Arquitecturas de software para *big data*

PID_00286214

Remo Suppi Boldrito

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Remo Suppi Boldrito

Ingeniero de Telecomunicaciones. Doctor en Informática por la UAB. Profesor del Departamento de Arquitectura de Computadores y Sistemas Operativos en la Universidad Autónoma de Barcelona.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Josep Jorba Esteve

Primera edición: febrero 2022

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Remo Suppi Boldrito
Producción: FUOC

Índice

Introducción

Big data **(BD)** es una de las tendencias tecnológicas que impregna nuestra sociedad y que de una forma significativa permite que las decisiones basadas en la experiencia den lugar a las decisiones basadas en datos.

Para complementar este punto el alumno deberá leer la parte 1 del libro *Big Data for Dummies* para analizar y contextualizar qué es *big data*, cuáles son sus fundamentos de *big data* y sus características, qué tipos de *big data* existen y dónde se encuentra el punto de transición entre los modelos de cómputos antes de *big data* y este nuevo concepto de cómputo distribuido.

Las preguntas que el alumno deberá plantearse serán:

- ¿Cómo obtener conocimiento de cantidades masivas de datos?
- ¿Qué es y cómo funciona *data analytics*?
- ¿En qué infraestructura se realiza este procesamiento?
- ¿Cómo se guarda –si es que se guarda– esta cantidad ingente de datos?
- ¿Qué herramientas se necesitan para hacer este procesamiento?
- ¿Se pueden almacenar y recuperar cantidades masivas de datos en BD relacionales y de forma eficiente?
- ¿Cuál es la arquitectura hardware y software para procesar *big data*?
- ¿Cómo se puede gestionar un gran volumen de datos sin causar problemas en el centro de datos actual?

Objetivos

Los objetivos principales de este módulo son los siguientes:

1. Conocer las diferentes infraestructuras software que se aplican al procesamiento de *big data*.

2. Analizar el concepto de MapReduce y su utilización en el procesamiento de grandes volúmenes de información.

3. Realizar pruebas de conceptos (ejemplos) con las herramientas más comunes en entornos *big data* como Hadoop y Spark.

4. Analizar distribuciones específicas basadas en Docker como prueba de concepto de los diferentes componentes.

El estudiante deberá centrar su atención en los siguientes conceptos fundamentales de este módulo:

1. Procesamiento de *big data*

2. Algoritmo de MapReduce

3. Hadoop + HDFS

4. Spark

5. Distribuciones basadas en Docker como pruebas funcionales del procesamiento de *big data*

1. Conceptos sobre el procesamiento de *big data*

Como ya se ha planteado anteriormente, *big data* se ha transformado en una expresión de uso habitual, ya que, tanto en centros de R+D+i como en las empresas, industrias e instituciones, hay cada vez más cantidad de datos y son necesarias nuevas técnicas y herramientas para procesarlos y obtener información de ellos. La revolución de las TIC en el nuevo siglo no solo ha causado un diluvio de datos (*data deluge*), sino también su variedad, es decir, estructurados, semiestructurados y no estructurados, y su velocidad (millones de dispositivos y aplicaciones que generan datos enormes cada segundo).

Este enorme crecimiento de los datos ha llegado al límite de procesamiento/almacenamiento de las infraestructuras de gestión de la información existentes, que ha obligado a las empresas a invertir más en hardware y en actualizaciones de bases de datos. Esta tendencia, que aparentemente soluciona el problema inicial, no es una solución real, ya que el conjunto de datos sigue creciendo y por tanto se entra en un ciclo de más hardware + almacenamiento y así continuamente. Además, la infraestructura tradicional no es eficiente debido a los altos costos, las limitaciones de escalabilidad (cuando se trata de petabytes) y la incompatibilidad de los sistemas de bases de datos relacionales con datos no estructurados.

Para abordar la enorme cantidad de datos en la web, Google presentó en 2004 un modelo de programación llamado **MapReduce** (MR) para realizar, de forma paralela, tareas de búsqueda sobre sus clústeres de servidores. Para lograr este objetivo publicaron sus ideas sobre un sistema de archivos distribuidos para tener los datos cerca de donde se tuvieran que procesar (2003) y luego el algoritmo de procesamiento MapReduce (en diciembre de 2004).

Basados en estas ideas, los investigadores Doug Cutting (desarrollador de Lucene, un creador de índices de búsqueda, y Nutch, una araña –*spider* o *crawler*– o motor de búsqueda de contenidos por Internet) y Mike Cafarella (profesor de la Universidad de Michigan y también implicado en el desarrollo de Nutch) crearon un **sistema de archivos y un entorno de procesamiento** y realizaron las implementaciones para ejecutar Nutch (el motor de búsqueda mencionado previamente) sobre esta infraestructura.

En 2006, cuando Doug Cutting estaba trabajando con Yahoo, hicieron mejoras en el software, que habían probado con Nutch, y crearon un entorno llamado **Hadoop** como un proyecto de código abierto en la Apache Software Foundation (versión 0.1.0). Desde entonces, Hadoop se ha convertido en el estándar

de facto para almacenar, procesar y analizar cientos de terabytes o petabytes de datos, y como un proyecto 100 % de código abierto y disponible para la comunidad que lo desee utilizar incluso con fines comerciales.

Hadoop permite el procesamiento distribuido de enorme cantidad de datos en un conjunto de servidores de bajo costo que almacenan y procesan los datos y que pueden escalar sin límites (este enfoque se denomina escalado horizontal).

Ejemplo: Hadoop en Yahoo!

De acuerdo con los datos de 2015, los clústeres de Hadoop en Yahoo! incluían más de cuarenta mil servidores y almacenaban cuarenta petabytes de datos de aplicaciones.

Después del lanzamiento del entorno Hadoop, *big data analythics* se transformó en el gran reto tecnológico que permitía a las empresas o instituciones dar el salto estratégico de la retrospectiva a la prospectiva extrayendo valor de los grandes conjuntos de datos. Como en toda nueva idea, existe una comunidad escéptica que manifiesta su rechazo al «*big data* y lo que comporta», ya que sostienen que «durante décadas, las empresas han tomado decisiones comerciales basadas en datos transaccionales almacenados en bases de datos relacionales (DBRMS) y ahora no es diferente», o que son «técnicas y herramientas (DBRMS) de demostrada eficiencia y que no es necesario cambiar».

En estas afirmaciones ignoran que existe un enorme potencial de datos no tradicionales y no estructurados, como los registros web, redes sociales, correo electrónico, datos de sensores, imágenes, audio y vídeo, que pueden contener información útil o valiosa. Estas opiniones tampoco tienen en cuenta lo que ocurre en el mundo real, donde la **toma de decisiones basada en datos** ha crecido exponencialmente en los últimos años con una reducción significativa en la toma de decisiones basadas en las evidencias o el instinto, la intuición o la experiencia.

1.1. Problemas a resolver en *big data*

Como ya se ha definido, en una arquitectura para el procesamiento de *big data*, como en cualquier entorno orientado a datos, existen cuatro niveles que interactúan: **recolección de datos**, **almacenamiento**, **procesamiento** y **visualización**, más uno más global de **administración**.

Esta arquitectura es la habitual que se puede encontrar en cualquier entorno orientado a datos (por ejemplo, *datamining*, *business intelligence* o *deep learning*), pero, cuando se asocia a estos entornos el concepto de *big data*, cada uno de estos niveles ha evolucionado o se han generado nuevas herramientas, algoritmos o entornos para adecuarse al *big data*.

En la **recolección** de los datos orientados a *big data* han surgido gran cantidad de herramientas orientadas a este fin (*data ingestion*), que permiten el **procesamiento secuencial habitual de datos almacenados** (*batch* o lotes) para obtener el siguiente tramo o secuencia (*chunk*) de datos desde el último leído; en

todo caso, en su mayoría, y dado el volumen de los mismos, las herramientas han evolucionado para recolectar eficientemente **flujos de datos** (*streams*) en tiempo real.

En el **almacenamiento** también han surgido grandes cambios para adaptarse a las características de los datos; ello se ha reflejado en una gran cantidad de desarrollos de motores de bases de datos y sistemas de archivos distribuidos para adecuarse a la realidad de procesamiento y, así, disponer de escalabilidad, fiabilidad y cercanía de los datos para su tratamiento.

Ejemplo: motores de bases de datos

NoSQL, documentos, columnas, llave/valor, grafos.

Ejemplo: sistemas de archivos distribuidos

Apache HDFS, BeeGFS, DiscoDDFS, Google GFS, BaiduFS, GlusterFS, QuantcastFS, CephFS, GridGain-in-memoryFS, LustreFS.

Probablemente, una de las mayores dificultades de *big data* es en el **procesamiento**, ya que los algoritmos y las técnicas han tenido que transformarse para mantener la escalabilidad y las prestaciones sobre un conjunto de datos y una plataforma de cómputo distribuida. Para ello, se han desarrollado nuevos paradigmas de procesamiento de la información que han caracterizado todo el entorno (y en cada una de las capas para adecuarlas a sus necesidades). Estos nuevos métodos y algoritmos se conocen como **MapReduce** (MR) o **Massive Paralell Processing** (MPP).

MapReduce es un paradigma de programación cuyo nombre viene dado por las dos funciones que lo componen (*map* y *reduce*) y del entorno (*open source*) Apache Hadoop en el que se implementa.

Ved también

Hablamos brevemente sobre MapReduce en el módulo «Introducción a las infraestructuras para *big data*» de esta asignatura.

Este paradigma, si bien no es la solución para todos los problemas, trabaja con grandes conjuntos de datos (petabytes) y se ejecuta sobre **sistemas de archivos distribuidos** (HDFS) y vinculados a herramientas como HBase, Hive, Impala o Cassandra (bases de datos).

Este paradigma es apto para procesar aquellos datos que pueden trabajar sobre tuplas del tipo [clave, valor] y donde la función **map()** procesa en paralelo las tuplas de un dominio y genera una lista de pares en un dominio diferente, que se agruparán bajo la misma clave utilizando un esquema de procesamiento *master-worker* en forma de árbol.

Posteriormente, la función **reduce()** se aplica de forma paralela para cada grupo y genera una colección de valores para cada dominio.

Para ver otro enfoque a una implantación de MR se analizará cómo sería el código Python de estas funciones para contar palabras. Para ello se desarrollará la función **mapper.py**, que leerá los datos de la entrada estándar (es decir, de donde le venga la entrada por defecto y que se identifica con STDIN), los dividirá en palabras y generará una lista de líneas que mapean palabras en salida estándar (donde esté asignada y que se identifica por STDOUT). Este

Ved también

Sobre el código Python podéis consultar las siguientes referencias:

Referencia 1.
Referencia 2.

código no calculará una suma (intermedia) de las ocurrencias de una palabra y en su lugar se generarán tuplas equivalentes de salida, aunque una palabra específica pueda aparecer varias veces en la entrada.

La función de **contar** se dejará para el siguiente paso (**reduce**), que hará el recuento de la suma final. Por supuesto, es una forma de implementar y se puede cambiar el comportamiento como se desee. Para hacer las pruebas funcionales se debe cambiar el atributo de ejecución con `chmod +x mapper.py`.

```
#!/usr/bin/env python3
"""mapper.py"""
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # generate tuples
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step.
        print ('%s\t%s' % (word, 1))
```

Si se ejecuta con la frase planteada inicialmente:

```
echo "La ciencia es una ecuación diferencial. La religión es una condición de frontera" |./mapper.py
```

El resultado será:

La	1
ciencia	1
es	1
una	1
ecuación	1
diferencial.	1
La	1
religión	1
es	1
una	1
condición	1
de	1
frontera	1

El siguiente código que se debe escribir será `reducer.py`, que leerá los resultados de mapper.py desde la entrada estándar (STDIN) y suma las ocurrencias generando la cuenta final a la salida estándar (STDOUT). Hay que recordar también que se debe ejecutar `chmod +x reducer.py` para activar los permisos de ejecución.

```
#!/usr/bin/env python3
"""reducer.py"""
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue
    # this IF-switch only works because Hadoop sorts map
    # output by key (here: word) before it is passed to the
    # reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
# do not forget to output the last word if needed!
if current_word == word:
    print ('%s\t%s' % (current_word, current_count))
```

Ahora se puede probar la funcionalidad ejecutando:

```
echo "La ciencia es una ecuación diferencial. La religión es una condición de frontera" |
./mapper.py | sort -k1,1 |./reducer.py
```

Como se puede ver, se simula lo que luego se verá que hace el entorno de ejecución (Hadoop), que en nuestro caso es ordenar las tuplas por palabras (*key*). La salida será:

ciencia	1
condición	1
de	1
diferencial.	1
ecuación	1
es	2
frontera	1
La	2
religión	1
una	2

Si se prueba con un texto extenso, por ejemplo *Alice's Adventures in Wonderland* (3.731 líneas) del proyecto Gutenberg (OpenBook), donde se ha ordenado la salida para ver las palabras más repetidas (solo se muestran las primeras líneas):

```
cat Alice.txt |./mapper.py | sort -k1,1 |./reducer.py | sort -k2 -rn
```

Figura 1. *Alice's Adventures in Wonderland*

```
└─ cat Alice.txt | ./mapper.py | sort -k1,1 | ./reducer.py | sort -k2 -rn
the     1675
and     780
to      777
a       667
of      600
she     485
said    416
in      402
it      355
was     329
you     303
I       249
as      246
that    225
Alice   221
```

El **paradigma MPP** ofrece una solución tradicional adaptada a *big data* basada en dividir los grandes conjuntos de datos en secciones (*slices*), que serán más fáciles de gestionar; cada uno de ellos será asignado a un elemento de cómputo para su procesamiento. El dispositivo de cómputo los procesará y, cuando termine, el sistema combinará los resultados parciales para dar un resultado final (equivalente a una secuencia *fork-join* en un modelo *master-workers*). Dado que los elementos de cómputo (procesadores) trabajarán sobre su segmento de datos asignado de la BD y se comunicarán mediante mensajes cuando generen los resultados, no hay interacción entre ellos; esto se conoce como «débilmente acoplados» (otros autores lo denominan *shared nothing*).

Entre las características principales de estos sistemas se pueden enumerar sus posibilidades de sintonización y escalado ilimitado (en principio al número de nodos disponible), con el consiguiente incremento de su rendimiento (prácticamente en forma lineal) y sin cuellos de botella.

Existe gran cantidad de bibliografía sobre la comparación MR y MPP; no obstante, muchas veces la solución no proviene de uno u otro paradigma, y dado que se complementan bien en cuanto a las prestaciones y los tipos de datos que obtienen mejores rendimientos, es habitual que se utilicen arquitecturas mixtas donde generalmente se aplica primero MR sobre datos no estructurados y, posteriormente, MPP para procesar y visualizar los datos estructurados generados por el primero.

Lectura recomendada

A. Grishchenko. «Hadoop vs MPP». *Distributed Systems Architecture*. https://0x0fff.com/hadoop-vs-mpp/

2. El entorno Hadoop

El proyecto Apache™ Hadoop® es una plataforma *open source* para el cómputo distribuido, confiable y escalable y utilizado por una gran cantidad de empresas e instituciones, que implementa el algoritmo de MapReduce.

> Es un entorno para el procesamiento distribuido de grandes conjuntos de datos por medio de clústeres de cómputo; utiliza modelos de programación sencillos y tiene la posibilidad de escalar desde servidores individuales a miles de procesadores.

Básicamente, Hadoop (V2.x o V3.x) está formado por cuatro módulos:

1) HadoopYARN: marco para la programación de tareas y gestión de recursos del clúster.

2) Hadoop MapReduce: sistema basado en YARN para el procesamiento paralelo de grandes conjuntos de datos.

3) Hadoop Distributed FileSystem (HDFS): sistema de archivos distribuido que proporciona acceso de alto rendimiento a los datos de la aplicación.

4) Hadoop Common: las utilidades comunes que soportan los otros módulos de Hadoop.

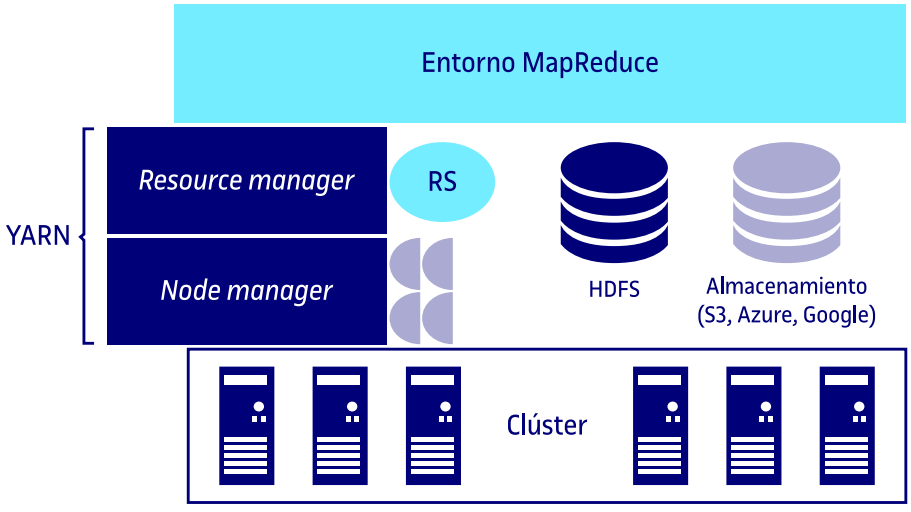
No obstante, Hadoop se puede relacionar, integrar o ampliar de forma fácil y rápida con los siguientes proyectos (listamos algunos de los más habituales o referenciados):

- **Ambari**: herramienta web para el aprovisionamiento, la gestión y la monitorización de clústeres Hadoop con HDFS, MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig y Sqoop.
- **Avro**: serialización de datos.
- **Cassandra**: base de datos escalable sin puntos de fallo únicos.
- **Chukwa**: recopilación de datos en sistemas distribuidos.
- **Drill**: *SQL query engine* de baja latencia que soporta aplicaciones distribuidas para el análisis interactivo de *datasets*.

- **Flume**: sistema distribuido para recoger, agregar y mover grandes cantidades de datos desde diferentes fuentes y también de *datastores* centralizados.
- **HBase**: base de datos orientada a columnas distribuida y escalable.
- **Hive**: almacén de datos que proporciona resumen de datos y consultas *ad hoc*.
- **Mahout**: entorno de *machine learning* y *data mining*.
- **Pig**: lenguaje de flujo de datos de alto nivel.
- **Spark**: motor de cálculo en memoria y especializado en procesar datos de *streams* (flujos de datos).
- **Sqoop**: herramienta para transferir datos entre Hadoop y bases de datos estructuradas, como por ejemplo bases de datos relacionales.
- **Tez**: entorno de programación de flujo de datos generalizado para grafos dirigidos.
- **ZooKeeper**: servicio de coordinación de alto rendimiento para aplicaciones distribuidas.

La relación entre los módulos de la arquitectura Hadoop con MapReduce (para más detalles consultar la documentación), quedan representados en la figura siguiente:

Figura 2. Arquitectura Hadoop con MapReduce



Donde:

1) Entorno MapReduce: capa de software que permite la ejecución de aplicaciones bajo este paradigma.

2) YARN (*Yet Another Resource Negotiator*): es la parte responsable de gestionar los recursos (CPU, memoria, etc.) que utilizarán las aplicaciones; se compone de dos grandes módulos:

a) *Resource manager*, RM (uno por clúster), que actúa como supervisor y conoce dónde están ubicados los *workers* y de cuántos recursos dispone. Incluye una serie de servicios, pero el más importante es el ***resource scheduler*** (RS), que decide cómo y a quién asignarlos.

b) *Node manager*, NM (>1 por clúster), es el *worker* de la infraestructura y mantiene informado al *resource manager* sobre su estado; gestiona la memoria y *vcores* que pueden ser asignados bajo la gestión de RS, que decidirá cómo se utiliza esta capacidad por medio de fracciones de la capacidad del NM llamadas *containers*.

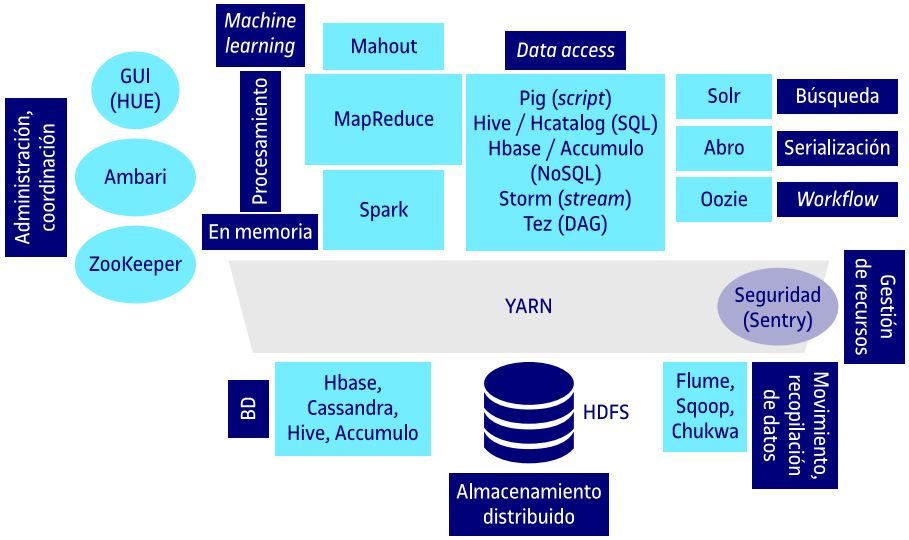
3) *HDFS Federation*: es la parte del entorno responsable de proveer almacenamiento permanente, fiable y distribuido; es utilizado para almacenar las entradas y salidas de la aplicación (no las intermedias). Se puede complementar con otras opciones de almacenamiento *cloud* (por ejemplo, S3, Google StorageObjects, etc.).

4) Clúster: es el conjunto de nodos de la infraestructura. Es importante destacar que la infraestructura YARN y el HDFS están totalmente desacoplados y sobre ellos se podrían ejecutar otros entornos. En el ciclo de vida de la aplicación, gestionado por YARN, intervienen tres módulos: *job submitter* (cliente), *resource manager* (supervisor o máster) y *node manager* (*worker*), por lo cual el *workflow* de una aplicación sería:

a) el cliente envía la aplicación al RM,
b) este sobre la base de la información de RS,
c) asigna el contenedor,
d) que además contacta con el *node manager*,
e) que a su vez lanza el contenedor
f) y este ejecuta la aplicación.

La figura siguiente muestra un entorno Hadoop con sus actores más importantes y su rol (uno de los posibles) dentro de la plataforma.

Figura 3. Entorno Hadoop



2.1. Casos de uso con Hadoop y Linux

Como primera prueba (luego se verán contenedores y otras distribuciones y métodos preinstalados) para la instalación de Hadoop se recomienda seguir las indicaciones de la página de los desarrolladores instalando lo que se llama *single cluster* (es decir, todos los componentes funcionando sobre un sistema que puede ser una máquina virtual). Para ello primero se debe verificar e instalar software necesario (ssh, java...), descargar la distribución de Hadoop y configurar lo que se denomina *pseudo-distributed operation,* ya que todos los componentes estarán en el mismo nodo. Con esta configuración ya se podrán probar todos los ejemplos que hay a continuación, pero si se desea configurar un clúster completo se pueden seguir las indicaciones de Hadoop Cluster Setup, que es la recomendada para entornos de procesamiento en producción.

Para esta prueba de concepto (analizar la funcionalidad de Hadoop) se utilizará el código MR para contar palabras, datos del tiempo y datos (*open data*) de recetas médicas utilizando la instalación indicada previamente preparada en una MV. El lenguaje habitual de trabajo en Hadoop es Java, y se probarán algunos ejemplos en este lenguaje, pero se dedicará tiempo a trabajar con Python para procesar los datos con un lenguaje que es utilizado en muchas disciplinas.

El «truco» para procesar código Python, como se mencionó anteriormente, es utilizar la Hadoop Streaming API, que permitirá pasar datos entre *mapper.py* y *reducer.py* por medio de STDIN (entrada estándar) y STDOUT (salida estándar). Para ello solo se deberá utilizar la librería *sys.stdin* de Python para leer los datos de entrada y generar los datos de salida con la librería *sys.stdout* y Hadoop Streaming se encargará de todo lo demás.

2.1.1. Ejemplo de WordCount (Java)

Para este ejemplo se utilizará el código Java provisto en MapReduce Tutorial, que utiliza las interfaces Java de Hadoop + HDFS + Yarn *framework* para contar palabras. Para simplificar la ejecución, este ejemplo se desarrollará sobre una única MV, pero con todo el entorno completo de Hadoop y su ecosistema, el cual puede rápidamente extenderse a un clúster (incluso de MV) para hacer un procesamiento con mayores prestaciones, que para esta prueba de concepto no es necesario.

Sobre la MV se encontrará un usuario **Hadoop** y en el *home* de este usuario un directorio ***wordcount*** con un archivo de código (WordCount.java) y dos archivos de datos (alice.txt y war-and-peace.txt). En el directorio $HOME/hadoop (/home/hadoop/hadoop) se encuentra instalado y configurado todo el software de Hadoop y en $HOME/.bashrc todas las variables de entorno para ejecutar Java y Hadoop.

1) Para comenzar se deben inicializar los procesos que soportan todo este cómputo (llamados *daemons*) ejecutando en un terminal:

```
start-dfs.sh y start-yarn.sh
```

Se puede observar abriendo un navegador el estado del sistema de procesamiento poniendo como URL: http://127.0.0.1:50070.

Los trabajos (*jobs*) enviados a procesar se pueden consultar en la URL: http://127.0.0.1:8088/cluster.

2) Para poder ejecutar el código Java se debe compilar (transformar el código del programa en código ejecutable por el procesador) con:

```
hadoop com.sun.tools.javac.Main WordCount.java
```

Esto generará un conjunto de archivos que se deben compactar en un único archivo llamado, en este ejemplo, *wc.jar*:

```
jar cf wc.jar WordCount*.class
```

3) A continuación se debe subir el archivo de datos a procesar (alice.txt) al sistema de archivos distribuido que «acercará» los datos a cada nodo de datos del entorno de ejecución:

```
hdfs dfs -put alice.txt input/alice.txt
```

4) Finalmente se podrá ejecutar con Hadoop:

```
hadoop jar wc.jar WordCount input/alice.txt output
```

5) Y se podrá analizar la salida con:

```
hdfs dfs -cat /output/*
```

Después se podrá repetir el mismo proceso con otro libro de mayor tamaño (war-and-peace.txt) y solo será necesario subir este al HDFS y volver a ejecutar cambiando solamente el archivo de entrada.

6) Comandos útiles:

- Cómo inicializar los directorios del HDFS necesarios para subir los datos a procesar por Hadoop: `hadoop namenode -format`
- Cómo iniciar los *Hadoop daemons*: `start-dfs.sh start-yarn.sh`
- Cómo crear un nuevo directorio en el HDFS:
  `hdfs dfs -mkdir /hduser/input`
- Cómo obtener la lista de comandos para trabajar con el HDFS: `hdfs dfs`
- Cómo copiar un archivo en mi directorio local al HDFS:
  `hdfs dfs -put /home/hadoop/file.txt input/file.txt`
- Cómo listar los archivos en el HDFS: `hdfs dfs -ls`
- Cómo ejecutar un programa:
  `hadoop jar file.jar MainClass input/input.txt output`
- Cómo visualizar los datos de salida generados por un programa en el HDFS:
  `hdfs dfs -cat output/*`
- Página de administración de Hadoop: `http://localhost:50070/`
- Página de administración de Yarn: `http://127.0.0.1:8088/cluster`
- Cómo terminar una sesión y apagar todo el entorno Hadoop:
  `/home/hadoop/Hadoop/sbin/stop-all.sh`

2.1.2. Ejemplo MapReduce Weather

Muchas aplicaciones basadas en datos generan grandes *datasets* que pueden ser descargados y en este caso se utilizarán las del National Centers for Environmental Information (NCEI) en un ejemplo que es mencionado en el libro *Hadoop: the Definitive Guide* cuyo código de ejemplo se encuentra en https://

github.com/tomwhite/hadoop-book (en este caso se utilizará el código Java del apartado 2). Este *dataset* está organizado por años y diferentes archivos de cada estación meteorológica donde cada registro tiene una serie de información indicada en el documento «Local Climatological Data (LCD) Dataset Documentation».

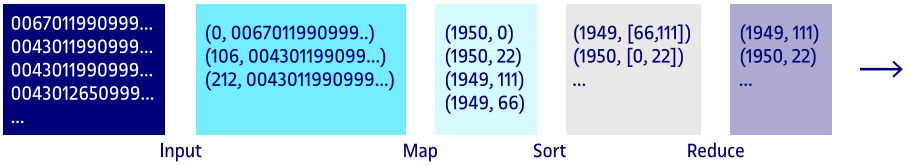
Por ejemplo, los datos del 2020 ocupan 4,1 GB (www.ncei.noaa.gov/data/local-climatological-data/archive/). El inventario de las estaciones, su ubicación y el mapa se encuentran en www.ncei.noaa.gov/products/land-based-station/station-histories.

Lectura recomendada

T. White (2015). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale* (4a edición, cap. II). Sebastopol: O'Reilly Media.

El problema a resolver es encontrar la temperatura máxima de cada año en el *dataset* de una estación, por lo cual deberemos centrar el estudio en la fecha y la temperatura de cada registro en un esquema similar al siguiente:

Figura 4. Esquema de registro



De las líneas que tienen un formato como:

(0, 006701199099999919500515070 04...9999999N9+00001+99999999999...)

(106, 004301199099999919500515120 04... 9999999N9+ 00221+ 99999999999...)

La función *map* solo se queda con el año y la temperatura y genera como resultado pares (1950, 0)(1950, 2)(1950, -11)(1949, 111)(1949,78)... y antes de enviar a la función *reduce* se agrupan y ordenan como (1949, [111,78]) (1950, [0,22,-11].

Con esto la función *reduce* iterará sobre la lista de cada año buscando la temperatura mayor y generará como resultado (1949, 111)(1950, 22) (recordad que las temperaturas están en ºF y por eso hay valores como 111F = 43º C).

1) Para compilar:

```
export CL=`hadoop classpath`
javac -classpath $CL -d maxTemperature_classes MaxTemperature.java MaxTemperatureMapper.java
MaxTemperatureReducer.java
```

2) Crear el archivo único (*jar*):

```
jar -cvf maxTemperature.jar -C maxTemperature_classes.
```

3) Poner el archivo de datos (data.txt) en el HDFS:

```
hdfs dfs -put data.txt /input/data.txt
```

Se pueden bajar los datos de interés desde las URL indicadas anteriormente o desde Github (donde se encuentran los ejemplos de *Hadoop: the Definitive Guide*).

4) Para ejecutar:

```
hadoop jar maxTemperature.jar MaxTemperature input/data.txt output
```

5) Consultar los resultados:

```
hdfs dfs -cat output/*
```

6) Utilizando las URL anteriores y el procedimiento indicado, se puede filtrar y procesar, por ejemplo, la temperatura máxima de la estación 010100 que contiene 715175 registros, y que resulta un ejercicio interesante para procesar y analizar, dada la magnitud de los datos.

2.1.3. Ejemplo de recetas médicas (Python)

Para este caso de uso se analizarán datos del archivo de recetas de Gencat mediante un entorno Hadoop (y que no es posible procesar por las herramientas habituales como hojas de cálculo por su tamaño).

Para cumplir con este objetivo se dispondrá del archivo de datos bajado de la plataforma antes mencionada con todas las recetas vendidas en Cataluña por el CatSalut, y que se han bajado de los datos originales en formato CSV, que se llamará *rece.csv*. También se desarrollará la función *mapper-rec.py* y *reduce-rec.py* y un programa (*shell-script*) llamado *to-execute* con la sintaxis para llamar a Hadoop con código Python.

Uno de los principales problemas a resolver es que, dentro del código Python, deberemos tener en cuenta que los campos son, por ejemplo:

> 2021,02,61,LLEIDA,0-1 any,Dona,A,TRACTO ALIMENTARIO Y METABOLISMO,A01, PREPARADOS ESTOMATOLOGICOS,A01A,PREPARADOS ESTOMATOLOGICOS, A01AB,Antiinfecciosos y antisepticos para el tratamiento oral-local,6,6,16.98,9.91

Donde se pueden ver los campos separados por «,», pero existen registros como por ejemplo el que está marcado:

> 2021,02,61,LLEIDA,0-1 any,Dona,P,"**PRODUCTOS ANTIPARASITARIOS, INSECTICIDAS Y REPELENTES**",P01,ANTIPROTOZOARIOS,P01A,AGENTES CONTRA LA AMEBIASIS Y OTRAS ENFERMEDADES POR PROTOZOARIOS,P01AB,Derivados del nitroimidazol,1,1,2.17,1.30

Donde se puede observar que hay el campo «PRODUCTOS ANTIPARASITARIOS, INSECTICIDAS Y REPELENTES» que incluye una «,» dentro del campo y que dificultará su procesamiento por Python, ya que no será tan fácil separar por campos. En este caso se podrá hacer sin problemas, ya que el *string* está entre «...».

El código comentado del *mapper-rec.py* es:

```
#!/usr/bin/env python3

import sys # las entrada vendrán de la STDIN (standard input)
for line in sys.stdin:

    line = line.strip()  # quitar espacios en blanco al inicio y final de la línea

    if line[0] == "a":  # quitar la cabecera que comienza por any
        continue

    words = line.split(",")        # separar los campos por ‘,’

    wordFinal = []  # array de campos finales (incluidos los separados por , dentro
    union = ""   # string temporal para ir juntando las partes de los campos
    for word in words:  # para todos los campos
        if word[0]=='"':  # si el campo comienza por “?
            union = word.replace(word[0], "")  # se agrega a unión y se quita la “ y continua
            continue
       if (union != "") and word[len(word)-1] !='"': # si ya se tienen las partes y no se ha llegado
       # al final se agrega
            union = union + word
       elif (union != "") and word[len(word)-1] =='"': # si ya se tienen partes y se ha llegado al
       # final, se quita y agrega al wordFinal
            word = word.replace('"', "")
            wordFinal.append(union + word)
            union = ""     # se borra union para el próximo
        else:
            wordFinal.append(word)    # en caso contrario se agrega
    #print ("line", line)
    #print ("-->", wordFinal)
    #print ("--->", wordFinal[13])
    wordsNoWhite = wordFinal[13].replace(" ", "_")  #cambia los espacios en blanco por _
    print ('%s\t%s' % (wordsNoWhite, 1))   # se generan las tuplas con el campo 13.
          # descripción de ATC4
```

El código del *reducer-rec.py* es exactamente igual al comentado anteriormente:

```
#!/usr/bin/env python3
```

```
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None

for line in sys.stdin:  # input comes from STDIN

    line = line.strip()  # remove leading and trailing whitespace
        # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
        # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
        continue
    # this IF-switch only works because Hadoop sorts map
    # output by key (here: word) before it is passed to the
    # reducer
    #print (word,count)
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
# do not forget to output the last word if needed!
if current_word == word:
    print ('%s\t%s' % (current_word, current_count))
```

Para ejecutar este código primero se puede probar simulando el entorno Hadoop simplemente cambiando los permisos al código y ejecutando:

```
chmod +x mapper-rec.py
chmod +x reducer-rec.py
cat rece.csv |./mapper-rec.py | sort -k1,1 |./reducer-rec.py
```

Que dará una lista como:

Acido_aminosalicilico_y_agentes_similares 14461

Acido_ascorbico_(vitamina_C)_monofarmaco 924

Acido_folico_y_derivados 17652

Acido_salicilico_y_derivados 6609

Acidos_biliares_y_derivados 13294

...

Para ejecutar en el entorno Hadoop, se deben seguir los siguientes pasos:

1) Poner en marcha el entorno: `start-dfs.sh` y a continuación ejecutar `start-yarn.sh`.

2) Subir el archivo rece.csv al HDFS al directorio *input*:

```
hdfs dfs -put rece.cvs input
```

3) Se puede verificar que está todo bien con `hdfs dfs -ls input/*`.

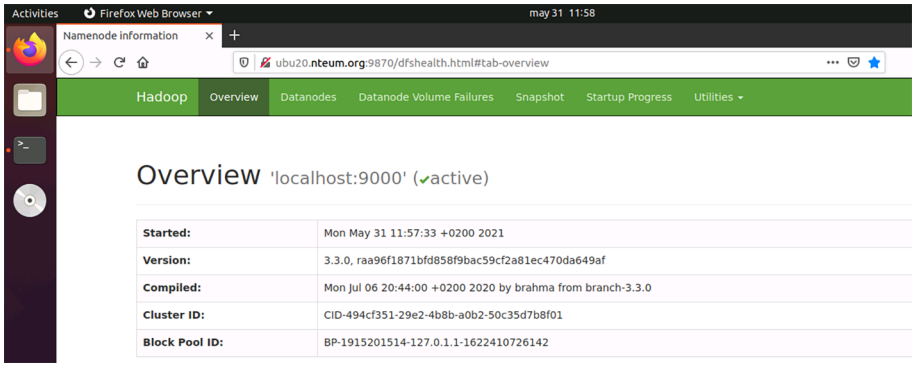
4) Finalmente ejecutar:

```
hadoop jar /home/hadoop/etc/hadoop/tools/lib/hadoop-streaming-3.3.0.jar -mapper "mapper-rec.py"
-file./mapper-rec.py -reducer "reducer-rec.py" -file./reducer-rec.py -input "input/rece.csv"
-output output
```

Como la sintaxis es compleja y se pueden cometer errores, se puede hacer un *script,* que es un comando en un archivo que simplifica su ejecución, llamado, por ejemplo, `to-execute`. Se le deben agregar los permisos de ejecución con `chmod +x to-execute` y luego se podrá ejecutar como: `./to-execute`.

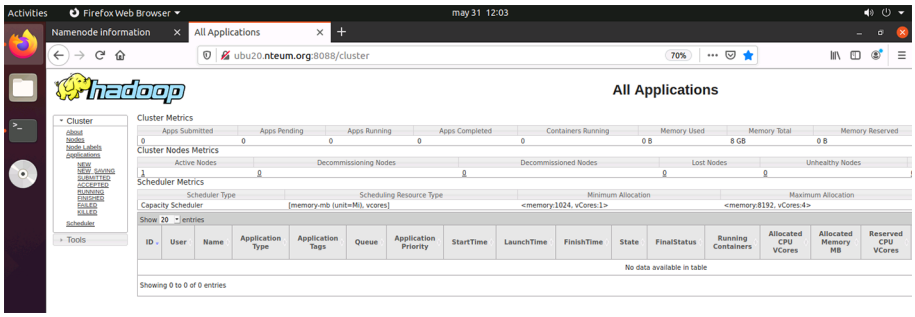
5) EL estado de HDFS se puede observar abriendo un navegador y poniendo como URL: localhost:9870 o el dominio de la máquina (en este caso ubu20.nteum.org), que mostrará una salida como la siguiente:

Figura 5. Estado de HDFS



También se puede ver la página de Hadoop en aplicaciones, indicando como URL (y en este caso debe ser el *nombre.dominio* de la máquina): **ubu20.nteum.org:8088**.

Figura 6. Aplicaciones de Hadoop



6) Para ver los resultados: `hdfs dfs -cat output/*`.

7) Para eliminar los resultados (es necesario para hacer otras pruebas sobre el mismo directorio de salida, ya que por protección no deja sobreescribir los directorios de salida): `hdfs dfs -rm output/*` y a continuación el directorio `hdfs dfs -rmdir output`.

8) Finalmente, cuando se hayan acabado las pruebas se deberá parar el entorno con: `stop-dfs.sh` y a continuación `stop-yarn.sh`.

3. El entorno Spark

Spark es un motor de procesamiento optimizado de datos en memoria que puede ejecutar operaciones con datos de Hadoop (en HDFS) y con un mejor rendimiento que MapReduce.

También puede procesar datos en otros entornos (como clústeres) y, al minimizar las lecturas y escrituras de disco, este ofrece muy alto rendimiento a colecciones de datos dinámicas y complejas y brinda funcionalidad interactiva como soporte de consultas específicas (*ad hoc*) de una manera escalable. Spark extiende el paradigma de procesamiento estático de datos almacenados y por secuencias (lotes o *batch*), que es el que utiliza MapReduce, hacia aplicaciones dinámicas y en tiempo real o sobre flujos de datos. Spark soporta programas en diferentes lenguajes de programación, que incluyen Java, Scala, Python y R y permite interactuar con datos incluidos en bases de datos SQL.

Si bien MapReduce ha sido durante años la metodología de desarrollo de software para el procesamiento distribuido por lotes (*batch*) de *big data*, este tipo de procesamiento no es la respuesta para todas las situaciones computacionales. Con el interés de buscar mejores prestaciones a MapReduce, los investigadores comenzaron a elaborar nuevas propuestas y dos de las más significativas han sido **Apache Drill** y **Spark**.

Drill es un motor de lenguaje de consulta estructurado (SQL) de alto rendimiento destinado a proporcionar funciones de exploración de datos, y Spark fue diseñado para ser un motor de procesamiento de propósito general para tareas interactivas, por lotes y de flujo que eran capaces de aprovechar los mismos tipos de recursos de procesamiento distribuidos que hasta ahora habían impulsado las iniciativas MapReduce.

No han sido los únicos intentos de mejorar las prestaciones sobre *big data,* ya que también se pueden mencionar los siguientes:

- **Apache Pig** (un lenguaje MapReduce especializado de alto nivel) desarrollado por Yahoo! para facilitar el acceso a los datos almacenados en Hadoop.

- **Apache Hive**, que aportó el poder de SQL a MapReduce como resultado de la investigación de Facebook para interacciones de datos más interactivas y basadas en SQL.

- **Apache Impala**, como entorno de consultas al estilo de la analítica de negocios para Hadoop.

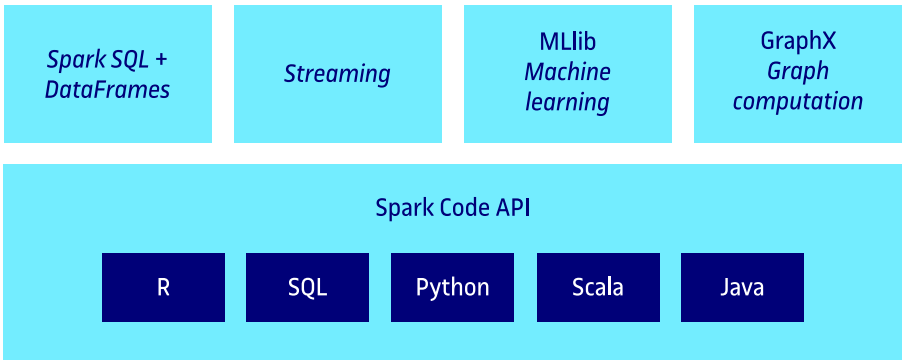
- **Presto**, como un motor de procesamiento distribuido de alta velocidad para ejecutar consultas SQL en bases de *big data*.

Partiendo desde el diseño, Spark estaba orientado a explotar el potencial, en particular la velocidad y la escalabilidad, ofrecido por procesamiento *in-memory* de datos, ya que el algoritmo de MapReduce tiende a escribir continuamente datos intermedios en el disco a lo largo de un ciclo de procesamiento. En este sentido, Spark puede ofrecer resultados con la misma configuración de datos hasta cien veces más rápido que MapReduce.

Además, sus premisas de diseño se han concentrado en ofrecer **rendimiento, simplicidad, facilidad de administración y desarrollo más rápido de aplicaciones,** por ello una gran cantidad de organizaciones y empresas lo han adoptado rápidamente.

Como se mencionó anteriormente, Spark utiliza un modelo simple de programación que puede ser en Python, R, Scala o Java y dispone de *application programming interfaces* (APIs) bien definidas para diferentes casos de uso como: Streaming, SQL, ML, DataFrames y Graph Processing. La figura siguiente muestra el ecosistema de Spark:

Figura 7. Ecosistema de Spark



3.1. Casos de uso comunes para Spark

Considerando sus características, como velocidad, simplicidad y productividad para el desarrollador, existen una gran cantidad de dominios que se han volcado a trabajar con Spark. Como por ejemplo: aplicaciones con flujo de datos (*streaming*), inteligencia artificial, aprendizaje automático y profundo, inteligencia empresarial o integración de datos.

Un caso específico dentro del ámbito de la salud es el procesamiento de datos generados *on-line* de monitores y sensores de un paciente, que permiten disponer de un registro continuo y escalable de grandes cantidades de informa-

ción para detectar variaciones o cambios en la información monitorizada para generar alertas, informes o registros simplemente de las partes que se consideran esenciales para guardar.

Otro dominio en el que Spark comienza a dar sus frutos es en el ámbito de la inteligencia artificial (IA), el aprendizaje automático (ML) y el aprendizaje profundo (DL), ya que las organizaciones comienzan cada vez más a utilizar estas técnicas y algoritmos para hacer frente a su estrategia de análisis de datos. En cualquier caso de uso asociado con IA, ML y DL se encuentran capacidades sofisticadas de clasificación y reconocimiento de patrones que sirven a aplicaciones de decisión basadas en datos. Pero se debe tener en cuenta que, según la opinión de expertos, las soluciones de IA necesitan de ocho a diez veces el volumen de datos utilizado que para las soluciones actuales de *big data*. Estos datos provienen de muchas fuentes, como datos de sistemas de gestión corporativa (ERP), bases de datos, *datalakes*, sensores, datos públicos, aplicaciones móviles, redes sociales y datos heredados y que pueden ser estructurados o no estructurados y de diversos formatos, con lo que el procesamiento en memoria y los flujos de datos altamente eficientes de Spark lo convierten en un motor de análisis ideal para la preparación, transformación y manipulación de datos en este tipo de proyectos basados en IA.

Spark proporciona el rendimiento necesario para permitir el análisis *on-line* (u *on-the-fly*) de sensores y monitores de señales que, en un escenario de seguimiento y vigilancia de pacientes, por ejemplo, permite una combinación de análisis por un sistema de información y por profesionales que pueden tomar decisiones más rápidas y precisas.

3.2. Estructura de los datos

Un aspecto importante en *big data* es vincular y gestionar datos de diversas fuentes. Esta tarea es ardua y requiere mucha dedicación y cómputo intensivo; este proceso se conoce como extraer, transformar y cargar (*extract, transform, and load* – ETL). Spark incluye mecanismos para realizar la integración de proveedores y plataformas de tecnología de almacenamiento de datos y permite realizar este proceso de forma rápida y sencilla por su procesamiento en memoria y sin almacenar los datos en el disco duro, lo cual logra eficiencias desde x10 hasta x100 en relación con las aplicaciones clásicas de MapReduce basadas en disco.

Para lograr estos resultados, Spark trabaja con diferentes estructuras de datos que han ido evolucionando en el tiempo: *resilient distributed datasets* (RDD), *dataframes* y *datasets*.

Inicialmente, en 2011 Spark fue desarrollado con RDD, pero viendo el auge de Spark en otros ámbitos, los desarrolladores implementaron *dataframes* en 2013 y *datasets* en 2015.

Los **conjuntos con resiliencia de datos distribuidos (*resilient distributed datasets*, RDD)** son la estructura de datos fundamental (inicial) de Spark. Permite almacenar los datos distribuidos en los múltiples nodos del clúster y hacer el procesamiento en paralelo. Es tolerante a fallos, ya que si se realizan múltiples transformaciones en el RDD y luego, por cualquier motivo, hay un fallo en algún nodo, el RDD es capaz de recuperarse automáticamente.

En más detalle son estructuras en memoria destinadas a contener la información que se desea cargar y luego procesar por Spark. Se las etiqueta «con **resiliencia**» porque mantienen un registro de su historial (por ejemplo, modificaciones y eliminaciones) para que puedan reconstruirse si hay fallos durante su procesamiento, y además son **distribuidos** porque el *dataset* se particiona y se reparte entre los diferentes nodos de la arquitectura subyacente del clúster para aumentar la eficiencia por medio del procesamiento paralelo.

Una aplicación de Spark puede crear una estructura RDD y luego cargar datos que pueden provenir de casi cualquier fuente de datos, como por ejemplo: Hadoop, Apache Cassandra, Amazon S3, Apache HBase, bases de datos relacionales, entre otros. Una vez que los datos se han colocado en el RDD, para crear un RDD se puede hacer en tres formas:

1) paralelizar una colección de datos existente,

2) leer un archivo de datos externo, o

3) crear RDD a partir de un RDD ya existente, por ejemplo:

```
# 1) paralelización de la recopilación de datos
my_list = [1, 2, 3, 4, 5, 6]
my_list_rdd = sc.parallelize (my_list)


# 2)Referencia a un archivo de datos externo
file_rdd = sc.textFile ("ruta_del_archivo")
```

Una pregunta que nos podemos hacer es: ¿cuándo utilizar un RDD? Es recomendable utilizar el RDD en las siguientes situaciones:

- Cuando se desea hacer transformaciones de bajo nivel en el conjunto de datos (más adelante se verán transformaciones sobre los RDD).

- Cuando los datos cargados no tienen un esquema y es necesario especificar manualmente el esquema de cada conjunto de datos cuando se crea el RDD.

Los ***dataframes*** se integraron en Spark a partir de la versión 1.3 para superar las limitaciones del RDD. Spark Dataframes es la colección distribuida de datos, pero organizados en las columnas nombradas (admiten cabeceras), y permite a los desarrolladores depurar el código durante el tiempo de ejecución, lo que no es posible con los RDD.

Los *dataframes* pueden leer y escribir los datos en varios formatos como CSV, JSON, AVRO, HDFS y tablas HIVE, y están optimizados para procesar grandes conjuntos de datos para la mayoría de las tareas de preprocesamiento, por lo que no es necesario que el desarrollador escriba funciones complejas. Para crear un *dataframe,* por ejemplo, en PySpark:

```
spark.createDataFrame (
 [
 (1, 'Pedro'),
 (2, 'Ana'),
.
.
.
.
 (100, 'Alicia')
 ],
 ['id', 'Nombre'] # etiqueta de las columnas
)
```

Los ***datasets*** son una extensión de los *dataframes* que incluyen sus beneficios y los de los RDD al permitir procesar de manera eficiente datos estructurados y no estructurados. Es una estructura que permite un procesamiento rápido y proporciona una interfaz segura, ya que el compilador de Spark validará los tipos de datos de todas las columnas del conjunto de datos durante la compilación únicamente y arrojará un error si hay alguna discrepancia en los tipos de datos. Quien utilice RDD encontrará que el código es muy similar, pero notará un incremento notable en la velocidad de procesamiento. Como se debe verificar el código durante la compilación, los *datasets* solo están disponibles para Scala y Java (no para PyPython).

Entre las características principales de las tres estructuras de datos se pueden mencionar:

Tabla 1. Características principales de las estructuras de datos

	RDD	***Dataframe***	***Dataset***
Representación	Colección de datos distribuida sin esquema	Colección de datos distribuida organizada en columnas con nombre	Extensión de *dataframes* con control de los tipos de datos
Optimización	Sin optimización	Optimizado	Optimizado
Esquema	Esquema manual	Búsqueda automática del esquema	Ídem *dataframe* pero con una SQL Engine
Operaciones	Lento para realizar operaciones como por ejemplo agrupar datos	Realiza agregaciones más rápido que RDD y *datasets*	Mejor que RDD pero peor que *dataframes*

Los *dataframes* permiten un conjunto de **operaciones** para la manipulación de datos estructurados (a continuación se muestran algunas de ellas). Antes de comenzar a trabajar con un *dataset* es necesario crear una SparkSession y estos parámetros se crean automáticamente si se accede por *pyspark*:

Ved también

En la bibliografía de este módulo encontraréis más información sobre manipulación de datos estructurados.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Spark Training').getOrCreate()
sc = spark.sparkContext
```

Para crear un *dataframe* desde un archivo CSV:

```
df = spark.read.format('csv').options(delimiter=',', header=True).load('/carpetas/archivo.csv')
```

Para convertir RDD a *dataframe*:

```
rdd = sc.parallelize([(1,2),(3,4),(5,6)])
rdf = rdd.toDF()
```

Para crearlo utilizando el método spark.createDataFrame:

```
tdf = spark.createDataFrame([('Pedro',24),('Ana',43)],['nombre','edad'])
tdf.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

Para mostrar las columnas del *dataframe*:

```
tdf.columns
['nombre', 'edad']
```

Para mostrar los datos de un *dataframe*:

```
df.show(5, truncate=False)
+------------+---------+
|nombre      |   edad  |
+------------+---------+
|Pedro       |   24    |
|Ana         |   43    |
+------------+---------+
```

Para los RDD, Spark permite dos tipos principales de operaciones: **transformaciones** y **acciones**.

1) Las **transformaciones** incluyen filtrado, mapeo o manipulación de los datos, generando un nuevo RDD, ya que los RDD son inmutables, es decir que el RDD original permanece sin cambios. Esta inmutabilidad permite que Spark realice un seguimiento de las modificaciones que se llevaron a cabo durante la transformación y, por lo tanto, deshace los cambios posteriores que se han realizado. Otro punto interesante del trabajo con RDD es que las transformaciones son «perezosas» (*lazy*), es decir, no se ejecutan hasta el momento en que se necesitan, por ejemplo, mediante una acción. Este comportamiento ayuda a mejorar el rendimiento al retrasar los cálculos hasta el momento en que se requieren, en lugar de realizar cálculos que podrían no ser necesarios según el flujo del programa.

2) Las **acciones** son funciones para obtener información de los datos, pero no los modifican, como por ejemplo Recuento, Recuperación de un elemento específico, Agregación (unión de información, no inserción). Cuando una aplicación solicita una acción, Spark evalúa el RDD inicial y luego crea instancias de RDD en función de lo que se le pide que haga, mirando qué lugar del clúster será el más adecuado, desde el punto de vista de la eficiencia, para colocar el nuevo RDD.

Como se ha mencionado, las transformaciones generan un nuevo RDD y entre las más utilizadas se pueden encontrar las siguientes:

- **map(func)**: retorna un nuevo RDD pasando cada elemento de la fuente por medio de la función *func*.

- **filter(func)**: retorna un nuevo RDD seleccionando aquellos elementos de la fuente cuyo *func* retorna *true*.

- **flatMap(func)**: similar a *map*, pero cada entrada puede ser mapeada a 0 o más salidas, lo que podría retornar una secuencia en lugar de un solo ítem.

- **union(otherDataset)**: retorna un nuevo *dataset* con la unión de los dos.

- **distinct([numTasks]))**: retorna un nuevo RDD que contiene aquellos elementos diferentes del RDD original.

- **groupByKey([numTasks])**: si el RDD está organizado por claves y valor (K, V) retorna un nuevo *dataset* agrupado por K (K, Seq[V]).

- **reduceByKey(func, [numTasks])**: genera un nuevo RDD donde los valores para cada clave están agregados por la *func*.

- **sortByKey([ascending], [numTasks])**: retorna un nuevo RDD ordenado.

Entre las acciones más importantes se encuentran:

- **reduce(func)**: agrega los elementos del *dataset* utilizando *func*.

- **collect()**: retorna todos los elementos del *dataset*.

- **count()**: retorna el número de elementos del *dataset*.

- **take(n)**: retorna un *array* con los primeros *n* elementos del *dataset*.

- **saveAsTextFile(path)**: escribe los elementos del *dataset* como un archivo de texto en el sistema de archivos local, en HDFS o cualquier otro sistema de archivos soportado por Hadoop.

- **countByKey()**: solo disponible en RDDs del tipo (K, V) y retorna un mapa de (K, Int) tuplas para cada *key* K.

- **foreach(func)**: ejecuta una función *func* sobre cada elemento del *dataset*.

3.3. Instalación de Spark, PySpark y Spider

PySpark es una librería de Spark escrita en Python para ejecutar una aplicación Python usando las capacidades de Apache Spark, es decir, PySpark es una API de Python para Spark.

Spark básicamente está escrito en Scala, pero dada la rápida adaptación a entornos de ciencia de datos se desarrolló la API PySpark para Python usando Py4J (librería de Java integrada dentro de PySpark), y permite que Python interactúe de forma dinámica con objetos Java, por lo cual es necesario tener instalado Java (se recomienda versión 8) para ejecutar PySpark.

Es simple realizar la instalación de PySpark, ya que se pueden seguir las instrucciones de la página de los desarrolladores y ejecutarlo en un nodo o extenderlo en un clúster.

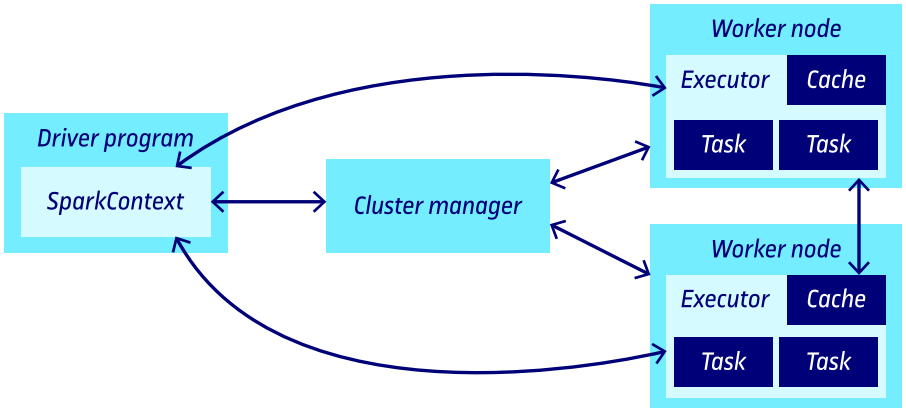
Para facilitar el desarrollo de programas en esta prueba de concepto se ha instalado sobre una máquina virtual (MV) en la cual se ha incluido, además de Spark y PySpark, un IDE (*integrated development environment*) llamado Spyder, que permite tener un entorno de edición, desarrollo y ejecución sobre PySark (también es muy utilizado Jupyter notebook).

Enlace recomendado

Para ver la configuración e integrar Spyder con PySpark se puede consultar la siguiente referencia: «Setup and run PySpark on Spyder IDE».

PySpark es una opción muy utilizada en ciencia de datos y aprendizaje automático, ya que hay numerosas bibliotecas de ciencia de datos escritas en Python que incluyen NumPy o TensorFlow, entre otras. Apache Spark funciona en una arquitectura maestro-trabajador (*master-worker*), en que el maestro se llama *driver* y los trabajadores *workers*. Cuando ejecuta una aplicación Spark, el *driver* crea un **contexto**, que es un punto de entrada a su aplicación, y todas las operaciones (transformaciones y acciones) se ejecutan en los nodos trabajadores, y el *cluster manager* administra los recursos (ver figura siguiente).

Figura 8. Contexto de Spark



Entre los *cluster managers* más utilizados por Spark se pueden enumerar los siguientes:

- **Standalone**: es un *cluster manager* simple incluido con Spark para una configuración fácil de un clúster.

- **Apache Mesos**: utiliza como *cluster manager* a Mesos, que permite ejecutar aplicaciones Hadoop MapReduce y PySpark (aunque los desarrolladores consideran que este método es obsoleto).

- **Hadoop YARN**: es el gestor de recursos de Hadoop 2 y normalmente es el *cluster manager* más utilizado.

- **Kubernetes**: es un entorno *open source* para automatizar, desplegar, gestionar y escalar aplicaciones en *containers* (Docker).

Enlace recomendado

Existe una gran cantidad de librerías y paquetes que permiten extender la funcionalidad de Spark y que se encuentran en https://spark-packages.org/

- Local: no es realmente un *cluster manager* pero es útil cuando se ejecuta Spark sobre un MV o un ordenador, y es el que se utilizará en estas pruebas de concepto. A local se le puede indicar un parámetro [k] para lanzar *k workers* (idealmente igual al número de *cores* de la máquina virtual –que se puede modificar con el parámetro VCPU en su creación– o física).

Después de instalado y configurado, Spark incluye, además de la consola a la que se puede acceder poniendo en un terminal **pyspark**, una interfaz web que permite analizar el desarrollo de los trabajos en ejecución poniendo como URL en un navegador http://localhost:4040. Esta interfaz mostrará, entre otra información: Jobs, Stages, Tasks, Storage, Environment, Executors y SQL para monitorizar el estado de la aplicación en ejecución como muestran las imágenes a continuación generada con la MV previamente mencionada.

Figura 9. Interfaz de Spark

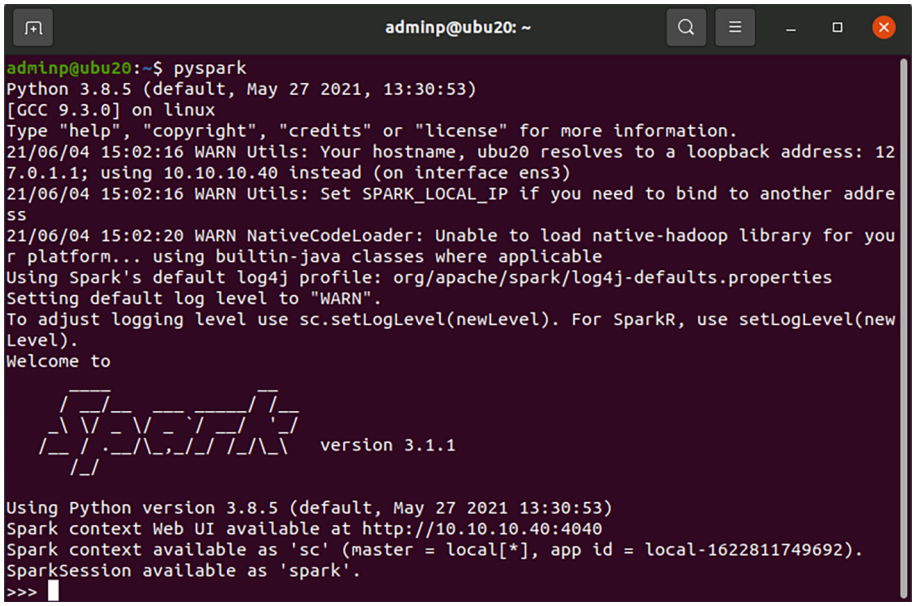
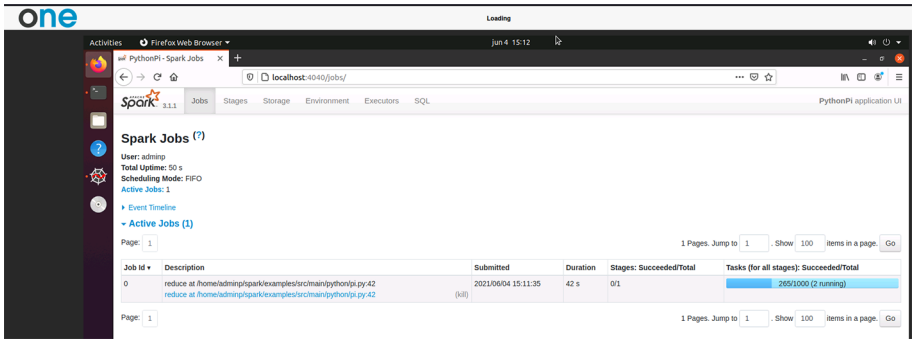


Figura 10. *Spark jobs*



¿Cómo se puede interactuar con Spark de forma fácil? Simplemente ejecutando pyspark y cuando se tenga el *prompt* >>> ejecutar, por ejemplo (véase que cuando se inicia pyspark ya genera el contexto como **sc** y crea una sesión como **spark**, por lo cual ya se puede entrar código Python para procesar los datos de forma interactiva):

Figura 11. Interactuar con Spark

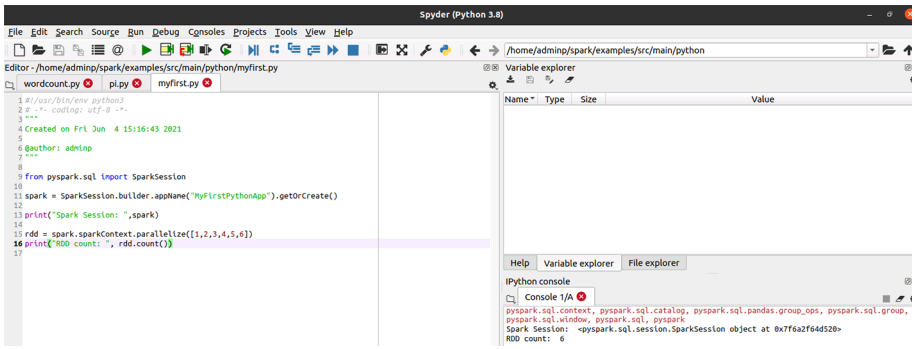
En este ejemplo se ha creado un vector de datos (*data*), luego se ha creado el RDD (*disDAta*) y paralelizado los datos, sobre los cuales luego se ha ejecutado una acción para contar los datos.

3.4. Programa en PySpark utilizando Spyder (IDE)

En la figura siguiente se puede ver un programa simple (*myfirst.py*), que se encuentra en el directorio /home/adminp/spark/examples/src/main/python/ y a continuación se describe cada una de las líneas:

- Línea 9: importación de las librerías.
- Línea 11: creación de una sesión.
- Línea 13: impresión de valor de la variable *spark*.
- Línea 15: creación de un RDD distribuido con un *array* de datos desde 1 a 6.
- Línea 16 impresión de una acción (*count*) sobre el RDD para contar el número de elementos del RDD.

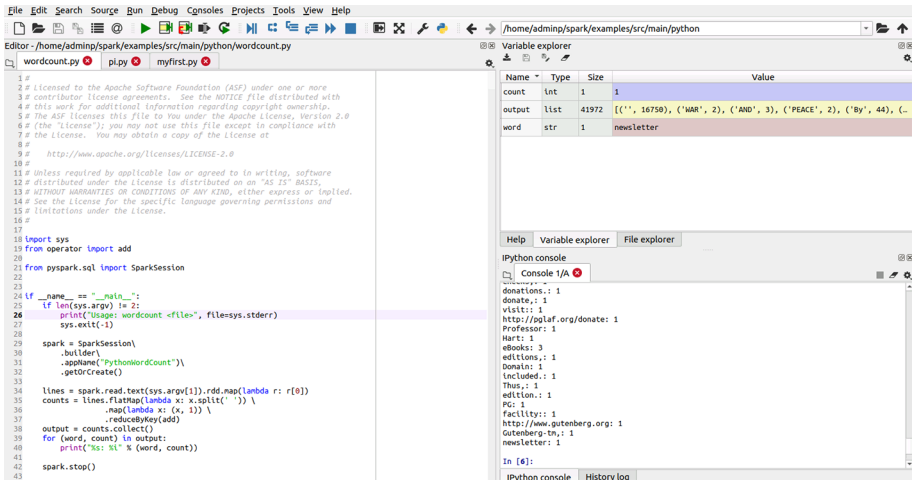
Figura 12. *myfirst.py*



Para ejecutar el programa en Spyder simplemente se abre el archivo y luego se clica la flecha de ejecución (la flecha de color verde del menú de iconos). En la ventana derecha inferior se puede ver el resultado de la ejecución.

En el mismo directorio antes mencionado se podrá encontrar el programa *wordcount.py,* que es similar a los que ya hemos tratado anteriormente.

Figura 13. *wordcount.py*



En este, el código de cada línea es:

- Líneas 18, 19, 21: importación de las librerías.
- Líneas 25, 26, 27: función de error si no se provee un archivo para procesar.
- Línea 29: creación de la sesión de Spark.
- Línea 34: lectura del archivo y creación de un RDD con cada línea del archivo.
- Línea 35: creación de un segundo RDD con las palabras separadas por espacios, generación de la tupla (w, 1) y reducción por la llave (*key*), en este caso la palabra.
- Línea 38: almacenamiento del RDD en un *array.*
- Línea 39: impresión de los datos como palabra: contador.
- Línea 42: final de la sesión Spark.

Para ejecutar se debe hacer clic en *play* (la flecha de color verde del menú de iconos), pero antes se deberá indicar el archivo de entrada (en este caso *war-peace.txt*); esto se realiza en el menú *Run* > *Configuration per file* > *marcar Command Line Options* y en el cuadro de texto incluir el directorio donde está

el archivo de entrada (en este caso el directorio txt/war-peace.txt). En la salida se puede observar a la derecha en la ventana superior las variables utilizadas y su valor, y en la inferior el resultado del procesamiento.

4. Distribuciones específicas

4.1. Cloudera

Es una compañía que, sobre la base de Apache Hadoop, ha desarrollado un ecosistema propio basado en soporte empresarial y servicios junto con programas de formación para grandes clientes.

La distribución original *open source* de Apache Hadoop se llamaba CDH (Cloudera Distribution Hadoop) y se enfocaba al desarrollo de esta tecnología orientada a empresas. Un aspecto interesante de esta compañía y beneficioso para el *open source* es que parte del tiempo de sus expertos se donaba (según la empresa hasta un 50 %) a diferentes proyectos *open source* (Apache Hive, Apache Avro, Apache HBase, etc.), y formaba también parte del directorio de la Apache Software Foundation.

En 2019, la empresa se unió con otra gran empresa que disponía de un entorno similar de gran calidad llamado Hortonworks, se juntaron las dos distribuciones y se generó una macrodistribución con lo mejor de cada una de ellas. En 2019-2020, orienta su negocio hacia el *cloud* (haciendo entornos consistentes en *cloud* y análisis de datos) y lanza su primera Cloudera Data Platform Private Cloud y actualiza sus plataformas con nuevos servicios (CDP Data Engineering, CDP Operational Database y CDP Data Visualization Services).

En la actualidad (2021), el producto se denomina Cloudera Data Platform (CDP), que es una plataforma estable y escalable centrada en entornos *cloud* y que se denomina First Enterprise Data Cloud. CDP permite análisis de datos por medio de entornos *cloud* (propios o *multicloud*, por ejemplo: Azure, Google, IBM...), pero **lamentablemente han cerrado sus licencias** (a pesar de que sus mayores componentes son *open source*) y ahora se rigen por un sistema de *paywall* y se requiere una suscripción válida para acceder a cualquiera de sus productos:

- Apache Hadoop (CDH)
- Hortonworks Data Platform (HDP)
- Data Flow (HDF/CDF)
- Cloudera Data Science Workbench (CDSW)

4.2. Apache Bigtop

El objetivo principal de Bigtop es un proyecto (*open source*), basado en la comunidad, que sienta las bases para el empaquetado, la implementación y las pruebas de interoperabilidad de proyectos relacionados con Hadoop.

Esto incluye pruebas en varios niveles (empaquetado, plataformas, tiempo de ejecución, actualización, etc.), y es desarrollado por una comunidad con un enfoque en el sistema como un todo, en lugar de proyectos individuales. Este proyecto desarrolla y publica código de prueba de compilación, empaquetado e integración que depende de las versiones oficiales de los proyectos relacionados con Apache Hadoop (HDFS, MapReduce, HBase, Hive, Pig, ZooKeeper, etc.). Su interés es que, a medida que se encuentren errores y otros problemas, poder solucionarlos cuanto antes.

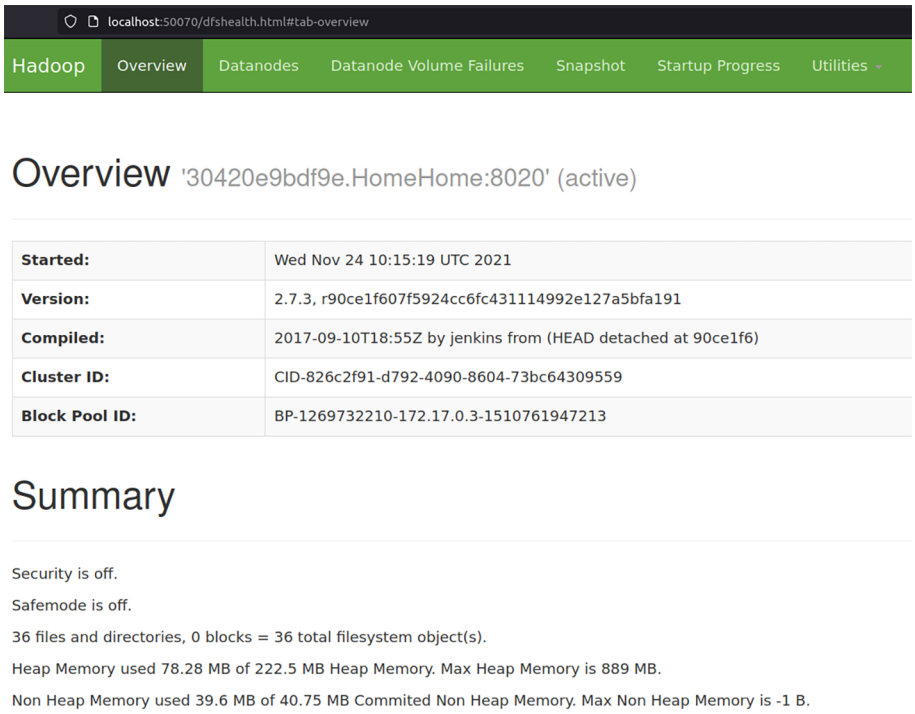
La última versión de Bigtop (3.0.0) es de octubre de 2021 e incluye los siguientes componentes: alluxio, ambari, bigtop-ambari-mpack, bigtop-groovy, bigtop-jsvc, bigtop-utils, elasticsearch, flink, gpdb, hadoop, hbase, hive, kafka, kibana, livy, logstash, oozie, phoenix, solr, spark, sqoop, tez, ycsb, zeppelin, zookeeper.

Como prueba de concepto se puede ejecutar o construir un pseudo clúster de Hadoop con Docker utilizando las imágenes que proveen los desarrolladores o construyendo las imágenes de Docker con las utilidades que se den. Por ejemplo, para desplegar Hadoop HDFS simplemente basta con hacer:

```
docker run -d -p 50070:50070 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs
```

En localhost:50070 se podrá ver la interfaz, como muestra la siguiente figura:

Figura 14. Interfaz de Bigtop



Como se puede observar, no es una versión actualizada, pero es totalmente funcional. Para finalizar la ejecución del contenedor hay que mirar el CONTAINER_ID con `docker ps --all` y después ejecutar `docker stop CONTAINER_ID`. Para eliminar el contenedor `docker rm CONATINER_ID`, para eliminar la imagen se debe mirar el IMAGE_ID con `docker images` y borrarla con `docker rmi IMAGE_ID`. Como se puede observar en la página del desarrollador también es posible almacenar el resultado de la ejecución del *docker run* en una variable (por ejemplo, BIGTOP, que será el CONTAINER_ID y luego utilizar $BIGTOP como valor de CONTAINER_ID).

Si se desea ejecutar Hadoop HDFS + Hbase:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 16010:16010 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_hbase)
docker exec -ti $BIGTOP hbase shell
```

Para ejecutar Hadoop HDFS + Spark Standalone:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 8080:8080 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_spark
-standalone)
docker exec -ti $BIGTOP spark-shell
```

Si se desea ejecutar Hadoop HDFS + YARN + Hive + Pig:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 8088:8088 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_yarn
_hive_pig)
docker exec -ti $BIGTOP hive
```

```
docker exec -ti $BIGTOP pig
```

Se podrían crear nuevos contenedores utilizando las instrucciones que hay en la página antes mencionada (o en el directorio bigtop-3.0.0/docker/sandbox/README.md) para la última versión, pero se debe hacer un trabajo importante ya que que solo está configurado para Ubuntu 16.04, por ejemplo, y este ya está obsoleto.

Enlace recomendado

En https://dlcdn.apache.org/bigtop/bigtop-3.0.0/repos/ se pueden encontrar los repositorios para instalarlo sobre Ubuntu20.04 o Debian 10, por ejemplo.

Otro aspecto interesante es el directorio *provisioner* dentro de la distribución que permite utillizar Docker o Vagrant para generar Virtual Hadoop Cluster. Es esta prueba se ha utilizado Docker y por medio de Docker Compose se puede crear un Bigtop Virtual Hadoop Cluster sobre contenedores Docker. Este clúster puede ser utilizado para:

- probar los test incluidos en Bigtop (*smoke tests*)
- probar las *Bigtop puppet recipes*
- ejecutar test de integración con la aplicación del usuario

En este caso se ha renombrado el archivo config_ubuntu-20.04.yaml como config.yaml y se ha ejecutado./docker-hadoop.sh --create 2 para crear un clúster Hadoop con dos nodos sobre Docker 20.10.7. En el archivo README.md se muestran diferentes formas de acceder o ejecutar comandos dentro de los contenedores, pero utilizando el *script* se puede entrar dentro del primer contenedor ejecutando por ejemplo:

```
./docker-hadoop.sh --exec 1 bash
```

Y en el directorio/archivo *config/hosts* se podrán ver las IP asignadas a cada contenedor y en el navegador de *host* se podrá poner como URL IP:50070 para ver el estado del clúster o IP:8088 para ver el estado de todas las aplicaciones, como se muestra a continuación. El estado de los contenedores puestos en marcha se puede ver con:

```
./docker-hadoop.sh -l
```

Figura 15. Estado de los contenedores

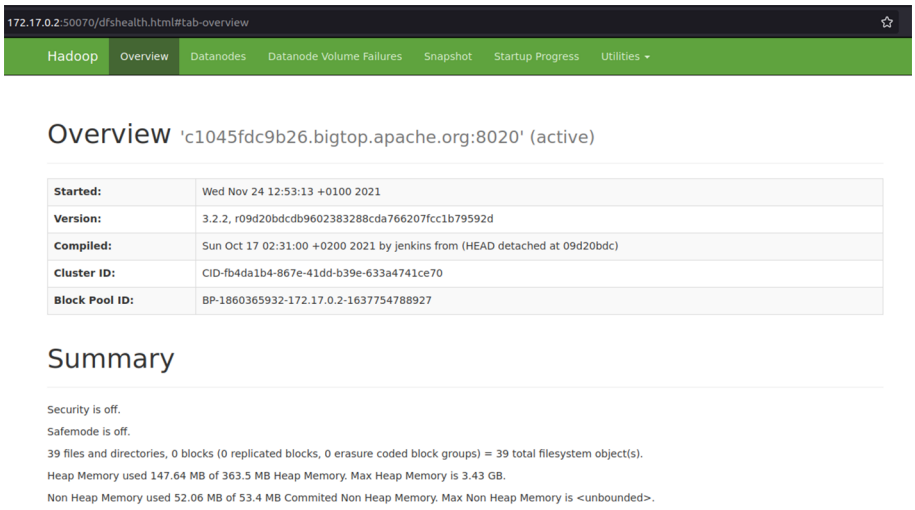
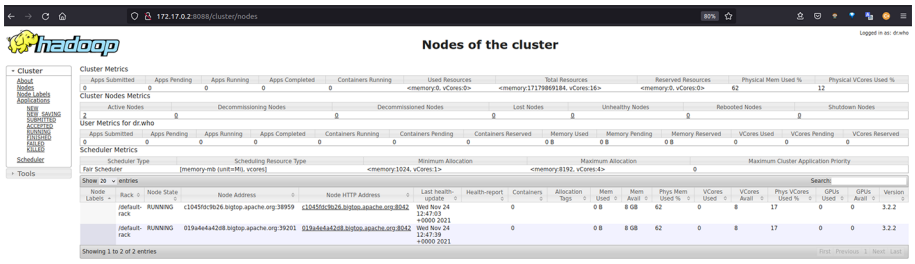


Figura 16. Estado de las aplicaciones



Enlace recomendado

Para más información sobre la última versión 3.0.0 consultad «Bigtop 3.0.0 Release».

4.3. Big Data Europe

Otro entorno interesante, como prueba de concepto, es el proyecto Big Data Europe, que permiten desplegar diferentes componentes de la infraestructura software para el procesamiento de *big data* (por ejemplo un clúster en Hadoop o Spark) por medio de Docker y utilizando Swarm como gestor de recursos. Por ejemplo, para desplegar un clúster Hadoop se debe hacer un clonado del repositorio big-data-europe/docker-hadoop:

```
git clone https://github.com/big-data-europe/docker-hadoop.git
```

A continuación:

```
docker-compose up
```

Para ejecutar el ejemplo de contador de palabras:

```
make wordcount
```

O para desplegar un entorno en Swarm:

```
docker stack deploy -c docker-compose-v3.yml hadoop
```

El `docker-compose` creará una red que puede ser localizada ejecutando `docker network list` (en nuestro caso *docker-hadoop_default*) y haciendo `docker network inspect docker-hadoop_default` se podrá ver la asignación de las IP a los diferentes nodos de las interfaces de Hadoop en las siguientes URL:

- Namenode: http://<*IP_address*>:9870/dfshealth.html#tab-overview
- History server: http://<*IP_address*>:8188/applicationhistory
- Datanode: http://<*IP_address*>:9864/
- Nodemanager: http://<*IP_address*>:8042/node
- Resource manager: http://<*IP_address*>:8088/

Introduciendo la URL del *resource manager* (http://<IP_address>:8088/) en el navegador se obtendrá el mismo entorno como el mostrado en figuras anteriores (p, ej. la figura 16).

Bibliografía

Abbasi, M. A. (2017). *Learning Apache Spark 2*. Birmingham: Packt Publishing. https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781785885136 (última consulta: noviembre de 2021).

Apache Spark. «Spark SQL, DataFrames and Datasets Guide». https://spark.apache.org/docs/3.1.1/sql-programming-guide.html (última consulta: noviembre de 2021).

Bengfort, B; Kim, J. (2016). *Data Analytics with Hadoop: An Introduction for Data Scientists*. Sebastopol: O'Reilly Media.

Chambers, B.; Zaharia. M. (2018). *Spark: The Definitive Guide. Big Data Processing Made Simple*. Sebastopol: O'Reilly.

Damji, J. S.; Wenig, B. *et al.* (2020). *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol: O'Reilly Media.

DataNoon (2018). «Pyspark DataFrame Operations - Basics | Pyspark DataFrames». https://datanoon.com/blog/pyspark_dataframe_operations/ (última consulta: noviembre de 2021).

DeRoos, D. (2014). *Hadoop For Dummies*. Hoboken: Wiley & Sons.

Hurwitz, J.; Nugent, A.; Halper, F.; Kaufman, M. (2013). *Big Data for Dummies*. Hoboken: Wiley & Sons. https://big-data.digital/big-data-for-dummies/ (última consulta: noviembre de 2021).

Mendelevitch, O.; Stella, C; Eadline, D. (2017). *Practical Data Science with Hadoop and Spark: Designing and Building Effective Analytics at Scale*. Boston: Addison-Wesley Professional.

Schneider, R. D.; Karmiol, J. (2019). *Spark for Dummies (2nd IBM Limited Edition)*. Hoboken: Wiley & Sons. www.ibm.com/downloads/cas/WEB4XBOR (última consulta: noviembre de 2021).

Scott, J. A. (2015). *Getting Started with Apache Spark*. San Jose (EE. UU.): MapR Technologies.

Shook, A.; Miner, D. (2012). *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. Sebastopol: O'Reilly Media.

Singh, C.; Kumar, M. (2019). *Mastering Hadoop 3: Big data processing at scale to unlock unique business insights*. Birmingham: Packt Publishing.

White, T. (2015). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale* (4a edición, cap. II). Sebastopol: O'Reilly Media.

Webs con iconos con licencia de uso libre

www.iconarchive.com (última consulta: noviembre de 2021).

www.customicondesign.com (última consulta: noviembre de 2021).

http://icons8.com (última consulta: noviembre de 2021).

Repositorios útiles de OpenData

«Cosmic»: es una base de datos analizada por expertos que abarca la amplia variedad de mecanismos de mutación somática que causan el cáncer humano. En este sitio, ved «Cosmic, the Catalogue Of Somatic Mutations In Cancer» (contiene detalles sobre millones de mutaciones en miles de tipos de cáncer. Está en constante crecimiento tanto en contenido como en alcance) y «Cancer Gene Census» (última consulta: noviembre de 2021).

«OpenData, COVID-19»: NCATS está generando una serie de conjuntos de datos mediante la selección de un panel de ensayos relacionados con el SARS-CoV-2 frente a todos los medicamentos aprobados. Estos conjuntos de datos, así como los protocolos de ensayo utilizados para generarlos, se pondrán de inmediato a disposición de la comunidad científica en este sitio web a medida que se completen (última consulta: noviembre de 2021).

«SARS-CoV-2 proteins – NCBI Datasets BETA»: se puede seleccionar y descargar un conjunto de datos sobre proteínas de genomas completos de SARS-CoV-2, incluidas secuencias de nucleótidos y proteínas, anotaciones, estructuras de proteínas y un informe de datos detallado (última consulta: noviembre de 2021).