UD 6 - Apache Spark

Apache Spark es un **motor de análisis unificado para el procesamiento de datos a gran escala**. Proporciona **API** de alto nivel en **Java, Scala, Python y R**, y un motor optimizado que admite gráficos de ejecución general. También admite un amplio conjunto de herramientas de nivel superior que incluyen **Spark SQL para SQL y procesamiento de datos estructurados**, API de **pandas en Spark** para cargas de trabajo de pandas, **MLlib** para aprendizaje automático, **GraphX** para procesamiento de gráficos y Structured Streaming para computación y transmisión incrementales.

Figura 6.1_Spark: Logo Spark. (Fuente: spark.apache.org)

1. Introducción

Apache Spark es un framework de computación distribuida similar a **Hadoop MapReduce** (así pues, Spark no es un lenguaje de programación), pero que en vez de almacenar los datos en un sistema de ficheros distribuidos o utilizar un sistema de gestión de recursos, lo hace en memoria. El hecho de almacenar en memoria los cálculos intermedios **implica que sea mucho más eficiente que MapReduce**.

En el caso de tener la necesidad de almacenar los datos o gestionar los recursos, se apoya en sistemas ya existentes como **HDFS, YARN o Apache Mesos**. Por lo tanto, Hadoop y Spark son sistemas complementarios.

2. Características

Las características clave de Apache Spark incluyen:

- **Velocidad**: Spark es conocido por su capacidad para procesar grandes volúmenes de datos a una velocidad significativamente más alta que otros motores de big data, como Hadoop MapReduce. Esto se debe en gran medida a su capacidad para realizar procesamientos en memoria (in-memory processing), lo que minimiza la lectura y escritura en el disco.
- **Facilidad de Uso**: Proporciona APIs sencillas para trabajar en lenguajes como Python, Java, Scala y R, lo que hace que sea más accesible para los desarrolladores y analistas de datos. También soporta una variedad de operaciones de datos, como transformaciones y acciones, lo que facilita la manipulación de datos.
- **Procesamiento de Datos en Tiempo Real**: A diferencia de Hadoop, que está diseñado principalmente para el procesamiento por lotes, Spark soporta tanto el procesamiento por lotes como el procesamiento en tiempo real, lo que lo hace adecuado para una gama más amplia de aplicaciones, como la detección de fraudes en tiempo real y el análisis de datos de streaming.
- **Librerías Integradas**: Viene con varias librerías integradas, incluyendo Spark SQL para el procesamiento de datos estructurados, MLlib para machine learning, GraphX para el procesamiento de gráficos y Spark Streaming para el procesamiento de datos en tiempo real.
- **Escalabilidad y Flexibilidad**: Apache Spark puede ejecutarse en una variedad de entornos, desde una sola máquina hasta clústeres de miles de nodos. Es compatible con varias plataformas de almacenamiento de datos y puede integrarse con sistemas de big data como Hadoop.
- **Extensible**: Al centrarse unicamente en el procesamiento, la gestión de los datos se puede realizar a partir de Hadoop, Cassandra, HBase, MongoDB, Hive o cualquier SGBD relacional, haciendo todo en memoria. Además, se puede extender el API para utilizar otras fuentes de datos, como Apache Kafka, Amazon S3 o Azure Storage.

En resumen, Apache Spark es una herramienta poderosa y versátil para el procesamiento de big data, apreciada por su velocidad, facilidad de uso y capacidades avanzadas de procesamiento de datos. Su diseño lo hace ideal tanto para el análisis de grandes volúmenes de datos históricos como para aplicaciones que requieren el procesamiento de datos en tiempo real.

3. Spark y Hadoop

La principal diferencia es que la computación se realiza en memoria, lo que puede implicar un mejora de hasta 100 veces mejor rendimiento. Para ello, se realiza una evaluación perezosa de las operaciones, de manera, que hasta que no se realiza una operación, los datos realmente no se cargan.

Para solucionar los problemas asociados a MapReduce, Spark crea un espacio de memoria RAM compartida entre los ordenadores del clúster. Este permite que los NodeManager/WorkerNode compartan variables (y su estado), **eliminando la necesidad de escribir los resultados intermedios en disco**. Esta zona de memoria compartida se traduce en el uso de **RDD, DataFrames y DataSets**, permitiendo realizar **procesamiento en memoria a lo largo de un clúster** con **tolerancia a fallos**.

4. Arquitectura

Un clúster de Spark tiene un Master solitario y muchos números de esclavos / trabajadores. El controlador y los agentes ejecutan sus procedimientos Java individuales y los usuarios pueden ejecutarlos en máquinas individuales.

- **Master Daemon** (Master / Driver Process)
- **Worker Daemon** (Slave Process)

A continuación se muestran los tres métodos para construir Spark con componentes Hadoop (estos tres componentes son pilares sólidos de Spark Architecture):

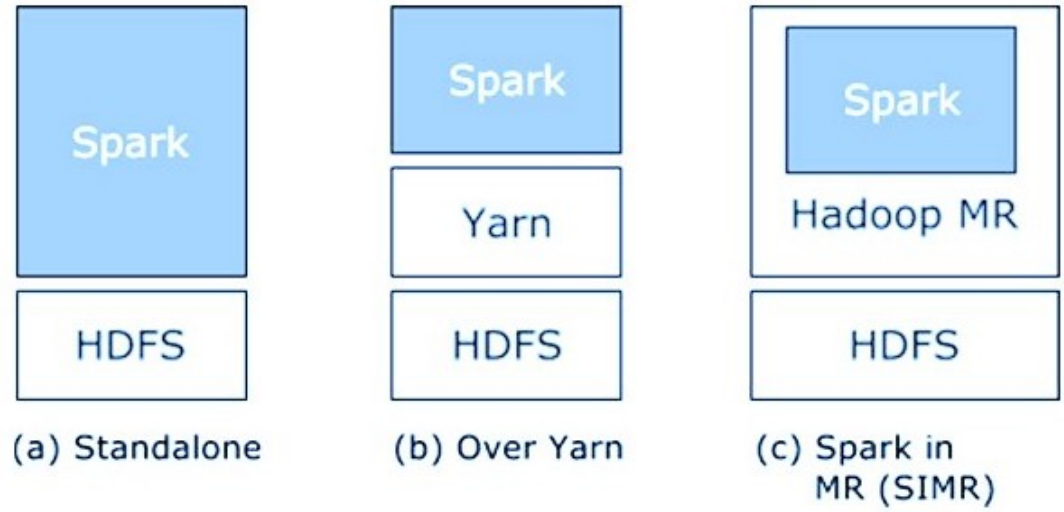

Figura 6.2_Spark: Arquitectura Spark con Hadoop. (Fuente: sitiobigdata.com)

1. **Independiente**

El arreglo implica que Spark posee el lugar en la parte superior de HDFS (Hadoop Distributed File System) y el espacio se asignan para HDFS, de manera inequívoca. Aquí, Spark y MapReduce se ejecutarán uno al lado del otro para cubrir todo en forma de Cluster.

2. **Hadoop Yarn**

El arreglo de Hadoop Yarn implica, básicamente, que Spark sigue funcionando en Yarn sin preestablecimiento o llegar a la raíz requerida. Incorpora Spark en el entorno de Hadoop o en la pila de Hadoop. Permite que diferentes partes se sigan ejecutando en la parte superior de la pila y tienen una asignación explícita para HDFS.

3. **Spark en MapReduce**

Spark en MapReduce se utiliza para despachar el trabajo de inicio a pesar de la disposición independiente. Con SIMR, el cliente puede comenzar a usar Spark y usar su shell sin acceso regulatorio.

5. Componentes del Ecosistema

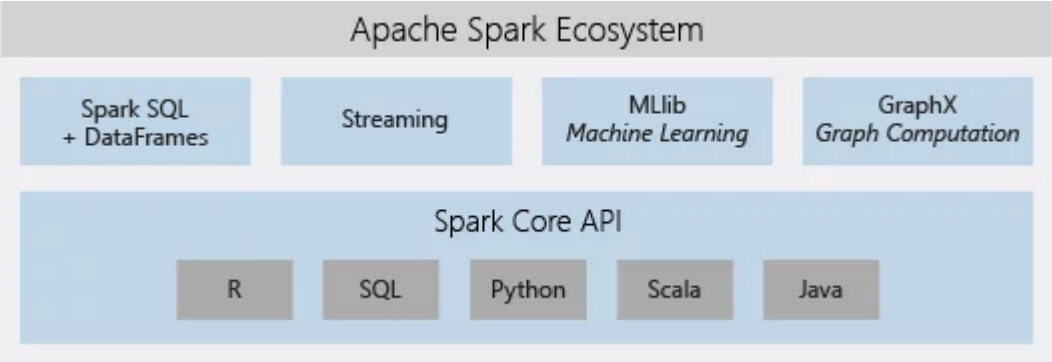

Figura 6.3_Spark: Ecosistema Spark (Fuente: aprenderbigdata.com)

- **Apache Spark Core**: Spark Core es el motor de ejecución general básico para la plataforma Spark en el que se basan todas las demás funcionalidades. Proporciona registro en memoria y conjuntos de datos conectados en marcos de almacenamiento externos.
- **Spark SQL**: Spark SQL es un segmento sobre Spark Core que presenta otra abstracción de información llamada SchemaRDD, que ofrece ayuda para sincronizar información estructurada y no estructurada.
- **Spark Streaming**: Spark Streaming utiliza la capacidad de programación rápida de Spark Core para realizar Streaming Analytics. Ingiere información en grupos a escala reducida y realiza cambios de RDD (Conjuntos de datos distribuidos resistentes) en esos grupos de información a pequeña escala.
- **MLlib (Machine Learning Library)**: MLlib es una estructura de aprendizaje automático distribuido por encima de Spark en vista de la arquitectura Spark basada en memoria distribuida. Es, como lo indican los puntos de referencia, realizado por los ingenieros de MLlib contra las ejecuciones de mínimos cuadrados alternos (ALS).
- **GraphX**: GraphX es un marco distribuido de procesamiento de gráficos de Spark. Proporciona una API para comunicar el cálculo del gráfico que puede mostrar los

diagramas caracterizados por el cliente utilizando la API de abstracción de Pregel. Asimismo, proporciona un tiempo de ejecución optimizado y mejorado a esta abstracción.

- **Spark R**: Esencialmente, para utilizar Apache Spark de R. Es el paquete R el que da una interfaz de usuario ligera. Además, permite a los investigadores de la información desglosar conjuntos de datos expansivos.

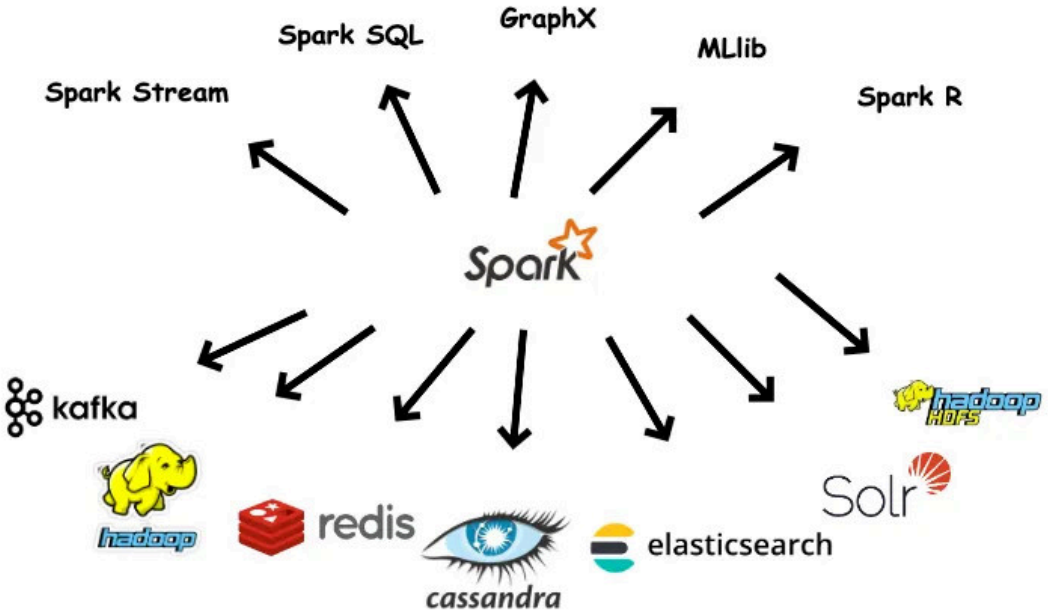

Figura 6.4_Spark: Third Party Systems Spark (Fuente: medium.com/rahasak).

6. Flujo de aplicaciones de Spark

Una aplicación Spark se compone de dos partes:

- La **lógica de procesamiento** de los datos, realizada mediante API de Spark(que veremos en el siguiente punto) o la propia shell interactiva.
- **Driver**: es el coordinador central encargado de interactuar con el clúster Spark y averiguar qué máquinas deben ejecutar la lógica de procesamiento. Para cada una de esas máquinas, el driver realiza una petición para lanzar un proceso conocido como ejecutor (**Executor**). Además, el driver Spark es responsable de gestionar y distribuir las tareas a cada ejecutor, y si es necesario, recoger y fusionar los datos resultantes para presentarlos al usuario. Estas tareas se realizan a través de la `SparkSession`.

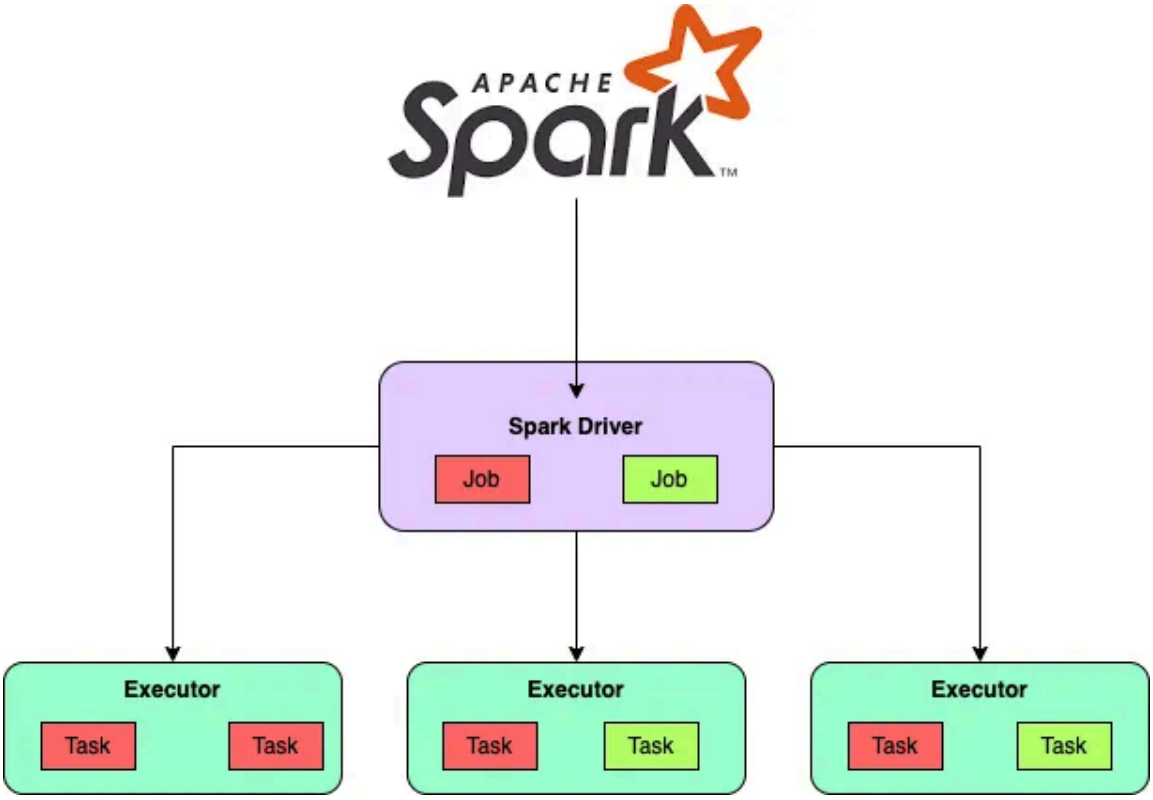

Figura 6.5_Spark: Flujo de ejecución de Aplicaciones Spark. (Fuente: https://pratikbarjatya.medium.com).

La ejecución se descompone en 3 partes a bajo nivel:

6.1 Spark Job

Un **Spark Job** representa una tarea informática completa que Spark realiza en un conjunto de datos. Consta de múltiples etapas y abarca todos los pasos necesarios para **transformar** y **analizar** datos. Los Jobs se envían a Spark a través del driver program y se dividen en unidades más pequeñas llamadas etapas de ejecución (**Executor**).

6.2 Spark Stage

Un **Spark Stage** es una unidad lógica de trabajo dentro de un Spark Job. Representa un conjunto de tareas que se pueden ejecutar juntas, generalmente como resultado de una transformación estrecha (por ejemplo, `map` o `filter`) o una operación aleatoria (por ejemplo, `groupByKey` o `reduceByKey`). Las Stages las determina el motor Spark durante la fase de optimización de consultas, en función de las dependencias entre RDDs o DataFrames.

6.3 Spark Task

Un **Spark Task** es la unidad de trabajo más pequeña en Spark. Representa una operación única que se puede ejecutar en un subconjunto de datos particionado. Los nodos trabajadores ejecutan las tareas en paralelo, aprovechando las capacidades informáticas distribuidas de Spark. Cada tarea opera sobre una porción de los datos, aplicando las transformaciones requeridas y produciendo resultados intermedios o finales.

Finalmente, al lanzar una aplicación Spark, podemos indicar el número de ejecutores que necesita la aplicación, así como la cantidad de memoria y número de núcleos que debería tener cada executor.

6.4 Cheat Sheet

1. Spark Job
    a. **Definición**: tarea de cálculo completa realizada por Spark en un conjunto de datos.
    b. **Relación**: Consta de múltiples etapas.
    c. **Envío**: enviado a Spark a través del driver program.
    d. **Ejecución**: Dividido en etapas para ejecución paralela.
2. Spark Stage
    a. **Definición**: unidad lógica de trabajo dentro de un job de Spark.
    b. **Relación**: Comprende un conjunto de tareas que se pueden ejecutar juntas.
    c. **Determinación**: determinada durante la optimización de consultas en función de las dependencias de RDD/DataFrame.
    d. **Transformaciones**: normalmente asociadas con operaciones estrechas o aleatorias.
3. Spark Task:
    a. **Definición**: unidad de trabajo más pequeña en Spark.
    b. **Ejecución**: ejecutada en paralelo por nodos workers.
    c. **Subconjunto de datos**: opera en un subconjunto de datos particionado.
    d. **Operaciones**: realiza las transformaciones requeridas en los datos.

6.5 Comprender el flujo:

- Se envía un Spark Job a Spark para su procesamiento.
- El Job se divide en Stage según las transformaciones y dependencias en el cálculo.
- Cada Stage consta de múltiples Tasks que se pueden ejecutar en paralelo.
- Las Tasks se asignan a nodos Workers, que realizan los cálculos necesarios en las particiones de datos que se les asignan.

- Los resultados intermedios y finales se producen a medida que las tareas completan sus cálculos.
- El trabajo general se completa cuando todas las Stages y Tasks terminan de ejecutarse.

7. Instalación

1. En primer lugar nos vamos a la página de descarga de Apache Spark y la descargamos. Ten en cuenta que hay varias opciones.

- Apache spark con hadoop incluido
- Apache spark con hadoop incluido con Scala
- Apache spark sin hadoop (lo proporcionamos nosotros)
- Source code

Instalamos la opción 3, ya que tenemos nuestra Hadoop ya instalado y configurado.

```
wget https://archive.apache.org/dist/spark/spark-3.5.4/spark-3.5.4-bin-without-hadoop.tgz
```

2. Descomprimimos

```
tar -zxvf spark-3.5.4-bin-without-hadoop.tgz
```

3. Movemos el directorio a nuestro directorio `$HADOOP_HOME` para una correcta organización

```
mv spark-3.5.4-bin-without-hadoop /opt/hadoop-3.4.1/spark-3.5.4
```

4. Accedemos a los "templates" de configuración que nos proporciona Apache Spark. Accedemos a directorio `conf` de Spark y ejecutamos los siguientes comandos

```
cp fairscheduler.xml.template fairscheduler.xml
cp log4j2.properties.template log4j2.properties
cp metrics.properties.template metrics.properties
cp spark-defaults.conf.template spark-defaults.conf
cp spark-env.sh.template spark-env.sh
cp workers.template workers
```

5. Para añadir Apache Spark a Apache Hadoop necesitamos incluir los paquetes jar de Hadoop. Para ello entramos en `conf/spark-env.sh` y añadimos las siguientes líneas *(Observa todas las variables de configuración que tiene Spark)*:

> **Warning**
>
> En la versión actual, aunque tengamos correctamente configurada la variable de entorno para ejecutar el comando `hadoop` debemos poner la ruta completa en la siguiente configuración para que funcione correctamente

spark-env.sh

```
# If 'hadoop' binary is on your PATH
export SPARK_DIST_CLASSPATH=$(/opt/hadoop-3.4.1/bin/hadoop classpath)
```

> **Java 11**
>
> Para Java 11, se requiere configurar `-Dio.netty.tryReflectionSetAccessible=true` para la librería A_pache Arrow_. Esto evita el error `java.lang.UnsupportedOperationException: sun.misc.Unsafe` o java.lang.UnsupportedOperationException: sun.misc.Unsafe or `java.nio.DirectByteBuffer.(long, int) not available error when Apache Arrow uses Netty internally`

6. Creamos las variables de entorno necesarias. Para ello abrimos el archivo `~/.bashrc` y añadimos al final el siguiente código y ejecuta el comando `source ~/.bashrc`

~/.bashrc

```
export SPARK_HOME=/opt/hadoop-3.4.1/spark-3.5.4
export SPARK_DIST_CLASSPATH=$(hadoop classpath)
export PATH=$PATH:$SPARK_HOME/bin
```

Spark Path - comandos y shell scripts

Apache Spark tiene 2 directorios de archivos de ejecución:

- `$SPARK_HOME/bin`: Comandos propios de Spark
- `$SPARK_HOME/sbin`: Shell Scripts propios de Spark

Puedes añadir, como hacemos normalmente, en el PATH los 2 directorios. Pero en este caso, hay comandos que tienen el mismo nombre que los de Apache Hadoop en `$SPARK_HOME/sbin`, como por ejemplo `start-all.sh`. Nosotros no la añadiremos para evitar problemas. Si añadimos `$SPARK_HOME/bin` que no entran en ningún conflicto. Por tanto, para ejecutar los Shell script de Spark, debemos tener en cuenta la ruta completa de los comandos: `$SPARK_HOME/sbin`. Si aún así prefieres añadirlo al PATH añadimos la siguiente línea en el archivo `~/.bashrc` después de las del punto anterior `source ~/.bashrc`

~/.bashrc

```
export PATH=$PATH:$SPARK_HOME/sbin
```

Success

No olvides tener iniciado Apache Hadoop `start-dfs.sh`

7. Iniciamos Spark en el master con el script `./sbin/start-master.sh` que se encuentra en `$SPARK_HOME`

```
#Desde $SPARK_HOME
./sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /opt/hadoop-3.4.1/spark-3.5.4/logs/spark-hadoop-org.apache.spark.deploy.master.Master-1-master.out
```

Ya tenemos Spark iniciado en el master. También se levanta un WebUI en el puerto 8080

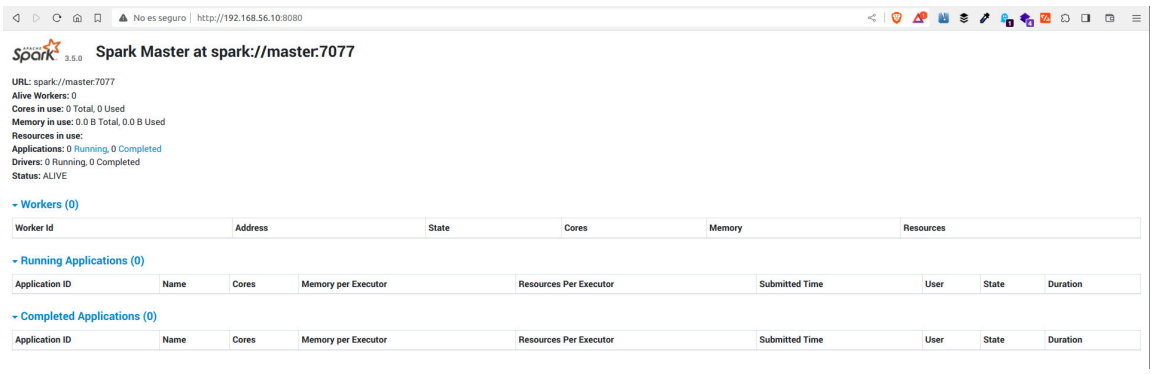

Figura 6.6_Spark: WebUI de Spark.

Para para Spark, ejecutamos el script `./sbin/stop-master.sh`

8. Quick Start

8.1 Análisis interactivo con la Shell de Spark

1. Para comprobar el correcto funcionamiento de Spark, vamos a abrir una shell interactiva para trabajar en consola con Spark, que proporciona una forma sencilla de aprender la API, así como una poderosa herramienta para analizar datos de forma interactiva.
2. Creamos un nuevo conjunto de datos a partir del texto del archivo README en el directorio fuente de Spark `$SPARK_HOME`:

Scala

Iniciamos

```
spark-shell
```

```
....
25/01/24 13:24:04 INFO Utils: Successfully started service 'SparkUI' on port 4040.
25/01/24 13:24:05 INFO Executor: Starting executor ID driver on host 10.0.2.15
25/01/24 13:24:05 INFO Executor: OS info Linux, 6.8.0-49-generic, amd64
25/01/24 13:24:05 INFO Executor: Java version 1.8.0_432
25/01/24 13:24:05 INFO Executor: Starting executor with user classpath (userClassPathFirst = false): ''
25/01/24 13:24:05 INFO Executor: Using REPL class URI: spark://10.0.2.15:35573/classes
25/01/24 13:24:05 INFO Executor: Created or updated repl class loader org.apache.spark.executor.ExecutorClassLoader@535a6697 for default.
25/01/24 13:24:05 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 44879.
25/01/24 13:24:05 INFO NettyBlockTransferService: Server created on 10.0.2.15:44879
25/01/24 13:24:05 INFO BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for block replication policy
25/01/24 13:24:05 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 10.0.2.15, 44879, None)
25/01/24 13:24:05 INFO BlockManagerMasterEndpoint: Registering block manager 10.0.2.15:44879 with 366.3 MiB RAM, BlockManagerId(driver, 10.0.2.15, 44879, None)
25/01/24 13:24:05 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 10.0.2.15, 44879, None)
25/01/24 13:24:05 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 10.0.2.15, 44879, None)
Spark context Web UI available at http://10.0.2.15:4040
```

```
Spark context available as 'sc' (master = local[*], app id = local-1737721444933).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.4
      /_/

Using Scala version 2.12.18 (OpenJDK 64-Bit Server VM, Java 1.8.0_432)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

```
scala> val textFile = spark.read.textFile("file:///opt/hadoop-3.4.1/spark-3.5.4/README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

Puedes obtener valores del Dataset directamente, llamando a algunas acciones o transformar el Dataset para obtener uno nuevo. Para obtener más detalles, consulta documentación de la API.

```
scala> textFile.count() // Número de items en este Dataset
res0: Long = 125

scala> textFile.first() // Primer item en este Dataset
res1: String = # Apache Spark
```

Ahora transformamos este Dataset en uno nuevo. Llamamos al filtro para devolver un nuevo Dataset con un subconjunto de los elementos del archivo.

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]
```

Podemos encadenar transformaciones y acciones:

```
scala> linesWithSpark.count() // Cuántas líneas contienen "Spark"?
res2: Long = 20
```

Salimos

```
scala> :q
```

Python

Iniciamos. Sería igual si PySpark está instalado con pip en su entorno actual:

```
pyspark
```

```
....
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.4
      /_/

Using Python version 3.10.12 (main, Nov  6 2024 20:22:13)
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-
1737721617607).
SparkSession available as 'spark'.
>>>
```

Debido a la naturaleza dinámica de Python, no necesitamos que el conjunto de datos esté fuertemente tipado en Python. Como resultado, todos los Datasets en Python son Datasets[Fila], y lo llamamos *DataFrame* para ser coherente con el concepto de data frame en Pandas y R. Vamos a crear un nuevo DataFrame a partir del texto del archivo README en el directorio fuente de Spark:

```
>>> textFile = spark.read.text("file:///opt/hadoop-3.4.1/spark-
3.5.4/README.md")
```

Puedes obtener valores del DataFrame directamente, llamando a algunas acciones o transformar el DataFrame para obtener uno nuevo. Para obtener más detalles, consulta documentación de la API.

```
>>> textFile.count() # Número de filas en este DataFrame
125

>>> textFile.first()  # Primer fila en este DataFrame
Row(value='# Apache Spark')
```

Ahora vamos a transformar este DataFrame en uno nuevo. Llamamos al filtro para devolver un nuevo DataFrame con un subconjunto de líneas en el archivo.

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
>>> textFile.count() # Otra forma
```

Podemos encadenar transformaciones y acciones:

```
>>> textFile.filter(textFile.value.contains("Spark")).count() // Cuántas
líneas contienen "Spark"?
20
```

Salimos

```
>>> quit()
```

8.2 Monitorización

Cada driver program (lo veremos en el siguiente punto) tiene una Web UI, generalmente en el puerto 4040, que muestra información sobre tareas en ejecución, executers y uso de almacenamiento. Simplemente accediendo a `http://<driver-node>:4040` en un navegador web para acceder a esta interfaz de usuario. La guía de seguimiento también describe otras opciones de seguimiento.

Por ejemplo, los ejemplos anteriores de shell interactiva nos da la siguiente info:

Job

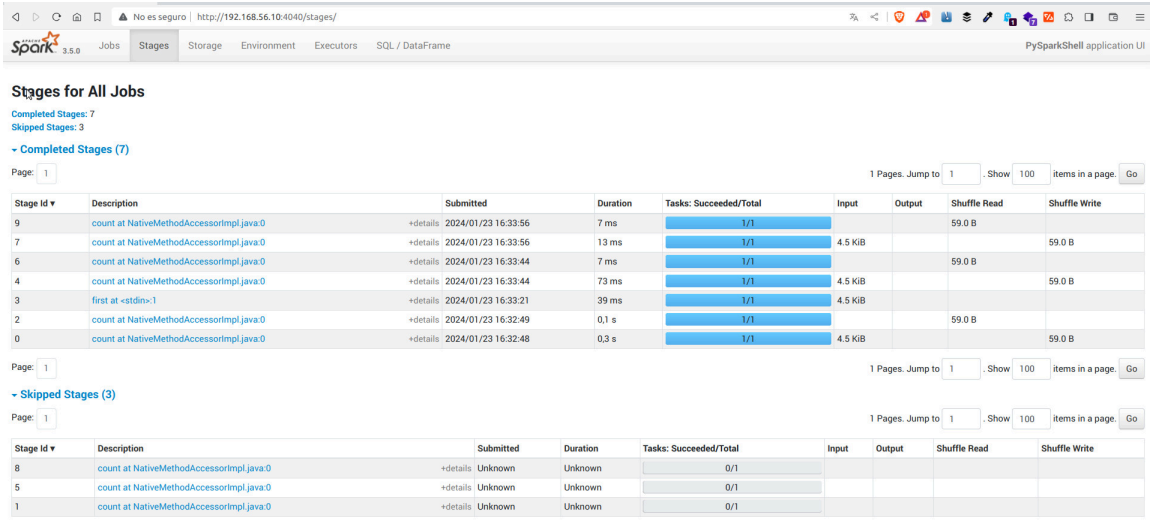

Figura 6.8_Spark: WebUI de Spark - Stage

Stage

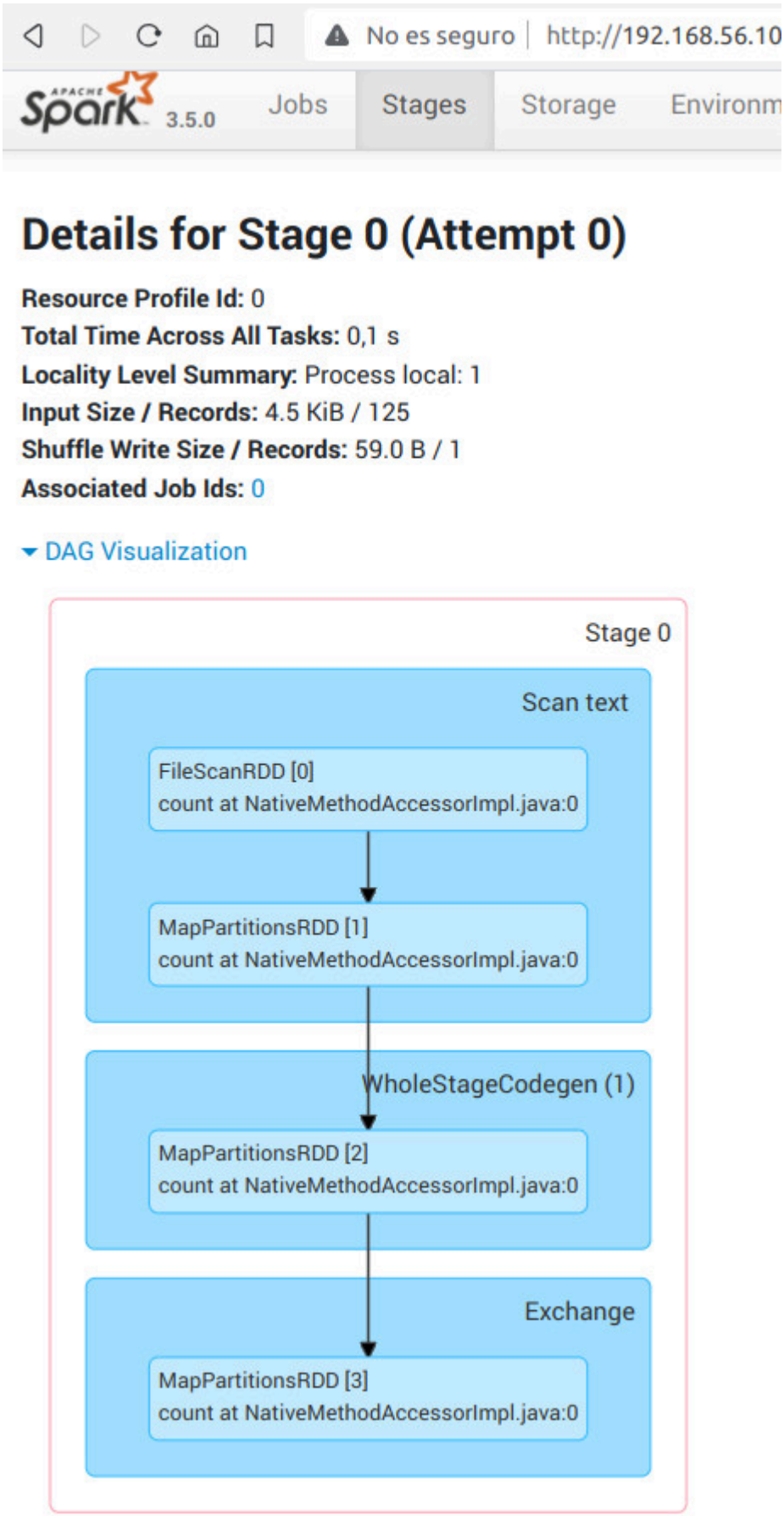
No es seguro | http://192.168.56.10
Spark 3.5.0
Jobs
Stages
Storage
Environm
Details for Stage 0 (Attempt 0)
Resource Profile Id: 0
Total Time Across All Tasks: 0,1 s
Locality Level Summary: Process local: 1
Input Size / Records: 4.5 KiB / 125
Shuffle Write Size / Records: 59.0 B / 1
Associated Job Ids: 0
DAG Visualization
Stage 0
Scan text
FileScanRDD [0]
count at NativeMethodAccessorImpl.java:0
MapPartitionsRDD [1]
count at NativeMethodAccessorImpl.java:0
WholeStageCodegen (1)
MapPartitionsRDD [2]
count at NativeMethodAccessorImpl.java:0
Exchange
MapPartitionsRDD [3]
count at NativeMethodAccessorImpl.java:0

Figura 6.9_Spark: WebUI de Spark - DAG

DAG

8.3 Aplicaciones autónomas

1. El script `spark-submit` de Spark en el directorio `$SPARK_HOME/bin` de Spark se utiliza para iniciar aplicaciones en un clúster. Puede utilizar todos los cluster managers con Spark a través de una interfaz uniforme para que no tengas que configurar tu aplicación para cada una de las configuraciones y situaciones de tu cluster.
2. Este script se encarga de configurar el classpath con Spark y sus dependencias, y puede admitir diferentes configuraciones y modos de implementación de clústeres que admite Spark

sintaxis

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

donde:

- `--class`: el punto de entrada de la aplicación (por ejemplo, `org.apache.spark.examples.SparkPi`)
- `--master`: la URL del master del clúster (por ejemplo, spark://23.195.26.187:7077)
- `--deploy-mode`: si queremos implementar un controlador en los nodos Workers (clúster) o localmente como un cliente externo (cliente) (predeterminado: cliente)
- `--conf`: argumento de configuración arbitraria de Spark en formato clave=valor. Para valores que contienen espacios, escriba **"clave=valor"** entre comillas (como se muestra). Se deben pasar varias configuraciones como argumentos separados. (por ejemplo, `--conf <clave>=<valor> --conf <clave2>=<valor2>`)
- `application-jar`: ruta a un jar incluido que incluye su aplicación y todas las dependencias. La URL debe ser visible globalmente dentro de su clúster, por ejemplo, una ruta hdfs:// o una ruta file:// que esté presente en todos los nodos.
- `application-arguments`: Argumentos pasados al método principal de su clase principal, si los hay

1. Para aplicaciones Python, simplemente pasamos un archivo .py en lugar de `application-jar` y agregamos los archivos Python .zip, .egg o .py a la ruta de búsqueda con --py-files.
2. A continuación se muestran algunos ejemplos de opciones comunes:

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a YARN cluster in cluster deploy mode
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# Run a Python application on a Spark standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000

# Run on a Mesos cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master mesos://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
```

```
    --total-executor-cores 100 \
    http://path/to/examples.jar \
    1000

  # Run on a Kubernetes cluster in cluster deploy mode
  ./bin/spark-submit \
    --class org.apache.spark.examples.SparkPi \
    --master k8s://xx.yy.zz.ww:443 \
    --deploy-mode cluster \
    --executor-memory 20G \
    --num-executors 50 \
    http://path/to/examples.jar \
    1000
```

5. Puedes ejecutar `spark-submit --help` para ver todas las opciones

8.4. Workers en aplicaciones autónomas

Para poder ejecutar cualquier aplicación autónoma, necesitamos, además del master, algún nodo worker funcionando

Para ello realizamos los siguientes pasos

1. Iniciamos Spark en el master con el script `./sbin/start-master.sh` que se encuentra en `$SPARK_HOME`. En este caso indicamos que escuche arranque en `0.0.0.0` para que escuche sin necesidad de hacer resolución de nombres *(Lo veremos cuando configuremos Spark en nuestro cluster)*

```
#Desde $Spark_HOME
./sbin/start-master.sh -h 0.0.0.0
starting org.apache.spark.deploy.master.Master, logging to /opt/hadoop-3.4.1/spark-3.5.4/logs/spark-hadoop-org.apache.spark.deploy.master.Master-1-master.out
```

Recordamos que también se levanta un WebUI en el puerto 8080

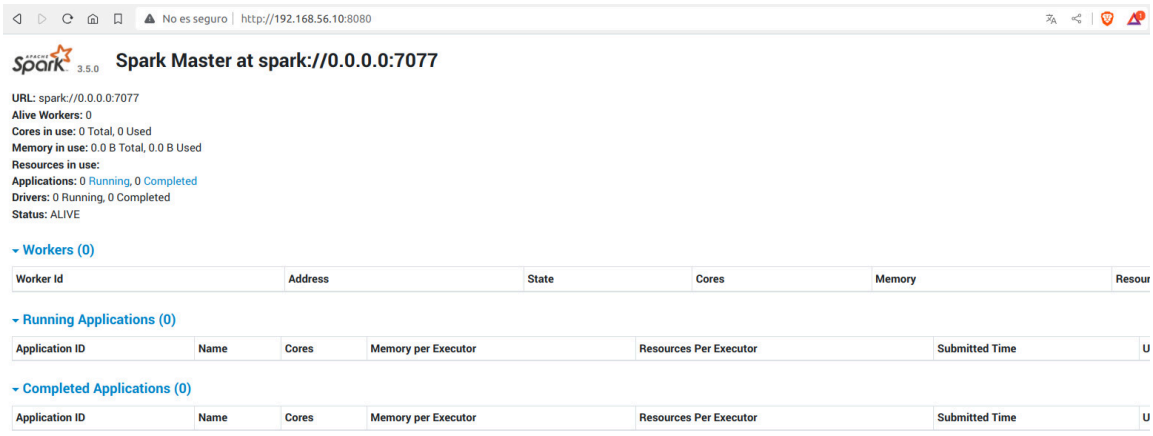

Figura 6.10_Spark: WebUI de Spark - master

1. Iniciamos un Spark Worker. Este **puede ser iniciado tanto en el mismo nodo del master, como en otro nodo**. Hacemos uso del script `./sbin/start-worker.sh` que se encuentra en `$SPARK_HOME` . Ahora si, le pasamos la IP del master. En nuestro caso `192.168.11.10`

```
./sbin/start-worker.sh spark://192.168.11.10:7077
```

Voy a levantar 2, uno en el mismo nodo donde inicio Spark master y otro en el nodo 1 (Recuerda las IPs que tiene, que van a aparecer en la WebUI)

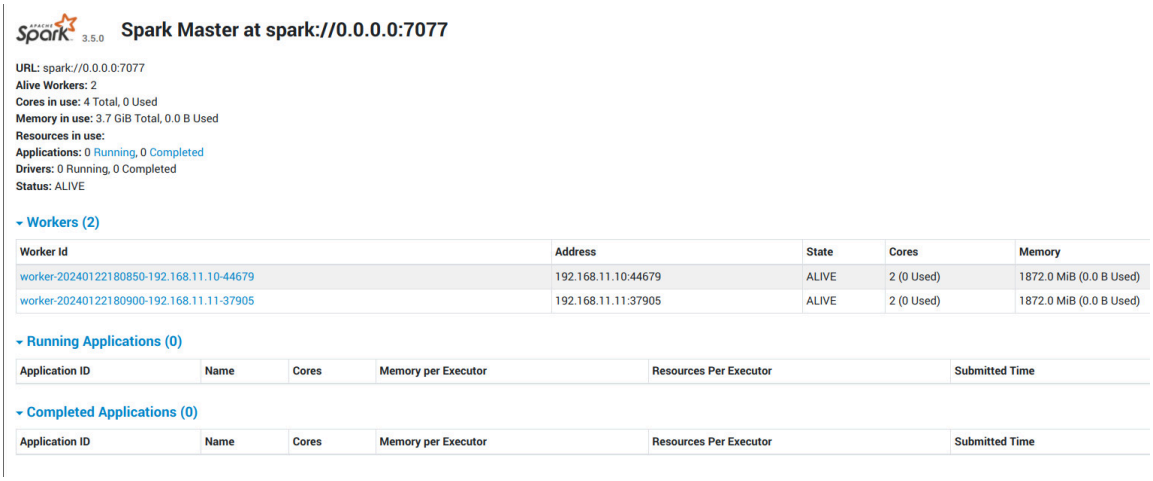

Figura 6.11_Spark: WebUI de Spark - Workers

Ya podemos ejecutar aplicaciones autónomas con `spark-submit`

8.5 Ejemplo Aplicación autónoma

1. Creamos un archivo `Ejemplo1_spark.py` con el siguiente código

Ejemplo1_spark.py

```
import sys
from pyspark.sql import SparkSession #SparkSession es el punto de entrada para crear RDD, DataFrames y DataSets

spark = SparkSession.builder.appName("Ejemplo1_spark").getOrCreate() # Dar nombre a la app

logFile = "file:///opt/hadoop-3.4.1/spark-3.5.4/README.md" #"YOUR_SPARK_HOME/README.md" Should be some file on your system
logData = spark.read.text(logFile).cache() # Carga de datos en caché

numAs = logData.filter(logData.value.contains('a')).count() # Contamos el número de veces que aparece el carácter a
numBs = logData.filter(logData.value.contains('b')).count() # Contamos el número de veces que aparece el carácter b

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

logData.show()

spark.stop()
```

1. Ejecutamos nuestra aplicación con `spark-submit`. Recuerda que tienes que tener al menos arrancado spark master y al menos un worker

```
spark-submit --master spark://192.168.11.10:7077 Ejemplo1_spark.py
```

```
24/01/22 18:16:26 INFO BlockManagerInfo: Removed broadcast_2_piece0 on cluster-bda:43799 in memory (size: 13.3 KiB, free: 366.2 MiB)
24/01/22 18:16:26 INFO BlockManagerInfo: Removed broadcast_2_piece0 on 192.168.11.11:35935 in memory (size: 13.3 KiB, free: 366.2 MiB)
24/01/22 18:16:26 INFO CodeGenerator: Code generated in 19.80941 ms
+--------------------+
|               value|
+--------------------+
|      # Apache Spark|
|                    |
|Spark is a unifie...|
|high-level APIs i...|
|supports general ...|
|rich set of highe...|
|pandas API on Spa...|
|and Structured St...|
|                    |
|<https://spark.ap...|
|                    |
|[![GitHub Actions...|
|[![AppVeyor Build...|
|[![PySpark Covera...|
|[![PyPI Downloads...|
```

```
|                    |
|                    |
|## Online Documen...|
|                    |
|You can find the ...|
+--------------------+
only showing top 20 rows
```

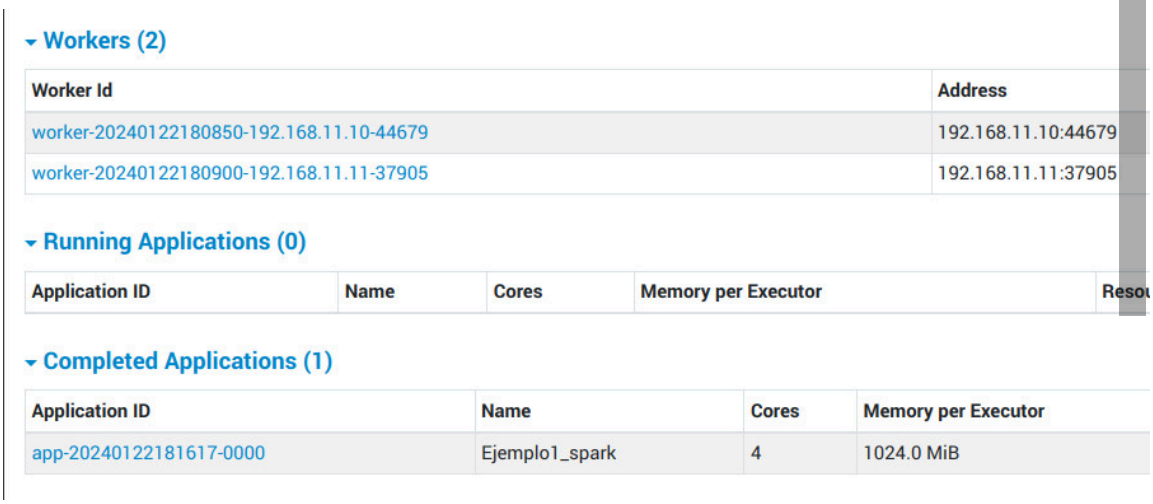

Figura 6.12_Spark: WebUI de Spark - Aplicación completada

9. Cluster Spark

9.1 Componentes

Las aplicaciones Spark se ejecutan como conjuntos independientes de procesos en un clúster, coordinados por el objeto **SparkContext** en su programa principal (llamado **driver program**).

Específicamente, para ejecutarse en un clúster, **SparkContext** puede conectarse a varios tipos de administradores de clústeres (ya sea el administrador de clústeres independiente de Spark, Mesos, YARN o Kubernetes), que asignan recursos entre aplicaciones. Una vez conectado, Spark adquiere **executors** en los nodos del clúster, que son procesos que ejecutan cálculos y almacenan datos para su aplicación. A continuación, envía el código de su aplicación (definido por archivos JAR o Python pasados a SparkContext) a los executors. Finalmente, SparkContext envía tareas a los executors para que las ejecuten.

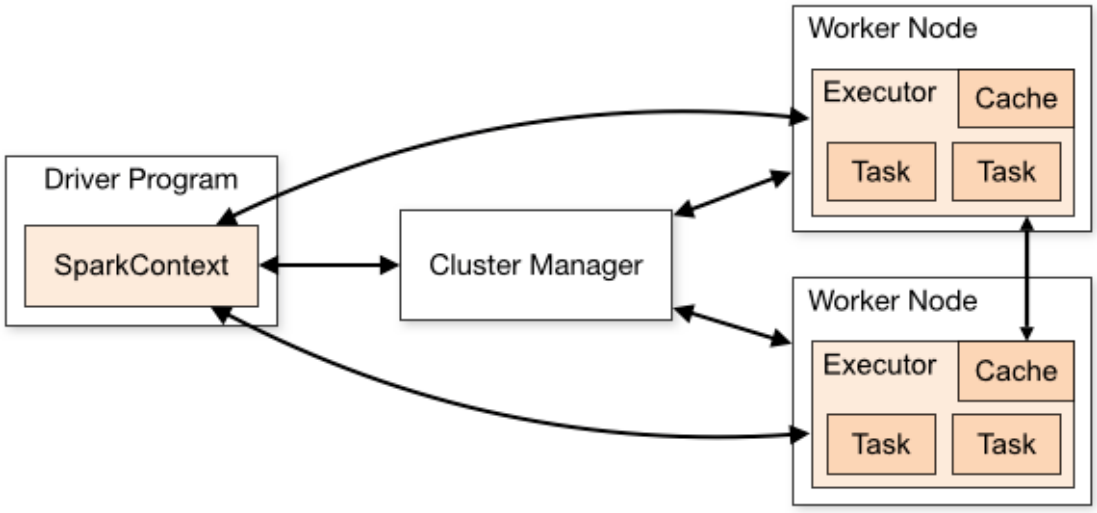

Figura 6.13_Spark: Cluster Overview (Fuente: spark.apache.org)

Hay varias cosas útiles a tener en cuenta sobre esta arquitectura:

- Cada aplicación tiene sus propios procesos executors, que permanecen activos durante toda la aplicación y ejecutan tareas en múltiples subprocesos. Esto tiene la ventaja de aislar las aplicaciones entre sí, tanto en el lado de la programación (cada controlador programa sus propias tareas) como en el lado del executor (las tareas de diferentes aplicaciones se ejecutan en diferentes JVM). Sin embargo, también significa que los datos no se pueden compartir entre diferentes aplicaciones Spark (instancias de SparkContext) sin escribirlos en un sistema de almacenamiento externo.
- Spark es independiente del *cluster manager* subyacente. Siempre que pueda adquirir procesos executors y estos se comuniquen entre sí, es relativamente fácil ejecutarlo incluso en un cluster manager que también admita otras aplicaciones (por ejemplo, Mesos/YARN/Kubernetes).
- El driver program debe escuchar y aceptar conexiones entrantes de sus ejecutores durante toda su vida útil (por ejemplo, consulte spark.driver.port en la sección de configuración de red). Como tal, el driver program debe ser direccionable en red desde los nodos workers.
- Debido a que el driver controla las tareas en el clúster, debe ejecutarse cerca de los nodos worker, preferiblemente en la misma red de área local. Si desea enviar solicitudes al clúster de forma remota, es mejor abrir un RPC para el controlador y hacer que envíe operaciones desde cerca que ejecutar un controlador lejos de los nodos worker.

9.2 Tipos de cluster manager

Actualmente, el sistema admite varios administradores de clústeres:

- **Standalone**: Un simple cluster manager incluido con Spark que facilita la configuración de un clúster.

- **Apache Mesos**: un cluster manager general que también puede ejecutar Hadoop MapReduce y aplicaciones de servicio. (Deprecated)
- **Hadoop YARN**: el resource manager en Hadoop 3.
- **Kubernetes**: un sistema de código abierto para automatizar la implementación, el escalado y la gestión de aplicaciones en contenedores.

9.3 Job Scheduling

Spark brinda control sobre la asignación de recursos tanto entre aplicaciones (en el nivel del cluster manager) como dentro de las aplicaciones (si se realizan múltiples cálculos en el mismo SparkContext). El job scheduling overview describe esto con más detalle.

9.4 Configuración del cluster "Standalone"

1. Sigue los pasos del punto anterior de Instalación para cada uno de los nodos del cluster
2. **Sólo en el nodo donde vas a lanzar Spark master** realiza las siguiente configuraciones
3. Entramos en `conf/spark-env.sh` y añadimos la IP del master a la variable de entorno `SPARK_MASTER_HOST`. En nuestro caso `192.168.11.10`:

spark-env.sh

```
export SPARK_MASTER_HOST=192.168.11.10
```

4. Entramos en `conf/workers` y añadimos los nodos que queremos que se inicien como workers:

workers

```
nodo1
nodo2
nodo3
```

Podrías iniciar tantos workers como quieras en los nodos. Sólo tienes que repetir los nodos. Comentamos `localhost` si no queremos que se inicie un worker en el master (nuestro caso)

5. Desde el nodo Spark master, iniciamos:

```
./sbin/start-master.sh
```

6. Desde el nodo Spark master, iniciamos los workers (**OJO, workers en plural**)

```
./sbin/start-workers.sh
```

Figura 6.14_Spark: Cluster con master y un worker en cada nodo (Fuente: propia)

Hadoop y Spark

No olvides tener iniciado Apache Hadoop `start-dfs.sh`