Trocear los datos y ejecutar secuencialmente el código sobre cada porción

2. **¿En qué casos el modelo de Hadoop es una buena solución?**
 - a) Para procesamiento por lotes, con cálculos y agregaciones sobre grandes cantidades de datos, donde no importa que el tiempo de respuesta sea elevado
 - b) Para procesamiento por lotes, con cálculos y agregaciones sobre grandes cantidades de datos, donde es importante que el tiempo de respuesta sea reducido
 - c) Para procesamiento en tiempo real, con cálculos y agregaciones sobre grandes cantidades de datos, donde es importante que el tiempo de respuesta sea reducido
 - d) Hadoop es una buena solución para todo tipo de problemas Big Data

3. ¿Permite HDFS modificar el contenido de un fichero?

- a) No permite modificar el contenido de un fichero una vez que lo hemos creado
- b) Permite modificar el contenido de un fichero, siempre que sepamos el bloque que hay que modificar
- c) Es posible modificar el contenido, pero puede suponer volver a trocearlo, repartir los bloques y sus réplicas, por lo que no es aconsejable
- d) Solo es posible si únicamente queremos añadir datos al final del fichero

4. ¿Cuáles son los principales recursos a gestionar por YARN?

- a) Los bloques de memoria RAM disponibles en los nodos de cálculo
- b) Los núcleos de computación (cores de CPU) y los bloques de memoria RAM del nodo gestor de recursos
- c) Los núcleos de computación (cores de CPU) y los bloques de memoria RAM de los nodos de cálculo
- d) El espacio de disco disponible para almacenar datos en HDFS

5. En un cluster YARN

- a) Hay un único ResourceManager activo en el nodo maestro, y un NodeManager activo en cada nodo de cálculo
- b) Hay un único ResourceManager activo en el nodo maestro, y varios NodeManagers activos (uno por cada aplicación) en cada nodo de cálculo
- c) Puede haber varios ResourceManagers activos en nodos distintos, y varios NodeManagers en un mismo nodo de cálculo
- d) Los NodeManagers deciden automáticamente entre sí cuál hace de ResourceManager de la plataforma

6. ¿A qué nos referimos con el término “colocalización” cuando hablamos de Hadoop?

- a) Consiste en procurar que los procesos de cálculo se lancen para ejecutarse de manera distribuida en las máquinas donde residen los bloques de datos a usar
- b) Consiste en que todos los nodos o máquinas que forman el cluster donde hay que ejecutar los cálculos distribuidos se encuentren todas cerca, en la misma subred
- c) Consiste en que todos los nodos o máquinas que forman el cluster para almacenar los datos se encuentren todas cerca, en la misma subred
- d) Consiste en que los nodos de datos y los nodos de cálculo se encuentren desplegados en máquinas distintas, conectadas por redes de alta velocidad.

7. ¿Cómo describirías el funcionamiento básico de MapReduce?

- a) Consiste en aplicar las funciones map y reduce de Python de forma distribuida sobre los datos de entrada, aprovechando los servicios de la plataforma Hadoop
- b) Consiste en aplicar una función a cada dato de entrada y después aplicar otra función a los resultados intermedios para combinarlos y reducirlos a un resultado final
- c) Consiste en aplicar una secuencia de funciones paralelizables a cada dato de entrada para después juntar los resultados
- d) Consiste en aplicar una función a los datos de entrada para repartirlos por el cluster y una segunda función para calcular los resultados de forma distribuida

8. La función Reduce

- a) Recibe como entrada una clave y una lista de valores asociados a esa clave, y devuelve uno o más pares clave-valor como resultado
- b) Recibe como entrada una clave y una lista de valores asociados a esa clave, y devuelve uno o más pares clave-valor, sin claves repetidas, como resultado
- c) Recibe como entrada una clave y una lista de valores asociados a esa clave, y devuelve uno o más valores como resultado

- d) Recibe como entrada una clave y una lista de valores asociados a esa clave, y devuelve una lista como resultado

9. ¿Qué hace el comando `hdfs dfs -ls`?

- a) Lee el contenido de un fichero almacenado en HDFS
- b) Limpia el contenido de un directorio almacenado en HDFS
- c) Lista los comandos disponibles desde línea de comandos
- d) Lista el contenido de un directorio almacenado en HDFS

10. ¿Qué es Hadoop Streaming?

- a) Es una herramienta de serie en Hadoop que permite lanzar trabajos MapReduce a partir de scripts creados por el usuario sin importar el lenguaje
- b) Es una herramienta de serie en Hadoop que permite lanzar trabajos MapReduce a partir de scripts en Python creados por el usuario
- c) Es una librería para procesar streams de datos en tiempo real usando Hadoop
- d) Es un intérprete de comandos para ejecutar funciones Map y Reduce definidas por el usuario

SOLUCIONARIO

1.	b	2.	a	3.	c	4.	c	5.	a
6.	a	7.	b	8.	b	9.	d	10.	a

BIBLIOGRAFÍA

- Página principal del proyecto:
<http://hadoop.apache.org/>
- Documentación oficial de Hadoop:
<https://hadoop.apache.org/docs/current/>
- Documentación oficial de HDFS:
<https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>
- Documentación oficial de MapReduce:
<https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- Documentación oficial de YARN:
<https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>
- Documentación de la librería mrjob:
<https://pythonhosted.org/mrjob/>
- "Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale".
Tom White. O'Reilly Media; 4 edition, 2015

