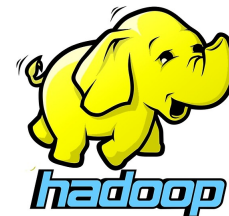


Hadoop

Si *Big Data* es la filosofía de trabajo para grandes volúmenes de datos, *Apache Hadoop* (<http://hadoop.apache.org/>) es la tecnología catalizadora. *Hadoop* puede escalar hasta miles de ordenadores creando un clúster con un almacenamiento del orden de *petabytes* de información.

Más que un producto, es un proyecto *open source* que aglutina una serie de herramientas para el procesamiento distribuido de grandes conjuntos de datos a través de clústers de ordenadores utilizando modelos de programación sencillos.



Logo de Apache Hadoop

Sus características son:

- **Confiable:** crea múltiples copias de los datos de manera automática y, en caso de fallo, vuelve a desplegar la lógica de procesamiento.
- **Tolerante a fallos:** tras detectar un fallo aplica una recuperación automática. Cuando un componente se recupera, vuelve a formar parte del clúster. En *Hadoop* los fallos de hardware se tratan como una regla, no como una excepción.
- **Escalable:** los datos y su procesamiento se distribuyen sobre un clúster de ordenadores (escalado horizontal), desde un único servidor a miles de máquinas, cada uno ofreciendo computación y almacenamiento local.
- **Portable:** se puede instalar en todo tipos de *hardware* y sistemas operativos.

En la actualidad se ha impuesto *Hadoop* v3 (la última versión a día de hoy es la 3.3.4), aunque todavía existe mucho código para *Hadoop* v2.

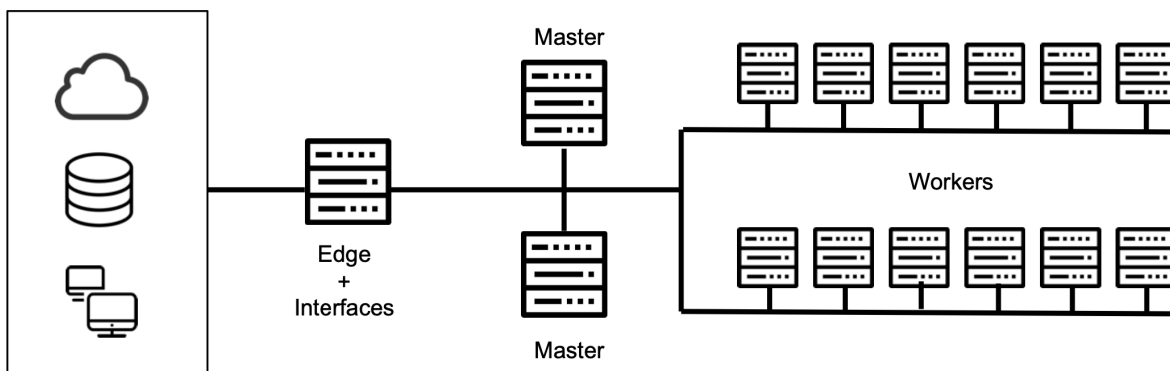
Procesamiento distribuido

Hadoop está diseñado para ejecutar sistemas de procesamiento en el mismo clúster que almacena los datos (*data local computing*). Su filosofía es almacenar todos los datos en un lugar y procesarlos en el mismo lugar, esto es, mover el procesamiento al almacén de datos y no mover

los datos al sistema de procesamiento.

Esto lo logra mediante un entorno distribuido de datos y procesos. El procesamiento se realiza en paralelo a través de nodos de datos en un sistema de ficheros distribuidos (HDFS), donde se distingue entre:

- Nodos maestros: encargados de los procesos de gestión global, es decir, controlar la ejecución o el almacenamiento de los trabajos y/o datos. Normalmente se necesitan 3. Su hardware tiene mayores requisitos.
- Nodos *workers*: tratan con los datos locales y los procesos de aplicación. Su número dependerá de las necesidades de nuestros sistemas, pero pueden estar comprendido entre 4 y 10.000. Su hardware es relativamente barato (*commodity hardware*) mediante servidores X86.
- Nodos *edge*: hacen de puente entre el clúster y la red exterior, y proporcionan interfaces.



Arquitectura hardware de Hadoop

Commodity Hardware

A veces el concepto *hardware commodity* suele confundirse con *hardware* doméstico, cuando lo que hace referencia es a hardware no específico, que no tiene unos requerimientos en cuanto a disponibilidad o resiliencia exigentes.

El *hardware* típico donde se ejecuta un cluster Hadoop es el siguiente:

- Nodos *worker*: 256 Gb RAM – 12 discos duros de 2-4 TB JBOD (*just a bunch of drives*) – 2 CPU x 6-8 cores.
- Nodos *master*: 256 Gb RAM – 2 discos duros de 2-3 TB en RAID – 2 CPU x 8 cores. En estos nodos es más importante la capacidad de la CPU que la de almacenamiento.
- Nodos *edge*: 256 Gb RAM – 2 discos duros de 2-3 TB en RAID – 2 CPU x 8 cores.

Cada vez que añadimos un nuevo nodo *worker*, aumentamos tanto la capacidad como el rendimiento de nuestro sistema.

Componentes y Ecosistema

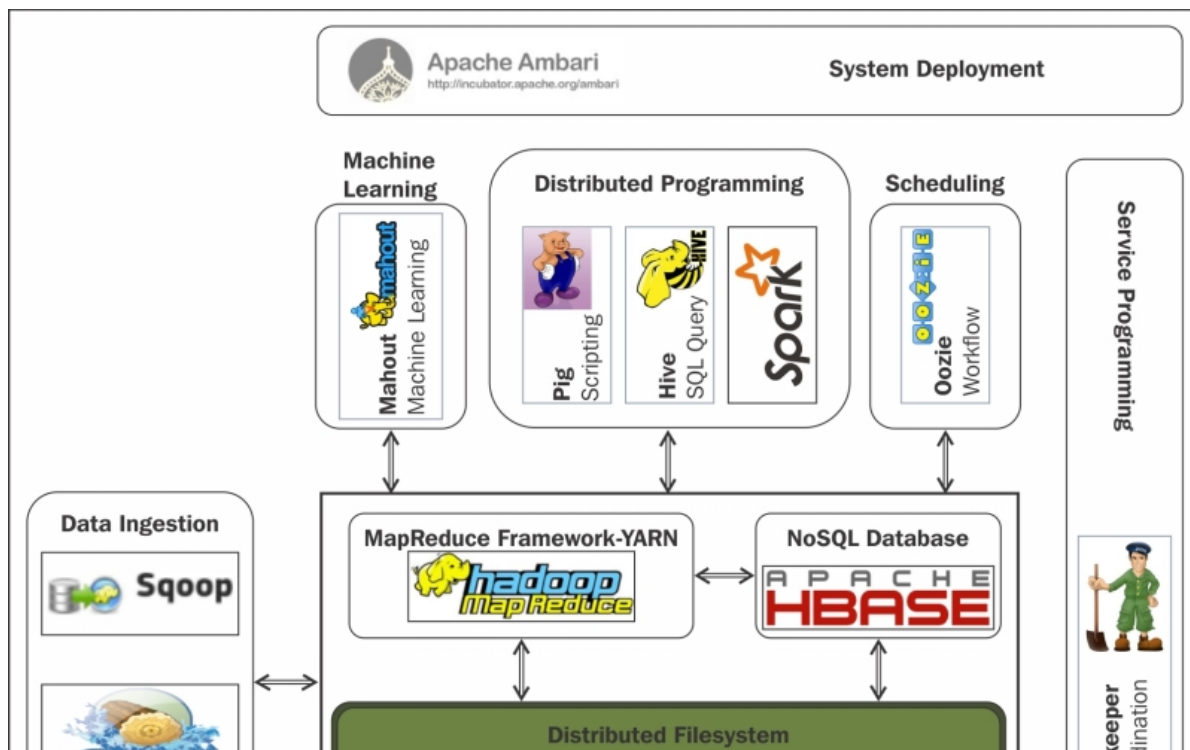
El núcleo de *Hadoop* se compone de:

- un conjunto de utilidades comunes (*Hadoop Common*)
- un sistema de ficheros distribuidos (*Hadoop Distributed File System* ↔ *HDFS*).
- un gestor de recursos para el manejo del clúster y la planificación de procesos (*YARN*)
- un sistema para procesamiento paralelo de grandes conjuntos de datos (*MapReduce*)

Estos elementos permiten trabajar casi de la misma forma que si tuviéramos un sistema de fichero locales en nuestro ordenador personal, pero realmente los datos están repartidos entre miles de servidores.

Las aplicaciones se desarrollan a alto nivel, sin tener constancia de las características de la red. De esta manera, los científicos de datos se centran en la analítica y no en la programación distribuida.

Sobre este conjunto de herramientas existe un ecosistema "infinito" con tecnologías que facilitan el acceso, gestión y extensión del propio Hadoop.





Ecosistema Hadoop

Las más utilizadas son:


- **Hive**: Permite acceder a HDFS como si fuera una Base de datos, ejecutando comandos muy parecido a SQL para recuperar valores (*HiveSQL*). Simplifica enormemente el desarrollo y la gestión con *Hadoop*.
- **HBase**: Es el sistema de almacenamiento *NoSQL* basado en columnas para *Hadoop*.
 - Es una base de datos de código abierto, distribuida y escalable para el almacenamiento de Big Data.
 - Escrita en Java, implementa y proporciona capacidades similares sobre *Hadoop* y HDFS.
 - El objetivo de este proyecto es el de trabajar con grandes tablas, de miles de millones de filas de millones de columnas, sobre un clúster *Hadoop*.
- **Pig**: Lenguaje de alto de nivel para analizar grandes volúmenes de datos. Trabaja en paralelo, lo que permite gestionar gran cantidad de información. Realmente es un compilador que genera comandos MapReduce, mediante el lenguaje textual denominado *Pig Latin*.
- **Sqoop**: Permite transferir un gran volumen de datos de manera eficiente entre *Hadoop* y sistemas gestores de base de datos relacionales.
- **Flume**: Servicio distribuido y altamente eficiente para distribuir, agregar y recolectar grandes cantidades de información. Es útil para cargar y mover información en *Hadoop*, como ficheros de logs, datos de Twitter/Reddit, etc. Utiliza una arquitectura de tipo *streaming* con un flujo de datos muy potente y personalizables
- **ZooKeeper**: Servicio para mantener la configuración, coordinación y aprovisionamiento de aplicaciones distribuidas. No sólo se utiliza en *Hadoop*, pero es muy útil en esa arquitectura, eliminando la complejidad de la gestión distribuida de la plataforma.
- **Spark**: Es un motor muy eficiente de procesamiento de datos a gran escala. Implementa procesamiento en tiempo real al contrario que *MapReduce*, lo que provoca que sea más rápido. Para ello, en vez de almacenar los datos en disco, trabaja de forma masiva en memoria. Puede trabajar de forma autónoma, sin necesidad de *Hadoop*.
- **Ambari**: Herramienta utilizada para instalar, configurar, mantener y monitorizar *Hadoop*.

Si queremos empezar a utilizar *Hadoop* y todo su ecosistema, disponemos de diversas distribuciones con toda la arquitectura, herramientas y configuración ya preparadas. Las más

reseñables son:

- [Amazon Elastic MapReduce \(EMR\)](#)  de AWS.
- [CDH](#)  de Cloudera
- [Azure HDInsight](#)  de Microsoft.
- [DataProc](#)  de Google.

HDFS

Es la capa de almacenamiento de *Hadoop*, y como tal, es un sistema de ficheros distribuido y tolerante a fallos que puede almacenar gran cantidad de datos, escalar de forma incremental y sobrevivir a fallos de hardware sin perder datos. Se basa en el [paper](#)  que publicó *Google* detallando su *Google File System* en 2003.

Es un sistema que reparte los datos entre todos los nodos del clúster de *Hadoop*, dividiendo los ficheros en bloques (cada bloque por defecto es de 128MB) y almacenando copias duplicadas a través de los nodos. Por defecto se replica en 3 nodos distintos (esto se conoce como el **factor de replicación**).

HDFS asegura que se puedan añadir servidores para incrementar el tamaño de almacenamiento de forma lineal, de manera que al introducir un nuevo nodo, se incrementa tanto la redundancia como la capacidad de almacenamiento.

Está planteado para escribir los datos una vez y leerlos muchos veces (*WORM / Write Once, Read Many*). Las escrituras se pueden realizar a mano, o desde herramientas como *Flume* y *Sqoop*, que estudiaremos más adelante.

No ofrece buen rendimiento para:

- Accesos de baja latencia. Realmente se utiliza para almacenar datos de entrada necesarios para procesos de computación.
- Ficheros pequeños (a menos que se agrupen). Funciona mejor con grandes cantidades de ficheros grandes, es decir, mejor millones de ficheros de 100MB que billones de ficheros de 1MB.
- Múltiples escritores.
- Modificaciones arbitrarias de ficheros.

Así pues, los datos, una vez escritos en HDFS son inmutables. Cada fichero de HDFS solo permite añadir contenido (*append-only*). Una vez se ha creado y escrito en él, solo podemos añadir

contenido o eliminarlo. Es decir, a priori, no podemos modificar los datos.



HBase / Hive

Tanto *HBase* como *Hive* ofrecen una capa por encima de HDFS para dar soporte a la modificación de los datos, como en cualquier base de datos.

Bloques

Un bloque es la cantidad mínima de datos que puede ser leída o escrita. El tamaño predeterminado de HDFS son 128 MB, ya que como hemos comentado, *Hadoop* está pensado para trabajar con ficheros de gran tamaño.

Todos los ficheros están divididos en bloques. Esto quiere decir que si subimos un fichero de 600MB, lo dividirá en 5 bloques de 128MB. Estos bloques se distribuyen por todos los nodos de datos del clúster de *Hadoop*.

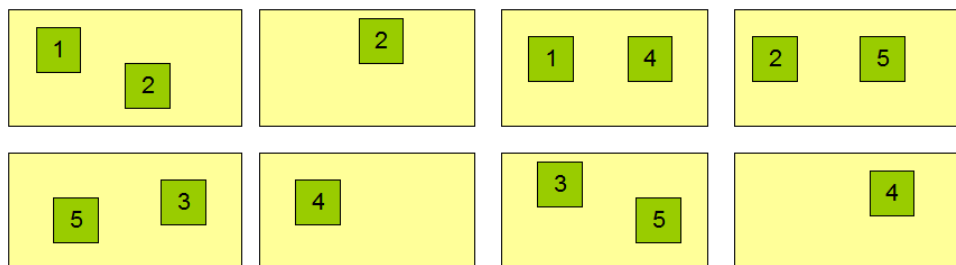
Si un fichero de HDFS es menor que el tamaño de un bloque, es decir, menor de 128MB, ocupará un bloque lógico pero en disco únicamente ocupará el espacio necesario. Es decir, un archivo de 1MB ocupará un bloque de 128MB, pero utilizará 1MB en disco.

A partir del *factor de replicación*, cada bloque se almacena varias veces en máquinas distintas. El valor por defecto es 3. Por lo tanto, el archivo de 600MB que teníamos dividido en 5 bloques de 128MB, si lo replicamos tres veces, lo tendremos repartido en 15 bloques entre todos los nodos del clúster.

Block Replication

```
Namenode (Filename, numReplicas, block-ids, ...)  
/users/sameerp/data/part-0, r:2, {1,3}, ...  
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
```

Datanodes



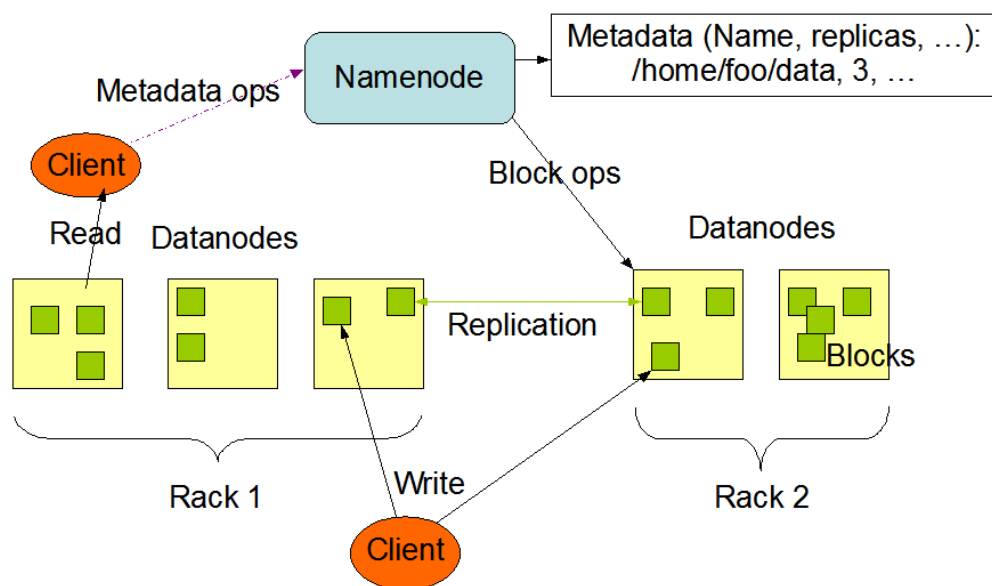
Factor de replicación HDFS

Respecto a los permisos de lectura y escritura de los ficheros, sigue la misma filosofía de asignación de usuarios y grupos que se realiza en los sistemas *Posix*. Es una buena práctica crear una carpeta `/user/` en el raíz de HDFS, de forma similar al `/home/` de Linux.

En HDFS se distinguen las siguientes máquinas:

- **Namenode:** Actúa como máster y almacena todos los metadatos necesarios para construir el sistema de ficheros a partir de sus bloques. Tiene control sobre dónde están todos los bloques.
- **Datanode:** Son los esclavos, se limitan a almacenar los bloques que compone cada fichero.
- **Secondary Namenode:** Su función principal es tomar puntos de control de los metadatos del sistema de archivos presentes en namenode.

HDFS Architecture



Arquitectura HDFS

En la siguiente sesión profundizaremos en la arquitectura de HDFS y cómo funcionan y gestionan los datos tanto el *namenode* como los *datanodes*.

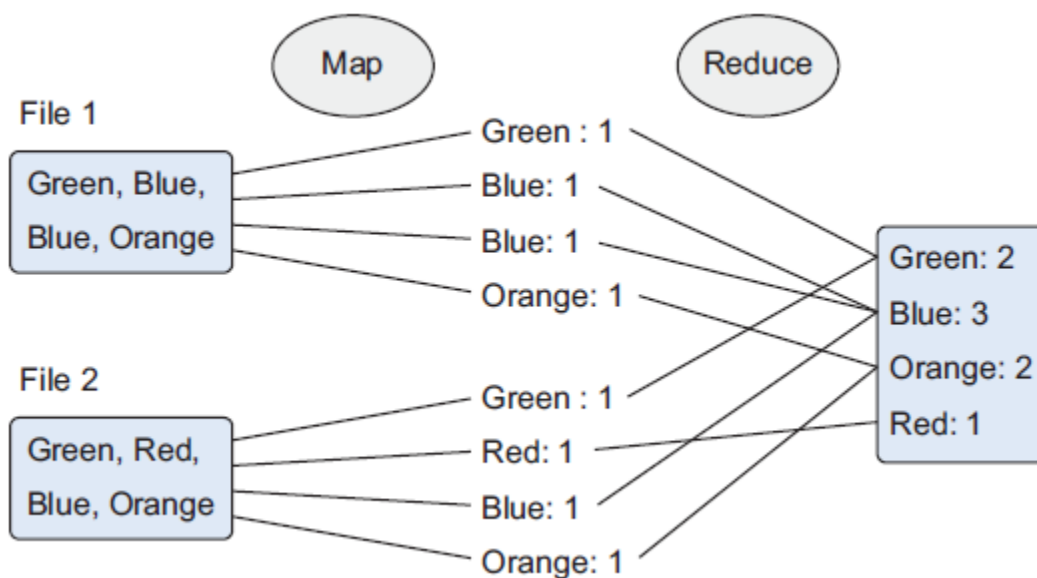
MapReduce

Se trata de un paradigma de programación funcional en dos fases, la de mapeo y la de reducción, y define el algoritmo que utiliza *Hadoop* para paralelizar las tareas. Un algoritmo MapReduce divide los datos, los procesa en paralelo, los reordena, combina y agrega de vuelta los resultados mediante un formato clave/valor.

Sin embargo, este algoritmo no casa bien con el análisis interactivo o programas iterativos, ya que persiste los datos en disco entre cada uno de los pasos del mismo, lo que con grandes *datasets* conlleva una penalización en el rendimiento.

Un **job** de *MapReduce* se compone de múltiples tareas *MapReduce*, donde la salida de una tarea es la entrada de la siguiente.

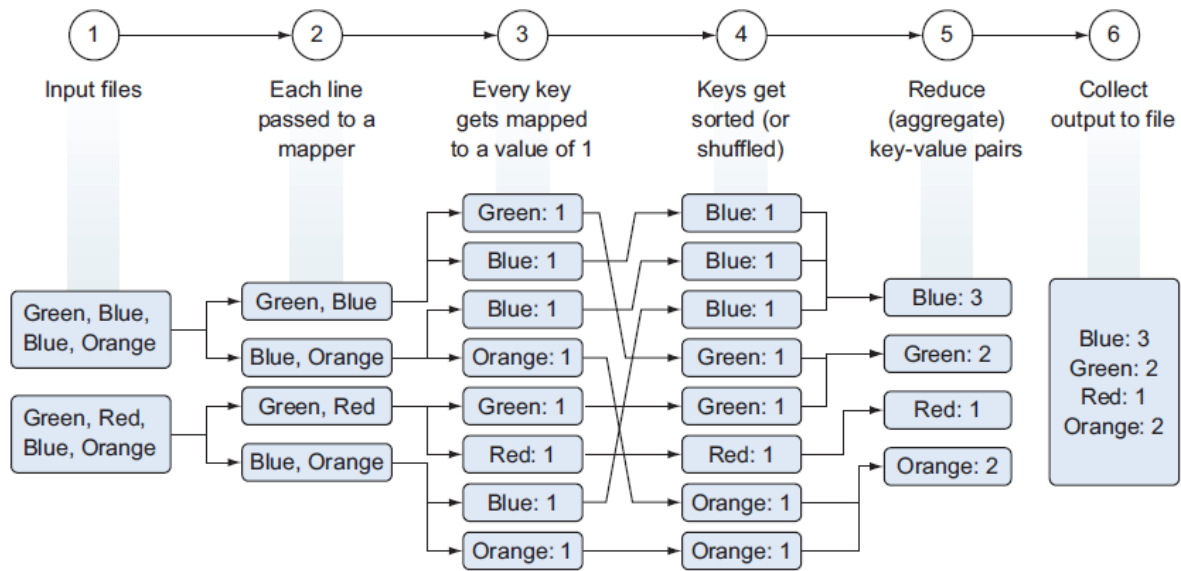
El siguiente gráfico muestra un ejemplo de una empresa que fabrica juguetes de colores. Cuando un cliente compra un juguete desde la página web, el pedido se almacena como un fichero en *Hadoop* con los colores de los juguetes adquiridos. Para averiguar cuantas unidades de cada color debe preparar la fábrica, se emplea un algoritmo *MapReduce* para contar los colores:



Como sugiere el nombre, el proceso se divide principalmente en dos fases:

- Fase de mapeo (*Map*) – Los documentos se parten en pares de clave/valor. Hasta que no se reduzca, podemos tener muchos duplicados.
- Fase de reducción (*Reduce*) – Es en cierta medida similar a un *"group by"* de SQL. Las ocurrencias similares se agrupan, y dependiendo de la función de reducción, se puede crear un resultado diferente. En nuestro ejemplo queremos contar los colores, y eso es lo que devuelve nuestra función.

Realmente, es un proceso más complicado:



1. Lectura desde HDFS de los ficheros de entrada como pares clave/valor.
2. Pasar cada línea de forma separada al mapeador, teniendo tantos mapeadores como bloques de datos tengamos.
3. El mapeador parsea los colores (claves) de cada fichero y produce un nuevo fichero para cada color con el número de ocurrencias encontradas (valor), es decir, mapea una clave (color) con un valor (número de ocurrencias).
4. Para facilitar la agregación, se ordenan y/o barajan los datos a partir de la clave.
5. La fase de reducción suma las ocurrencias de cada color y genera un fichero por clave con el total de cada color.
6. Las claves se unen en un único fichero de salida que se persiste en HDFS.



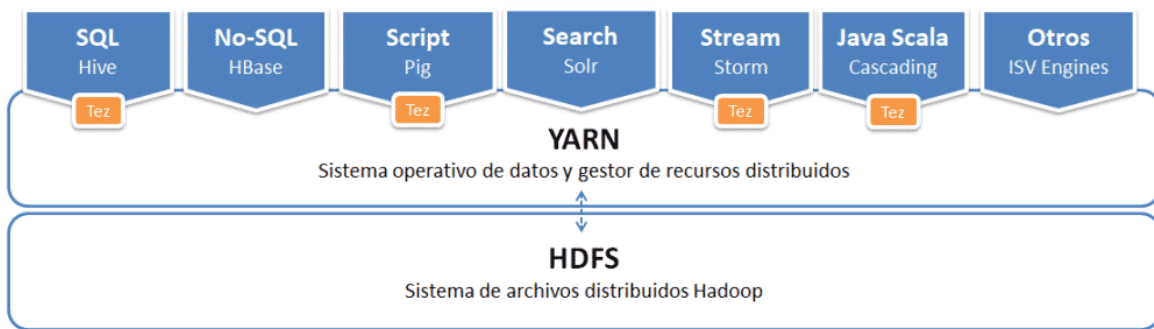
No es oro todo lo que reluce

Hadoop facilita el trabajo con grandes volúmenes de datos, pero montar un clúster funcional no es una cosa trivial. Existen gestores de clústers que hacen las cosas un poco menos incómodas (como son [Apache Ambari](#) ^{lk} o [Apache Mesos](#) ^{lk}), aunque la tendencia es utilizar una solución *cloud* que nos evita toda la instalación y configuración.

Tal como comentamos al inicio, uno de los puntos débiles de *Hadoop* es el trabajo con algoritmos iterativos, los cuales son fundamentales en la parte de IA. La solución es el uso de [Spark](#) ^{lk} (que estudiaremos próximamente), que mejora el rendimiento por una orden de magnitud.

YARN

Yet Another Resource Negotiator es un distribuidor de datos y gestor de recursos distribuidos. Forma parte de Hadoop desde la versión 2, y abstrae la gestión de recursos de los procesos *MapReduce* lo que implica una asignación de recursos más efectiva. YARN soporta varios frameworks de procesamiento distribuido, como *MapReduce v2*, *Tez*, *Impala*, *Spark*, etc..



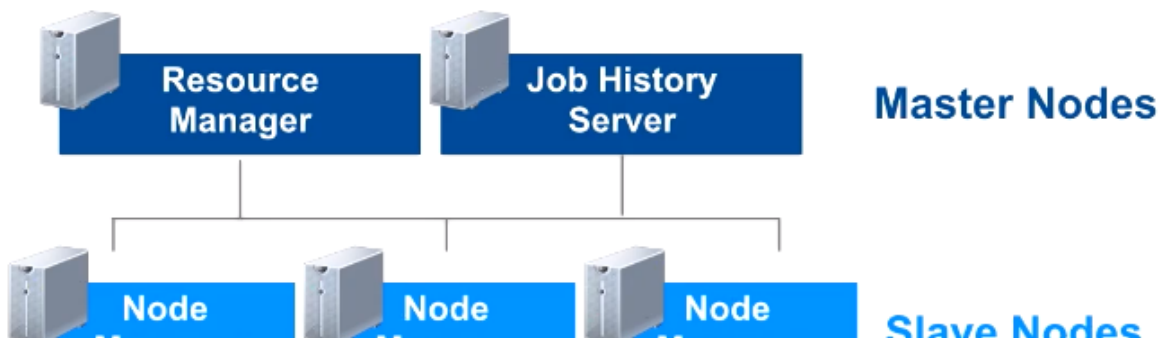
YARN y Hadoop

El objetivo principal de YARN es separar en dos servicios las funcionalidades de gestión de recursos de la monitorización/planificación de tareas. Por un lado, un gestor de los procesos que se ejecutan en el clúster, que permite coordinar diferentes aplicaciones, asignar recursos y prioridades, permitir su convivencia, etc. Y por otro lado, las aplicaciones, que pueden desarrollarse utilizando un marco de ejecución más ligero, no atado a un modelo estricto sobre cómo ejecutarse, lo que da más libertad para poder desarrollar las aplicaciones.

Componentes

Se divide en tres componentes principales: un *Resource Manager*, múltiples *Node Manager* y varios *ApplicationMaster*.

La idea es tener un *Resource Manager* por clúster y un *Application Master* por aplicación, considerando una aplicación tanto un único *job* como un conjunto de *jobs* cíclicos.





Componentes en YARN

El *Resource Manager* y el *Node Manager* componen el framework de computación de datos. En concreto, el *ResourceManager* controla el arranque de la aplicación, siendo la autoridad que orquesta los recursos entre todas las aplicaciones del sistema. A su vez, tendremos tantos *NodeManager* como *datanodes* tenga nuestro clúster, siendo responsables de gestionar y monitorizar los recursos de cada nodo (CPU, memoria, disco y red) y reportar estos datos al *Resource Manager*.

El *Application Master* es una librería específica encargada de negociar los recursos con el *ResourceManager* y de trabajar con los *Node Manager* para ejecutar y monitorizar las tareas.

Finalmente, en nuestro clúster, tendremos corriendo un *Job History Server* encargado de archivar los ficheros de log de los *jobs*. Aunque es un proceso opcional, se recomienda su uso para monitorizar los jobs ejecutados.

Resource Manager

El *Resource Manager* mantiene un listado de los *Node Manager* activos y de sus recursos disponibles. Dicho de otro modo, es el equivalente al *Namenode* de HDFS.

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

El gestor de recursos, a su vez, se divide en dos componentes:

- El *Scheduler* o planificador es el encargado de gestionar la distribución de los recursos del clúster de YARN. Además, las aplicaciones usan los recursos que el *Resource Manager* les ha proporcionado en función de sus criterios de planificación. Este planificador no monitoriza el estado de ninguna aplicación ni les ofrece garantías de ejecución, ni recuperación por fallos de la aplicación o el hardware, sólo planifica. Este componente realiza su planificación a partir de los requisitos de recursos necesarios por las aplicaciones (CPU, memoria, disco y red).
- *Applications Manager*: responsable de aceptar las peticiones de trabajos, negociar el contenedor con los recursos necesarios en el que ejecutar la *Application Master* y proporcionar reinicios de los trabajos en caso de que fuera necesario debido a errores.

Node Manager

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

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

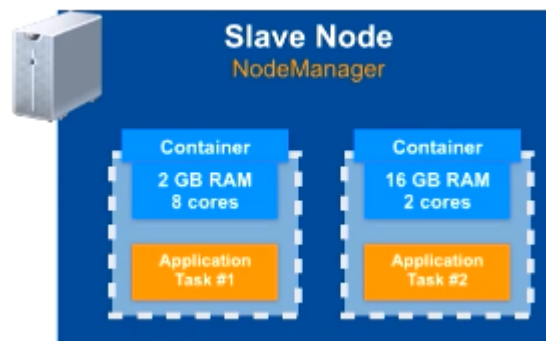


Contenedores

Es la unidad mínima de recursos de ejecución para las aplicaciones, y que representa una cantidad específica de memoria, núcleos de procesamiento (cores) y otros recursos (disco, red), para procesar las aplicaciones. Por ejemplo, un contenedor puede representar 4 gigabytes de memoria y 1 núcleo de procesamiento.

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

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



Contenedores en NodeManager

Los contenedores YARN tienen una asignación de recursos (CPU, memoria, disco y red) fija de un host del clúster y el *Node Manager* es el encargado de monitorizar esta asignación. Si un proceso sobrepasase los recursos asignados, por ejemplo, sería el encargado de detenerlo. Además, mapean las variables de entorno necesarias, las dependencias y los servicios necesarios para crear los procesos.

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

Application Master

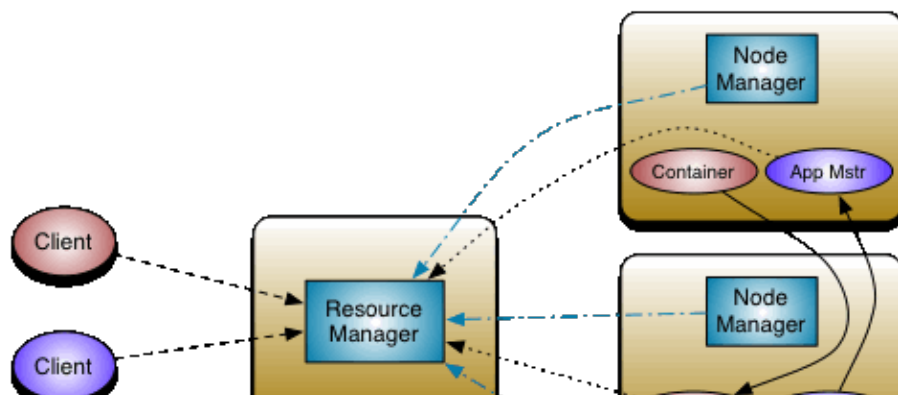
El *Application Master* es el responsable de negociar los recursos apropiados con el *Resource Manager* y monitorizar su estado y su progreso. También coordina la ejecución de todas las tareas en las que puede dividirse su aplicación.

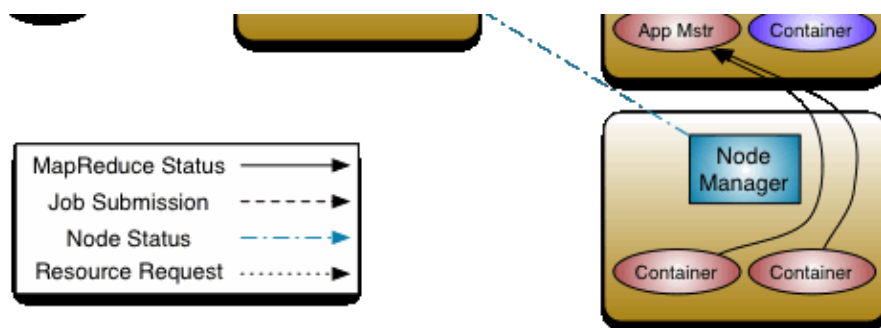
Existe un proceso *ApplicationMaster* por aplicación, y se ejecuta en uno de los nodos worker, para garantizar la escalabilidad de YARN, ya que si se ejecutaran todos los *ApplicationMaster* en el nodo maestro, junto con el *ResourceManager*, éste sería un cuello de botella para poder escalar o poder lanzar un gran número de aplicaciones sobre el clúster.

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

Funcionamiento

Podemos ver la secuencia de trabajo y colaboración de estos componentes en el siguiente gráfico:





Secuencia de trabajo YARN

1. El cliente envía una aplicación YARN.
2. *Resource Manager* reserva los recursos en un contenedor para su ejecución.
3. El *Application Manager* se registra con el *Resource Manager* y pide los recursos necesarios.
4. El *Application Manager* notifica al *Node Manager* la ejecución de los contenedores. Se ejecuta la aplicación YARN en el/los contenedor/es correspondiente.
5. El *Application Master* monitoriza la ejecución y reporta el estado al *Resource Manager* y al *Application Manager*.
6. Al terminar la ejecución, el *Application Manager* lo notifica al *Resource Manager*.

YARN soporta la reserva de recursos mediante el [Reservation System](#)¹⁶, un componente que permite a los usuarios especificar un perfil de recurso y restricciones temporales (*deadlines*) y posteriormente reservar recursos para asegurar la ejecución predecibles de las tareas importantes. Este sistema registra los recursos a lo largo del tiempo, realiza control de admisión para las reservas, e informa dinámicamente al planificador para asegurarse que se produce la reserva.

Para conseguir una alta escalabilidad (del orden de miles de nodos), YARN ofrece el concepto de [Federación](#)¹⁶. Esta funcionalidad permite conectar varios clústeres YARN y hacerlos visibles como un clúster único. De esta forma puede ejecutar trabajos muy pesados y distribuidos.

Hadoop v1

MapReduce en hadoop-2.x mantiene la compatibilidad del API con versiones previas (hadoop-1.x). De esta manera, todo los *jobs* de *MapReduce* funcionan perfectamente con YARN sólo recompilando el código.

En *Hadoop v1* los componentes encargados de realizar el procesamiento eran el *JobTracker* (situado en el *namenode*) y los *TaskTracker* (situados en los *datanodes*).

Instalación

Para trabajar en esta y las siguientes sesiones, vamos a utilizar la máquina virtual que tenemos compartida en *Aules*. A partir de la OVA de VirtualBox, podrás entrar con el usuario *iabd* y la contraseña *iabd*.

Si quieres instalar el software del curso, se recomienda crear una máquina virtual con cualquier distribución Linux. En mi caso, yo lo he probado en la versión *Lubuntu 20.04 LTS* y la versión 3.3.1 de *Hadoop*. Puedes seguir las instrucciones del artículo

[Cómo instalar y configurar Hadoop en Ubuntu 20.04 LTS](#) .

Para trabajar en local tenemos montada una solución que se conoce como *pseudo-distribuida*, porque es al mismo tiempo maestro y esclavo. En el mundo real o si utilizamos una solución cloud tendremos un nodo maestro y múltiples nodos esclavos.

Configuración

Los archivos que vamos a revisar a continuación se encuentran dentro de la carpeta

```
$HADOOP_HOME/etc/hadoop.
```

El archivo que contiene la configuración general del clúster es el archivo `core-site.xml`. En él se configura cual será el sistema de ficheros, que normalmente será `hdfs`, indicando el dominio del nodo que será el maestro de datos (*namenode*) de la arquitectura. Por ejemplo, su contenido será similar al siguiente:

core-site.xml

```
1 <configuration>
2   <property>
3     <name>fs.defaultFS</name>
4     <value>hdfs://iabd-virtualbox:9000</value>
5   </property>
6 </configuration>
```

El siguiente paso es configurar el archivo `hdfs-site.xml` donde se indica tanto el factor de replicación como la ruta donde se almacenan tanto los metadatos (*namenode*) como los datos en sí (*datanode*):

hdfs-site.xml

```
1 <configuration>
2   <property>
3     <name>dfs.replication</name>
4     <value>1</value>
```

```
5     </property>
6
7     <property>
8         <name>dfs.namenode.name.dir</name>
9         <value>/opt/hadoop-data/hdfs/namenode</value>
10    </property>
11
12    <property>
13        <name>dfs.datanode.data.dir</name>
14        <value>/opt/hadoop-data/hdfs/datanode</value>
15    </property>
16 </configuration>
```



Recuerda

Si tuviésemos un clúster, en el nodo maestro sólo configuraríamos la ruta del *namenode* y en cada uno de los nodos esclavos, únicamente la ruta del *datanode*.

Para configurar YARN, primero editaremos el archivo `yarn-site.xml` para indicar quien va a ser el nodo maestro, así como el manejador y la gestión para hacer el *MapReduce*:

yarn-site.xml

```
1 <configuration>
2   <property>
3       <name>yarn.resourcemanager.hostname</name>
4       <value>iabd-virtualbox</value>
5   </property>
6   <property>
7       <name>yarn.nodemanager.aux-services</name>
8       <value>mapreduce_shuffle</value>
9   </property>
10  <property>
11      <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
12      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
13  </property>
14 </configuration>
```

Y finalmente el archivo `mapred-site.xml` para indicar que utilice YARN como framework *MapReduce*:


```

1 <configuration>
2   <property>
3     <name>mapreduce.framework.name</name>
4     <value>yarn</value>
5   </property>
6 </configuration>

```



Hadoop en Docker

Si no quieres (o puedes) ejecutar la máquina virtual, bien puedes utilizar un servicio en la nube como AWS EMR (que veremos en la próxima sesión), o lanzar un contenedor *Docker*. La red contiene múltiples imágenes ya creadas, tanto con el core como con los servicios ya configurados. Para esta sesión y la siguiente, con una imagen sencilla como la que podemos crear desde <https://www.section.io/engineering-education/set-up-containerize-and-test-a-single-hadoop-cluster-using-docker-and-docker-compose/>



es suficiente.

Puesta en marcha

Para arrancar Hadoop/HDFS, hemos de ejecutar el comando `start-dfs.sh`. Al finalizar, veremos que ha arrancado el *namenode*, los *datanodes*, y el *secondary namenode*.

Si en cualquier momento queremos comprobar el estado de los servicios y procesos en ejecución, tenemos el comando `jps`.

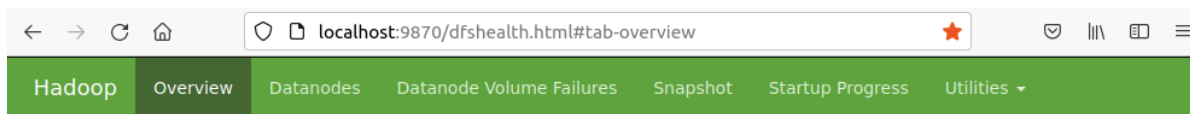
```

hadoop@hadoop-VirtualBox:~$ start-dfs.sh
Starting namenodes on [hadoop-VirtualBox]
Starting datanodes
Starting secondary namenodes [hadoop-VirtualBox]
hadoop@hadoop-VirtualBox:~$ jps
4513 Jps
4002 NameNode
4361 SecondaryNameNode
4141 DataNode

```

Arrancando HDFS

Si accedemos a `http://iabd-virtualbox:9870/` podremos visualizar su interfaz web.



Overview 'hadoop-VirtualBox:9000' (active)

Started:	Mon Sep 20 12:02:16 +0200 2021
Version:	3.2.2, r7a3bc90b05f257c8ace2f76d742649060f7a932
Compiled:	Sun Jan 03 10:26:00 +0100 2021 by hexiaoqiao from branch-3.2.2

Cluster ID:	CID-83ed185a-1907-476b-bfda-7ce072ed045c
Block Pool ID:	BP-1410034788-192.168.0.101-1618596221610

Summary

Security is off.

Safemode is off.

225 files and directories, 70 blocks (70 replicated blocks, 0 erasure coded block groups) = 295 total filesystem object(s).

Heap Memory used 100.23 MB of 192 MB Heap Memory. Max Heap Memory is 1.69 GB.

Non Heap Memory used 50.48 MB of 51.77 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	60.03 GB
-----------------------------	----------

Interfaz Web de Hadoop

Para arrancar YARN utilizaremos el comando

`start-yarn.sh` para lanzar el *Resource*

Manager y el *Node Manager*:

Y a su vez, YARN también ofrece un interfaz web para obtener información relativa a los jobs ejecutados. Nos conectaremos con el nombre del nodo principal y el puerto `8088`. En nuestro

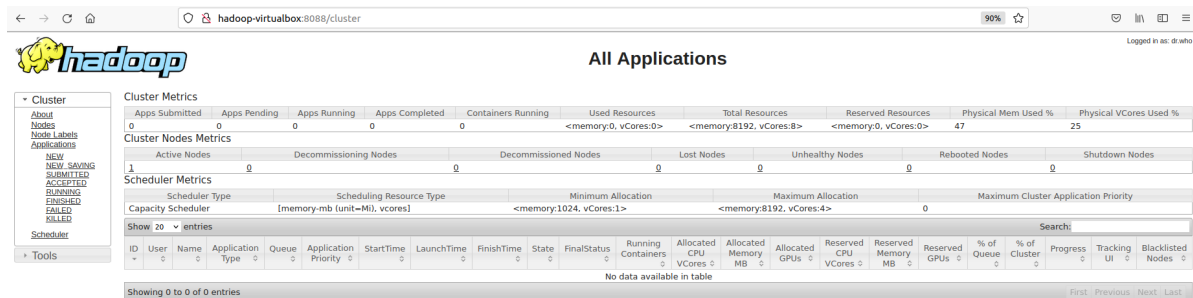
caso lo hemos realizado a `http://iabd-`

`virtualbox:8088` obteniendo la siguiente

página:

```
hadoop@hadoop-VirtualBox:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
hadoop@hadoop-VirtualBox:~$ jps
5537 Jps
4002 NameNode
5077 ResourceManager
4361 SecondaryNameNode
5226 NodeManager
4141 DataNode
```

Arrancando YARN




The screenshot shows the Hadoop Web UI interface. The top navigation bar includes the Hadoop logo and the title 'All Applications'. The left sidebar contains a menu with options like 'Cluster', 'Nodes', 'Applications', 'Scheduler', and 'Tools'. The main content area displays various metrics and tables. The 'Cluster Metrics' table shows 0 apps submitted, 0 pending, 0 running, and 0 completed. The 'Scheduler Metrics' table shows 0 active nodes, 0 decommissioning nodes, 0 decommissioned nodes, 0 lost nodes, 0 unhealthy nodes, 0 rebooted nodes, and 0 shutdown nodes. The 'Applications' table is empty, showing 0 entries.

Interfaz Web de YARN

Hola Mundo

El primer ejemplo que se realiza como *Hola Mundo* en *Hadoop* suele ser una aplicación que cuente las ocurrencias de cada palabra que aparece en un documento de texto.

En nuestro caso, vamos a contar las palabras del libro de *El Quijote*, el cual podemos descargar desde <https://gist.github.com/jsdario/6d6c69398cb0c73111e49f1218960f79> .

Una vez arrancado *Hadoop* y *YARN*, vamos a colocar el libro dentro de HDFS (estos comandos los estudiaremos en profundidad en la siguiente sesión):

```
1  hdfs dfs -put el_quijote.txt /user/iabd/
```

Hadoop tiene una serie de ejemplos ya implementados para demostrar el uso de MapReduce en la carpeta `$HADOOP_HOME/share/hadoop/mapreduce`. Así pues, podemos ejecutar el programa `wordcount` de la siguiente manera:

Comando Hadoop

```
1  hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-  
examples-3.3.1.jar \  
2      wordcount /user/iabd/el_quijote.txt /user/iabd/salidaWC
```

Comando Yarn

```
1  yarn jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-  
examples-3.3.1.jar \  
2      wordcount /user/iabd/el_quijote.txt /user/iabd/salidaWC
```

Si nos fijamos en la salida del comando podremos ver una traza del proceso *MapReduce*:

```
1  2022-11-28 09:24:02,763 INFO client.DefaultNoHARMFailoverProxyProvider:  
Connecting to ResourceManager at iabd-virtualbox/127.0.1.1:8032  
2  2022-11-28 09:24:03,580 INFO mapreduce.JobResourceUploader: Disabling  
Erasure Coding for path: /tmp/hadoop-yarn/staging/iabd/.staging/  
job_1669623168732_0001  
3  2022-11-28 09:24:04,473 INFO input.FileInputFormat: Total input files to  
process : 1  
4  2022-11-28 09:24:04,623 INFO mapreduce.JobSubmitter: number of splits:1  
5  2022-11-28 09:24:05,313 INFO mapreduce.JobSubmitter: Submitting tokens for  
job: job_1669623168732_0001  
6  2022-11-28 09:24:05,313 INFO mapreduce.JobSubmitter: Executing with tokens:  
[]  
7  2022-11-28 09:24:05,820 INFO conf.Configuration: resource-types.xml not  
found  
8  2022-11-28 09:24:05,821 INFO resource.ResourceUtils: Unable to find  
'resource-types.xml'.  
9  2022-11-28 09:24:06,483 INFO impl.YarnClientImpl: Submitted application  
application_1669623168732_0001  
10 2022-11-28 09:24:06,644 INFO mapreduce.Job: The url to track the job:  
http://iabd-virtualbox:8088/proxy/application_1669623168732_0001/  
11 2022-11-28 09:24:06,645 INFO mapreduce.Job: Running job:
```

```

job_1669623168732_0001
12  2022-11-28 09:24:20,124 INFO mapreduce.Job: Job job_1669623168732_0001
    running in uber mode : false
13  2022-11-28 09:24:20,126 INFO mapreduce.Job: map 0% reduce 0%
14  2022-11-28 09:24:28,406 INFO mapreduce.Job: map 100% reduce 0%
15  2022-11-28 09:24:35,623 INFO mapreduce.Job: map 100% reduce 100%
16  2022-11-28 09:24:36,687 INFO mapreduce.Job: Job job_1669623168732_0001
    completed successfully

```

Podemos observar como se crea un *job* que se envía a YARN, el cual ejecuta el proceso *MapReduce*, el cual tarda alrededor de 40 segundos. A continuación aparecen estadísticas del proceso:

```

1  2022-11-28 09:24:36,824 INFO mapreduce.Job: Counters: 54
2      File System Counters
3          FILE: Number of bytes read=347063
4          FILE: Number of bytes written=1241507
5          FILE: Number of read operations=0
6          FILE: Number of large read operations=0
7          FILE: Number of write operations=0
8          HDFS: Number of bytes read=1060376
9          HDFS: Number of bytes written=257233
10         HDFS: Number of read operations=8
11         HDFS: Number of large read operations=0
12         HDFS: Number of write operations=2
13         HDFS: Number of bytes read erasure-coded=0
14     Job Counters
15         Launched map tasks=1
16         Launched reduce tasks=1
17         Data-local map tasks=1
18         Total time spent by all maps in occupied slots (ms)=6848
19         Total time spent by all reduces in occupied slots (ms)=4005
20         Total time spent by all map tasks (ms)=6848
21         Total time spent by all reduce tasks (ms)=4005
22         Total vcore-milliseconds taken by all map tasks=6848
23         Total vcore-milliseconds taken by all reduce tasks=4005
24         Total megabyte-milliseconds taken by all map tasks=7012352
25         Total megabyte-milliseconds taken by all reduce
tasks=4101120
26     Map-Reduce Framework
27         Map input records=2186
28         Map output records=187018
29         Map output bytes=1808330
30         Map output materialized bytes=347063
31         Input split bytes=117
32         Combine input records=187018
33         Combine output records=22938
34         Reduce input groups=22938
35         Reduce shuffle bytes=347063
36         Reduce input records=22938
37         Reduce output records=22938

```

```


38      Spilled Records=45876
39      Shuffled Maps =1
40      Failed Shuffles=0
41      Merged Map outputs=1
42      GC time elapsed (ms)=1173
43      CPU time spent (ms)=4890
44      Physical memory (bytes) snapshot=751063040
45      Virtual memory (bytes) snapshot=5099941888
46      Total committed heap usage (bytes)=675282944
47      Peak Map Physical memory (bytes)=520818688
48      Peak Map Virtual memory (bytes)=2548346880
49      Peak Reduce Physical memory (bytes)=230244352
50      Peak Reduce Virtual memory (bytes)=2551595008
51      Shuffle Errors
52          BAD_ID=0
53          CONNECTION=0
54          IO_ERROR=0
55          WRONG_LENGTH=0
56          WRONG_MAP=0
57          WRONG_REDUCE=0
58      File Input Format Counters
59          Bytes Read=1060259
60      File Output Format Counters
61          Bytes Written=257233

```

Para poder obtener toda la información de un *job* necesitamos arrancar el **Job History Server**:

```
1 mapred --daemon start historyserver
```

De manera que si accedemos a la URL que se visualiza en el log, podremos ver de forma gráfica la información obtenida:



MapReduce Job job_1642247847632_0001

Logged in as: dr.who

Application

Job

- Overview
- Counters
- Configuration
- Map tasks
- Reduce tasks

Tools

Job Overview			
Job Name:	word count		
User Name:	iabd		
Queue:	default		
State:	SUCCEEDED		
Uberized:	false		
Submitted:	Sat Jan 15 12:59:52 CET 2022		
Started:	Sat Jan 15 13:00:07 CET 2022		
Finished:	Sat Jan 15 13:00:46 CET 2022		
Elapsed:	38sec		
Diagnostics:			
Average Map Time	20sec		
Average Shuffle Time	9sec		
Average Merge Time	0sec		
Average Reduce Time	2sec		

ApplicationMaster			
Attempt Number	Start Time	Node	Logs
1	Sat Jan 15 12:59:58 CET 2022	iabd-virtualbox:8042	logs

Task Type	Total	Complete
Map	1	1
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Resultado del History Server

Si accedemos al interfaz gráfico de HDFS (

<http://iabd-virtualbox:9870/explorer.html#/user/iabd/salidaWC>), podremos ver cómo se ha creado la carpeta `salidaWC` y dentro contiene dos archivos:

- `_SUCCESS` : indica que el job de *MapReduce* se ha ejecutado correctamente
- `part-r-00000` : bloque de datos con el resultado

The screenshot shows the Hadoop web interface with a modal window titled "File information - part-r-00000". The modal displays the following details:

- Block information:** Block 0 (selected)
- Block ID: 1073743447
- Block Pool ID: BP-481169443-127.0.1.1-1639217848073
- Generation Stamp: 2623
- Size: 257233
- Availability:
 - iabd-virtualbox

The "File contents" section shows the following text:

```
"Apenas 1
"Caballero 4
"Conde 1
"Donde 1
"Más 1
"Miau", 1
"No 1
"Rastrea 1
```

The background interface shows the "Browse Directory" view for the path `/user/iabd/salidaWC`, displaying two files: `_SUCCESS` and `part-r-00000`.

Contenido HDFS de salidaWC

? Autoevaluación

- ¿Qué comando HDFS utilizarías para obtener el contenido de la carpeta `/user/iabd/salidaWC`? ¹
- ¿Y para obtener el contenido del archivo generado? ²

MapReduce en Python

El API de *MapReduce* está escrito en *Java*, pero mediante *Hadoop Streaming* podemos utilizar *MapReduce* con cualquier lenguaje compatible con el sistema de tuberías Unix (`|`).

Para entender cómo funciona, vamos a reproducir el ejemplo anterior mediante *Python*.

Mapper

Primero creamos el *mapeador*, el cual se encarga de parsear línea a línea el fragmento de documento que reciba, y va a generar una nueva salida con todas las palabras de manera que cada nueva línea la compongan una tupla formada por la palabra, un tabulador y el número 1 (hay una ocurrencia de dicha palabra)

mapper.py

```
1  #!/usr/bin/python3
2  import sys
3  for linea in sys.stdin:
4      # eliminamos los espacios de delante y de detrás
5      linea = linea.strip()
6      # dividimos la línea en palabras
7      palabras = linea.split()
8      # creamos tuplas de (palabra, 1)
9      for palabra in palabras:
10         print(palabra, "\t1")
```

Si queremos probar el mapper, podríamos ejecutar el siguiente comando:

```
1  cat el_quijote.txt | python3 mapper.py
```

Obteniendo un resultado similar a:

```
1  ...
2  gritos 1
3  al 1
4  cielo 1
5  allí 1
6  se 1
7  renovaron 1
8  las 1
9  maldiciones 1
10 ...
```

Reducer

A continuación, en el *reducer* vamos a recibir la salida del *mapper* y parsearemos la cadena para separar la palabra del contador.

Para llevar la cuenta de las palabras, vamos a meterlas dentro de un diccionario para incrementar las ocurrencias encontradas.



Cuidado con la memoria

En un caso real, hemos de evitar almacenar todos los datos que recibimos en memoria, ya que es posible que al trabajar con *big data* no quepa en la RAM de cada *datanode*. Para ello, se recomienda el uso de la librería *itertools*, por ejemplo, utilizando la función `groupby()`.

Finalmente, volvemos a crear tuplas de palabra, tabulador y cantidad de ocurrencias.

reducer.py

```
1  #!/usr/bin/python3
2  import sys
3
4  # inicializamos el diccionario
5  dictPalabras = {}
6
7  for linea in sys.stdin:
8      # quitamos espacios de sobra
9      linea = linea.strip()
10     # parseamos la entrada de mapper.py
11     palabra, cuenta = linea.split('\t', 1)
12     # convertimos cuenta de string a int
13     try:
14         cuenta = int(cuenta)
15     except ValueError:
16         # cuenta no era un numero, descartamos la linea
17         continue
18
19     try:
20         dictPalabras[palabra] += cuenta
21     except:
22         dictPalabras[palabra] = cuenta
23
24 for palabra in dictPalabras.keys():
25     print(palabra, "\t", dictPalabras[palabra])
```

Para probar el proceso completo, ejecutaremos el siguiente comando:

```
1  cat el_quijote.txt | python3 mapper.py | python3 reducer.py > salida.tsv
```


Si abrimos el fichero, podemos ver el resultado:

salida.tsv

1	don	1072
2	quijote	812
3	de	9035
4	la	5014
5	mancha	50
6	miguel	3
7	cervantes	3
8	...	

Hadoop Streaming

Una vez comprobados que los algoritmos de mapeo y reducción funcionan, vamos a procesarlos dentro de *Hadoop* para aprovechar la computación distribuida.

Para ello, haremos uso de [Hadoop Streaming](#) , el cual permite ejecutar *jobs Map/Reduce* con cualquier script (y por ende, codificados en cualquier lenguaje de programación) que pueda leer de la entrada estándar (*stdin*) y escribir a la salida estándar (*stdout*). De este manera, *Hadoop Streaming* envía los datos en crudo al *mapper* vía *stdin* y tras procesarlos, se los pasa al *reducer* vía *stdout*.

La sintaxis para ejecutar los *jobs* es:

```
1 mapred streaming \  
2   -input miCarpetaEntradaHDFS \  
3   -output miCarpetaSalidaHDFS \  
4   -mapper scriptMapper \  
5   -reducer scriptReducer
```



Versiones 1.x

En versiones más antiguas de Hadoop, en vez de utilizar el comando `mapred`, se utiliza el comando `hadoop jar rutaDeHadoopStreaming.jar <parámetros>`, siendo normalmente la ruta del jar `$HADOOP_HOME/share/hadoop/tools/lib`.

Así pues, en nuestro caso ejecutaríamos el siguiente comando si tuviésemos los archivos (tanto los datos como los scripts) dentro de HDFS:

```
1 mapred streaming \  
2   -input el_quijote.txt \  
3   -output salidaPy \  
4   -mapper mapper.py \  
5   -reducer reducer.py
```

```
5      -reducer reducer.py
```

⚠ Permisos de ejecución

Recuerda darle permisos de ejecución a ambos scripts (`chmod u+x mapper.py` y `chmod u+x reducer.py`) para que *Hadoop Streaming* los pueda ejecutar





Como queremos usar los archivos que tenemos en local, debemos indicar cada uno de los elementos mediante el parámetro `-file`:

```
1  mapred streaming \  
2      -input el_quijote.txt \  
3      -output salidaPy \  
4      -mapper mapper.py -file mapper.py \  
5      -reducer reducer.py -file reducer.py
```

Una vez finalizado el *job*, podemos comprobar cómo se han generado el resultado en HDFS mediante:

```
1  hdfs dfs -head /user/iabd/salidaPy/part-00000
```

Referencias

- Documentación de [Apache Hadoop](#) .
- [Hadoop: The definitive Guide, 4th Ed - de Tom White - O'Reilly](#) .
- Artículo de [Hadoop por dentro](#) .
- [Tutorial de Hadoop](#)  de *Tutorialspoint*.

Actividades

Para los siguientes ejercicios, copia el comando y/o haz una captura de pantalla donde se muestre el resultado de cada acción.

1. (RA5075.2 / CE5.2b / 1p) Sobre *Hadoop*, ejecuta el siguiente comando y explica qué sucede:

```
1  yarn jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-  
examples-3.3.1.jar pi 16 32
```

2. (RA5075.2 / CE5.2b / 2p) Vuelve a contar las palabras que tiene *El Quijote*, pero haciendo usos de los scripts *Python*, teniendo en cuenta que el proceso de mapeo va a limpiar las palabras de signos ortográficos (quitar puntos, comas, paréntesis) y en el *reducer* vamos a considerar que las palabras en mayúsculas y minúsculas son la misma palabra.

- *Tip*: para la limpieza, puedes utilizar el método de `string.translate` de manera que elimine las `string.punctuation`.

Debes ejecutar ambos *script* como procesos *MapReduce* mediante *Hadoop Streaming* y comprobar en HDFS el archivo que se ha creado.

3. (RA5075.4 / CE5.4a / 1p) Entra en *Hadoop UI* y en *YARN*, y visualiza los procesos que se han ejecutado en las actividades 1 y 2.

-
1. `hdfs dfs -ls /user/iabd/salidaWC` ↩
 2. `hdfs dfs -cat /user/iabd/salidaWC/part-r-00000` ↩

