```
# Instalar SDK Java 8

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

# Descargar Spark 3.2.2

!wget -q https://archive.apache.org/dist/spark/spark-3.2.3/spark-3.2.3-bin-hadoop3.2.tgz

# Descomprimir el archivo descargado de Spark

!tar xf spark-3.2.3-bin-hadoop3.2.tgz

# Establecer las variables de entorno

import os

os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.2.3-bin-hadoop3.2"

# Instalar la librería findspark

!pip install -q findspark

# Instalar pyspark

!pip install -q pyspark

### verificar la instalación ###

import findspark

findspark.init()

from pyspark.sql import SparkSession

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

# Probando la sesión de Spark
df = spark.createDataFrame([{"Hola": "Mundo"} for x in range(4)])

df.show(10, False)
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 281.4/281.4 MB 5.5 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.7/199.7 KB 14.1 MB/s eta 0:00:00
  Building wheel for pyspark (setup.py) ... done
+-----+
|Hola |
+-----+
|Mundo|
|Mundo|
|Mundo|
|Mundo|
+-----+
```

```
#Iniciar mi sesion de spark

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

spark
```

SparkSession - in-memory

SparkContext

Spark UI

Version
v3.2.3
Master
local[*]
AppName
pyspark-shell

```
# Diferentes formas de crear un RDD

sc = spark.sparkContext

# Crear un RDD vacío
```

```
rdd_vacio = sc.emptyRDD

# #Crear un rdd con la función parallelize vacio, pero con 3 particiones:

rdd_vacio3 = sc.parallelize([], 3)

rdd_vacio3.getNumPartitions()
```

```
3
```

```
# Crear un RDD con parallelize con una lista de datos:

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

rdd.collect()
```

```
[1, 2, 3, 4, 5]
```

```
# Crear un RDD desde un archivo de texto
from google.colab import files
from google.colab import drive
drive.mount('/content/drive')

#rdd_source = spark.read.text(files.upload())
rdd_texto = spark.read.text('/content/drive/MyDrive/pyspark/rdd_source.txt')

rdd_texto.collect()
```

```
Mounted at /content/drive
[Row(value='Así podemos crear'),
 Row(value='un RDD desde un'),
 Row(value='archivo de texto!!!')]
```

```
# Crear un RDD desde un archivo de texto dode todo el archivo es un solo registro:
rdd_texto_completo = sc.wholeTextFiles('/content/drive/MyDrive/pyspark/rdd_source.txt')

rdd_texto_completo.collect()
```

```
[('file:/content/drive/MyDrive/pyspark/rdd_source.txt',
  'Así podemos crear\nun RDD desde un\narchivo de texto!!!')]
```

```
rdd_suma = rdd.map(lambda x: x +1)

rdd_suma.collect()

df = spark.createDataFrame([(1, 'jose'), (2, 'juan')], ['id', 'nombre'])

df.show()

rdd_df = df.rdd

rdd_df.collect()
```

```
+---+------+
| id|nombre|
+---+------+
|  1|  jose|
|  2|  juan|
+---+------+

[Row(id=1, nombre='jose'), Row(id=2, nombre='juan')]
```

```
#TRANSFORMACIONES SOBRE RDD's:
"""
map
filter
flatMap
groupByKey
sortByKey
combineByKey
sampleByKey
randomSplit
partitionBy
repartition
```

```
zipwithIndex
coalesce
"""
```

```
'\nmap\nfilter\nflatMap\ngroupByKey\nsortByKey\ncombineByKey
```

```
#TRANSFORMACION MAP

#Utilizo un rdd creado anteriormente para aplicar la transformación map y mostar true para valores pares

rdd_par = rdd.map(lambda x: x % 2 == 0)

rdd_par.collect()
```

```
[False, True, False, True, False]
```

```
#transformar de minusculas a mayusculas:

rdd_texto = sc.parallelize(['jose', 'juan', 'lucia'])

rdd_mayuscula = rdd_texto.map(lambda x: x.upper())

rdd_mayuscula.collect()
```

```
['JOSE', 'JUAN', 'LUCIA']
```

```
#Concatenar 'hola' con el rrd anterior:

rdd_hola = rdd_texto.map(lambda x: 'Hola ' + x)

rdd_hola.collect()
```

```
['Hola jose', 'Hola juan', 'Hola lucia']
```

```
# Transformaciones: función flatMap

#cada particion del rdd da lugar a nuevas particiones de rdd

import findspark
findspark.init()
from pyspark.sql import SparkSession

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


#Utilizando el map anterior obtengo una lista de tuplas
rdd_cuadrado = rdd.map(lambda x: (x, x ** 2))

rdd_cuadrado.collect()
```

```
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

```
#Si aplano ese mismo rdd (utilizando flatmap):

rdd_cuadrado_flat = rdd.flatMap(lambda x: (x, x ** 2))

rdd_cuadrado_flat.collect()
```

```
[1, 1, 2, 4, 3, 9, 4, 16, 5, 25]
```

```
#Igualmente pero con texto:

rdd_texto = sc.parallelize(['jose', 'juan', 'lucia'])

rdd_mayuscula = rdd_texto.flatMap(lambda x: (x, x.upper()))

rdd_mayuscula.collect()
```

```
['jose', 'JOSE', 'juan', 'JUAN', 'lucia', 'LUCIA']
```

```
# Transformaciones: función filter

#Obtener elementos pares solamente

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

rdd_par = rdd.filter(lambda x: x % 2 == 0)
```

```
rdd_par.collect()
```

```
[2, 4, 6, 8]
```

```
#Obtener elementos impares solamente
rdd_impar = rdd.filter(lambda x: x % 2 != 0)

rdd_impar.collect()
```

```
[1, 3, 5, 7, 9]
```

```
#Filtrar por nombres que contengan la letra k:

rdd_texto = sc.parallelize(['jose', 'juaquin', 'juan', 'lucia', 'karla', 'katia'])

rdd_k = rdd_texto.filter(lambda x: x.startswith('k'))

rdd_k.collect()
```

```
['karla', 'katia']
```

```
#Que empiece por la letra 'j' y contengan la letra 'u'
rdd_filtro = rdd_texto.filter(lambda x: x.startswith('j') and x.find('u') == 1)

rdd_filtro.collect()
```

```
['juaquin', 'juan']
```

```
#Genero 10 particiones
rdd = sc.parallelize([1,2,3.4,5], 10)

rdd.getNumPartitions()
```

```
10
```

```
#Creo un rdd a partir del existente, esta vez con 5 particiones
rdd5 = rdd.coalesce(5)

rdd5.getNumPartitions()
```

```
5
```

```
#Transformación repartition crea un rdd a partir del existente, distribuyendo las particiones combinandolas o uniendo según el caso.
rdd = sc.parallelize([1,2,3,4,5], 3)
rdd.getNumPartitions()
rdd7 = rdd.repartition(7) #incremento el número de particiones original a 7

rdd7.getNumPartitions()
```

```
7
```

Coalesce se usa solo para reducir el número de particiones. Esta es una versión optimizada de repartition donde el movimiento de los datos a través de las particiones es menor. Ambas son operaciones muy costosas.

```
#Trasformación reduceByKey
rdd = sc.parallelize(
    [('casa', 2),
     ('parque', 1),
     ('que', 5),
     ('casa', 1),
     ('escuela', 2),
     ('casa', 1),
     ('que', 1)]
)

print(rdd.collect())

rdd_reduciodo = rdd.reduceByKey(lambda x,y: x + y)

print(rdd_reduciodo.collect())
```

```
[('casa', 2), ('parque', 1), ('que', 5), ('casa', 1), ('escuela', 2), ('casa', 1), ('que', 1)]
[('parque', 1), ('que', 6), ('casa', 4), ('escuela', 2)]
```

```
#Tipos de acciones:
"""
Driver
Distributed

"""
```

```
#Función reduce

#Suma de todos los elementos del rdd

rdd = sc.parallelize([2,4,6,8])

rdd.reduce(lambda x,y: x + y)
```

```
20
```

```
#Multiplicación de los elementos de los elementos del rdd
rdd1 = sc.parallelize([1,2,3,4])

rdd1.reduce(lambda x,y: x * y)
```

```
24
```

```
#Función count:
#Contar elementos de mi rdd
rdd = sc.parallelize(['j', 'o', 's', 'e'])

rdd.count()
```

```
4
```

```
#Contar elementos de mi rdd generado
rdd1 = sc.parallelize([item for item in range(10)])

rdd1.count()
```

```
10
```

```
#Función collect:
#Crear un rdd de un texto
rdd = sc.parallelize('Hola Apache Spark!'.split(' '))

rdd.collect()
```

```
['Hola', 'Apache', 'Spark!']
```

```
#Misma idea pero con números
rdd1 = sc.parallelize([(item, item ** 2) for item in range(5)])

rdd1.collect()
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
```

```
# take

rdd = sc.parallelize('La programación es bella'.split(' '))

print(rdd.take(2))

print(rdd.take(4))
```

```
['La', 'programación']
['La', 'programación', 'es', 'bella']
```

```
# max

rdd1 = sc.parallelize([item/(item + 1) for item in range(10)])

print(rdd1.max())
```

```
print(rdd1.collect())
```

```
0.9
[0.0, 0.5, 0.6666666666666666, 0.75, 0.8, 0.8333333333333334, 0.8571428571428571, 0.875, 0.8888888888888888, 0.9]
```

```
# saveAsTextFile
"""
rdd.collect()

rdd.saveAsTextFile('./rdd')

rdd.coalesce(1).saveAsTextFile('./rdd1')

"""
```

```
'\nrdd.collect()\n\nrdd.saveAsTextFile('./rdd')\n\nrdd.coales
```

El almacenamiento en caché permite que Spark conserve los datos en todos los cálculos y operaciones.

De hecho, esta es una de las técnicas más importantes de Spark para acelerar los cálculos, especialmente cuando se trata de cálculos interactivos.

El almacenamiento en caché funciona almacenando el RD tanto como sea posible en la memoria.

Si los datos que se solicitan para almacenar en caché son más grandes que la memoria disponible, el rendimiento disminuirá porque se utilizará disco en lugar de memoria.

Puede marcar una red como almacenado en caché usando PER o caché.

Caché simplemente un sinónimo de perfil con la opción memory only.

Ver si puede usar memoria o disco o ambos.

Los siguientes son los valores posibles para el nivel de almacenamiento.

Como pueden observar, la primera opción Memory Only, esta almacena la RD como un objeto de serialización en la máquina virtual de Java.

Si el RD no cabe memoria, algunas particiones no se almacenarán en caché y se volverán a calcular sobre la marcha cada vez que se necesiten.

Hay que tener en cuenta que este es el nivel por defecto que trae par.

El siguiente nivel de almacenamiento es Memory and Disk.

Esta almacena los RDF como objetos ya realizados en la máquina virtual de Java.

Si el RD no cabe en memoria, almacena las particiones que no quepan en el disco y las lee desde allí cuando sea necesario.

El siguiente nivel de almacenamiento es this only.

Esta almacena las particiones del RD solo en disco.

Y por último, a nivel de almacenamiento tenemos Memory Only dos o Memory Only Disk dos, igual que los niveles anteriores, pero replica cada partición en los nodos del cluster.

El nivel de almacenamiento a elegir depende de la situación.

Por ejemplo, si los rebeldes caben en la memoria, use memory only, ya que es la opción más rápida para el rendimiento de ejecución.

This only no debe usarse a menos que sus cálculos sean costosos.

Por último, utiliza almacenamiento replicado para una mejor tolerancia falla si puede ahorrar la memoria adicional necesaria.

Esto evitará que se vuelvan a calcular las particiones perdidas para obtener la mejor disponibilidad.

Podemos utilizar la función un perfil simplemente para liberar el contenido en caché.

```
# Persistir con la función storagelevel.MEMORY.ONLY

rdd = sc.parallelize([item for item in range(10)])

from pyspark.storagelevel import StorageLevel
```

```
rdd.persist(StorageLevel.MEMORY_ONLY)
```

```
ParallelCollectionRDD[87] at readRDDFromFile at PythonRDD.scala:274
```

```
rdd.unpersist() #El unpersist es necesario para hacer nuevos cambios de persistencia
```

```
ParallelCollectionRDD[87] at readRDDFromFile at PythonRDD.scala:274
```

```
#Ahora puedo utilizar un nuevo nivel de persistencia

rdd.persist(StorageLevel.DISK_ONLY)
```

```
ParallelCollectionRDD[87] at readRDDFromFile at PythonRDD.scala:274
```

```
#Un nuevo cambio
rdd.unpersist()

rdd.cache() #Memory only
```

```
ParallelCollectionRDD[87] at readRDDFromFile at PythonRDD.scala:274
```

PARTICIONADO DE DATOS:

Los RBD operan con datos no como una sola masa de datos, sino que administran y operan los datos en particiones repartidas por todo el cluster.

Por lo tanto, el concepto de partición de datos es fundamental para el correcto funcionamiento de los chop de Apache span y puede tener un gran efecto en el rendimiento y en la forma en que se utilizan los recursos.

Los RD constan de particiones de datos y todas las operaciones se realizan en las particiones de datos en el RD.

Varias operaciones, como las transformaciones, son funciones ejecutadas por un ejecutor en la partición específica de datos en la que se opera.

Sin embargo, no todas las operaciones pueden realizarse simplemente realizando operaciones aisladas en las particiones de datos por parte de los respectivos ejecutores.

Las operaciones como las agregaciones, requieren que los datos se muevan a través del cluster en una fase conocida como mezcla o shuffle.

```
#Aplicar un hash partitioner de forma manual:

rdd = sc.parallelize(['x', 'y', 'z'])

hola = 'Hola'

hash(hola)
```

```
746465152603520417
```

```
num_particiones = 6

# indice = hash(item) % num_particiones

hash('x') % num_particiones #indice de particion

hash('y') % num_particiones

hash('z') % num_particiones
```

```
4
```

```
#Variables Broadcast:

rdd = sc.parallelize([item for item in range(10)])

uno = 1
```

```
br_uno = sc.broadcast(uno) #creo la variable

rdd1 = rdd.map(lambda x: x + br_uno.value) #Utilizo esa variable broadcast

rdd1.collect()
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
br_uno.unpersist() #Retiro el valor del broadcast anterior y vuelvo a aplicarlo

rdd1  = rdd.map(lambda x: x + br_uno.value)

rdd1.collect()
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
br_uno.destroy() #tambien puedo destruir el dato bradcast en vez de retirarlo
```

ACUMULADORES:

Los acumuladores son variables compartidas entre los ejecutores que normalmente se usan para agregar contadores a su programa en Spark.

Algunos puntos que debemos de tener en cuenta son.

A través de Spark con un punto acumulador se puede usar para definir una variable de acumulador.

La función AD se usa para agregar o actualizar un valor en el acumulador.

La propiedad value de la variable del acumulador se utiliza para recuperar el valor del acumulador.

```
#valor acumulativo de los valores de un rdd:

acumulador = sc.accumulator(0) #creo el acumulador

rdd = sc.parallelize([2,4,6,8,10])

rdd.foreach(lambda x: acumulador.add(x))

print(acumulador.value)
```

```
30
```

```
#Número de elementos de un rdd:
#Contador de palabras
rdd1 = sc.parallelize('Mi nombre es Jose Miguel y me siento genial'.split(' '))

acumulador1 = sc.accumulator(0)

rdd1.foreach(lambda x: acumulador1.add(1))

print(acumulador1.value)
```

```
9
```