

# Introducción a Spark



**alan TURING**

CENTRO PÚBLICO INTEGRADO DE FORMACIÓN PROFESIONAL



**accenture**



# Spark

- Spark es un **framework** de computación distribuida similar a Hadoop-MapReduce:
  - Es un sistema para procesar grandes volúmenes de datos de manera distribuida, es decir, repartiendo el trabajo a través de múltiples nodos en un clúster.
  - Esto permite a Spark procesar datos a una escala que no sería manejable en una sola máquina, de manera similar a cómo funciona Hadoop-MapReduce.

# Spark

- Spark, a diferencia de Hadoop, en lugar 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.



# Spark

- No es un lenguaje de programación: Spark proporciona APIs en varios lenguajes de programación como Scala, Python o Java.
- Esto significa que Spark es una plataforma o un entorno que se utiliza para desarrollar aplicaciones de procesamiento de datos en estos lenguajes, no un lenguaje de programación por sí mismo.

# Spark

- Permite trabajar con todo el ciclo del datos, desde la ingesta y la validación de los datos en crudo, limpieza, transformación y agregación de los datos, así como la realización de un análisis exploratorio de los mismos.



# Spark

- 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.



# Spark

## Diferencias con Hadoop MapReduce

Factores	Hadoop MapReduce	Apache Spark
Velocidad	+ rápido que sistemas tradicionales	100 veces más rápido que MapReduce
Programado en	Java	Scala (interfaz para Java, Python y R)
Procesamiento de datos	Batch	Batch, Tiempo real, Iterativo o Grafos
Facilidad de uso	Largo y complejo	Compacto y + fácil
Caché	No soportado	Se cachean los datos en memoria (+ rendimiento)




# Spark vs Hadoop

## Almacenamiento en memoria vs. en disco:

- La principal diferencia entre Spark y Hadoop radica en cómo manejan el almacenamiento de datos durante el procesamiento.
- Spark está diseñado para almacenar datos en la RAM de los nodos del clúster.
- Esto le permite realizar operaciones de procesamiento mucho más rápido comparado con Hadoop, que lee y escribe datos en el disco (a través de HDFS) entre cada paso del proceso MapReduce.
- Almacenar datos en memoria permite a Spark optimizar el procesamiento de datos iterativos y las consultas interactivas.

# Spark vs Hadoop

Sistema de gestión de recursos:

- Tanto Hadoop como Spark necesitan de un sistema de gestión de recursos para distribuir el trabajo a través de los nodos del clúster.
  - Hadoop utiliza YARN, su propio sistema de gestión de recursos.
  - Spark es agnóstico en cuanto al sistema de gestión de recursos y puede ejecutarse sobre YARN, Mesos, Kubernetes o incluso en su propio clúster independiente gestionado por el Spark Standalone Scheduler.
- 

# Spark vs Hadoop

- Apache Spark almacena datos en memoria utilizando una estructura distribuida llamada RDD (Resilient Distributed Dataset) y optimiza su almacenamiento en memoria de los nodos del clúster mediante una serie de estrategias.




# Spark vs Hadoop

## 1. División y Distribución de Datos

Cuando Spark carga un archivo muy grande (por ejemplo, desde HDFS, Amazon S3 o local), lo divide en particiones.

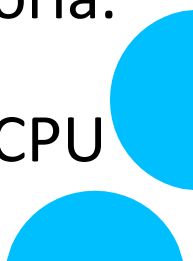
Cada partición se almacena y procesa en uno de los nodos del clúster:

- Spark asigna una partición por núcleo disponible en los ejecutores para balancear la carga de trabajo.
  - Si el archivo es demasiado grande, se fragmenta en múltiples particiones que pueden repartirse entre los nodos.
- 

# Spark vs Hadoop

## 2. Almacenamiento en Memoria con RDDs

Spark utiliza RDDs para mantener datos en memoria de forma distribuida y tolerante a fallos. Los datos pueden almacenarse en diferentes formatos:

- `MEMORY_ONLY`: Almacena los datos en RAM.
  - `MEMORY_AND_DISK`: Guarda los datos en RAM, pero si la memoria es insuficiente, almacena el resto en disco.
  - `DISK_ONLY`: Usa solo el disco, sin almacenar en memoria.
  - `MEMORY_ONLY_SER`: Guarda los datos en RAM pero serializados, reduciendo el uso de memoria a costa de más CPU para deserializar.
- 

# Spark vs Hadoop

Ejemplo:

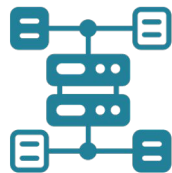
```
rdd = spark.sparkContext.textFile("hdfs://path/to/large/file.txt")  
  
rdd.persist(storageLevel=StorageLevel.MEMORY_AND_DISK)  
# Esto asegura que Spark almacene el RDD en memoria, pero si  
#no hay suficiente espacio, se volcará a disco.
```

# Spark

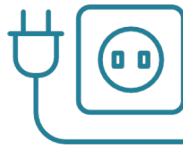
## Características principales



Batch/Streaming



Cluster



API de  
programación



SQL



Bibliotecas

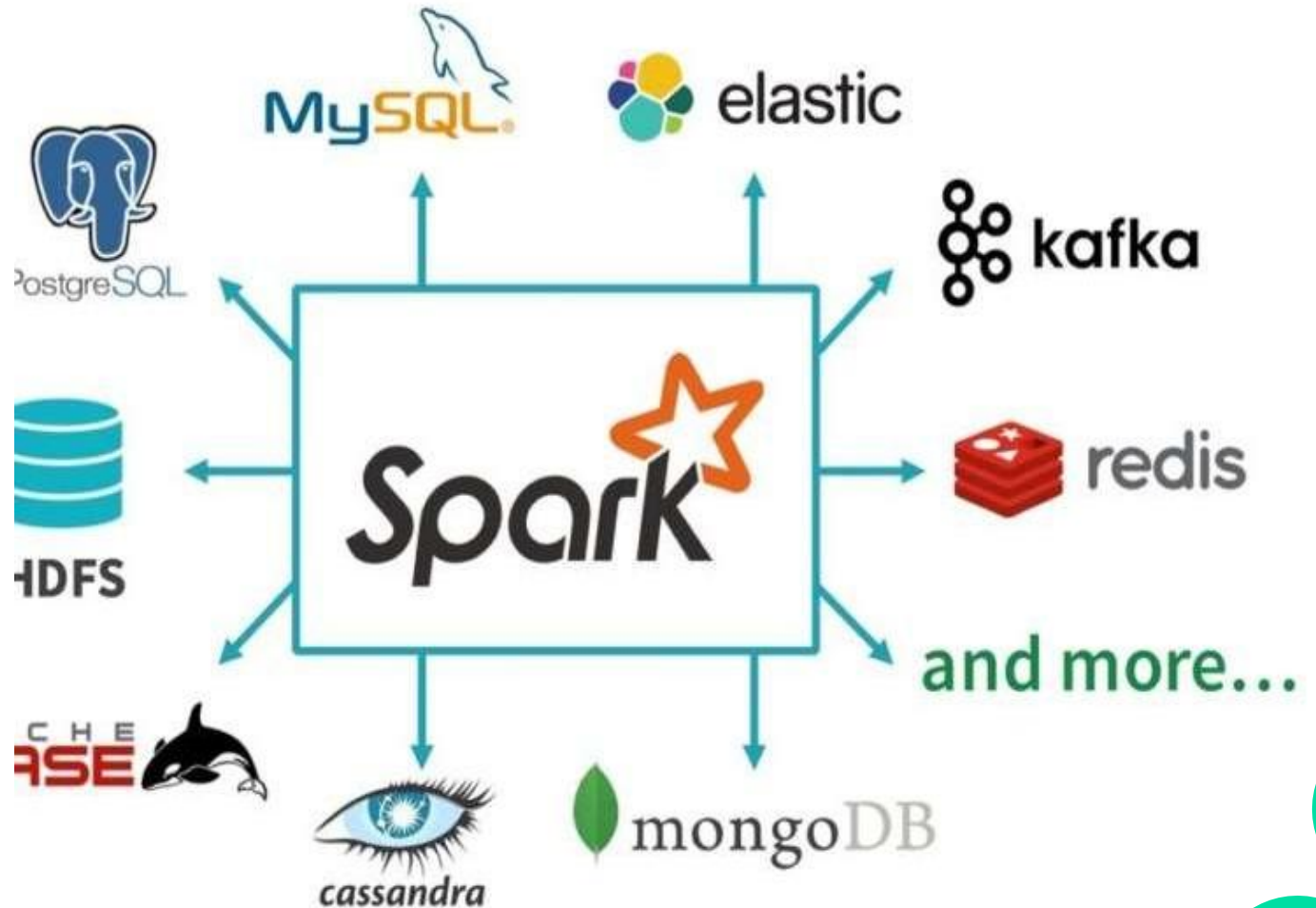
# Spark

## Características principales

- **Velocidad:** enfocado al uso en un clúster de commodity hardware con una gestión eficiente del multihilo y procesamiento paralelo.
- **Facilidad de uso:** Spark ofrece varias capas de abstracción sobre los datos, como son los RDD, DataFrames y Dataset.
- **Modularidad:** soporte para todo tipo de cargas mediante: Scala, Java, Python, SQL y R.
- **Extensibilidad:** La gestión de los datos se puede realizar a partir de Hadoop, Cassandra, HBase, MongoDB, Hive o cualquier SGBDR, haciendo todo en memoria.

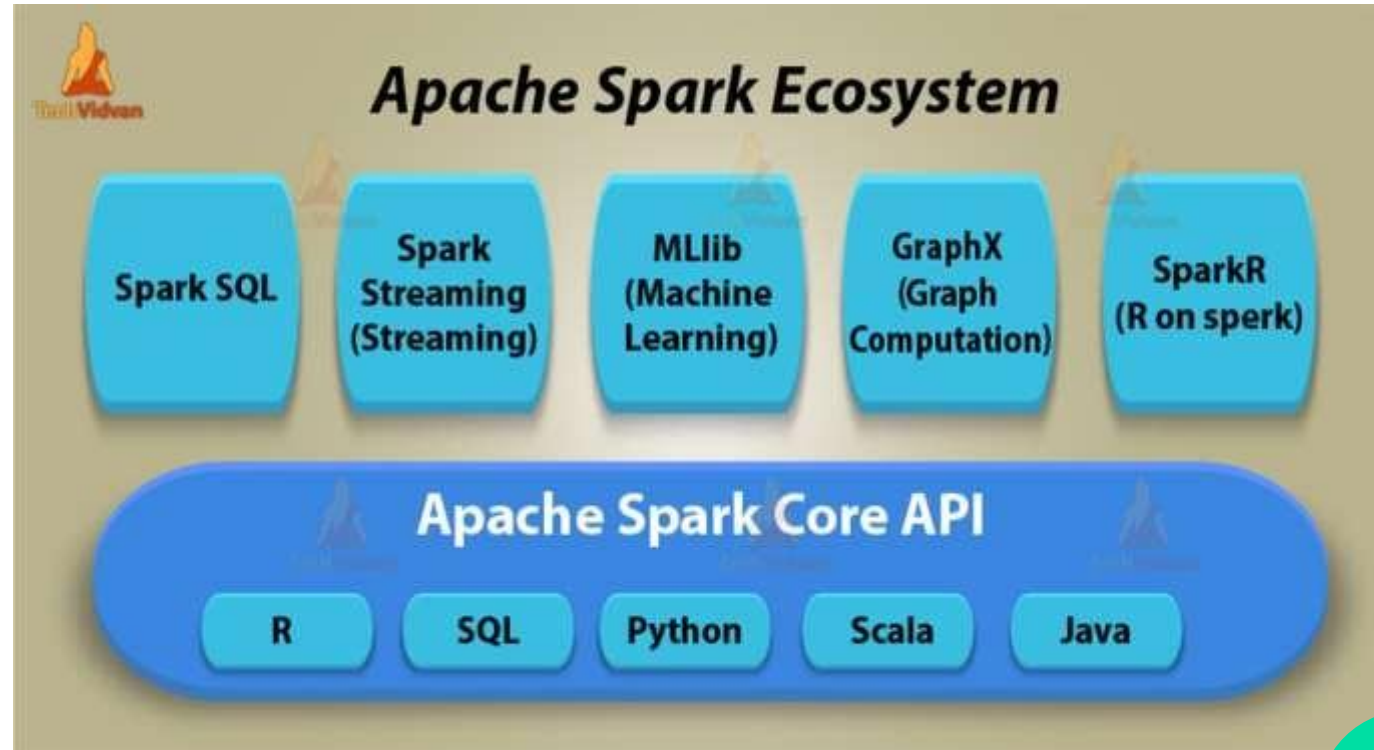


# Spark



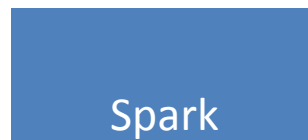
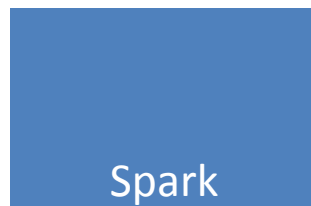
# Spark

## Ecosistema



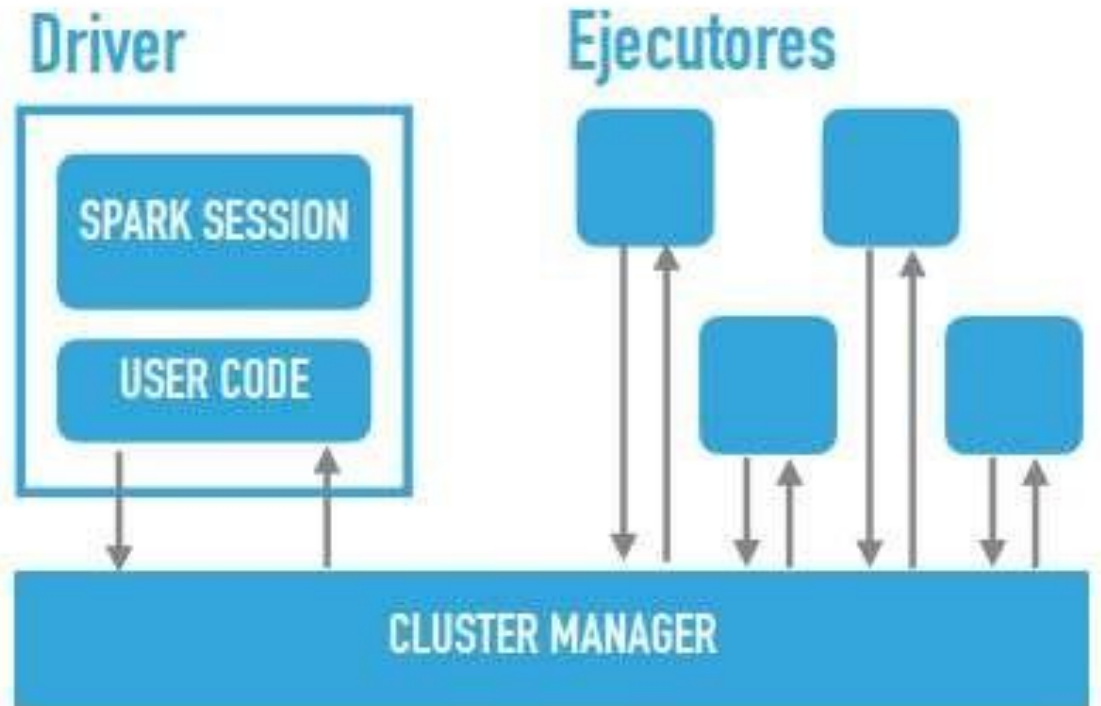
# Spark

## Características principales



# Spark

## Arquitectura




# Spark

- El proceso driver (**Driver Node**) sabe dónde se localizan los trabajadores, cuanta memoria disponen y el número de cores CPU de cada nodo. Su mayor responsabilidad es orquestar el trabajo asignándolo a los diferentes nodos.
- Los procesos ejecutores (**Workers Nodes**) procesan las tareas asignadas por el driver, y ofrece recursos (memoria, CPU, etc...) al gestor del clúster.
- Esta arquitectura es independiente del sistema de gestión del cluster (Spark, Yarn...)

# Aplicaciones Spark


Una aplicación Spark se compone de dos partes:

1) La **lógica de procesamiento** de los datos, la cual realizamos mediante alguna de las API que ofrece Spark (Java, Scala, Python, etc...), desde algo sencillo que realice una ETL sobre los datos a problemas más complejos que requieran múltiples iteraciones y tarden varias horas, como entrenar un modelo de ML.



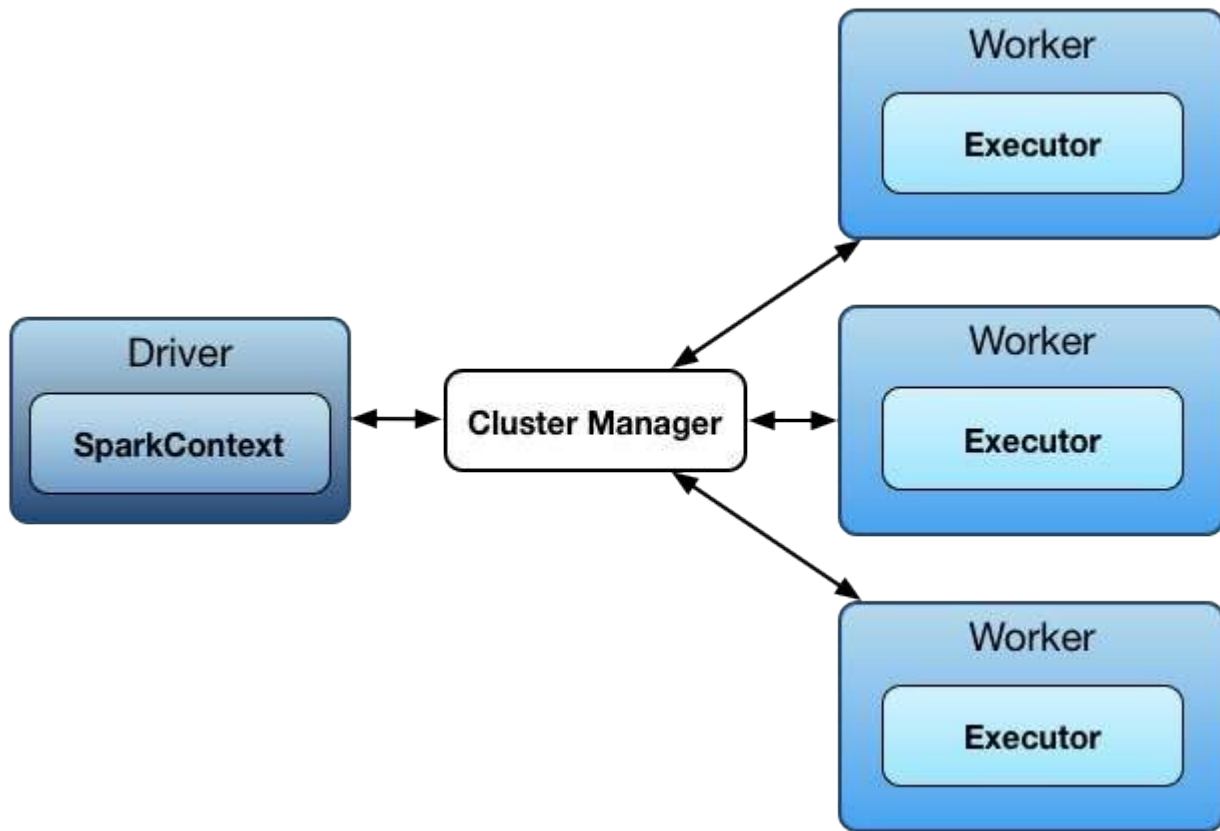
# Aplicaciones Spark

**2) 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 al clúster 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**.



# Spark

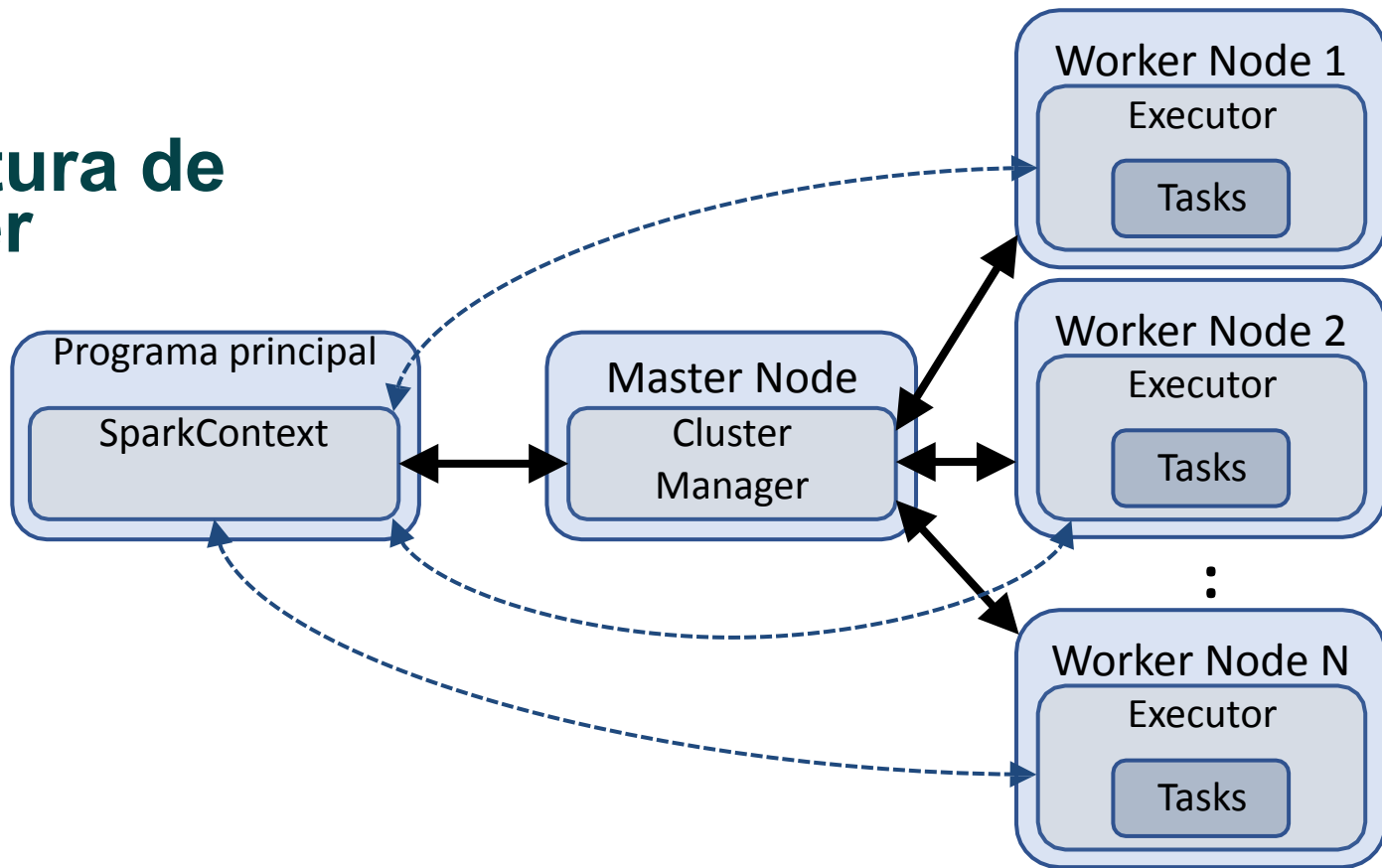
## Arquitectura





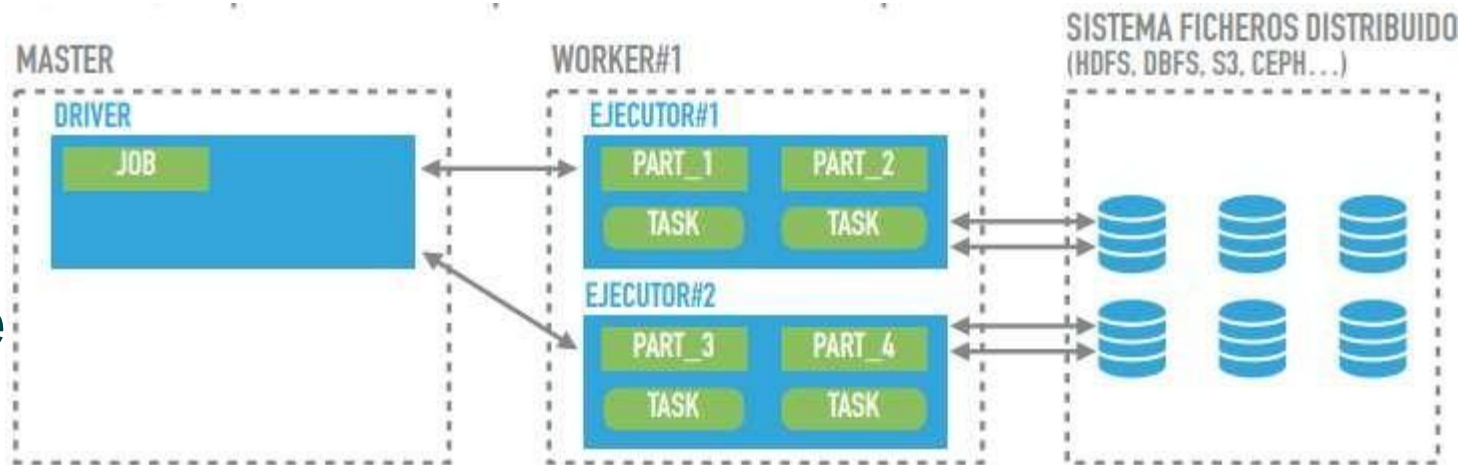
# Spark

## Arquitectura de un cluster



# Spark

## Modelo de Ejecución



- **Particiones:** Fragmento distribuido de los datos
- **Trabajos (job):** Operación sobre un conjunto de datos distribuido
- **Tareas (tasks):** Operación en paralelo sobre una partición


# Modelo de Ejecución

- Cada nodo puede tener más de un ejecutor.
- La cantidad de ejecutores por nodo depende de la configuración de Spark y de los recursos disponibles en el clúster.



# Modelo de Ejecución

## Definiciones clave

- **Nodo** (Node): Una máquina en el clúster de Spark.
  - **Ejecutor** (Executor): Un proceso que ejecuta tareas en un nodo y administra su propia memoria y CPU.
  - **Núcleos** (Cores): Unidades de procesamiento en un nodo utilizadas por los ejecutores.
  - **Driver**: El proceso central que coordina la ejecución del trabajo en Spark.
- 

# Modelo de Ejecución


Ejemplo:

Si tienes un nodo con 8 cores y 16 GB de RAM, puedes configurarlo así:

- 1 ejecutor con 8 cores y 16GB de RAM (1 ejecutor por nodo).
- 2 ejecutores con 4 cores y 8GB de RAM cada uno (2 ejecutores por nodo).
- 4 ejecutores con 2 cores y 4GB de RAM cada uno (4 ejecutores por nodo).

```
spark-submit --master yarn \  
  --num-executors 4 \  
  --executor-cores 2 \  
  --executor-memory 4G \  
  my_spark_job.py
```

# RDD (Resilient Distributed Dataset)

- Es el concepto básico de trabajo en Spark.
  - Representa una colección inmutable y distribuida de objetos que pueden ser procesados en paralelo.
  - Se pueden transformar para crear nuevos RDDs o realizar acciones sobre ellos pero no modificar.
  - Se guarda la secuencia de transformaciones para poder recuperar RDDs de forma eficiente si alguna máquina se cae.
  - Están distribuidos en el clúster en los nodos workers
- 

# Spark

## RDDs

```
from pyspark import SparkContext
```

```
sc = SparkContext()
```

```
# Crear un RDD a partir de una lista
```

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

```
# transformación map para multiplicar cada número por sí mismo
```

```
squared_rdd = rdd.map(lambda x: x * x)
```

```
# Recolectar y mostrar los resultados
```

```
print(squared_rdd.collect())
```

# DataFrame

- La principal abstracción de los datos en Spark es el **Dataset**.
- Por consistencia con el concepto de Pandas, los llamaremos **DataFrame**.
- Un **Dataset** es similar a un **DataFrame** pero con tipado fuerte y optimizaciones adicionales en Scala y Java.
- Si usas PySpark, DataFrame = Dataset (aunque no son lo mismo).
- Ofrecen una interfaz de alto nivel, similar a las tablas en una base de datos relacional, que permite realizar operaciones complejas de manera eficiente.



# DataFrame

```
from pyspark.sql import SparkSession

# Iniciar una sesión de Spark
spark = SparkSession.builder.appName("iabd").getOrCreate()

# Crear un DataFrame a partir de una lista de tuplas
data = [("Aitor", 28), ("Menta", 30), ("Verde", 26)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, schema=columns)

# Mostrar el DataFrame
df.show()

# Seleccionar personas mayores de 27 años
df.filter(df.Age > 27).show()
```

# RDD vs DataFrame

- Entonces, ¿un RDD es una fila y un DataFrame una tabla?
  - NO.
  - Cada elemento en un RDD puede ser una fila de datos (por ejemplo, un registro de log, una fila de una tabla, etc.), pero el RDD en sí es el conjunto completo de estos elementos distribuidos a través del clúster.



# RDD vs DataFrame

- Entonces, ¿un RDD es una fila y un DataFrame una tabla?
  - NO.
  - Cada elemento en un RDD puede ser una fila de datos (por ejemplo, un registro de log, una fila de una tabla, etc.), pero el RDD en sí es el conjunto completo de estos elementos distribuidos a través del clúster.

```
from pyspark import SparkContext  
sc = SparkContext()
```

# Crear un RDD a partir de una lista de tuplas (cada tupla representa una "fila" de datos)

```
data = [("John", 28), ("Jane", 30), ("Doe", 26)]  
rdd = sc.parallelize(data)
```

# Ahora, rdd contiene un conjunto de filas (cada fila es una tupla), y cada elemento (fila) puede ser procesado en paralelo.

# RDD vs DataFrame

## Uso de RDD para procesamiento de datos no estructurados

- Situación: Tienes un archivo de texto grande con registros de logs que quieres filtrar para encontrar errores específicos.

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()

# Leer datos de log como RDD
logs_rdd = sc.textFile("path/to/logfile.txt")

# Filtrar para encontrar registros de error
errors_rdd = logs_rdd.filter(lambda line: "ERROR" in line)

# Recolectar y mostrar algunos errores
for line in errors_rdd.take(10):
    print(line)
```

# RDD vs DataFrame

## Uso de DataFrame para análisis de datos estructurados

- Situación: Tienes un archivo CSV con datos de usuarios y quieres realizar un análisis para obtener la edad promedio.

```
from pyspark.sql import SparkSession

# Iniciar una sesión de Spark
spark = SparkSession.builder.appName("DataFrameExample").getOrCreate()

# Leer datos desde un CSV en un DataFrame
users_df = spark.read.csv("path/to/users.csv", header=True, inferSchema=True)

# Calcular la edad promedio de los usuarios
avg_age = users_df.groupBy().avg("age").collect()[0][0]

print(f"Edad promedio de los usuarios: {avg_age}")
```

# Spark

## ■ Transformaciones

- Son operaciones fundamentales que actúan sobre los RDDs
- Transforman un RDD en otro RDD (inmutables).
- Las transformaciones en Spark son **perezosas**:
  - No computan sus resultados inmediatamente.
  - Solo se ejecutan cuando se realiza una acción (como `collect()`, `count()`, `save()`, etc.), que requiere que Spark produzca un resultado tangible.
- Ejemplos: `map`, `filter`, `flatMap`, `join`, `reduceByKey`, `groupBy`.

# Spark

## ■ Acciones

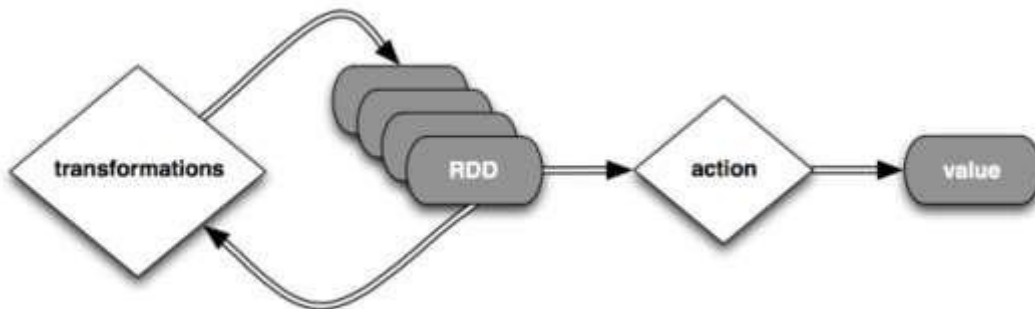
- Las acciones son operaciones que devuelven un resultado final del procesamiento de los datos.
- Cuando se ejecuta una acción, Spark inicia la ejecución de las transformaciones acumuladas.
- Ejecución Inmediata: A diferencia de las transformaciones, las acciones son ejecutadas inmediatamente.
- Ejemplos: collect, count, first, take, reduce, saveAsTextFile.

# Spark

## Acciones

### Actions

```
reduce(func)  
collect()  
count()  
first()  
take(n)  
saveAsTextFile(path)  
countByKey()  
foreach(func)  
...
```






# Componentes Spark

- **Spark Streaming** es una herramienta para la creación de aplicaciones de procesamiento en streaming que ofrece un gran rendimiento con soporte para la tolerancia a fallos.
- Los datos pueden venir desde fuentes de datos tan diversas como Kafka, Flume, Twitter (X) y tratarse en tiempo real.



# Componentes Spark

- **Spark SQL** ofrece un interfaz SQL para trabajar con Spark, permitiendo la lectura de datos tanto de una tabla de cualquier base de datos relacional como de ficheros con formatos estructurados (CSV, texto, JSON, etc...) y construir tablas permanentes o temporales en Spark.
  - Tras la lectura, permite combinar sentencias SQL para trabajar con los datos y cargar los resultados en un DataFrame.
- 

# Componentes Spark

- Por ejemplo, con este fragmento leemos un fichero JSON desde S3, creamos una tabla temporal y mediante una consulta SQL cargamos los datos en un DataFrame de Spark:

```
df = spark.read.json("s3://apache_spark/data/committers.json", header=True, sep=",")  
df.createOrReplaceTempView("committers")  
  
resultado = spark.sql("""SELECT name, org, module, release, num_commits FROM  
committers WHERE module = 'mllib' AND num_commits > 10 ORDER BY num_commits  
DESC""")
```

# Componentes Spark

- **Spark MLlib** es un módulo que ofrece la gran mayoría de algoritmos de ML y permite construir pipelines para el entrenamiento y evaluación de los modelos IA.
- **GraphX** permite procesar estructuras de datos en grafo, siendo muy útiles para recorrer las relaciones de una red social u ofrecer recomendaciones sobre gustos o afinidades.

# SparkContext

**SparkContext** es el punto de entrada a Spark desde las versiones 1.x. Su objeto **sc** está disponible en el spark-shell y se puede crear mediante la clase SparkContext.

```
from pyspark import SparkContext  
sc = SparkContext.getOrCreate()
```

# SparkSession

**SparkSession** se introdujo en la versión 2.0 y es el punto de entrada para crear RDD y DataFrames. El objeto **spark** se encuentra disponible por defecto en el spark-shell y se puede crear mediante el **builder** de SparkSession.

```
from pyspark.sql import SparkSession  
spark = SparkSession.builder.getOrCreate()
```

# SparkSession

Además, desde una sesión de Spark podemos obtener un contexto a través de la propiedad `sparkContext`:

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.getOrCreate()  
  
sc = spark.sparkContext
```

# SparkSession vs SparkContext

## Diferencia entre SparkSession y SparkContext

- En Spark, tanto SparkSession como SparkContext son esenciales para la ejecución de tareas, pero tienen usos distintos.






# SparkSession vs SparkContext

## 1. SparkContext (Nivel Bajo, RDD API)

SparkContext es la primera API utilizada en Spark (antes de Spark 2.0). Sirve para gestionar la conexión con el clúster, la configuración y la ejecución de tareas sobre RDDs.

Cuándo usar SparkContext:

- Si trabajas con RDDs directamente (`sc.textFile()`, `sc.parallelize()`).
  - Si necesitas un control más bajo sobre los recursos.
  - Si usas una versión de Spark anterior a 2.0 (donde SparkSession no existía).
- 

# SparkSession vs SparkContext

## Ejemplo con SparkContext

```
from pyspark import SparkContext  
  
sc = SparkContext.getOrCreate() # Crear o recuperar el SparkContext  
  
rdd = sc.parallelize([1, 2, 3, 4, 5]) # Crear un RDD  
print(rdd.collect()) # [1, 2, 3, 4, 5]
```

# SparkSession vs SparkContext

## 2. SparkSession

SparkSession fue introducido en Spark 2.0 como una API unificada que incluye SparkContext, SQLContext y HiveContext. Es la forma recomendada de interactuar con Spark en versiones modernas.

Cuándo usar SparkSession:

- Si trabajas con DataFrames o Datasets (`spark.read.csv()`, `spark.sql()`).
- Si usas SQL en Spark (`spark.sql("SELECT * FROM tabla")`).
- Si quieres un código más limpio y moderno.

# SparkSession vs SparkContext

## Ejemplo con SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Ejemplo").getOrCreate()
# Crear una sesión Spark

df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
# Leer CSV como DataFrame

df.show() # Mostrar las primeras filas
```

# SparkSession vs SparkContext

¿Cuál usar?

Característica	SparkContext (sc)	SparkSession (spark)
Trabaja con RDDs	✓ Sí	⚠ Puede, pero no recomendado
Trabaja con DataFrames	✗ No	✓ Sí (Optimizado)
Uso de SQL (spark.sql())	✗ No	✓ Sí
Uso con APIs modernas	✗ No	✓ Sí
Recomendado para nuevos proyectos	✗ No	✓ Sí

# SparkSession vs SparkContext

## Conclusión:

- Si trabajas con DataFrames y SQL → Usa **SparkSession** (recomendado).
- Si trabajas con RDDs y bajo nivel → Usa **SparkContext** (aunque se desaconseja para nuevos proyectos).



# Hola Spark

Lo primero que debemos hacer siempre es conectarnos a la sesión de *Spark*, el cual le indica a *Spark* cómo acceder al clúster.

## ejemploSpark.py

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext    # SparkContext a partir de la sesión
# Suma de los 100 primeros números
rdd = sc.parallelize(range(100 + 1))
rdd.sum()
```

## Colab: Ejemplo 0 - Computación paralela con Spark



```
Ejemplo 5.3 - Computación paralela con Spark
Archivo Editar Ver Insertar Entorno de ejecución Herramientas Ayuda Guardado con última vez: 10:04

+ Código + Texto

servidor de Spark. Además configuraremos el entorno Spark con las variables que sean necesarias.

NOTA:

1. la última versión de PySpark es la 3.1.2 link
2. Puede tardar un poco tiempo en hacer todos los procesos de descarga de datos pyspark tardar en

[1] # Install spark-related dependencies
apt-get install openjdk-8-jdk-headless -q > /dev/null
wget -q http://apache.osuosl.org/spark/spark-3.1.2/spark-3.1.2-bin-hadoop2.7.tgz
tar -xzf spark-3.1.2-bin-hadoop2.7.tgz

pip install -q findspark
pip install pyspark
# Set up required environment variables

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop2.7"
```



# Spark Submit

De la misma manera que mediante Hadoop podíamos lanzar un proceso al clúster para su ejecución, Spark ofrece el comando **spark-submit** para enviar un script al driver para su ejecución de forma distribuida.

```
spark-submit ejemploSpark.py
```

# AWS desde Spark

Para conectarse a **AWS** desde **Spark** hace falta:

1. Descargar dos librerías y configurarlas en `$SPARK_HOME/conf/spark-defaults.conf` (o colocarlas directamente en la carpeta `$SPARK_HOME/jars`):

```
# https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-aws
# https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-bundle/1.11.901
spark.driver.extraClassPath =
/opt/spark-3.3.1/conf/hadoop-aws-3.3.4.jar:/opt/spark-3.3.1/conf/aws-java-sdk-bundle-1.12.367.jar
spark.executor.extraClassPath =
/opt/spark-3.3.1/conf/hadoop-aws-3.3.4.jar:/opt/spark-3.3.1/conf/aws-java-sdk-bundle-1.12.367.jar
```

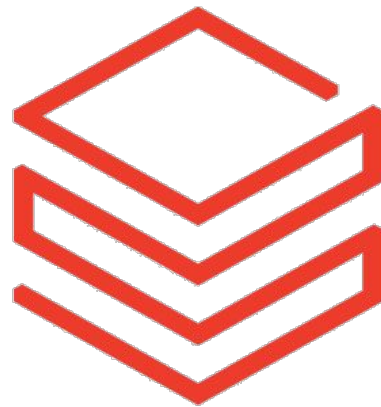
# AWS desde Spark

2. Configurar las **credenciales** de AWS en .aws/credentials
3. Tras crear la sesión de Spark, configurar el proveedor de credenciales:

```
spark = SparkSession.builder.getOrCreate()


spark._jsc.hadoopConfiguration().set("fs.s3a.aws.credentials.provider",
"com.amazonaws.auth.profile.ProfileCredentialsProvider")

df = spark.read.csv("s3a://s8a-spark-s3/departure_delays.csv")
```



**databricks**

# Databricks

- Databricks es una plataforma analítica de datos basada en Apache Spark.
  - Creada en 2013 por los desarrolladores principales de Spark, permite realizar analítica de Big Data e IA con Spark de una forma sencilla y colaborativa.
  - Databricks se integra de forma transparente con AWS, Azure y Google Cloud.
- 
- Two overlapping teal circles are located in the bottom right corner of the slide, serving as a decorative element.

# Databricks



## Try Databricks free

Test-drive the full Databricks platform free for 14 days on your choice of AWS, Microsoft Azure or Google Cloud. Sign-up with your work email to elevate your trial experience.

✓ Simplify data ingestion and automate ETL

Ingest data from hundreds of sources. Use a simple declarative approach to build data pipelines.

✓ Collaborate in your preferred language

Code in Python, R, Scala and SQL with coauthoring, automatic versioning, Git integrations and RBAC.

✓ 12x better price/performance than cloud data warehouses

See why over 7,000 customers worldwide rely on Databricks for all their workloads from BI to AI.



Mercedes-Benz

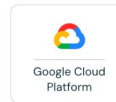
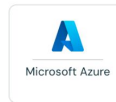


How will you be using Databricks?

2/2

### Professional use

Pick your cloud provider. You'll need admin access to your cloud account to get started.



Continue

By clicking "Get Started," you agree to the [Privacy Policy](#) and [Terms of Service](#).

### Personal use

Community Edition is a limited, single node version of Databricks for personal or educational use.

Get started with Community Edition

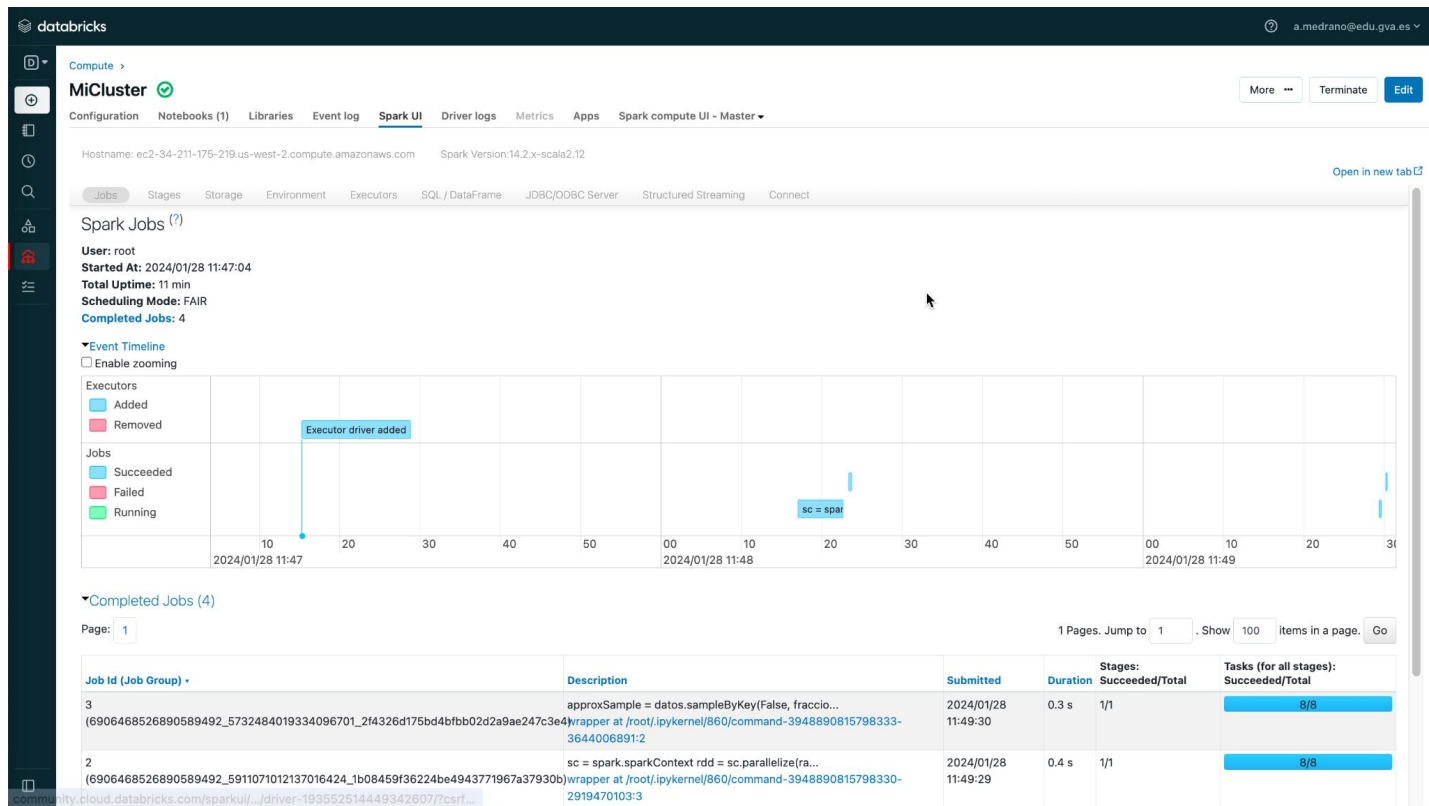
By selecting "Databricks Community Edition," you agree to the [Privacy Policy](#) and [Terms of Service](#).

Para poder trabajar de forma gratuita, podemos hacer uso de Databricks Community Edition, donde podemos crear nuestros propios cuadernos Jupyter y trabajar con Spark sin necesidad de instalar nada.

# Databricks UI

Databricks provee de un interfaz gráfico donde podemos monitorizar y analizar el código Spark ejecutado. La barra superior muestra un menú con las opciones para visualizar los jobs, stages, el almacenamiento, el entorno y sus variables de configuración, los ejecutores, etc.





Para acceder a la herramienta de monitorización en Databricks, una vez creado un clúster, en la opción Compute/Calcular podremos seleccionar el clúster creado y en la pestaña IU de Spark acceder al mismo interfaz gráfico.