

Primeros pasos con Spark

Este laboratorio tiene el propósito de proporcionar instrucciones para que el lector se familiarice con el proceso de instalación de Apache Spark.

Luego, comenzaremos nuestra primera interacción con Apache Spark haciendo un par de ejercicios prácticos con Spark CLI, conocido como REPL.

Pasaremos a analizar los componentes de Spark y las terminologías comunes asociadas con Spark y, finalmente, analizaremos el ciclo de vida de un trabajo de Spark.

Ejecutaremos Apache Spark en modo local. Primero configuraremos Scala, que es el requisito previo para Apache Spark. Después de la configuración de Scala, configuraremos y ejecutaremos Apache Spark. También realizaremos algunas operaciones básicas sobre él.

Necesitamos Scala o Python ya que el shell no está disponible para Java

Dado que Apache Spark está escrito en Scala, necesita que Scala esté configurado en el sistema. Puede descargar Scala desde <http://www.scala-lang.org/download/>

Ahora es hora de descargar Apache Spark. Puede descargarlo desde <http://spark.apache.org/downloads.html>.

La instalación de Apache Spark implica **extraer el archivo descargado** a la ubicación deseada. Cree una nueva carpeta llamada *Spark* en la raíz de su unidad C:.

Busque el archivo Spark que descargó. Haga clic derecho en el archivo y extráigalo a *C:\Spark*. Ahora, su carpeta *C:\Spark* tiene una nueva carpeta *spark-2.4.5-bin-hadoop2.7* con los archivos necesarios dentro.

Descargue el **archivo winutils.exe** para la versión subyacente de Hadoop para la instalación de Spark que descargó. Navegue a esta URL <https://github.com/cdarlint/winutils> y dentro de la **carpeta bin**, busque **winutils.exe** y haga clic en él.

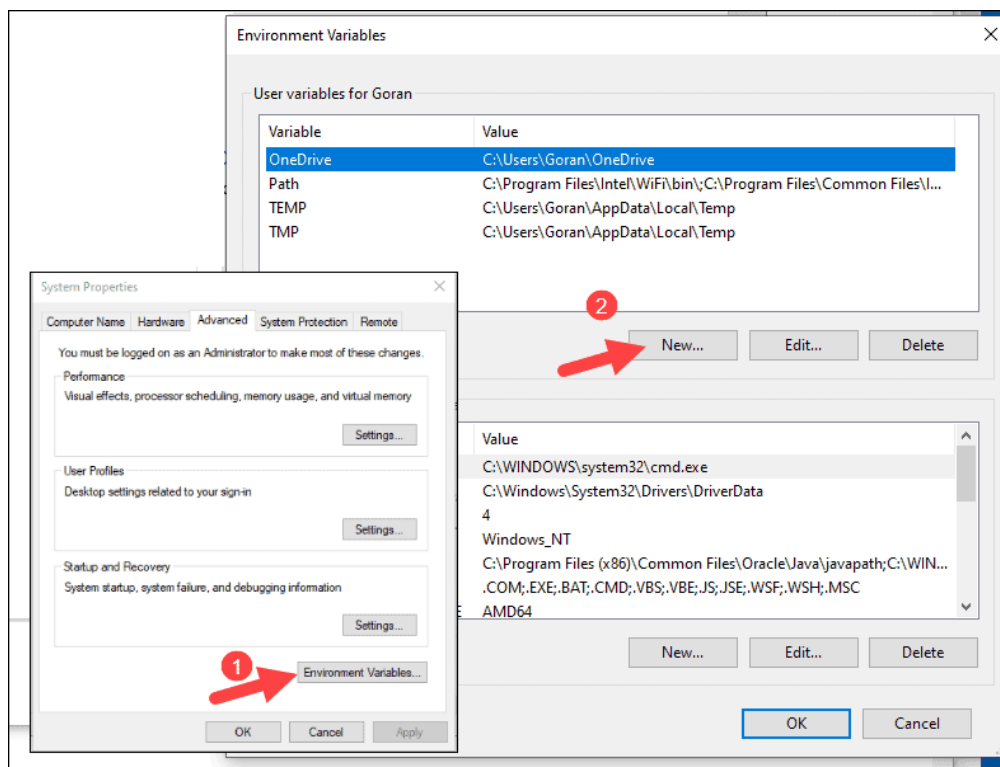
 mapred	some binaries from 273 to 311
 mapred.cmd	some binaries from 273 to 311
 rcc	some binaries from 273 to 311
 winutils.exe	fixed exe and lib 265-312
 winutils.pdb	fixed exe and lib 265-312
 yarn	some binaries from 273 to 311
 yarn.cmd	some binaries from 273 to 311

Busque el **botón Descargar** en el lado derecho para descargar el archivo. Ahora, cree nuevas carpetas **Hadoop** y **bin** en C: usando el Explorador de Windows o el Símbolo del sistema. Copie el archivo winutils.exe de la carpeta Descargas a **C:\hadoop\bin** .

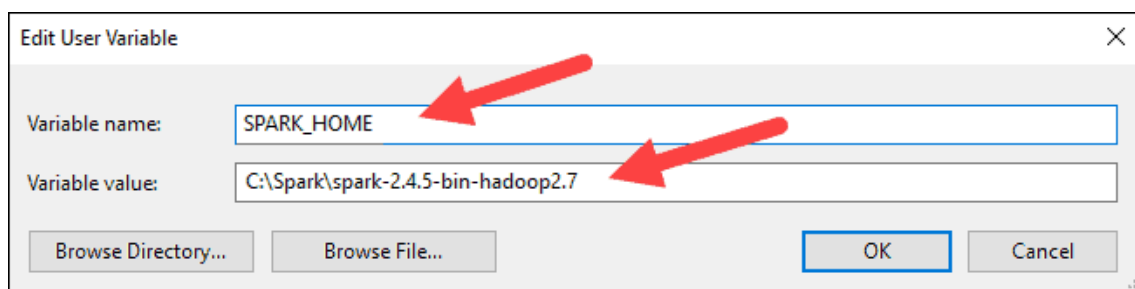
La configuración [de variables de entorno en Windows](#) agrega las ubicaciones de Spark y Hadoop a la RUTA de su sistema. Le permite ejecutar el shell de Spark directamente desde una ventana del símbolo del sistema.

Haga clic en **Inicio** y escriba *entorno*. Seleccione el resultado etiquetado como **Editar las variables de entorno del sistema** .

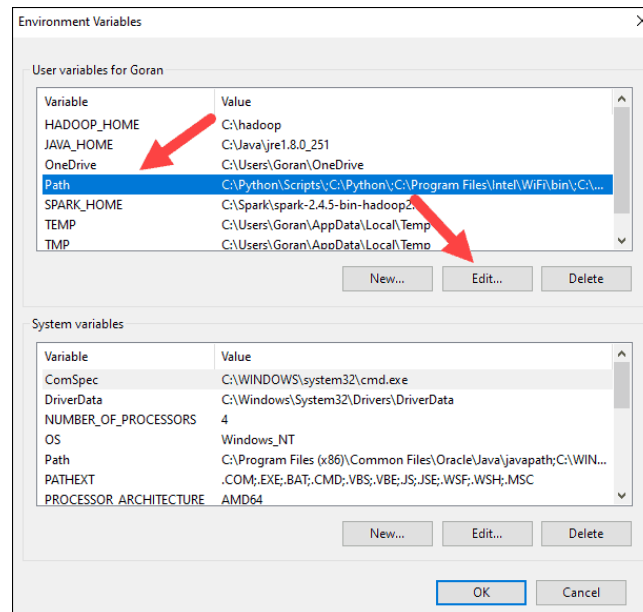
Aparece un cuadro de diálogo Propiedades del sistema. En la esquina inferior derecha, haga clic en **Variables de entorno** y luego haga clic en **Nuevo** en la siguiente ventana.



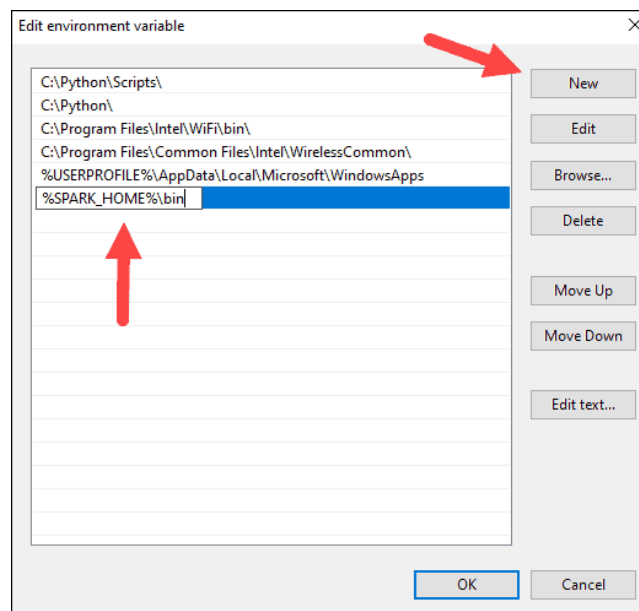
Para *Nombre de variable*, escriba **SPARK_HOME** . Para *Valor de variable*, escriba **C:\Spark\spark-2.4.5-bin-hadoop2.7** y haga clic en Aceptar. Si cambió la ruta de la carpeta, use esa en su lugar.



En el cuadro superior, haga clic en la **entrada Ruta**, luego haga clic en **Editar** . Tenga cuidado al editar la ruta del sistema. Evite eliminar cualquier entrada que ya esté en la lista.




Debería ver un cuadro con entradas a la izquierda. A la derecha, haz clic en **Nuevo**. El sistema resalta una nueva línea. Ingrese la ruta a la carpeta Spark **C:\Spark\spark-2.4.5-bin-hadoop2.7\bin**. Recomendamos usar **%SPARK_HOME%\bin** para evitar posibles problemas con la ruta.



Repita este proceso para Hadoop y Java.


Para Hadoop, el nombre de la variable es **HADOOP_HOME** y para el valor use la ruta de la carpeta que creó anteriormente: **C:\hadoop**. Agregue **C:\hadoop\bin** al campo de la **variable Ruta** , pero recomendamos usar **%HADOOP_HOME%\bin** .

Para Java, el nombre de la variable es **JAVA_HOME** y para el valor use la ruta a su directorio Java JDK (en nuestro caso es **C:\Program Files\Java\jdk1.8.0_251**).

Para iniciar Spark, ingrese en 

Abra un navegador web y vaya a **http://localhost:4040/**

Debería ver una interfaz de usuario web de shell de Apache Spark. El siguiente ejemplo muestra la *página Ejecutores*

2.4.5

JobsStagesStorageEnvironmentExecutors

Spark shell application U

Executors

[Show Additional Metrics](#)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	To
Active(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0
Total(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0

Executors

Show 20 entries

Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Con
driver	DESKTOP-SFBGHOU:61547	Active	0	0.0 B / 434 MB	0.0 B	4	0	0	0

Showing 1 to 1 of 1 entries

Previous1Next

Algunos ejercicios básicos usando Spark shell

Tenga en cuenta que Spark shell solo está disponible en el lenguaje Scala. Sin embargo, he propuesto ejemplos sencillos y enfocados a los desarrolladores de Java.

Ejecute el siguiente comando para verificar la versión de Spark:

```
sc.version
```

Comencemos creando un RDD de strings:

```
val  
stringRdd=sc.parallelize(Array("Java","Scala","Python","Ruby","JavaScript",  
"Java"))
```

Ahora, filtraremos este RDD para mantener solo aquellas cadenas que comiencen con la letra J:

```
val filteredRdd = stringRdd.filter(s => s.startsWith("J"))
```

En el primer capítulo, aprendimos que si una operación en RDD devuelve un RDD entonces es una transformación.

El resultado del comando anterior muestra claramente que filtrar la operación devolvió un RDD, por lo que el filtro es una transformación.

Ahora, vamos a ejecutar una acción en filteredRdd para ver sus elementos.

```
val list = filteredRdd.collect
```

La operación devolvió una matriz de cadenas. Entonces, es una acción.

Ahora, veamos los elementos de la variable list:

```
list
```

Nos quedamos solo con elementos que comienzan con J, que era nuestro resultado deseado:

```
scala> val stringRdd=sc.parallelize(Array("Java","Scala","Python","Ruby","JavaScript","Java"))  
stringRdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[7] at parallelize at <console>:24  
  
scala> val filteredRdd = stringRdd.filter(s => s.startsWith("J"))  
filteredRdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[8] at filter at <console>:26  
  
scala> val list = filteredRdd.collect  
list: Array[String] = Array(Java, JavaScript, Java)  
  
scala> list  
res3: Array[String] = Array(Java, JavaScript, Java)  
  
scala> 
```

Hagamos un problema de conteo de palabras en stringRDD. El conteo de palabras es el HelloWorld del mundo del Big Data. Word count significa que contaremos la aparición de cada palabra en el RDD:

Así que primero vamos a crear pairRDD como sigue:

```
val pairRDD=stringRdd.map( s => (s,1))
```

El pairRDD consta de pares de la palabra y un entero donde la palabra representa cadenas de stringRDD.

Ahora, ejecutaremos el reduceByKey operación en este RDD para contar la aparición de cada palabra de la siguiente manera:

```
val wordCountRDD=pairRDD.reduceByKey((x,y) => x+y)
```

Ahora, vamos a correr collect en él para ver el resultado:

```
val wordCountList=wordCountRDD.collect
```

```
wordCountList
```

Según la salida de wordCountList, cada cadena en stringRDD aparece una vez excepto Java, que apareció dos veces.

Se muestra en la siguiente captura de pantalla:

```
scala> val pairRDD=stringRdd.map( s => (s,1))
pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[9] at map at <console>:26

scala> val wordCountRDD=pairRDD.reduceByKey((x,y) => x+y)
wordCountRDD: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[10] at reduceByKey at <console>:28

scala> val wordCountList=wordCountRDD.collect
wordCountList: Array[(String, Int)] = Array((Python,1), (JavaScript,1), (Java,2), (Scala,1), (Ruby,1))

scala> wordCountList
res4: Array[(String, Int)] = Array((Python,1), (JavaScript,1), (Java,2), (Scala,1), (Ruby,1))

scala> █
```

Vamos ahora a encontrar la suma de todos los números pares en un RDD de enteros. Primero creemos un RDD de enteros de la siguiente manera:

```
val intRDD = sc.parallelize(Array(1,4,5,6,7,10,15))
```

El siguiente paso es filtrar todos los elementos pares en este RDD. Entonces, ejecutaremos un filter en el RDD, de la siguiente manera:

```
val evenNumbersRDD=intRDD.filter(i => (i%2==0))
```

La operación anterior obtendrá aquellos elementos pares. Ahora, sumaremos todos los elementos de este RDD de la siguiente manera:

```
val sum = evenNumbersRDD.sum
```

```
sum
```

Se muestra en la siguiente captura de pantalla:

```
scala> val intRDD = sc.parallelize(Array(1,4,5,6,7,10,15))
intRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val evenNumbersRDD=intRDD.filter(i => (i%2==0))
evenNumbersRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at filter at <console>:26

scala> val sum = evenNumbersRDD.sum
sum: Double = 20.0

scala> sum
res0: Double = 20.0

scala> █
```

Vamos ahora a contar el número de palabras en un archivo. Leamos el archivo people.txt. El método `textFile()` se puede utilizar para leer el archivo de la siguiente manera:

```
val file=sc.textFile("file:///C:/Users/Usuario/Desktop/AF-PYSPARK/RECURSOS/people.txt")
```

El siguiente paso es aplanar el contenido del archivo, es decir, crearemos un RDD dividiendo cada línea con “ , “

```
val flattenFile = file.flatMap(s => s.split(", " ))
```

Los contenidos de `flattenFileRDD` se ve de la siguiente manera:

```
flattenFile.collect
```

Ahora, podemos contar todas las palabras en este RDD de la siguiente manera:

```
val count = flattenFile.count
```

```
count
```

Se muestra en la siguiente captura de pantalla:

```
scala> val file=sc.textFile("/usr/local/spark/examples/src/main/resources/people.txt")
file: org.apache.spark.rdd.RDD[String] = /usr/local/spark/examples/src/main/resources/people.txt MapPartitionsRDD[4] at textFile at <console>:24

scala> val flattenFile = file.flatMap(s => s.split(", "))
flattenFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at flatMap at <console>:26

scala> flattenFile.collect
res1: Array[String] = Array(Michael, 29, Andy, 30, Justin, 19)

scala> val count = flattenFile.count
count: Long = 6

scala> count
res2: Long = 6

scala> █
```

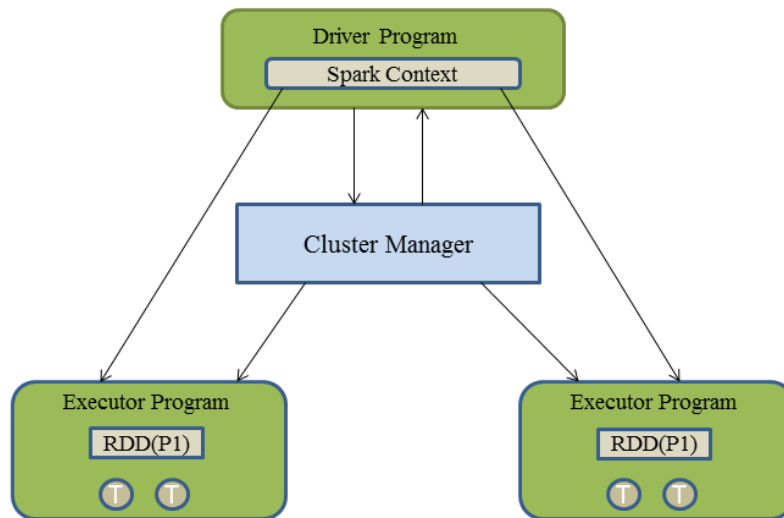
Cada vez que cualquier acción como count recibe una llamada, Spark crea un **gráfico acíclico dirigido (DAG)**.

Componentes de Spark

Antes de continuar, primero comprendamos las terminologías comunes asociadas con Spark:

- **Controlador:** este es el programa principal que supervisa la ejecución de un extremo a otro de un trabajo o programa de Spark. Negocia los recursos con el administrador de recursos del clúster para delegar y orquestar el programa en la unidad de programación paralela local de datos más pequeña posible.
- **Ejecutores:** En cualquier trabajo de Spark, puede haber uno o más ejecutores, es decir, procesos que ejecutan tareas más pequeñas delegadas por el controlador.
- **Maestro:** Apache Spark se ha implementado en la arquitectura maestro-esclavo y, por lo tanto, maestro se refiere al nodo del clúster que ejecuta el programa del controlador.
- **Esclavo:** en un modo de clúster distribuido, esclavo se refiere a los nodos en los que se ejecutan los ejecutores y, por lo tanto, puede haber más de un esclavo en el clúster.
- **Trabajo:** Esta es una colección de operaciones realizadas en cualquier conjunto de datos. Un trabajo típico de conteo de palabras consiste en leer un archivo de texto de una fuente arbitraria y dividir y luego agregar las palabras.
- **DAG:** cualquier trabajo de Spark en un motor de Spark está representado por un DAG de operaciones. El DAG representa la ejecución lógica de las operaciones de Spark en un orden secuencial.
- **Tareas:** Un trabajo se puede dividir en unidades más pequeñas para operar en silos que se denominan **Tareas**. Cada tarea es ejecutada por un ejecutor en una partición de datos.
- **Etapas:** los trabajos de Spark se pueden dividir lógicamente en etapas, donde cada etapa representa un conjunto de tareas que tienen las mismas dependencias aleatorias, es decir, donde se produce la combinación aleatoria de datos.

El siguiente diagrama muestra una representación lógica de cómo interactúan los diferentes componentes de la aplicación Spark:



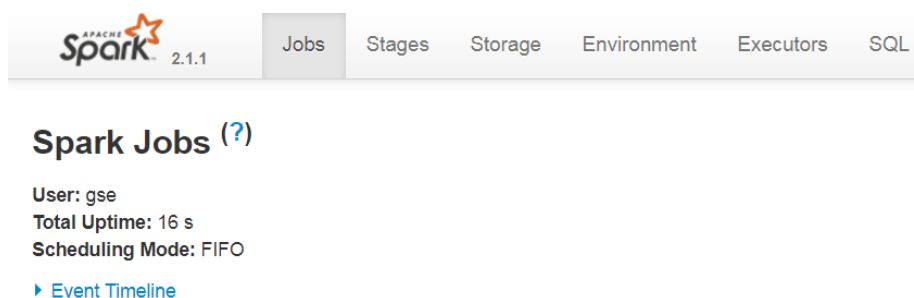
Un Spark Job puede constar de una serie de operaciones que se realizan sobre un conjunto de datos. Por grande o pequeño que sea un trabajo de Spark, requiere un SparkContext para ejecutar cualquier trabajo. En los ejemplos anteriores de trabajar con REPL, uno notaría el uso de una variable de entorno llamada `sc`, que es como un SparkContext es accesible en un entorno REPL.

Interfaz de usuario web del controlador Spark

Esta sección proporcionará algunos aspectos importantes de la interfaz de usuario del controlador Spark. Veremos las estadísticas de los trabajos que ejecutamos usando Spark Shell en Spark UI.

La interfaz de usuario del controlador de Spark se ejecuta en <http://localhost:4040/>

Cuando inicie Spark shell, la interfaz de usuario del controlador de Spark tendrá el siguiente aspecto:



SparkContext es un punto de entrada a todas las aplicaciones de Spark. Cada trabajo de Spark se inicia con un SparkContext y puede constar de una sola SparkContext.

Spark shell, siendo una aplicación Spark comienza con SparkContext y cada SparkContext lanza su propia interfaz de usuario web. El puerto predeterminado es 4040. Spark UI se puede habilitar/deshabilitar o se puede iniciar en un puerto separado usando las siguientes propiedades:

Propiedad	Valor por defecto
spark.ui.enabled	True
spark.ui.port	4040

Por ejemplo, la aplicación Spark Shell con Spark UI ejecutándose en 5050El puerto se puede lanzar como:

```
spark-shell --confspark.ui.port=5050
```

Si se inician varias aplicaciones Spark en paralelo en un sistema sin proporcionar ninguno de los anteriores confparámetros, luego Spark UI para esas aplicaciones se iniciará en puertos sucesivos a partir de 4040.

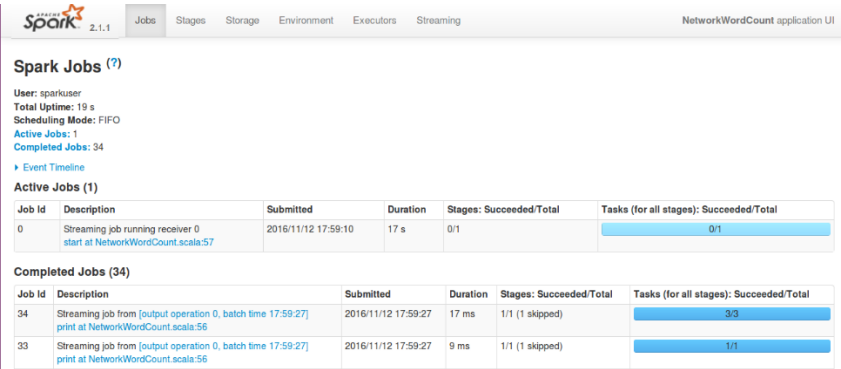
Spark UI consta de las siguientes pestañas:

Jobs es la pestaña predeterminada de Spark UI. Muestra el estado de todas las aplicaciones ejecutadas dentro de un SparkContext.

Consta de tres secciones:

- **Active Jobs:** esta sección es para los trabajos que se están ejecutando actualmente
- **Completed Jobs:** Esta sección es para los trabajos que se completaron con éxito
- **Failed Jobs:** Esta sección es para los trabajos que fallaron

Se muestra en la siguiente captura de pantalla:

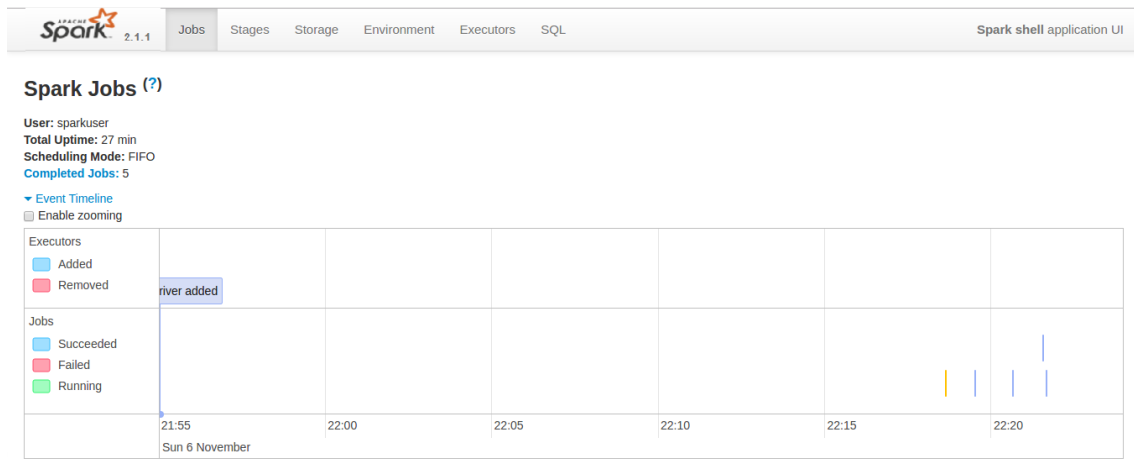


The screenshot shows the Spark UI interface with the 'Jobs' tab selected. It displays the 'Spark Jobs' section with a summary of active and completed jobs. Below the summary, there are two tables: 'Active Jobs (1)' and 'Completed Jobs (34)'. The 'Active Jobs' table shows a single job (ID 0) that is a streaming job running receiver 0. The 'Completed Jobs' table shows two jobs (IDs 34 and 33) that are streaming jobs from [output operation 0, batch time 17:59:27].

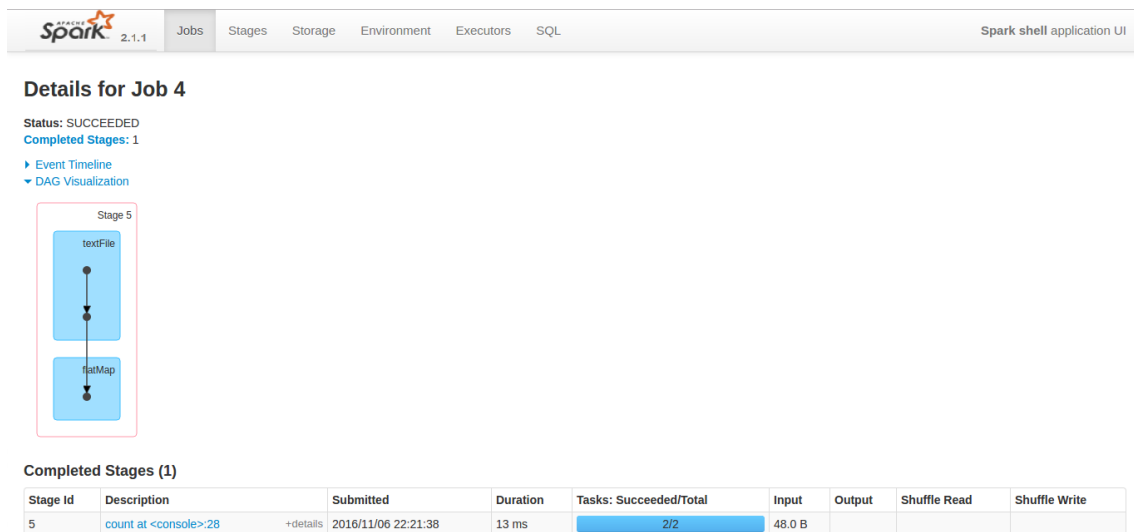
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	Streaming job running receiver 0 start at NetworkWordCount.scala:57	2016/11/12 17:59:10	17 s	0/1	0/1

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
34	Streaming job from [output operation 0, batch time 17:59:27] print at NetworkWordCount.scala:56	2016/11/12 17:59:27	17 ms	1/1 (1 skipped)	3/3
33	Streaming job from [output operation 0, batch time 17:59:27] print at NetworkWordCount.scala:56	2016/11/12 17:59:27	9 ms	1/1 (1 skipped)	1/1

Además, si expande el **Event Timeline**, se puede ver el tiempo en el que se inicio SparkContext (es decir, se inició el controlador) y los trabajos que se ejecutaron junto con su estado:



Además, al hacer clic en cualquiera de los trabajos, puede ver los detalles del trabajo, es decir, el **Event Timeline** del trabajo y el DAG de las transformaciones y etapas ejecutadas durante la ejecución del trabajo, así:



Stages enumera y proporciona el estado actual de todas las etapas de cada trabajo ejecutado en el SparkContext.

La pestaña también consta de tres secciones:

- **Active Stages:** Esta sección es para las etapas de un activo que se están ejecutando actualmente
- **Completed Stages:** Esta sección es para las etapas de trabajos activos/fallidos/completados que se completaron con éxito
- **Failed Stages:** Esta sección es para las etapas de trabajos activos/fallidos/completados que fallaron

Si hace clic en cualquiera de las etapas, puede ver varios detalles de la etapa, es decir, el DAG de tareas ejecutadas, **Event Timeline**, etcétera. Además, esta página proporciona métricas de varios detalles de ejecutores y tareas ejecutadas en la etapa:



Si durante la ejecución de un trabajo, el usuario guarda en caché un RDD, la información sobre ese RDD se puede recuperar en esta pestaña.

Iniciemos Spark Shell nuevamente, leamos un archivo y ejecutemos una acción en él. Sin embargo, esta vez almacenaremos en caché el archivo antes de ejecutar una acción en él.

Inicialmente, cuando inicia Spark shell, el **Storage** aparece en blanco.



Leamos el archivo usando SparkContext, como sigue:

```
val file=sc.textFile("file:///C:/Users/Usuario/Desktop/AF-PYSPARK/RECURSOS/people.txt")
```

Esta vez almacenaremos en caché este RDD. De forma predeterminada, se almacenará en caché en la memoria:

```
file.cache
```

Como se explicó anteriormente, el DAG de transformaciones solo se ejecutará cuando se realice una acción, por lo que el paso de caché también se ejecutará cuando ejecutemos una acción en el RDD. Así que vamos a ejecutar una recopilación en él:

file.collect

Ahora, puede encontrar información sobre un RDD que se almacena en caché en el **Storage**.

<div>Spark 2.1.1</div> <div>JobsStagesStorageEnvironmentExecutorsSQL</div> <div>Spark shell application UI</div>					
Storage					
RDDs					
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
/usr/local/spark/examples/src/main/resources/people.txt	Memory Deserialized 1x Replicated	2	100%	232.0 B	0.0 B

Si hace clic en el nombre de RDD, proporciona información sobre las particiones en el RDD junto con la dirección del host en el que se almacena el RDD.

ASAP

Spark

2.1.1

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application UI

RDD Storage Info for MapPartitionsRDD

Storage Level: Memory Deserialized 1x Replicated

Cached Partitions: 2

Total Partitions: 2

Memory Size: 360.0 B

Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	Memory Usage	Disk Usage
192.168.0.106:38543	360.0 B (366.3 MB Remaining)	0.0 B

2 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_9_0	Memory Deserialized 1x Replicated	232.0 B	0.0 B	192.168.0.106:38543
rdd_9_1	Memory Deserialized 1x Replicated	128.0 B	0.0 B	192.168.0.106:38543

El **Environment** tenemos información sobre varias propiedades y variables de entorno utilizadas en la aplicación Spark (o SparkContext).

Los usuarios pueden obtener información extremadamente útil sobre varias propiedades de Spark en esta pestaña sin tener que revisar los archivos de propiedades.

<div>Spark 2.1.1</div> <div>JobsStagesStorageEnvironmentExecutorsSQL</div> <div>Spark shell application UI</div>	
Environment	
Runtime Information	
Name	Value
Java Home	/usr/lib/jvm/java-8-oracle/jre
Java Version	1.8.0_101 (Oracle Corporation)
Scala Version	version 2.11.8
Spark Properties	
Name	Value
spark.app.id	local-1478449502954
spark.app.name	Spark shell
spark.driver.host	192.168.0.106
spark.driver.port	39321
spark.executor.id	driver
spark.home	/usr/local/spark
spark.jars	
spark.master	local[*]
spark.repl.class.outputDir	/tmp/spark-b7ebaww9-jd56-44df-87ca-aa88209248c/repl-6e37e708-f229-4b38-99e7-d04d1ac8e2ba
spark.repl.class.uri	spark://192.168.0.106:39321/classes
spark.scheduler.mode	FIFO
spark.sql.catalogImplementation	hive
spark.submit.deployMode	client

Executors proporciona información sobre la memoria, los núcleos y otros recursos que utilizan los ejecutores. Esta información está disponible tanto a nivel de cada ejecutor como a nivel agregado.

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(1)	1	16.3 KB / 366.3 MB	0.0 B	4	0	0	20	20	1.2 s (0 ms)	96.0 B	0.0 B	309.0 B
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(1)	1	16.3 KB / 366.3 MB	0.0 B	4	0	0	20	20	1.2 s (0 ms)	96.0 B	0.0 B	309.0 B

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	192.168.0.106:38543	Active	1	16.3 KB / 366.3 MB	0.0 B	4	0	0	20	20	1.2 s (0 ms)	96.0 B	0.0 B	309.0 B	Thread Dump

La pestaña **SQL** en Spark UI proporciona las consultas Spark SQL realmente útiles. Aprenderemos sobre el funcionamiento del marco Spark SQL mas adelante . Por ahora, ejecutemos los siguientes comandos en Spark Shell. Es accesible en <http://localhost:4040/SQL>.

En el siguiente ejemplo, leeremos un archivo JSON en Spark usando el SparkSession y creando vistas temporales de datos JSON y luego ejecutaremos una consulta Spark SQL en la tabla:

```
import org.apache.spark.sql.SparkSession
import spark.implicits.
```

```
val spark = SparkSession.builder().appName("Spark SQL basic example").getOrCreate()
```

```
val df = spark.read.json("C:/Users/Usuario/Desktop/AF-PYSPARK/RECURSOS/people.json")
```

```
df.createOrReplaceTempView("people")
```

```
val sqlDF = spark.sql("SELECT * FROM people")
```

```
sqlDF.show()
```

Después de ejecutar el ejemplo anterior, los detalles de la consulta SQL se pueden encontrar en la pestaña SQL.