# SPARK

Imagen que contiene dibujo

Descripción generada automáticamente

## INTRODUCCIÓN

Spark es un framework de computación distribuida en memoria.

El hecho de almacenar en memoria los cálculos intermedios implica que sea mucho más eficiente que otros, como por ejemplo MapReduce.

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

El diseño de Spark se basa principalmente en cuatro características:

* 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. Al ofrecer un conjunto de transformaciones y acciones como operaciones de su API, Spark facilita el desarrollo de aplicaciones Big data.
* Modularidad: soporte para todo tipo de cargas mediante cualquiera de los lenguajes de programación soportados: Scala, Java, Python, SQL y R, así como los módulos de Spark SQL para consultas interactivas, Spark Structured Streaming para procesamiento de datos en streaming, Spark MLlib para machine learning y GraphX para trabajar con grafos.

De esta manera, mediante una única aplicación Spark se puede hacer todo sin necesidad de utilizar APIs separadas.

* Extensibilidad: Al centrarse únicamente 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 términos de flexibilidad, Spark permite resolver múltiples tipos de procesamiento de datos, tanto aplicaciones batch como consultas interactivas, algoritmos de machine learning que requieren muchas iteraciones, aplicaciones de ingesta en streaming con rendimiento cercado al tiempo real, etc...

Antes de Spark, para cada uno de estos tipos de procesamiento necesitábamos una herramienta diferente, ahora con Spark tenemos una eficaz herramienta que reduce los costes y recursos necesarios.

### COMPONENTES DE SPARK

El elemento principal es **Spark Core** el cual aporta toda la funcionalidad necesaria para preparar y ejecutar las aplicaciones distribuidas, gestionando la planificación y tolerancia a fallos de las diferentes tareas.

Spark se puede ejecutar en batch o en modo interactivo y tiene soporte para Python, Java, Scala, R o SQL.

Además, contiene otros 4 grandes componentes construidos sobre el core:

Captura de pantalla de un celular

Descripción generada automáticamente

Componentes de Spark

1. **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 y tratarse en tiempo real.
2. **Spark SQL & Dataframes** 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, Avro, ORC, Parquet, etc...) y construir tablas en Spark. Tras permite combinar sentencias SQL y cargar los resultados en un DataFrame de Spark.

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

*df\_zips = spark.read.json("zips.json")*

*df\_zips.createOrReplaceTempView("zips")*

*df\_resultado = spark.sql("""SELECT city, state FROM zips WHERE pop > 10000 ORDER BY pop DESC""")*

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

Además, la comunidad de Spark dispone de un gran número de conectores para diferentes fuentes de datos, herramientas de monitorización, etc... que conforman su propio ecosistema:

Imagen de la pantalla de un celular con la imagen de una caricatura

Descripción generada automáticamente con confianza baja

### PYSPARK VS SPARK

PySpark y Spark son estrechamente relacionados, pero no son exactamente lo mismo:

* **Apache Spark**: Es un framework de procesamiento de datos en clústeres, escrito en Scala. Spark proporciona una API para Scala, Java, Python (a través de PySpark) y R. Al instalar Spark, obtienes acceso a todas estas APIs y la capacidad de ejecutar tareas de procesamiento de datos distribuidos, procesamiento en memoria, etc.
* **PySpark**: Es la interfaz para Apache Spark en Python. Permite a los usuarios interactuar con Apache Spark usando Python. PySpark ofrece la mayoría de las funcionalidades de Spark, pero en un lenguaje de programación que es más accesible para la comunidad de análisis de datos y científicos de datos.

Cuando instalas PySpark (por ejemplo, usando pip), internamente viene con una versión empaquetada de Spark, lo que significa que no necesitas instalar Spark por separado para trabajar con PySpark. Esta versión empaquetada está diseñada para funcionar de forma autónoma y está optimizada para un entorno de desarrollo y experimentación local.

Sin embargo, hay algunas consideraciones:

* Uso en Producción: Para entornos de producción, especialmente cuando se trabaja con clústeres de Spark distribuidos, es recomendable instalar y configurar una versión completa de Apache Spark que coincida con las necesidades específicas del entorno de producción.
* Funcionalidades Completas: Mientras que PySpark ofrece muchas de las funcionalidades de Spark, puede haber características específicas de Spark (especialmente aquellas en Scala o Java) que no estén disponibles o que sean menos accesibles en PySpark.
* Rendimiento y Optimización: En un entorno de producción, es posible que necesites ajustar y configurar Spark más allá de lo que permite la instalación predeterminada de PySpark.

En resumen, para propósitos de aprendizaje, desarrollo y experimentación en Python, instalar PySpark es suficiente. Pero para aplicaciones de producción o para usar todas las características y optimizaciones de Apache Spark, es recomendable instalar y configurar Apache Spark específicamente para tu entorno.

## PUESTA EN MARCHA

### PYSPARK EN UBUNTU

Para configurar PySpark con Jupyter en una máquina virtual con Ubuntu, sigue estos pasos:

1. **Instalar Java**: PySpark requiere Java para funcionar:

*sudo apt update*

*sudo apt install default-jdk*

1. **Verificar la instalación de Java:** Para asegurarte de que Java se ha instalado correctamente, ejecuta:

*java -version*

1. **Instalar Python y PIP:** Asegúrate de tener Python y pip instalados. Si no es así, puedes instalarlos con:

*python3 --version*

*sudo apt install python3-pip*

1. **Instalar PySpark**: Puedes instalar PySpark utilizando pip:

*pip install pyspark*

*nano ~/.bashrc*

Añadir las siguientes líneas:

*export PATH="$HOME/.local/bin:$PATH"*

*export JAVA\_HOME=$(dirname $(dirname $(readlink -f $(which javac))))*

*export SPARK\_HOME=$(python3 -c "import os; import pyspark; print(os.path.dirname(os.path.abspath(pyspark.\_\_file\_\_)))")*

*export PATH=$PATH:$SPARK\_HOME/bin:$SPARK\_HOME/sbin*

*export PYSPARK\_PYTHON="python3"*

Para aplicar los cambios

*source ~/.bashrc*

Con pyspark ya podemos abrir Spark (para salir exit())

*pyspark*

1. **Instalar Jupyter Notebook**: Si aún no tienes Jupyter Notebook, puedes instalarlo con:

*pip install notebook*

1. **Iniciar Jupyter Notebook**: Una vez que todo está instalado y configurado, puedes iniciar Jupyter Notebook con:

*jupyter notebook*

1. **Importar PySpark en Jupyter**: En tu notebook de Jupyter, puedes importar PySpark para empezar a usarlo:

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.master("local[1]").appName('PySparkExample').getOrCreate()*

SparkSession.builder: Inicia la construcción de una sesión de Spark.

.master("local[1]"): Define el entorno de ejecución de Spark. En este caso, local[1] significa que Spark se ejecutará localmente con un solo núcleo de procesador. local[\*] utilizaría todos los núcleos disponibles.

.appName('PySparkExample'): Asigna un nombre a la aplicación, que será útil para identificarla en la interfaz de usuario de Spark.

.getOrCreate(): Crea una nueva SparkSession si no existe ninguna o devuelve una ya existente.

1. **Verificación**: Para verificar que todo funciona correctamente, puedes ejecutar una tarea simple como:

*sc = spark.sparkContext*

*# Suma de los 100 primeros números*

*rdd = sc.parallelize(range(100 + 1))*

*rdd.sum()*

1. **PySpark y Jupyter Notebook**: Podemos hacer que al ejecutar pyspark en vez de que se abra la Shell de Spark se abra directamente Jupyter Notebook.

*nano ~/.bashrc*

Añadir las siguientes líneas:

*export PYSPARK\_DRIVER\_PYTHON=jupyter*

*export PYSPARK\_DRIVER\_PYTHON\_OPTS='notebook'*

Para aplicar los cambios

*source ~/.bashrc*

Ahora al ejecutar pyspark se debería abrir Jupyter Notebook

*pyspark*

### PYSPARK EN VISUAL STUDIO CODE

Para poder usar PySpark en nuestro equipo para pruebas tenemos que seguir los siguientes pasos:

1. **Instalar PySpark**: Puedes instalar PySpark utilizando pip:

*pip install pyspark*

1. **Importar PySpark**: En Visual Studio Code crear un fichero de Jupyter Notebook e importar PySpark

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.master("local[\*]").appName('PySparkExample').getOrCreate()*

1. **Verificación**: Para verificar que todo funciona correctamente, puedes ejecutar una tarea simple como:

*sc = spark.sparkContext*

*# Suma de los 100 primeros números*

*rdd = sc.parallelize(range(100 + 1))*

*rdd.sum()*

Nota para Windows: hay que instalar java, añadir las siguientes variables de entorno:

* Variables de usuario
  + Añadir a la variable Path: C:\Program Files\Java\jre-1.8\bin ( o la versión que sea)
* Variables del sistema
  + JAVA\_HOME = C:\Program Files\Java\jre-1.8 (o la versión que sea)
  + PYSPARK\_PYTHON = python

### SPARK EN LA NUBE

Para trabajar con Spark desde la nube disponemos de varias alternativas, ya sean herramientas que permiten trabajar con cuadernos Jupyter como pueden ser Google Colab o Databricks, o montar un clúster mediante AWS EMR (Elastic MapReduce) o Azure HDInsight.

#### GOOGLE COLAB

Para que funcione Spark dentro de Google Colab, únicamente hemos de instalar las librerías.

1. Instalar las dependencias

*!apt-get install openjdk-8-jdk-headless -qq > /dev/null*

*!wget -q https://downloads.apache.org/spark/spark-3.5.0/spark-3.5.0-bin-hadoop3.tgz*

*!tar xvf spark-3.5.0-bin-hadoop3.tgz*

*!pip install -q pyspark*

1. Configurar el entorno

*import os*

*os.environ["JAVA\_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"*

*os.environ["SPARK\_HOME"] = f"/content/spark-3.5.0-bin-hadoop3"*

1. Cargar Pyspark

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.appName("s8a").master("local[\*]").getOrCreate()*

*spark*

1. Probar Pyspark

*sc = spark.sparkContext*

*# Suma de los 100 primeros números*

*rdd = sc.parallelize(range(100 + 1))*

*rdd.sum()*

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

#### DATABRICKS

Databricks es una plataforma analítica de datos basada en Apache Spark que permite realizar analítica Big Data e Inteligencia Artificial con Spark de una forma sencilla y colaborativa.

Databricks se integra de forma transparente con AWS, Azure y Google Cloud.

Para poder trabajar con Databricks de forma gratuita, podemos hacer uso de [Databricks Community Edition](https://community.cloud.databricks.com/login.html), donde podemos crear nuestros propios cuadernos Jupyter y trabajar con Spark sin necesidad de instalar nada.

Tras registrarnos, tenemos que crear un clúster básico (con 15.3GB de memoria y dos núcleos) desde la opción Create del menú de la izquierda:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Tras un par de minutos se habrá creado y lanzado el clúster y ya estaremos listos para crear un nuevo notebook con acceso a Spark:

*spark*

*sc = spark.sparkContext*

*# Suma de los 100 primeros números*

*rdd = sc.parallelize(range(100 + 1))*

*rdd.sum()*

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Por defecto, el navegador del sistema de archivos de DataBricks está oculto. Para facilitar el acceso a los datos y visualizar la estructura y ruta de los mismos, lo podemos activar. Pinchar arriba a la derecha en nuestro usuario e ir a Admin Settings, ir a la pestaña Workspace settings y el apartado Advanced hay que activar la opción DBFS File Browser.

## PRIMEROS PASOS

### SPARKCONTEXT VS SPARKSESSION

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

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.getOrCreate()*

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*

### SPARK SUBMIT

Spark ofrece el comando spark-submit para enviar un script al driver para su ejecución de forma distribuida.

Así pues, si colocamos nuestro código en un archivo de Python:

**holaSpark.py**

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.getOrCreate()*

*sc = spark.sparkContext*

*rdd = sc.parallelize(range(100 + 1))*

*suma=rdd.sum()*

*print("------------------")*

*print(suma)*

*print("------------------")*

Lo podemos ejecutar desde fuera de Jupyter Notebook con el siguiente comando:

*spark-submit holaMundo.py*

Si en nuestro equipo hemos modificado las variables de entorno para que abrir spark con Jupyter Notebook habrá que deshacer el cambio:

*unset PYSPARK\_DRIVER\_PYTHON*

Si nuestro servidor estuviera en otra dirección IP, deberíamos indicarle donde encontrar el master:

*spark-submit --master spark://<ip-servidor-driver>:7077 holaMundo.py*

## ARQUITECTURA

Spark es un sistema distribuido diseñado para procesar grandes volúmenes de datos de forma rápida y eficiente.

Este sistema normalmente se despliega en un conjunto de máquinas que se conocen como un clúster Spark, pudiendo estar compuesta de unas pocas máquinas o miles de ellas.

A la hora del despliegue, se utiliza un sistema de gestión de recursos como el gestor propio de Spark (conocido como [Spark](https://spark.apache.org/docs/latest/spark-standalone.html) [Standalone](https://spark.apache.org/docs/latest/spark-standalone.html)), Apache Mesos, Kubernetes o YARN para gestionar de forma inteligente y eficiente el clúster.

Los dos componentes principales del clúster son:

* el **gestor de clúster**: nodo maestro que sabe dónde se localizan los esclavos, 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 nodos trabajadores (**workers**): cada nodo ofrece recursos (memoria, CPU, etc..) al gestor del clúster y realiza las tareas que se le asignen.

### 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 machine learning.
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.

Cada ejecutor es un proceso JVM (Java Virtual Machine) dedicado para una aplicación Spark específica. Un ejecutor vivirá tanto como dure la aplicación Spark, lo cual puede ser segundos, minutos o días, dependiendo de la complejidad de la aplicación. Conviene destacar que los ejecutores son elementos aislados que no se comparten entre aplicaciones Spark, por lo que la única manera de compartir información entre diferentes ejecutores es mediante un sistema de almacenamiento externo como HDFS.

Diagrama

Descripción generada automáticamente

Así pues, Spark utiliza una arquitectura maestro/esclavo, donde el driver es el maestro, y los ejecutores los esclavos. Cada uno de estos componentes se ejecutan como un proceso independiente en el clúster Spark. Por lo tanto, una aplicación Spark se compone de un driver y múltiples ejecutores. Cada ejecutor realiza lo que se le pide en forma de tareas, ejecutando cada una de ellas en un núcleo CPU separado. Así es como el procesamiento paralelo acelera el tratamiento de los datos. Además, cada ejecutor, bajo petición de la lógica de la aplicación, se responsabiliza de cachear un fragmento de los datos en memoria y/o disco.

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 ejecutor.

Diagrama

Descripción generada automáticamente

### JOB, STAGE Y TASK

Cuando creamos una aplicación Spark, por debajo, se distinguen los siguientes elementos:

* **Job** (trabajo): computación paralela compuesta de múltiples tareas que se crean tras una acción de Spark (sabe, collect…). Al codificar nuestro código mediante PySpark, el driver convierte la aplicación Spark en uno o más jobs, y a continuación, estos jobs los transforma en un DAG (grafo). Este grafo, en esencia, es el plan de ejecución, donde cada elemento dentro del DAG puede implicar una o varias stages (escenas).
* **Stage** (etapa): cada job se divide en pequeños conjuntos de tareas que forman un etapa. Como parte del grafo, las stages se crean a partir de si las operaciones se pueden realizar de forma paralela o de forma secuencial. Como no todas las operaciones pueden realizarse en una única stage, en ocasiones se dividen en varias, normalmente debido a los límites computacionales de los diferentes ejecutores.
* **Task** (tarea): unidad de trabajo más pequeña que se envía a los ejecutores Spark. Cada etapa se compone de varias tareas. Cada una de las tareas se asigna a un único núcleo y trabaja con una única partición de los datos. Por ello, un ejecutor con 16 núcleos puede tener 16 o más tareas trabajando en 16 o más particiones en paralelo.

Imagen de la pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza baja

### DATAFRAME

La principal abstracción de los datos en Spark es el **Dataset**.

Se pueden crear desde las fuentes de entrada de Hadoop (como ficheros que provienen de HDFS o S3) o mediante transformaciones de otros Datasets. Dado el cariz de Python, no necesitamos que los Dataset estén fuertemente tipados, por eso, todos los Dataset que usemos serán Dataset[Row] (si trabajásemos mediante Java o Scala sí deberíamos indicar el tipo de sus datos), y por consistencia con Pandas los llamaremos DataFrame.

Veamos cómo podemos crear un DataFrame a partir de un fichero de texto:

*from pyspark.sql import SparkSession*

*spark = SparkSession.builder.getOrCreate()*

*quijoteTxt = spark.read.text("el\_quijote.txt")*

*quijoteTxt.count() # número de filas del DataFrame*

*>>>2186*

*quijoteTxt.first() # primera fila*

*>>>Row(value='DON QUIJOTE DE LA MANCHA')*

*# Transformamos un DataFrame en otro nuevo, con las líneas que contiene la palabra Quijote*

*lineasConQuijote = quijoteTxt.filter(quijoteTxt.value.contains("Quijote"))*

*lineasConQuijote.count() # cantidad de líneas con la palabra Quijote*

*>>>584*

*# Las transformaciones se pueden encadenar*

*quijoteTxt.filter(quijoteTxt.value.contains("Quijote")).count()*

*>>584*

Más adelante veremos los Dataframe en profundidas