

Spark: Breve resumen

Por qué Spark es bueno para *Data Scientists*?

Resumiendo las diapositivas anteriores

Transformaciones VS Acciones

Transformaciones: *lazy*

Acciones: *eager*

La Latencia es importante

Demasiada latencia nos hace perder mucho tiempo.

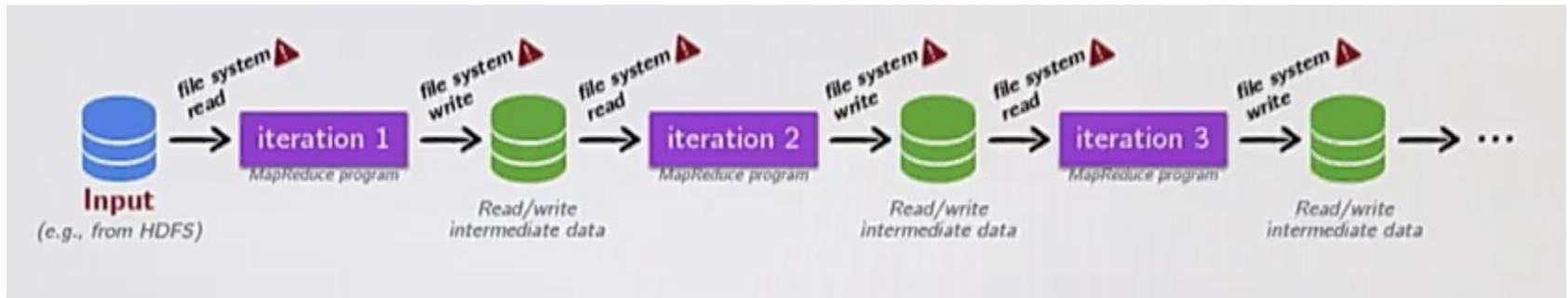
Mejor trabajar en memoria.

La mayoría de los problemas que tiene que resolver un DS contienen iteraciones

Spark: Iteraciones Hadoop vs Spark

Las iteraciones en *Hadoop* son algo así

Enorme % de tiempo perdido en IO



Las iteraciones en *Spark* son así



Spark: ejemplo iteración

En el mundo de los Data Scientists, el Hello, World es la Regresión Logística, que tiene la siguiente fórmula

$$w \leftarrow w - \alpha \cdot \sum_{l=1}^n g(w; x_l, y_l)$$

La implementación de una RL en *Spark* se puede hacer de la siguiente manera

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

→ RDD de puntos formato double

→ Vector de pesos

→ Durante un n° de iteraciones

→ iteramos sobre todo

→ el dataset (points)

→ Hacemos el reduce


→ Actualizamos el peso

Hacemos varios Map y Reduce durante el proceso

Pero hay algo que se está haciendo y que no es necesario. **Qué es??**

Spark: ejemplo iteración

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```



Está utilizando el RDD inicializado al comienzo. Al ejecutar una acción, se estará calculando el RDD 'points' en cada iteración

En cada iteración ejecutamos 2 maps y un reduce sobre todos el dataset, cosa que es totalmente innecesaria

¿Qué solución se os ocurre?

Spark: ejemplo iteración

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```



Persistir

1. Persistir el primer map en memoria para que no se tenga que evaluar en cada iteración.
2. Reusarlo en cada iteración una vez persistido. Lo tomará de memoria en vez de tener que volverlo a calcular.

Hay que recordar que un RDD es evaluado cada vez que se ejecuta una acción sobre él.

Spark: caché y persistencia

En el caso del ejemplo de la regresión logística

```
val points = sc.textFile(...).map(parsePoint).persist()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x))) - 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

De este modo, `points` es evaluado una vez y usado las veces que sean necesarias en el bucle sin tener que volver a ejecutar la carga `textFile()`

Spark: caché y persistencia

Spark permite controlar fácilmente qué se cachea y qué se persiste en memoria mediante dos métodos aplicados a los datos:

- `cache()`
- `persist()`

Volviendo al caso anterior

```
val lastYearsLogs: RDD[String] = ...  
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()  
val firstLogsWithErrors = logsWithErrors.take(10)
```

Persistimos el RDD: "*logsWithErrors*".

En este caso no sería necesario, porque solo se ejecuta una acción, que es *take(10)*. El RDD se calculará una única vez, será entonces cuando se persista en memoria, por lo que no nos beneficiamos de esta persistencia.

Pero si añadimos una línea más que contenga otra acción...

Spark: caché y persistencia

```
val lastYearsLogs: RDD[String] = ...  
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()  
val firstLogsWithErrors = logsWithErrors.take(10)  
val numErrors = logsWithErrors.count() // faster
```

Nos beneficiamos de tenerlo persistido, porque no tiene que volverse a evaluar.

Spark: caché y persistencia

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Si usamos la opción de *Memory_only* y el RDD no entra en memoria, desecha lo que no entre, y si hace falta usarlo de nuevo volverá a ejecutar la secuencia de transformaciones previas. En estos casos es muy útil la opción compartida de memoria y disco

La opción memoria y disco deja datos en disco cuando la memoria se llena

La opción memoria y disco *ser*, guarda en memoria la representación serializada de los datos

Una de las razones por las que Spark resulta ineficiente a veces es por el mal uso de estos parámetros. ✉ reevaluación

Spark: caché y persistencia

Hay varias formas de persistir los datos

Con el método `cache()`

Que establece el sistema de persistencia por defecto, que es memoria

Con el método *persist*



Que permite elegir el tipo de persistencia

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(", "))
```

Ya visto en
anteriores
diapositivas, los
tipos de persistencia
están fuera del
alcance de este
curso.

Spark: Topología

Para trabajar eficientemente con *Spark*, hay que entender cómo funciona internamente.

Ejemplo 1:

Asumimos que tenemos un RDD con objetos del tipo persona

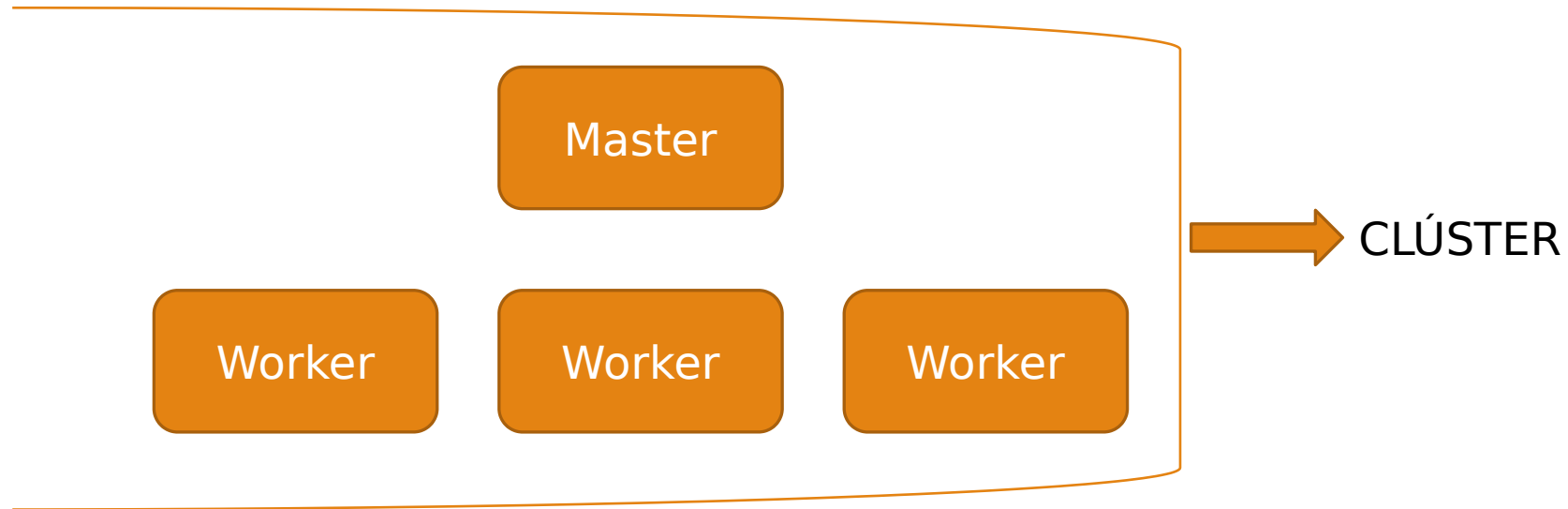
```
case class persona(nombre: string, edad: int)
```

¿Qué devolverá el siguiente código?

```
val people: RDD[persona]=....  
people.foreach(println)
```

Veremos la respuesta luego

Spark: Topología



2 modos de ejecución *SPARK*:

- Modo local (se usa para realizar pruebas)
- Modo distribuido (se usa en entornos reales)



En modo distribuido,
Spark utiliza una
arquitectura
maestro/esclavo

En modo distribuido,
Spark utiliza una
arquitectura
maestro/esclavo



Spark: Topología. ¿Cómo funciona Spark?

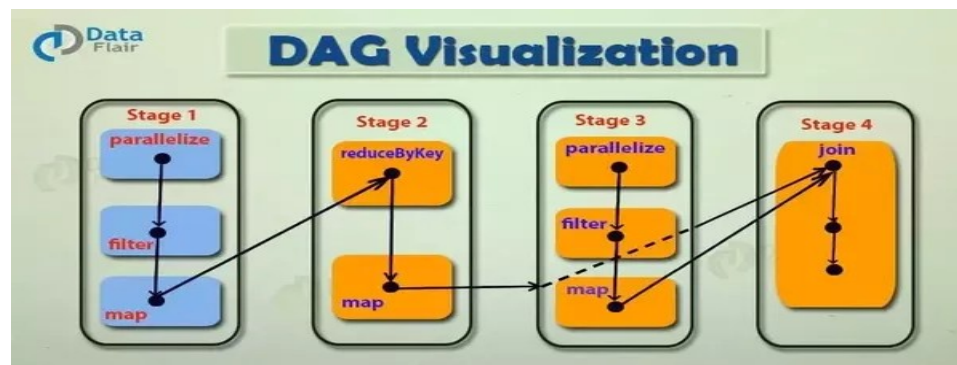
Todos los programas *Spark* tienen la misma estructura:

1. Crean RDD's para una determinada entrada.
2. Crean RDD's derivados de los originales mediante **transformaciones**
3. Realizan **acciones** para mostrar o guardar los datos.

Durante su ejecución, un programa *Spark* crea un DAG lógico (*Directed Acyclic Graph*) de operaciones. Cuando el programa se inicia, convierte el gráfico lógico en un plan de ejecución físico.

Spark realiza multitud de operaciones para transformar ese gráfico en una serie de etapas. Por su parte, cada etapa consta de multitud de tareas (*tasks*).

Las *Tasks* son la unidad de trabajo más pequeña de *Spark*. Todas las tareas son enviadas al clúster, donde se dividirán entre los diferentes ejecutores disponibles.



Spark: Topología. Driver

El **Driver** es el proceso donde corre el método `main()` del programa, crea el *sparkContext*, *RDDs* y ejecuta Transformaciones y Acciones.

El *Spark Context* está en el Driver Program (Recordar que lo usamos para crear nuevos *RDDs* (`sc.parallelize`, `sc.textFile`, ...))

Cuando se inician, los nodos Worker se registran en el driver, de modo que éste tiene una completa visión de los ejecutores todo el tiempo

Driver Program

Spark
Context

El Driver tratará de programar cada tarea en el nodo adecuado, basándose en dónde se encuentren los datos

Los workers se encargan de ejecutar las tareas (*tasks*)

Worker

Executor

Worker

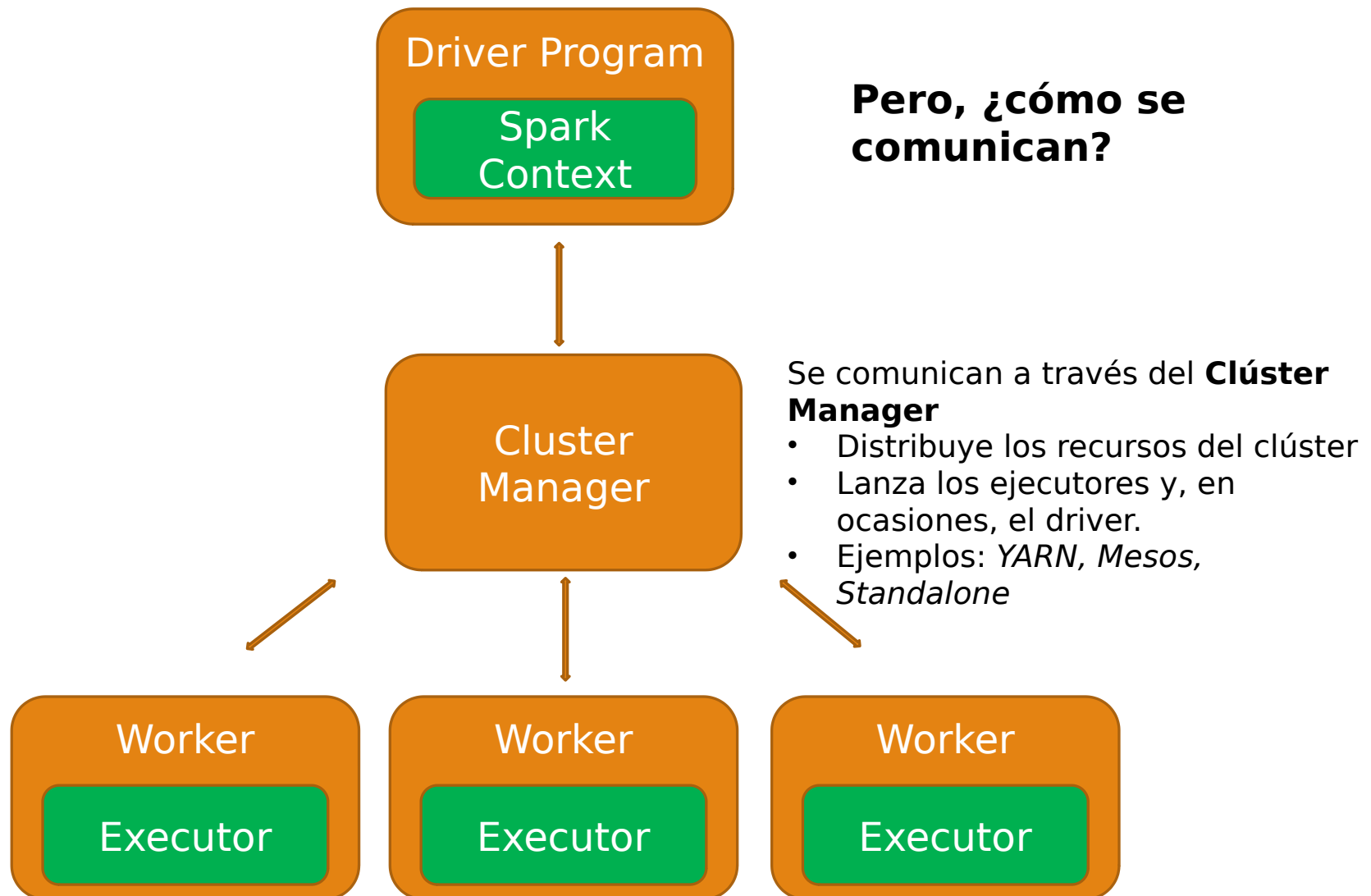
Executor

Worker

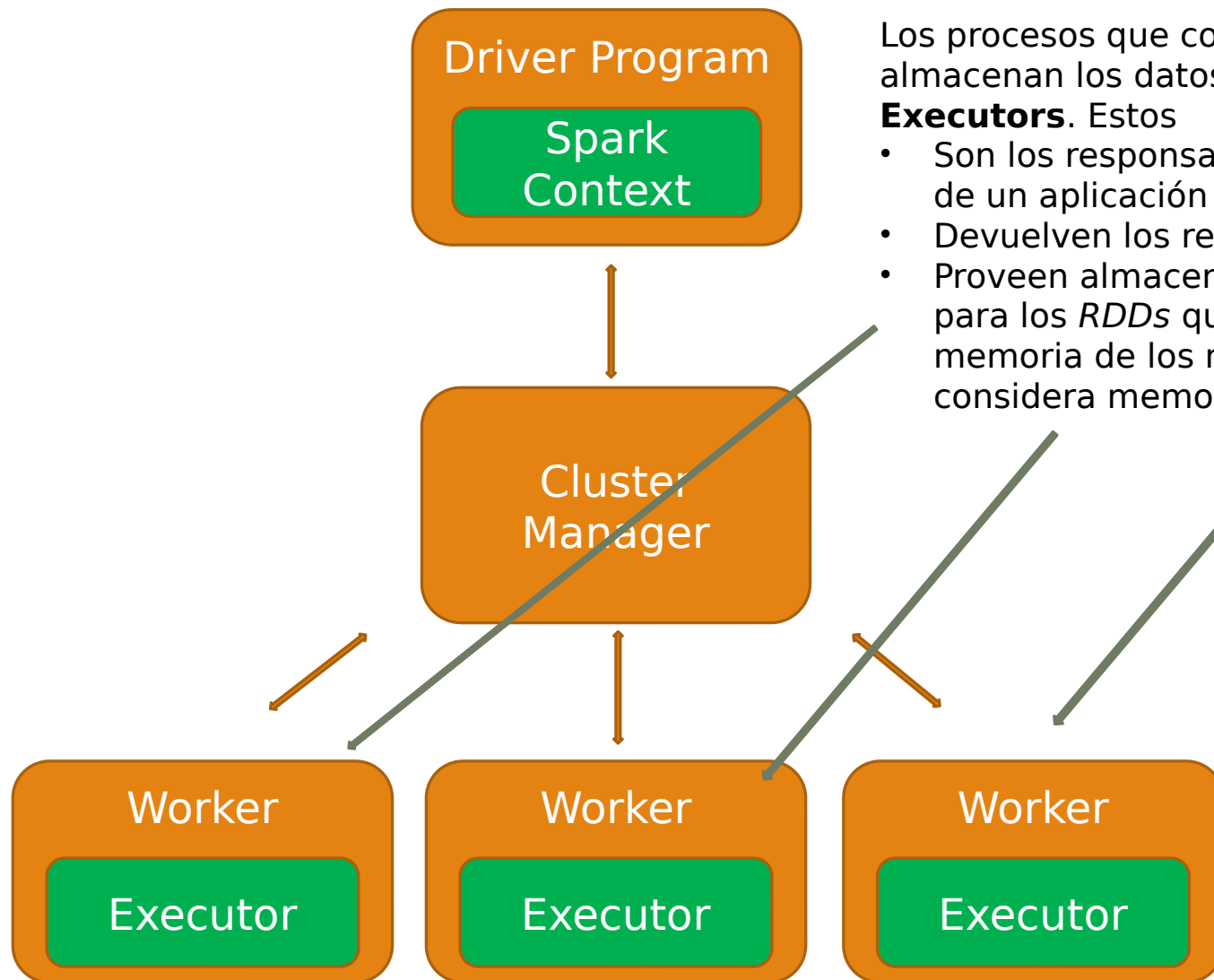
Executor

Pero, ¿cómo se comunican?

Spark: Topología. Clúster Manager



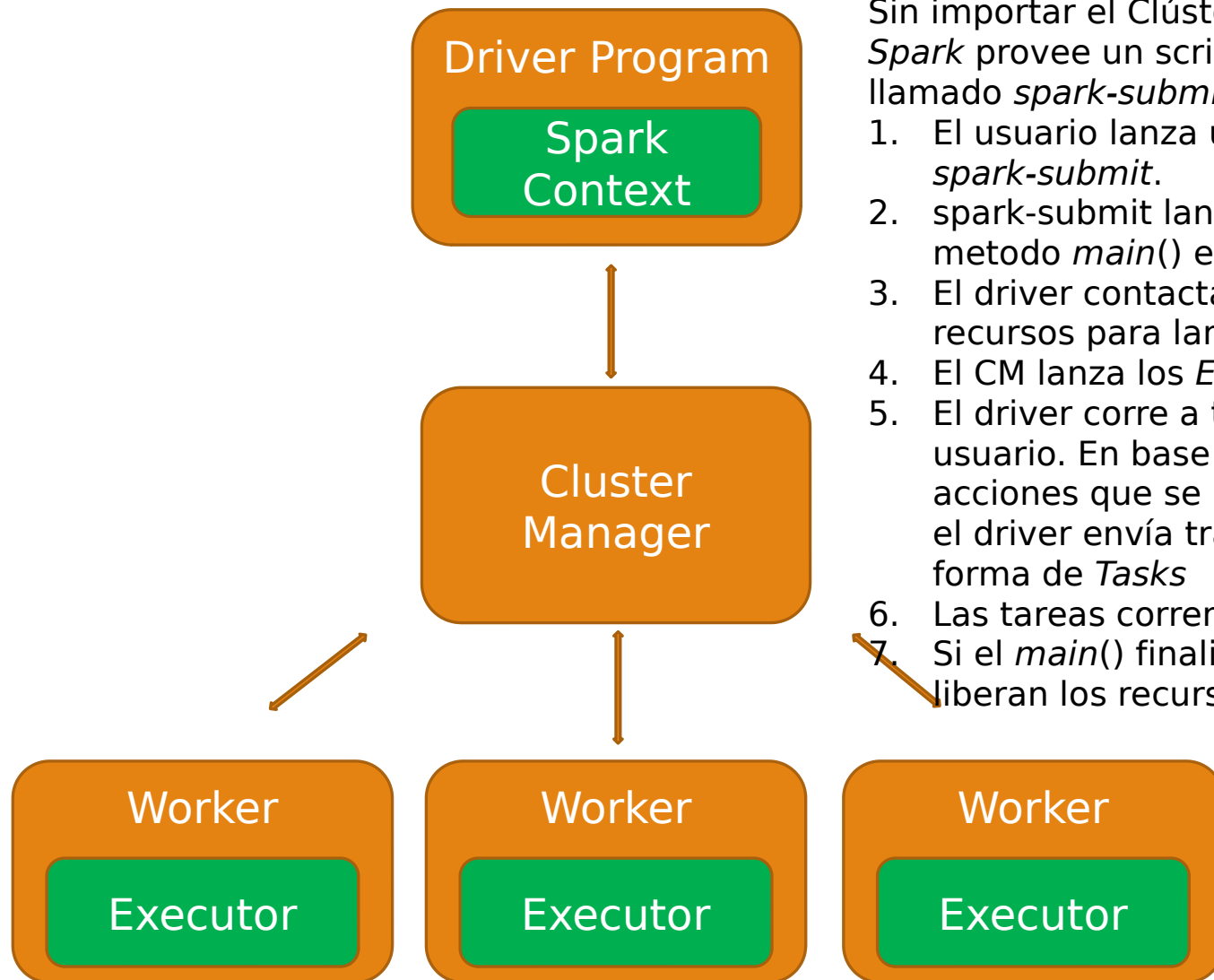
Spark: Topología. Ejecutores



Los procesos que corren la computación y almacenan los datos de la aplicación son los **Executors**. Estos

- Son los responsables de ejecutar las *Tasks* de un aplicación *Spark*.
- Devuelven los resultados al driver
- Proveen almacenamiento en memoria para los *RDDs* que han sido cacheados. La memoria de los nodos *Worker* se considera memoria del clúster.

Spark: Topología



Pasos de la ejecución de un programa Spark:

Sin importar el Clúster Manager que usemos, *Spark* provee un script para lanzar programas llamado *spark-submit*

1. El usuario lanza una aplicación usando el *spark-submit*.
2. *spark-submit* lanza el driver e invoca al metodo *main()* especificado por el usuario
3. El driver contacta el *Clúster Manager* y pide recursos para lanzar los *Executors*
4. El CM lanza los *Executors* de parte del driver
5. El driver corre a través de la aplicación del usuario. En base a las transformaciones y acciones que se han de hacer en los *RDDs*, el driver envía trabajo a los *Executors* en forma de *Tasks*
6. Las tareas corren en los *Executors*
7. Si el *main()* finaliza, finalizan los *Executors* y liberan los recursos del *Clúster Manager*

Spark: Topología

Ahora que sabemos un poco más sobre la anatomía de *Spark*

Ejemplo 1: asumimos que tenemos un RDD con objetos del tipo *persona*

```
case class persona(nombre: string, edad: int)
```

¿Qué devolverá el siguiente código?

```
val people: RDD[persona]=....  
people.foreach(println)
```

Spark: Topología

Ahora que sabemos un poco más sobre la anatomía de *Spark*

Ejemplo 1: asumimos que tenemos un RDD con objetos del tipo persona

```
case class persona(nombre: string, edad: int)
```

¿Qué devolverá el siguiente código?

```
val people: RDD[persona]=....
```

```
people.foreach(println)
```

No devuelve *NADA*

Mejor dicho, no devuelve nada el driver, dado que el proceso de `println` se está ejecutando en los workers, y nosotros no tenemos visión de lo que hacen los workers ni de su salida. Nosotros solo vemos la salida del Driver.

El comando `foreach(println)` no envía datos al Driver (al `sc`), por eso no vemos nada. Lo que necesitamos es enviar los datos de los RDD a la sesión, de esta manera tendremos los datos localmente en el driver y podremos imprimirlos.

Spark: Topología

Para trabajar eficientemente con *Spark*, hay que entender cómo funciona internamente.

Ejemplo 2: asumimos que tenemos un RDD con objetos del tipo persona

```
case class persona(nombre: string, edad: int)
```

¿Qué devolverá el siguiente código?

```
val people: RDD[persona]=....
```

```
val primeros10=people.take(10)
```

Spark: Topología

Para trabajar eficientemente con Spark, hay que entender cómo funciona internamente.

Ejemplo 2: asumimos que tenemos un RDD con objetos del tipo persona

```
case class persona(nombre: string, edad: int)
```

Qué devolverá el siguiente código?

```
val people: RDD[persona]=....
```

```
val primeros10=people.take(10)
```

Vemos los 10 primeros elementos impresos en la sdout del Driver.

Es decir, nosotros vemos esta salida, porque interactuamos con el Shell (*sdout*) del driver.

En este caso el comando *take()* se encarga de recolectar las 10 primeras líneas de los RDDs de los workers y las envía al Driver.

Podríamos usar el comando *collect()*, que coge todas los RDDS de los workers y los envía a la memoria del Driver, pero si tenemos muchos datos, se colapsaría la memoria. Por eso usamos *take()*

Spark: Topología

Para el ejemplo 1, ¿qué ocurriría si tuviéramos un único nodo? Veríamos los resultados?

Ejecución de Spark en clúster

Despliegue de aplicaciones con *spark-submit*

- Cuando se llama a *spark-submit* seguido del nombre de un script o un JAR, Spark lo ejecuta en modo local.
- Para lanzarlo de otra manera, debemos añadir más parámetros. Ejemplo
- `bin/spark-submit --master spark://host:7077 my_script.py`
- La opción `--master` especifica al cluster al que se debe conectar. Las opciones que hay para esta opción son las siguientes

Value	Explanation
<code>spark://host:port</code>	Connect to a Spark Standalone cluster at the specified port. By default Spark Standalone masters use port 7077.
<code>mesos://host:port</code>	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050.
<code>yarn</code>	Connect to a YARN cluster. When running on YARN you'll need to set the <code>HADOOP_CONF_DIR</code> environment variable to point the location of your Hadoop configuration directory, which contains information about the cluster.
<code>local</code>	Run in local mode with a single core.
<code>local[N]</code>	Run in local mode with N cores.
<code>local[*]</code>	Run in local mode and use as many cores as the machine has.

Ejecución de Spark en clúster

Despliegue de aplicaciones con *spark-submit*

- La estructura general de *spark-submit* es la siguiente:
- `bin/spark-submit [options] <app jar | python file> [app options]`
- [options] son la lista de opciones para spark-submit

Flag	Explanation
<code>--master</code>	Indicates the cluster manager to connect to. The options for this flag are described in Table 7-1 .
<code>--deploy-mode</code>	Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster"). In client mode <code>spark-submit</code> will run your driver on the same machine where <code>spark-submit</code> is itself being invoked. In cluster mode, the driver will be shipped to execute on a worker node in the cluster. The default is client mode.
<code>--class</code>	The "main" class of your application if you're running a Java or Scala program.
<code>--name</code>	A human-readable name for your application. This will be displayed in Spark's web UI.
<code>--jars</code>	A list of JAR files to upload and place on the classpath of your application. If your application depends on a small number of third-party JARs, you can add them here.
<code>--files</code>	A list of files to be placed in the working directory of your application. This can be used for data files that you want to distribute to each node.
<code>--py-files</code>	A list of files to be added to the PYTHONPATH of your application. This can contain <code>.py</code> , <code>.egg</code> , or <code>.zip</code> files.

Ejecución de *Spark* en clúster

Despliegue de aplicaciones con *spark-submit*

- `<app jar | python file>` se refiere al JAR o script *python* que contiene el punto de entrada de la aplicación
- `[app options]` son las opciones que se pasan a la aplicación.
- *Spark-submit* también permite configurar otras opciones para el *SparkConf* usando `--conf prop=value` o pasar archivos de propiedades a través de `--properties-file` que contienen pares (K,V)
- Cuando hay varios usuarios lanzando aplicaciones al mismo tiempo, el *Cluster Manager* se encarga de gestionarlo ayudándose de los *Fair Schedulers*, que permiten gestionar prioridades y hacer posible que aplicaciones cortas convivan con aplicaciones muy largas

Clúster Managers

Como hemos visto en anteriores diapositivas, existen varios tipos de Cluster Managers para Spark:

- *Standalone*
- *Yarn*
- *Mesos*
- Otros: *Amazon EC2*, etc...

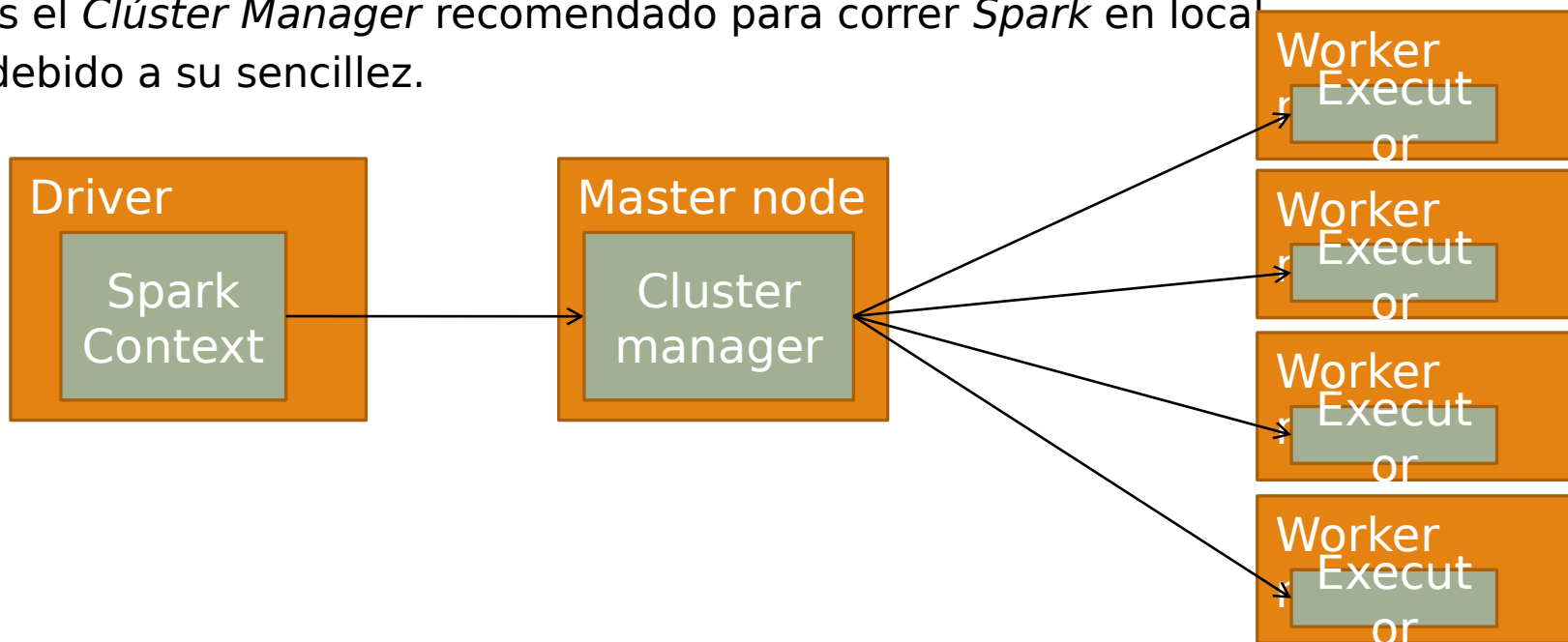


Ejecución de Spark en clúster

Standalone Cluster Manager

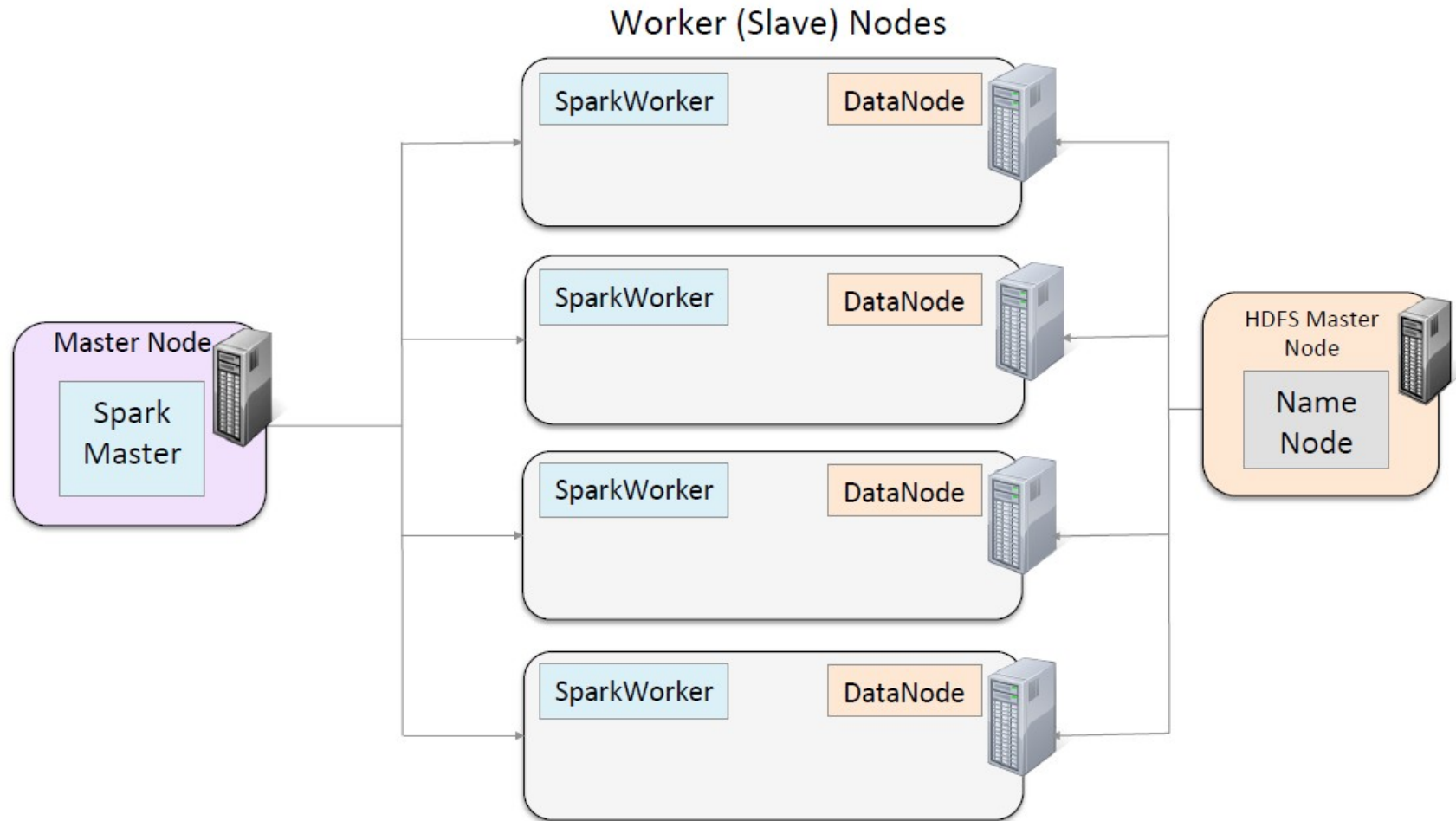


- Es la forma más sencilla de correr una aplicación en un clúster.
- Es el Clúster Manager
- Consta de un *master* y varios *workers*, cada uno de ellos con su cantidad de RAM y CPU asignada. Esta cantidad se puede escoger al lanzar la aplicación.
- Básicamente hay que instalar *Spark* en cada nodo del clúster e indicar qué nodos son *Masters* y *Slaves* junto con su configuración
- Es el *Clúster Manager* recomendado para correr *Spark* en local debido a su sencillez.



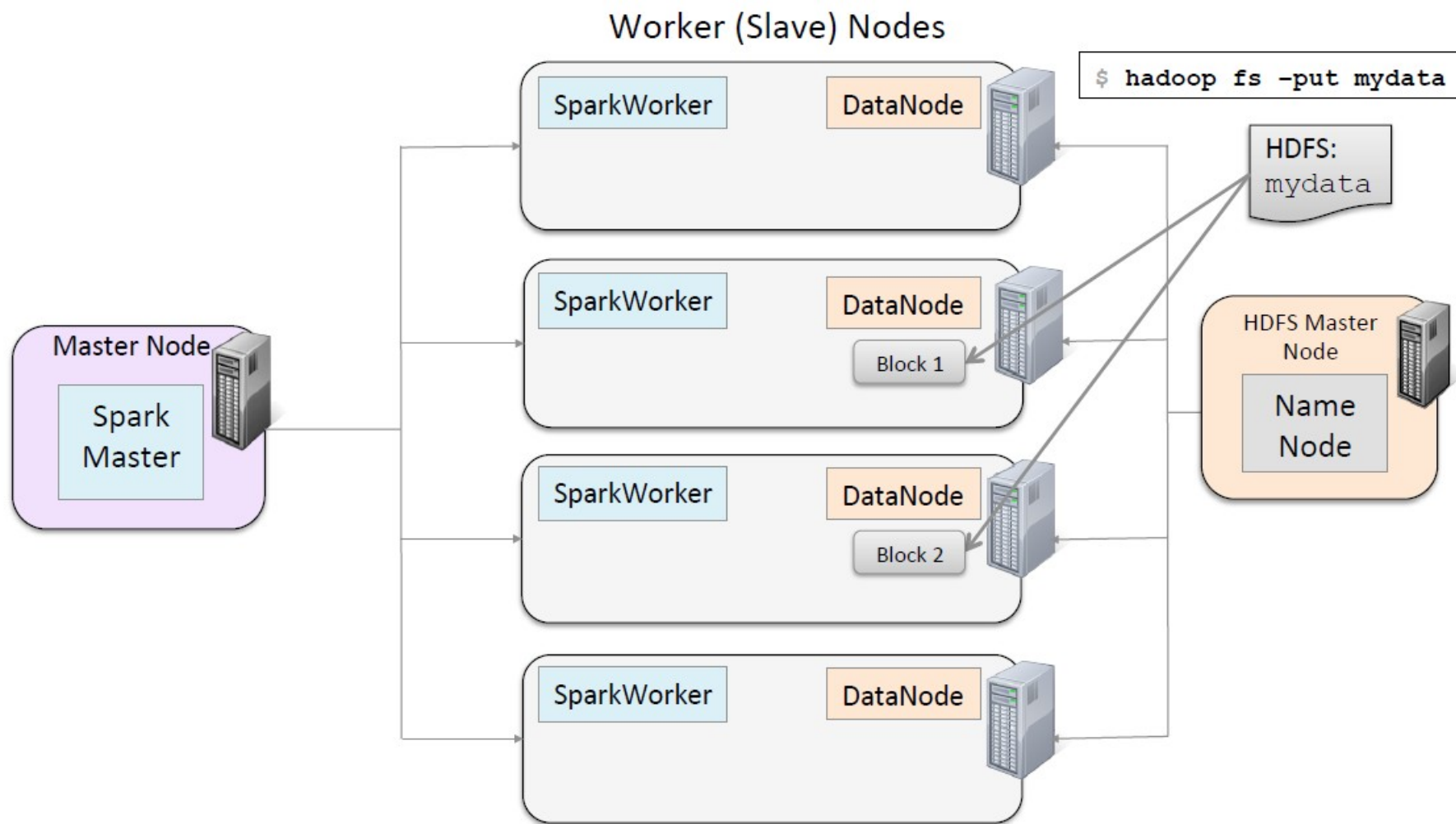
Ejecución de Spark en clúster

Standalone Cluster Manager



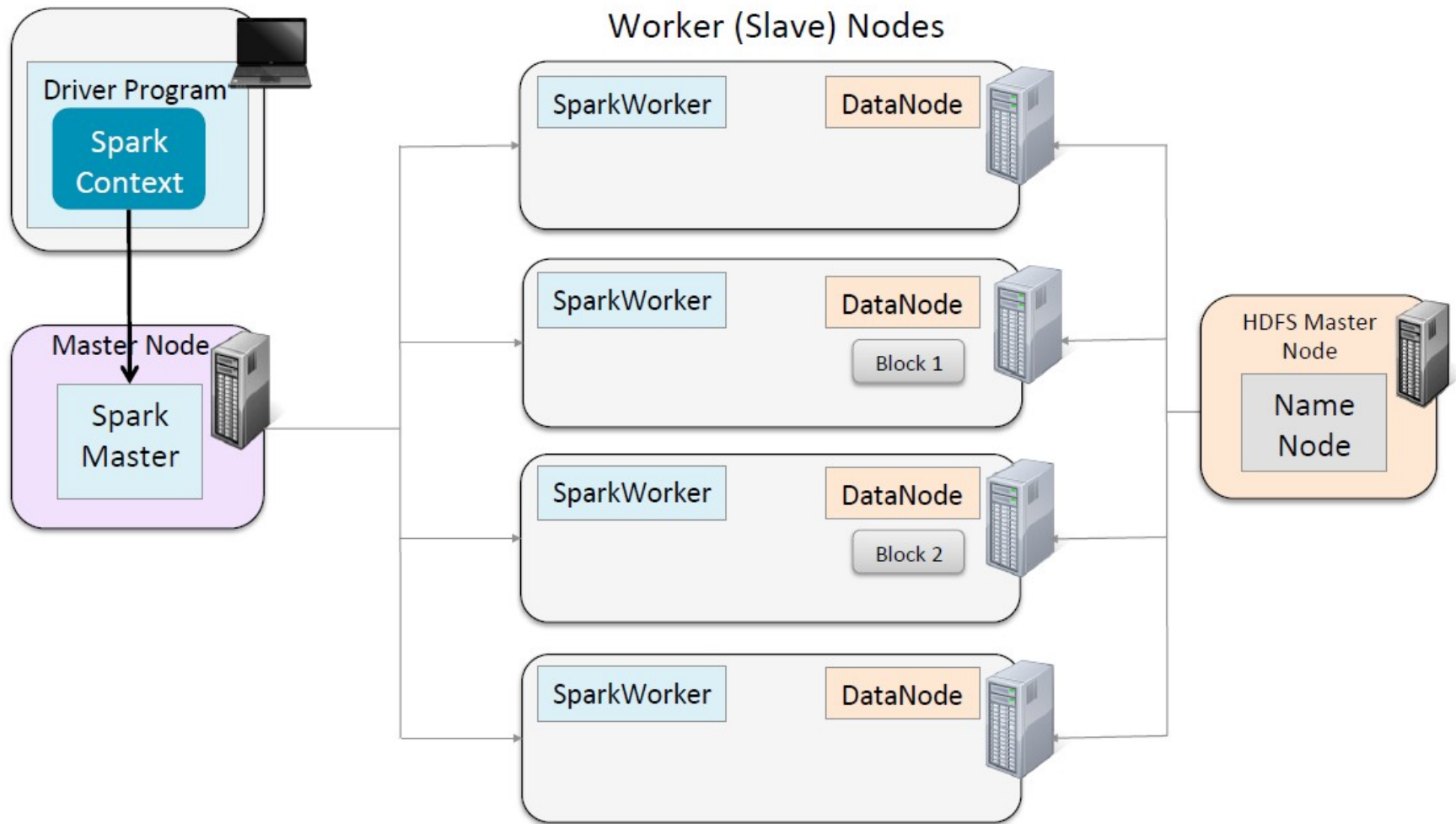
Ejecución de Spark en clúster

Standalone Cluster Manager



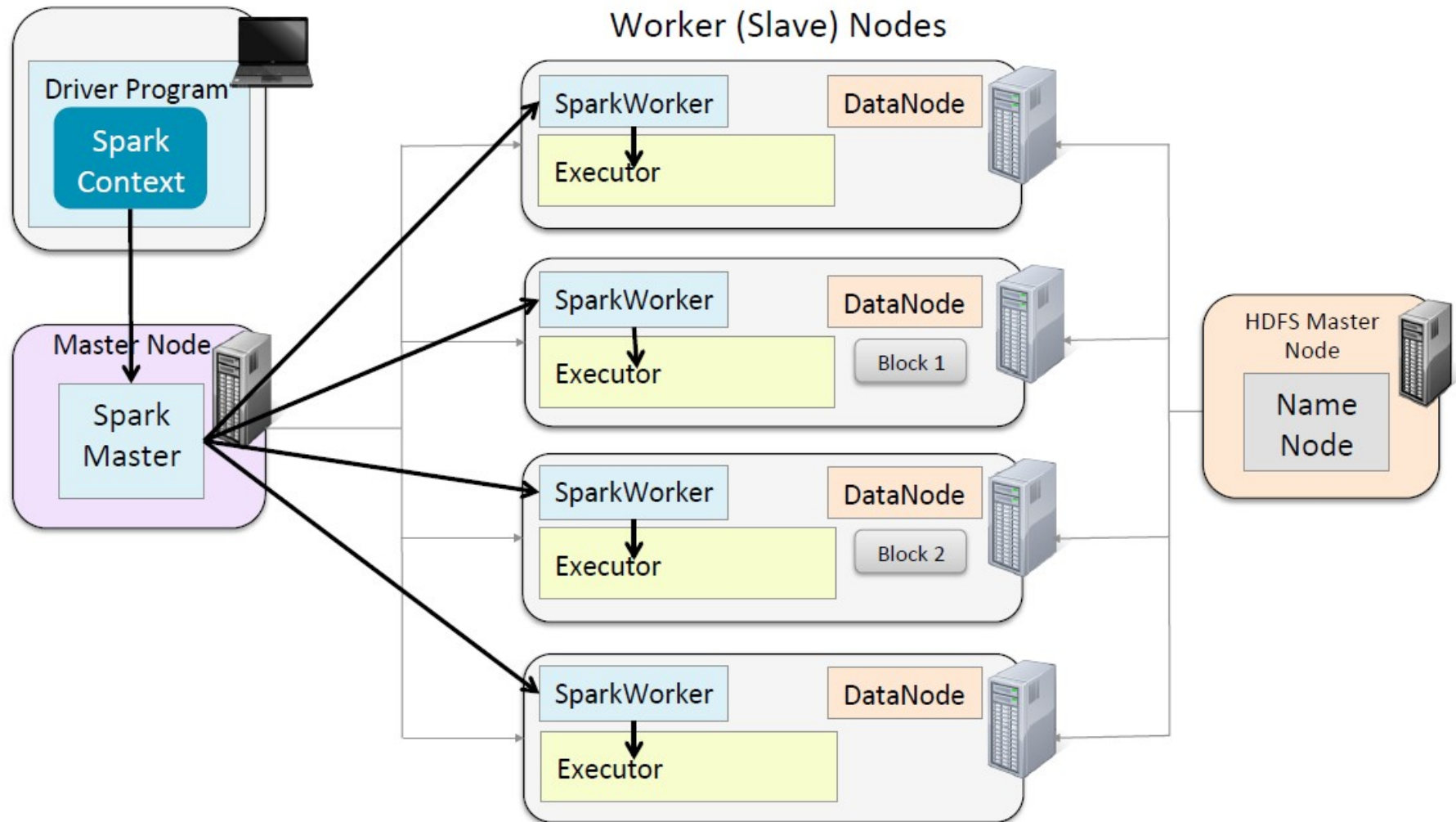
Ejecución de Spark en clúster

Standalone Cluster Manager



Ejecución de Spark en clúster

Standalone Cluster Manager



Ejecución de Spark en clúster

YARN Cluster Manager



- Introducido en Hadoop 2.0
- Es muy útil porque permite acceder a los datos almacenados en HDFS en el nodo donde están almacenados.
- Su uso es simple. Los encargados de configurar el uso de YARN son los Administradores del clúster. El usuario final solo tiene que ejecutar las aplicaciones
- A través del *shell* solo tenemos que indicar **--master yarn**
- Si usamos YARN, disponemos de un número fijo de *Executors*, aunque se pueden modificar por parámetro **--num-Executors** (2 por defecto)
- Ocurre de igual manera con memoria y *cores*: **--Executor-memory**, **--Executor-cores**

Ejecución de *Spark* en clúster



Apache Mesos Cluster Manager

- Ofrece dos modos de compartir recursos entre ejecutores en el mismo clúster:
 - "*Fine-Grained*": Asigna dinámicamente el número de CPU's a cada ejecutor, permitiendo a diferentes aplicaciones de usuario compartir el clúster.
 - "*Coarse-Grained*": Asigna un número fijo de CPU'S a cada ejecutor, y no los desasigna hasta que la aplicación termina, aunque el ejecutor no esté realizando tareas.
- En el *shell* solo tenemos que indicar `--master mesos`

Ejecución de Spark en clúster

¿Qué Cluster Manager usar?

- Se recomienda comenzar con *Spark Standalone* para hacer pruebas porque es sencillo y rápido de poner a punto.
- Cuando queramos ejecutar las aplicaciones con el total de los datos usando el total de la potencia del clúster, usar *YARN* o *Mesos*.
- La ventaja de *YARN* es que viene preinstalado por defecto con Hadoop
- *Mesos* es recomendable en entornos donde múltiples usuarios están ejecutando shells interactivas.



Ejecución de Spark en clúster

Standalone Cluster Manager (WebUI)



Spark Master at spark://localhost:7077

URL: spark://localhost:7077

REST URL: spark://localhost:6066 (*cluster mode*)

Alive Workers: 1

Cores in use: 2 Total, 2 Used

Memory in use: 2.0 GB Total, 2.0 GB Used

Applications: 1 Running, 1 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160109142947-192.168.1.12-53888	192.168.1.12:53888	ALIVE	2 (2 Used)	2.0 GB (2.0 GB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160109143144-0001 (kill)	Spark shell	2	2.0 GB	2016/01/09 14:31:44	jacek	RUNNING	52 s

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160109143059-0000	Spark shell	2	1024.0 MB	2016/01/09 14:30:59	jacek	FINISHED	24 s

Ejecución de Spark en clúster

 Spark Master at `spark://ec2-23-20-24-104.compute-1.amazonaws.com:7077`

URL: `spark://ec2-23-20-24-104.compute-1.amazonaws.com:7077`

Workers: 5

Cores: 20 Total, 20 Used

Memory: 68.2 GB Total, 63.2 GB Used

Applications: 1 Running, 2 Completed

Master URL

Worker Nodes

Workers

Id	Address	State	Cores	Memory
worker-20140121065745-ip-10-236-129-42.ec2.internal-60105	ip-10-236-129-42.ec2.internal:60105	ALIVE	4 (4 Used)	13.6 GB (12.6 GB Used)
worker-20140121065747-ip-10-137-18-53.ec2.internal-54087	ip-10-137-18-53.ec2.internal:54087	ALIVE	4 (4 Used)	13.6 GB (12.6 GB Used)
worker-20140121065747-ip-10-138-3-46.ec2.internal-50661	ip-10-138-3-46.ec2.internal:50661	ALIVE	4 (4 Used)	13.6 GB (12.6 GB Used)
worker-20140121065747-ip-10-236-151-85.ec2.internal-60016	ip-10-236-151-85.ec2.internal:60016	ALIVE	4 (4 Used)	13.6 GB (12.6 GB Used)
worker-20140121065748-ip-10-238-128-41.ec2.internal-42252	ip-10-238-128-41.ec2.internal:42252	ALIVE	4 (4 Used)	13.6 GB (12.6 GB Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20140121215958-0002	PageRank	20	12.6 GB	2014/01/21 21:59:58	root	RUNNING	13 s

Completed Applications

ID	Name	Cores	Memory	Submitted Time	User	State	Duration
app-20140121215522-0001	SparkPi	20	12.6 GB	2014/01/21 21:55:22	root	FINISHED	10 s
app-20140121215016-0000	Spark shell	20	12.6 GB	2014/01/21 21:50:16	root	FINISHED	1.2 min

Applications