Arquitecturas Cloud y Big Data

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
red.es
Centro de Referencia Nacional en Comercio Electrónico y Marketing CRN Digital
GARANTÍA JUVENIL
UNIÓN EUROPEA
Barrabés
The Valley
"El FSE invierte en tu futuro"
Fondo Social Europeo

Índice

1. **Dataframe: operaciones de agregación**
2. **Consideraciones sobre rendimiento RDDs: particiones**
3. **Consideraciones sobre rendimiento RDDs: persistencia**
4. **Aplicación PySpark: funcionamiento**

1. Dataframe: operaciones agregación

Introducción

- Agregar, resumir información es una operación importante y común en el análisis y tratamiento de datos.
- PySpark ofrece las funciones de agregación más comunes dentro de **“pyspark.sql.functions”** (count(); countDistinct(); avg(); sum() …)
- Al igual que en SQL, estas funciones se pueden llamar sobre el conjunto completo de datos (Dataframe). Aunque el mayor uso que se le da y mayor provecho es aplicarlo a agrupaciones.
- Para formar en un Dataframe agrupaciones por una columna o columnas, utilizamos **“groupBy(cols)”.** Lo más usual es a continuación realizar agregados por grupo: **“.agg()”** (facilita alias, varios agregados)

“Sales”, ejemplos de agregación: count(); countDistinct()

- Número de ventas realizadas por región (o por región y país).

```
from pyspark.sql.functions import count, countDistinct

sales_df.groupBy('Region','Country').agg(count('Item_Type').alias('NumVentas')).orderBy('Region','Country').show(10)
```

Ojo: count(“*”) contabilizaría también los valores nulos si los hubiera

- Número de productos distintos (Item_Type) vendidos por región (o por región y país).

```
# ¿Cuantos tipos de productos hay?
sales_df.select('Item_Type').distinct().show()

sales_df.groupBy('Region','Country').agg(countDistinct('Item_Type').alias('TiposArtículo')).orderBy('Region','Country').show(10)
```

approxCountDistinct (…): ¿vale la pena siempre valor exacto (Big Data)?

“Sales”, ejemplos de agregación: sum(); avg(); stddev()

- Unidades vendidas de cada tipo de producto por región, de mayor a menor unidades:

```
from pyspark.sql.functions import col, sum, avg, stddev

sales_df.groupBy('Item_Type','Region').agg(sum('Units_Sold').alias('TotalU
nidades')).orderBy(col('TotalUnidades').desc()).show(40,truncate=False)
```

- Media y desviación típica de las unidades vendidas por producto y región, orden descendente media.

```
(sales_df.groupBy('Item_Type','Region')
.agg(avg('Units_Sold').alias('MediaUnidades'),stddev('Units_Sold').alias('
DesvTip'))
.orderBy(col('MediaUnidades').desc())).show(40,truncate=False)
```

“Sales”, ejemplos de agregación: max(); min()

- Podemos seguir indagando con las unidades vendidas de cada tipo de producto por región: pedidos máximos y mínimos.

```
from pyspark.sql.functions import min,max

sales_df.groupBy('Item_Type','Region').agg(max('Units_Sold').alias('Pedido
Máximo'),min('Units_Sold').alias('PedidoMínimo')).orderBy(col('PedidoMáxim
o').desc()).show(40,truncate=False)
```

Vamos a practicar……

EJERCICIOS AGREGACIÓN: Sales

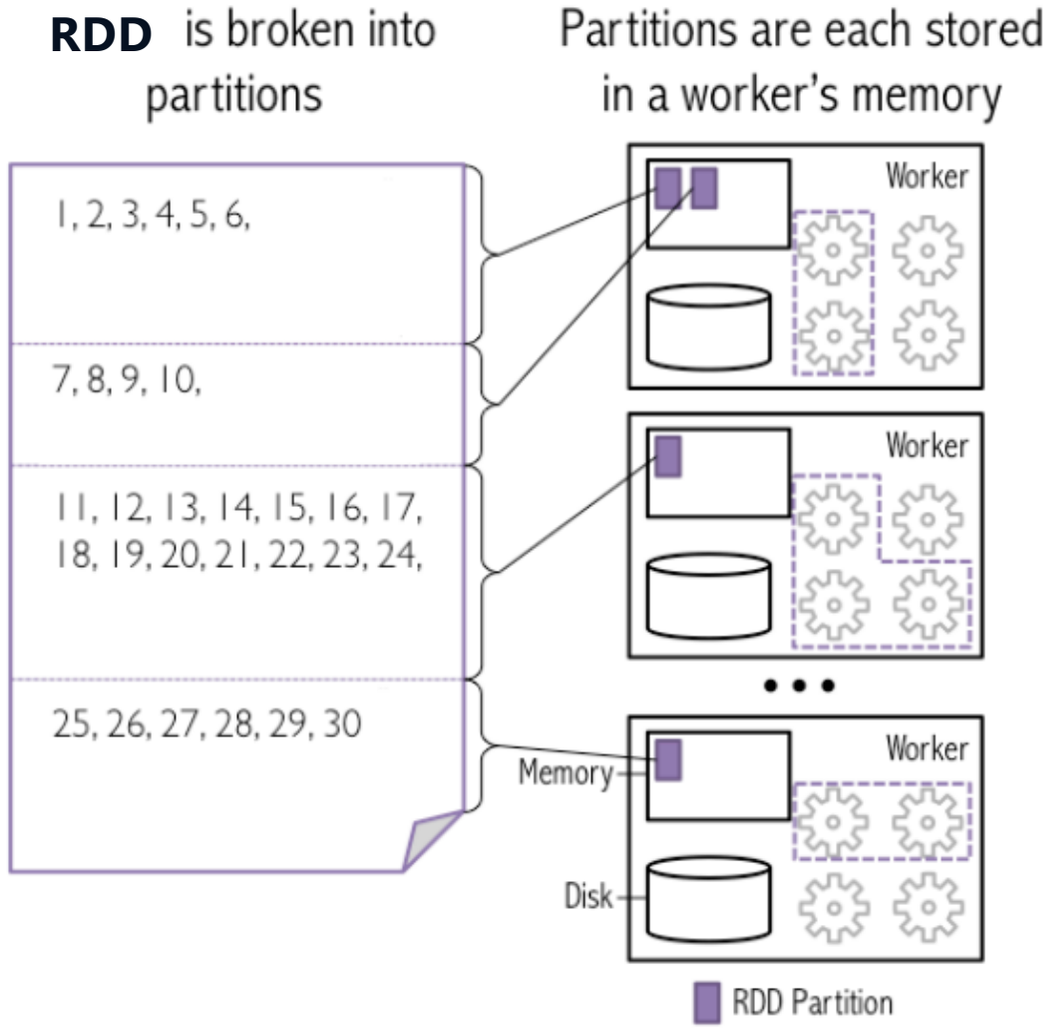
- **Partir del Dataframe “Sales” original (desde cero)**

- **Preguntas:**
  - **¿Cuántos tipos de ítems (distintos) hay? ¿Cuáles son?**
  - **¿Cuales son los 5 países que más unidades reciben? ¿Y los que menos?**
  - **Nº de transacciones (no unidades) por región y país, en orden descendente.**
  - **¿Cuántos países por región reciben mercancías, ordenados de más a menos países?**

2. Consideraciones sobre rendimiento RDDs: particiones

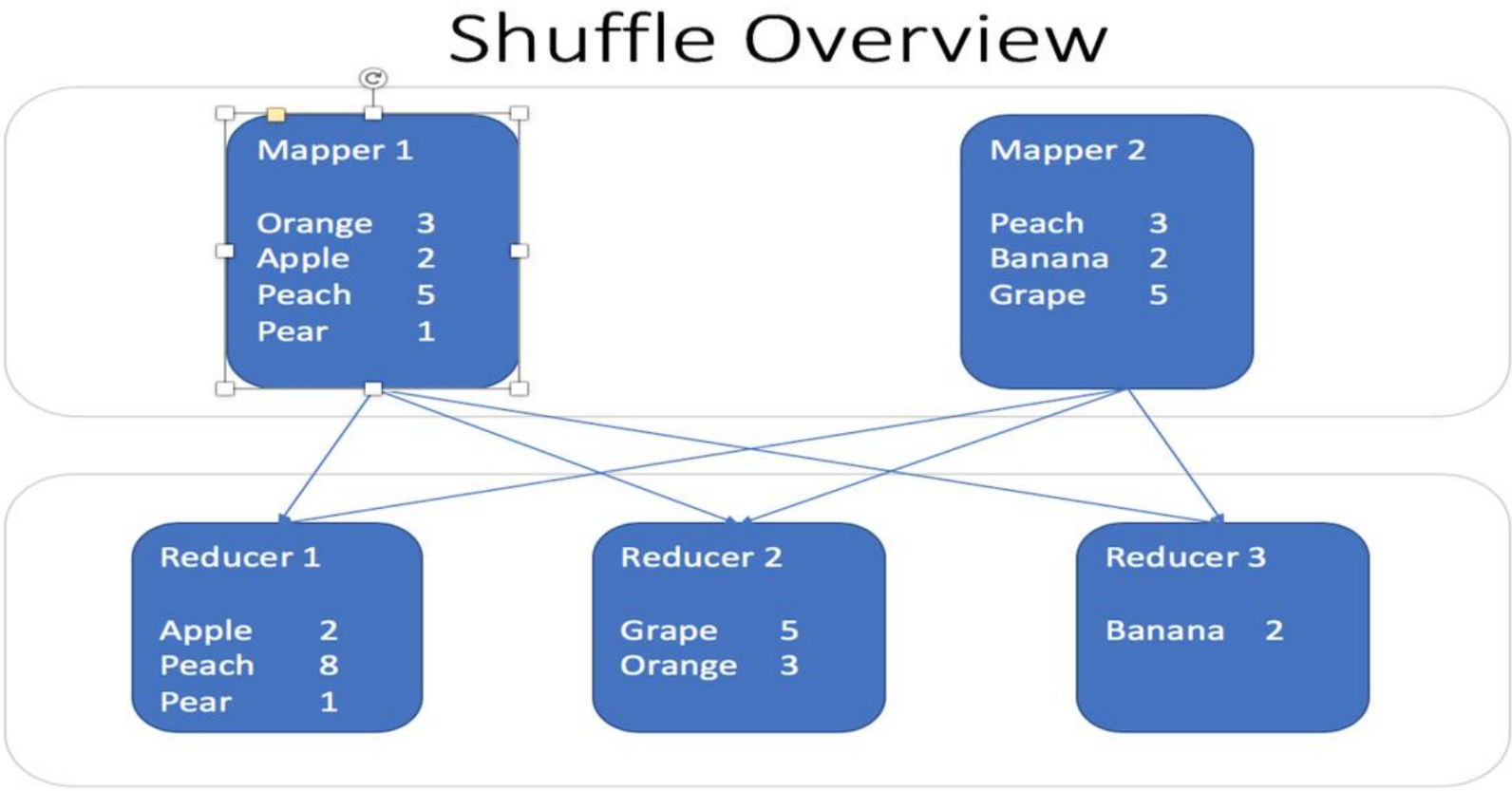
PARTICIONES DATOS (sistema distribuido)

- Los datos se dividen en particiones que se distribuyen en memoria de distintos nodos (**nodos workers)**
- Aunque esto no siempre es posible, a cada ejecutor de Spark se le adjudica preferentemente una tarea que requiera leer la partición más cercana a él en la red.



SHUFFLE: movimiento de datos en el clúster

- Se produce en determinadas situaciones, como las operaciones de ordenación o reducción.
- Ejemplo: reduceByKey(…). Hay que agrupar en un nodo los elementos de una misma clave, que provendrán de distintos nodos.
- Esto puede ser muy costoso (tiempo), pero es a veces es necesario.



Coalesce(numPartitions)

- Reduce el número de particiones de un RDD a numPartitions (crea uno nuevo)
- Es útil para ejecutar operaciones de forma más eficiente, por ejemplo después de filtrar un número elevado de datos (particiones con pocos elementos, una tarea a cada partición, ineficiente).
- A favor: minimiza el movimiento de datos (“colapsa” particiones en un mismo nodo si puede)
- Contra: No obtiene particiones homogéneas en número de datos (óptimo rendimiento)

```
rdd = sc.parallelize([3,2,1,4,5,7,9,6,2,8],4)

rdd2 = rdd.coalesce(3)

print(rdd2.getNumPartitions())

print(rdd2.glom().collect())
```

Devuelve nº particiones

Crea RDD cuyos elementos son las particiones

Repartition(numPartitions)

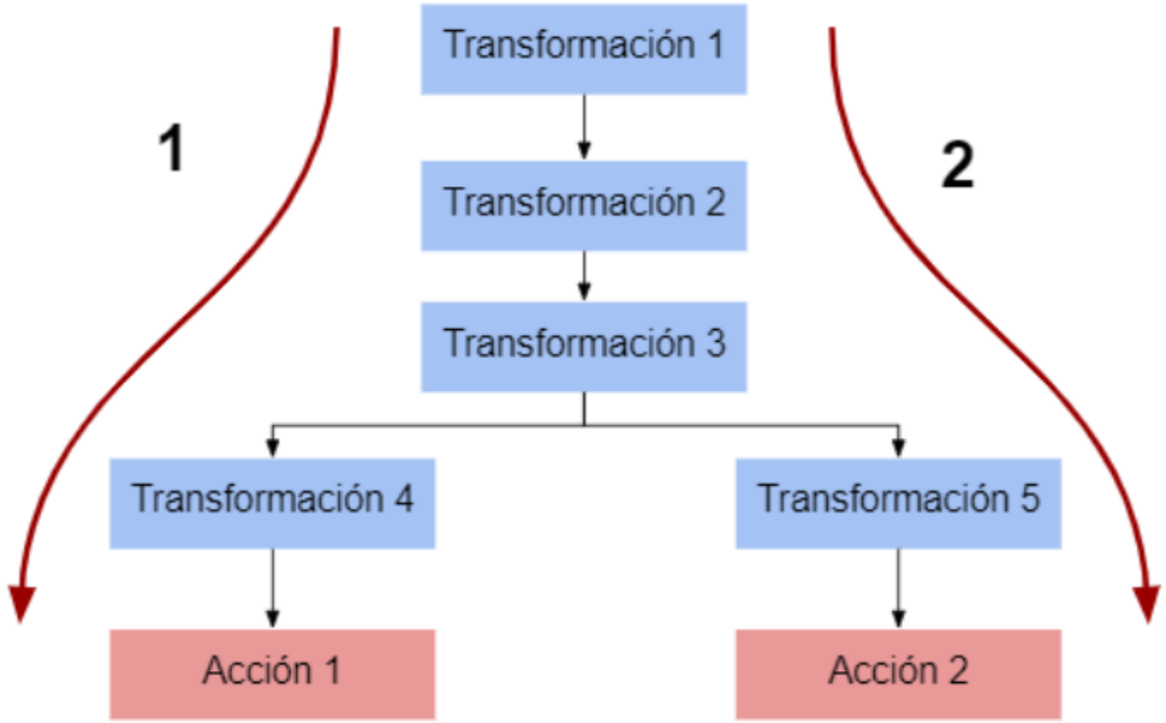
- Puede aumentar o reducir el número de particiones del RDD (crea uno nuevo).
- Las particiones resultantes son de igual tamaño (aprox.) lo que permite ganar posteriormente en velocidad
- A cambio mayor coste: siempre produce movimiento de datos a través del clúster (full shuffle)
- A veces son necesarias (valorar pros/contras en cada caso)
- “repartition” distribuye los datos de forma equitativa, a costa del movimiento de datos.

```
rdd = sc.parallelize([4,5,7,9,6,2,8,3],3)

rdd2 = rdd.repartition(4)

print(rdd2.getNumPartitions())
```

3. Consideraciones sobre rendimiento RDDs: persistencia

PERSISTENCIA: casos de uso

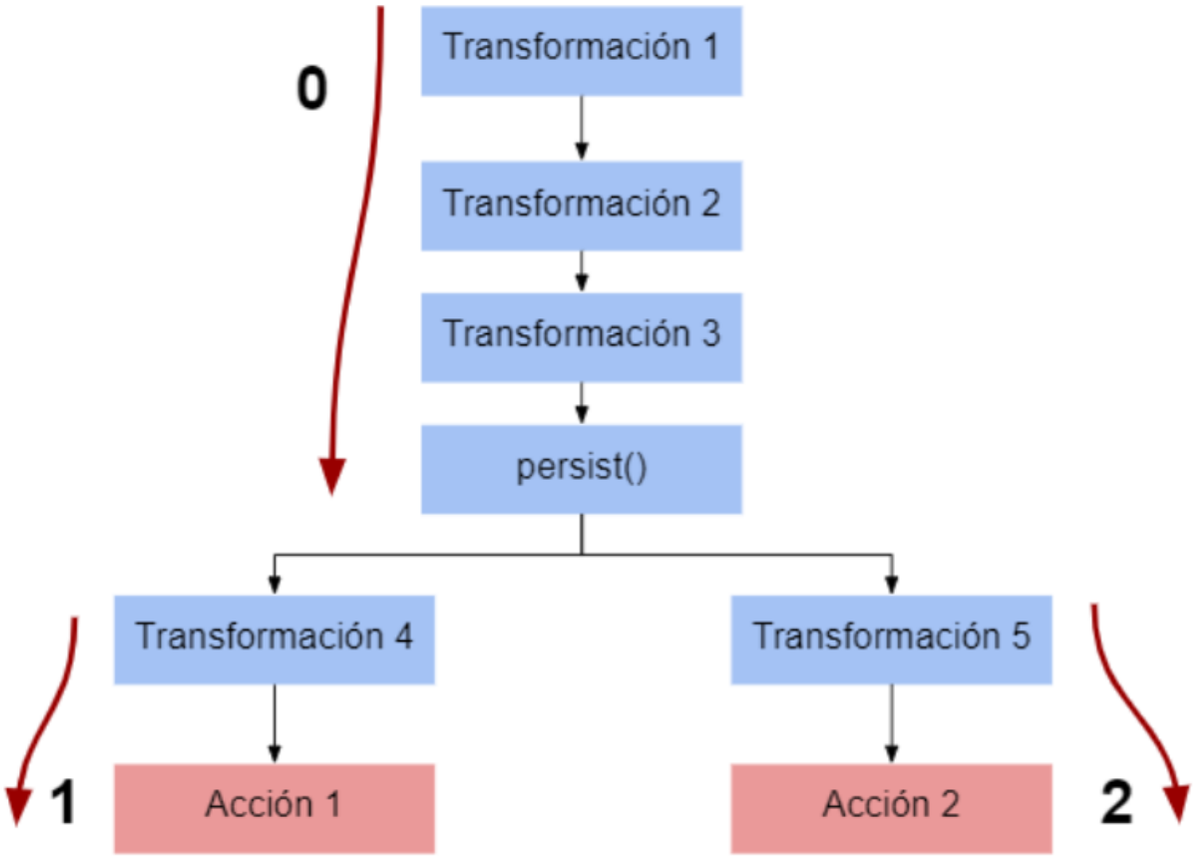
- Evaluación perezosa: no se almacena ningún dato hasta la acción (símil receta)
- Transformaciones 1 a 3 comunes, se ejecutarían 2 veces (p. ejplo. bifurcaciones por sentencias Joins o GroupBy)
- Mayor tiempo de procesamiento y mayor uso de memoria
- SOLUCIÓN: **persistencia**



PERSISTENCIA: casos de uso

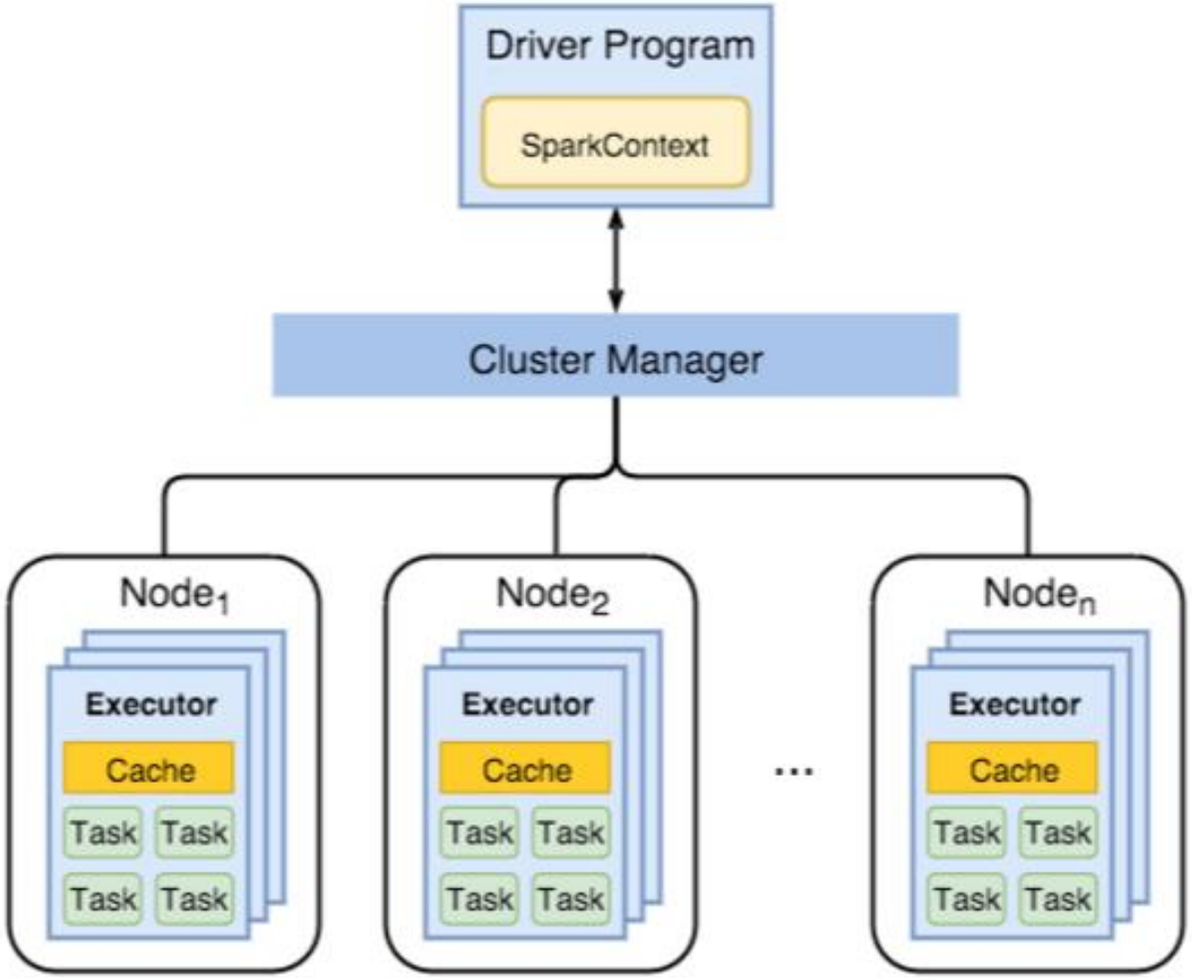
- Se almacena en memoria el RDD resultado de la transformación 3 (proceso 0)
- Procesos 1 y 2 recuperarían el RDD almacenado
- No solo esta aplicación, basta con que tengamos un proceso que lleve mucho tiempo o un conjunto de datos que se utilice con frecuencia
- Dos métodos disponibles:

cache() y **persist()**



PERSISTENCIA: nodos

- Cuando se persiste un RDD, cada nodo con particiones del mismo las mantiene en memoria y las reutiliza en otras acciones. Esto permite que las acciones futuras sean mucho más rápidas (a menudo por más de 10 veces).



Persist(StorageLevel.<>)

- Permite especificar distintos niveles de almacenamiento como parámetro <>:
  - MEMORY_ONLY (por defecto)
  - DISK_ONLY
  - MEMORY_AND_DISK (si no cabe en memoria utiliza disco)

```
from pyspark import StorageLevel


file = 'dbfs:/FileStore/shared_uploads/edurf.cld@gmail.com/quijote-1.txt'
lineas = sc.textFile(file)
long_lineas = lineas.map(lambda elemento: len(elemento))
long_lineas.persist(StorageLevel.MEMORY_ONLY)


print(long_lineas.reduce(lambda elem1,elem2: elem1 + elem2))
```

Cache()

- Es un “atajo” para el nivel de persistencia por defecto (MEMORY_ONLY): rdd.cache()
- Tanto con “cache()” como con “persist()” es necesario usar una acción posterior para ejecutarlas
- Spark monitorea automáticamente todas las llamadas de persist() y cache() que realiza (consola Spark -> Storage)

```
file = 'dbfs:/FileStore/shared_uploads/edurf.cld@gmail.com/quijote-
1.txt'
lineas = sc.textFile(file)
long_lineas = lineas.flatMap(lambda elemento: elemento.split())
long_lineas.cache()


print(long_lineas.count()
```

4. Aplicación PySpark: funcionamiento

Aplicación PySpark: funcionamiento

- Spark es un **motor de procesamiento de datos distribuido,** con sus componentes trabajando en colaboración en un **clúster de máquinas**.
- Estos componentes trabajan juntos y se comunican. Hay un nodo maestro **DRIVER** y uno o más nodos esclavos **WORKERS** (dependiendo modo despliegue).
- A alto nivel una **aplicación Spark** consiste en **un programa controlador (Spark driver o Driver Program)** que es responsable de orquestar las operaciones paralelas en el clúster de Spark.
- El controlador (Driver Program) contiene la función 'main ()' con el código que queremos ejecutar. En este código se debe crear la SparkSession (SparkContext).
- El controlador accede a los componentes del clúster (workers y el gestor del clúster) a través de una SparkSession (SparkContext).

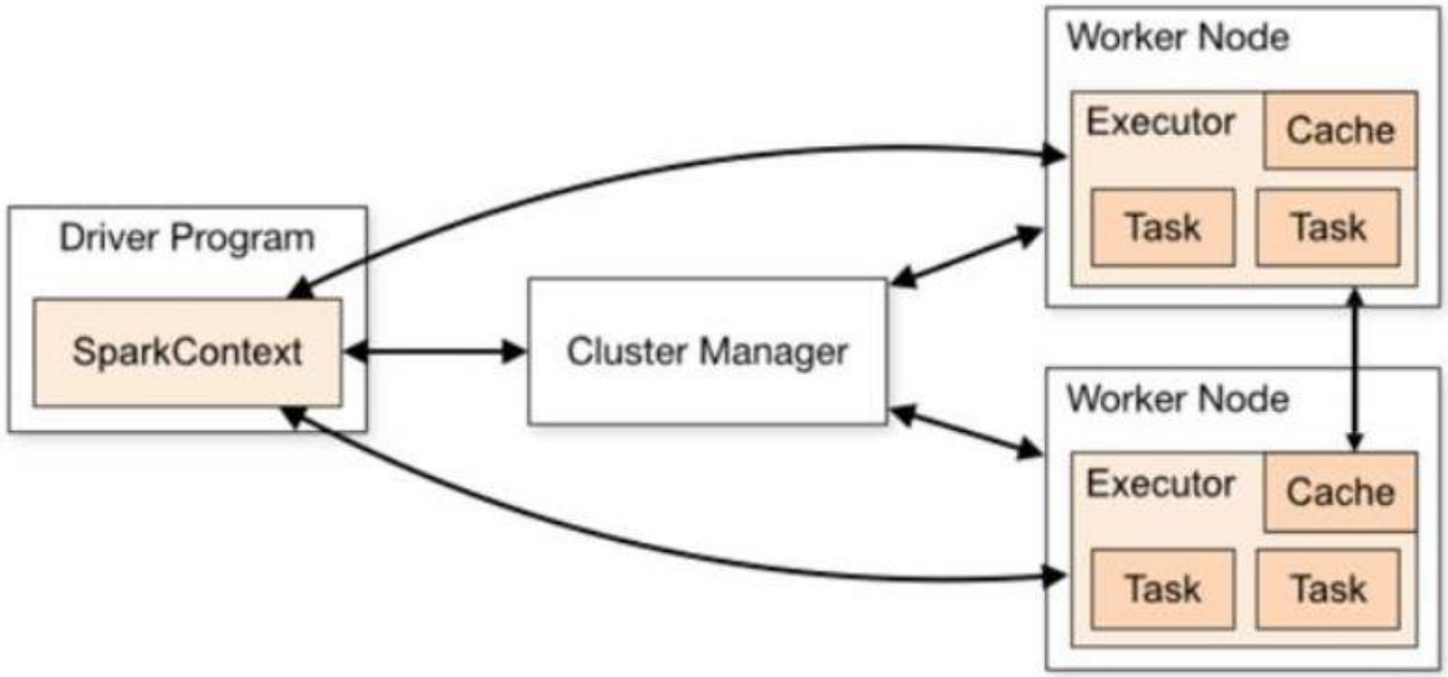
```python
# Import the necessary libraries.
# Since we are using Python, import the SparkSession and related functions
# from the PySpark module.
import sys

from pyspark.sql import SparkSession
from pyspark.sql.functions import count

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: mnmcount <file>", file=sys.stderr)
        sys.exit(-1)

    # Build a SparkSession using the SparkSession APIs.
    # If one does not exist, then create an instance. There
    # can only be one SparkSession per JVM.
    spark = (SparkSession
      .builder
      .appName("PythonMnMCount")
      .getOrCreate())
    # Get the M&M data set filename from the command-line arguments
    mnm_file = sys.argv[1]
    # Read the file into a Spark DataFrame using the CSV
    # format by inferring the schema and specifying that the
    # file contains a header, which provides column names for comma-
    # separated fields.
    mnm_df = (spark.read.format("csv")
      .option("header", "true")
      .option("inferSchema", "true")
      .load(mnm_file))
```

"main"

Se crea la "SparkSession"

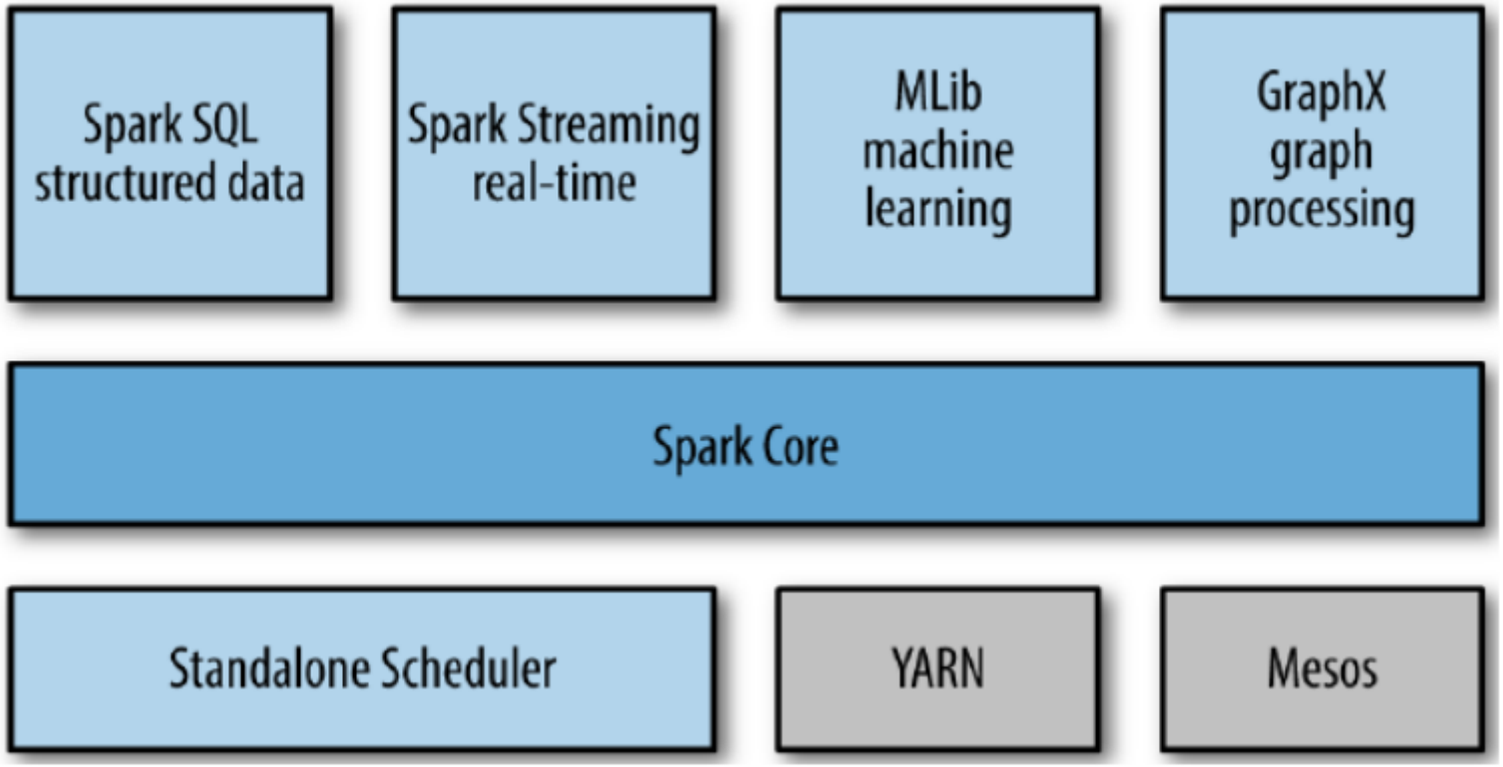
Spark funcionamiento interno

- El controlador (Driver program) crea la sesión (contexto) , hace petición de recursos al clúster manager y declara las operaciones sobre los datos utilizando transformaciones y acciones de RDD: crea el grafo DAG y distribuye su ejecución como tareas (Task) a los workers.
- El clúster manager (gestor de recursos) maneja y asigna recursos del clúster. Coordina las diferentes etapas del trabajo. Spark admite varios (YARN, Mesos…)
- Los workers son donde las tareas se ejecutan realmente. Tienen los recursos y la conectividad de red requeridos para ejecutar las operaciones solicitadas en los RDD.



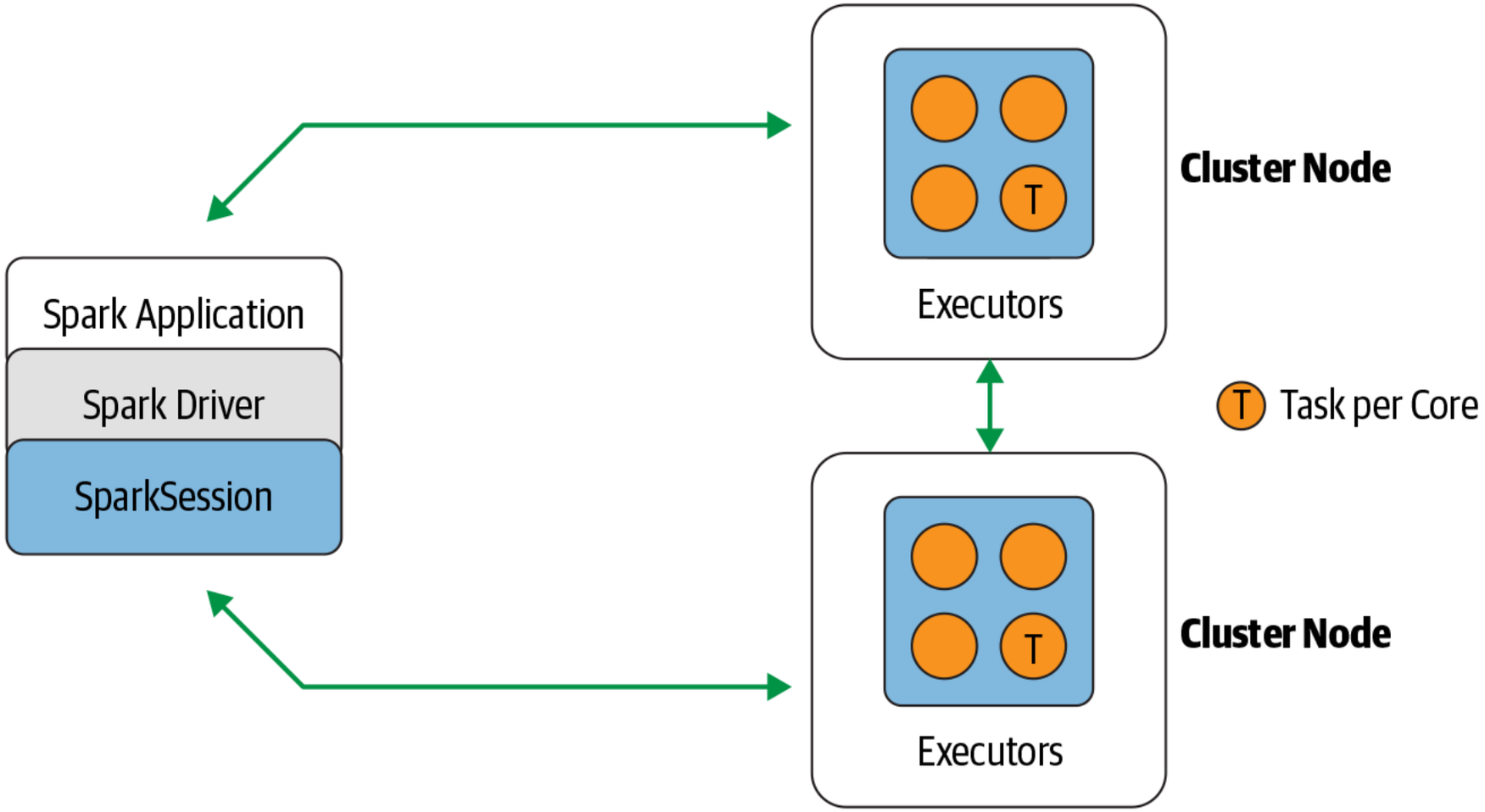
Gestores de recursos

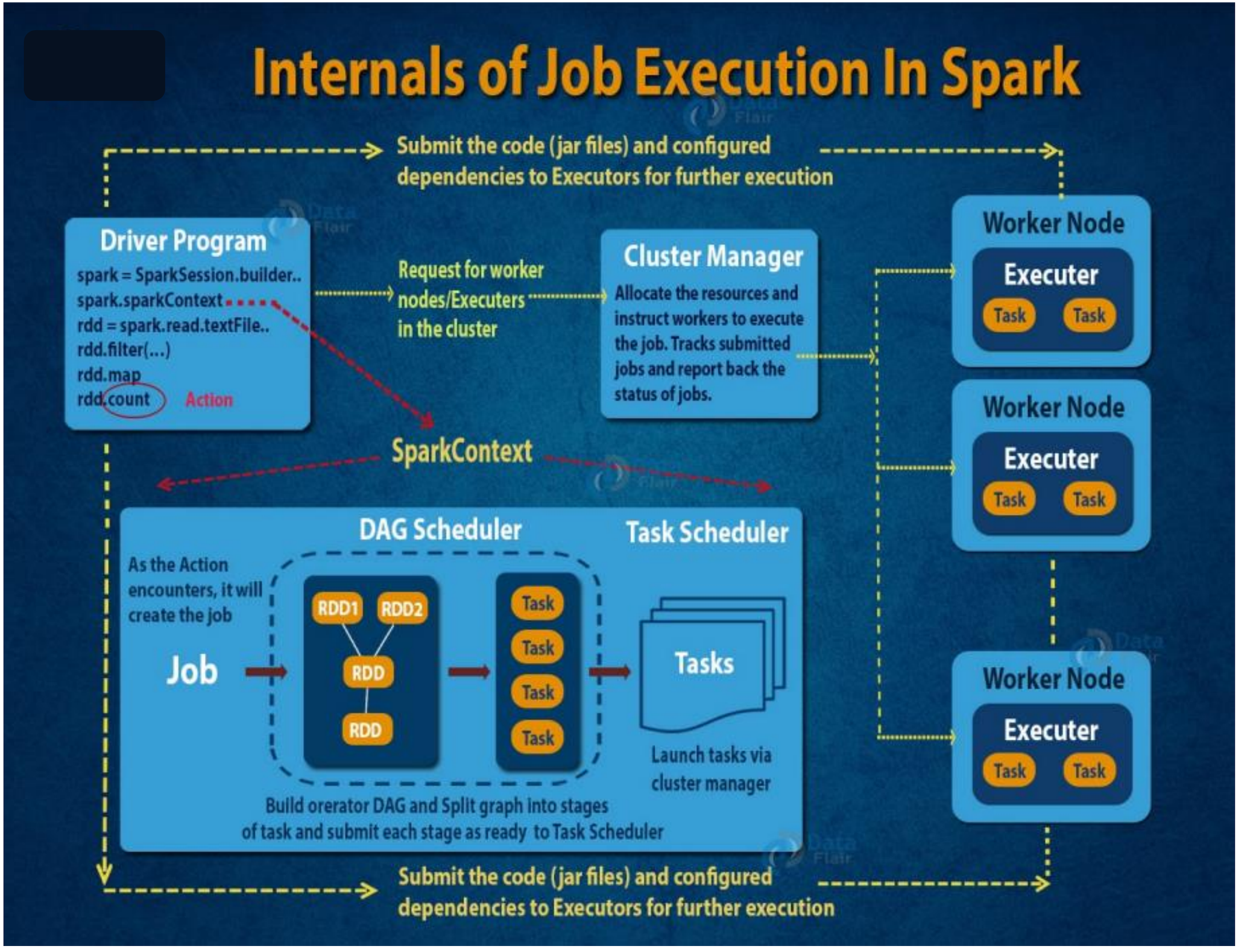
- Spark admite varios gestores de recursos, entre ellos un gestor de recursos autónomo integrado (Standalone Sheduler), Apache Hadoop YARN y Apache Mesos.



SPARK: reparto de tareas (Task)

- Las tareas se asignan a los “executors” (procesos en los nodos “workers”)




Internals of Job Execution In Spark
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

red.es

CRN Digital

GARANTÍA JUVENIL

UNIÓN EUROPEA

"El FSE invierte en tu futuro"

Fondo Social Europeo

Barrabés

The Valley