1. **Instalar Spark**:

   a. Revisar archivo “**Instalar_Spark.ipynb**”.

```
print("\n\nPREPARANDO EL ENTORNO\n\n")
import os
 # Instalar SDK java 8
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
# Descargar Spark
!wget -q https://archive.apache.org/dist/spark/spark-3.3.1/spark-3.3.1-bin-hadoop3.tgz
# Descomprimir la versión de Spark
!tar xf spark-3.3.1-bin-hadoop3.tgz
# Establecer las variables de entorno
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.3.1-bin-hadoop3"
# Descargar findspark
!pip install -q findspark
# Descargar pyspark
!pip install -q pyspark
print("\n\n******** INSTALACIÓN CORRECTA *******")
```

2. **SparkSession**:

   a. Revisar archivo “**SparkSession.ipynb**”.

```
# Para identificar la ruta de instalación
import findspark
findspark.init()

from pyspark.sql import SparkSession
# master, en este caso indica que es spark local y utiliza todos los cores
spark = SparkSession.builder.master("local[*]").appName("Curso PySpark").getOrCreate()
```

```
spark
```

SparkSession - in-memory

SparkContext

Spark UI

Version
    v3.3.1
Master
    local[*]
AppName
    Curso PySpark

3. **¿Qué es un RDD?**: Resident Distributed Dataset (RDD) es la abstracción principal de Spark.

   a. Revisar archivo “**CrearRDD.ipynb**”. Ejemplo,

```
# Creamos variable de sparkContext
sc = spark.sparkContext
```

```
# Crear RDD vacio (Sin particion)
rdd_vacio = sc.emptyRDD
```

```
lista = []
lista = RDD_PRIMO.filter(lambda x: x > 10).collect()
RDD_MAYOR_10 = sc_rdd.parallelize(lista)
RDD_MAYOR_10.collect()
```

4. **Transformaciones en un RDD**:

   a. Ver archivo "**TransformacionesRDD.ipynb**".

   b. Son **3**, dividir, filtrar, realizar cálculos.

   c. Categorías de las **transformaciones**:

   i. Transformaciones generales. Ejemplo,
      1. Map.
      2. Filter.
      3. flatMap.
      4. groupByKey.
      5. sortByKey.
      6. combineByKey.

   ii. Transformaciones matemáticas o estadísticas. Ejemplo,
      1. sampleByKey.
      2. randomSplit.

   iii. Transformaciones de conjunto o relacionales. Ejemplo,
      1. cogroup.
      2. join.
      3. subtractByKey.
      4. fullOuterJoin.
      5. leftOuterJoin.
      6. rightOuterJoin.

   iv. Transformaciones basadas en estructura de datos.
      1. partitionBy.
      2. repartition.
      3. zipwithIndex.
      4. coalesce.

d. Función **tuple**, retorna una tupla.

e. Función **replace**, reemplaza.

f. Función **split**, separa.

5. Acciones sobre un RDD en Spark

a. Ver archivo "**AccionesRDD.ipynb**".

b. Función reduce:

```
1 rdd = sc.parallelize([2, 4, 6, 8])
2 rdd.reduce(lambda x,y: x + y)
```

c. Función count:

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

```
1 rdd1.count()
```

d. Función reduce:

```
1 rdd = sc.parallelize([2, 4, 6, 8])
2 rdd.reduce(lambda x,y: x + y)
```

```
20
```

```
1 rdd1 = sc.parallelize([1,2,3,4])
2 rdd1.reduce(lambda x,y: x * y)
```

```
24
```

e. Función count:

```
1 rdd = sc.parallelize(['j', 'o', 's', 'e'])
```

```
1 rdd.count()
```

```
4
```

f. Función collect:

```
1 rdd = sc.parallelize('Hola Apache Spark!'.split(' '))
```

```
1 rdd.collect()
```

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

g. Función take, max y saveTextFile:

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

```
1 rdd.take(2)
```

```
['La', 'programación']
```

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

```
1 rdd1.max()
```

```
0.9
```

```
1 rdd.saveAsTextFile('./rdd')
```

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

h. Ejercicio:

1. Cree un RDD llamado importes a partir del archivo adjunto a esta lección como recurso.

```
1 importes = sc.textFile("./num.txt")
2 importes = importes.map(lambda _: int(_))
3 importes.collect()
```

1.a. ¿Cuántos registros tiene el RDD importes?

```
[32]  1 importes.count()
```

```
233
```

1.b. ¿Cuál es el valor mínimo y máximo del RDD importes?

```
[33]  1 importes.max()
```

```
99
```

```
[34]  1 importes.min()
```

```
1
```

1.c. Cree un RDD top15 que contenga los 15 mayores valores del RDD importes. Tenga en cuenta que pueden repetirse los valores. Por último, escriba el RDD top15 como archivo de texto en la carpeta data/salida.

```
[37]  1 top15 = importes.sortBy(lambda x: -x)
      2 top15.take(15)
```

```
[99, 99, 98, 98, 97, 97, 97, 97, 97, 96, 96, 95, 95, 95, 95]
```

```
[38]  1 top15.coalesce(1).saveAsTextFile("./top15")
```

Otra forma:

```
1 lista_top15 = importes.top(15)
2 top15 = sc.parallelize(lista_top15)
```

```
1 top15.coalesce(1).saveAsTextFile("./data/salida/")
```

2. Cree una función llamada factorial que calcule el factorial de un número dado como parámetro. Utilice RDDs para el cálculo.

```
]  1 import math
   2
   3 def factorial(c):
   4   return math.factorial(c)
   5
   6 rdd_factorial = sc.parallelize([(item, factorial(item)) for item in range(10)])
   7 rdd_factorial.collect()
   8
```

```
[(0, 1),
 (1, 1),
 (2, 2),
 (3, 6),
 (4, 24),
 (5, 120),
 (6, 720),
 (7, 5040),
 (8, 40320),
 (9, 362880)]
```

Otra forma:

```
1 def factorial_sin_funcion(n):
2   if n == 0:
3     return 1
4   else:
5     rdd = sc.parallelize(list(range(1, n+1)))
6     return rdd.reduce(lambda x,y: x * y)
7
8 factorial_sin_funcion(4)
```

6. Aspectos avanzados sobre RDD

a. Ver archivo "**AvanzadoRDD.ipynb**".

b. Almacenamiento en caché:

i. permite conservar los datos en todos los cálculos y operaciones.

ii. Se puede marcar un RDD como almacenado en caché usando **persist()** o **cache()**.

iii. **Cache()** es un sinónimo de **persist(MEMORY_ONLY)**

iv. **Persist()** puede usar memoria o disco o ambos.

v. Nivel de almacenamiento:

Nivel de Almacenamiento	Significado
MEMORY_ONLY	Almacena RDD como objetos Java deserializados en la JVM. Si el RDD no cabe en la memoria, algunas particiones no se almacenarán en caché y se volverán a calcular sobre la marcha cada vez que se necesiten. Este es el nivel por defecto.
MEMORY_AND_DISK	Almacena RDD como objetos Java deserializados en la JVM. Si el RDD no cabe en la memoria, almacena las particiones que no quepan en el disco y las lee desde allí cuando sea necesario.
DISK_ONLY	Almacene las particiones RDD solo en el disco.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Igual que los niveles anteriores, pero replica cada partición en dos nodos del clúster.

c. Particionado:

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

ii. Si la cantidad de particiones es demasiado pequeña, usaremos solo unas pocas CPU/núcleos en una gran cantidad de datos, por lo que tendremos un rendimiento más lento y dejaremos el clúster subutilizado.

iii. Si la cantidad de particiones es demasiado grande, utilizará más recursos de los que realmente necesita y, en un entorno de múltiples procesos, podría estar provocando la falta de recursos para otros procesos que usted u otros miembros de su equipo ejecutan.

d. Mezcla de datos (shuffling)

i. El movimiento de datos para el reparticionamiento se denomina **shuffling**.

e. Broadcasts variables:

i. Las variables broadcast son variables compartidas entre todos los ejecutores. Estas se crean una vez en el controlador y luego se leen sólo en los ejecutores.

f. Acumuladores:

i. Los acumuladores son variables compartidas entre ejecutores que normalmente se utilizan para agregar contadores a su programa Spark.

g. Ejercicio:

1. Cree un RDD importes a partir de los datos adjuntos a esta lección como recurso. Emplee acumuladores para obtener el total de ventas realizadas y el importe total de las ventas.

```
5]  1 # Cargamos el fichero
    2 importes = sc.textFile("./rdd.txt")
    3 importes = importes.map(lambda _: int(_))
    4 importes.take(5)
```

```
[527, 386, 701, 240, 941]
```

```
3]  1 acumulador = sc.accumulator(0)
    2 importes.foreach(lambda x: acumulador.add(1))
    3 print(acumulador.value)
```

```
10000
```

```
4]  1 importes.foreach(lambda x: acumulador.add(x))
    2 print(acumulador.value)
```

```
5052335
```

2. Si se conoce que a cada venta hay que restarle un importe fijo igual a 10 pesos por temas de impuestos.

2.1. ¿Cómo restaría este impuesto de cada venta utilizando una variable broadcast para acelerar el proceso?

R. broadcast = 10, se resta en el lambda

2.2. Cree un RDD llamado ventas_sin_impuestos a partir de la propuesta del inciso a que contenga las ventas sin impuestos.

```
[47]  1 dies = 10
      2 br_dies = sc.broadcast(dies)
      3 ventas_sin_inpuestos = importes.map(lambda x: x - br_dies.value)
      4 ventas_sin_inpuestos.take(5)
```

```
[517, 376, 691, 230, 931]
```

2.3. Destruya la variable broadcast creada luego de emplearla para crear el RDD del inciso b.

```
[48]  1 br_dies.destroy()
```

3. Persista el RDD ventas_sin_impuestos en los siguientes niveles de persistencia.

```
49]  1 ventas_sin_inpuestos.persist(StorageLevel.MEMORY_ONLY)
```

```
PythonRDD[30] at RDD at PythonRDD.scala:53
```

```
51]  1 ventas_sin_inpuestos.unpersist()
```

```
PythonRDD[30] at RDD at PythonRDD.scala:53
```

```
52]  1 ventas_sin_inpuestos.persist(StorageLevel.DISK_ONLY)
```

```
PythonRDD[30] at RDD at PythonRDD.scala:53
```

```
53]  1 ventas_sin_inpuestos.unpersist()
```

```
PythonRDD[30] at RDD at PythonRDD.scala:53
```

```
54]  1 from pyspark import storagelevel
     2 ventas_sin_inpuestos.persist(StorageLevel.MEMORY_AND_DISK)
```

7. Spark SQL

a. Ver archivo “**SparkSQL.ipynb**”.

b. En Spark 1.6, se introdujo una nueva abstracción de programación llamada API estructurada.

c. En esta nueva forma de hacer el procesamiento de datos, los datos deben organizarse en un formato estructurado y la lógica de cálculo de datos debe seguir una determinada estructura.

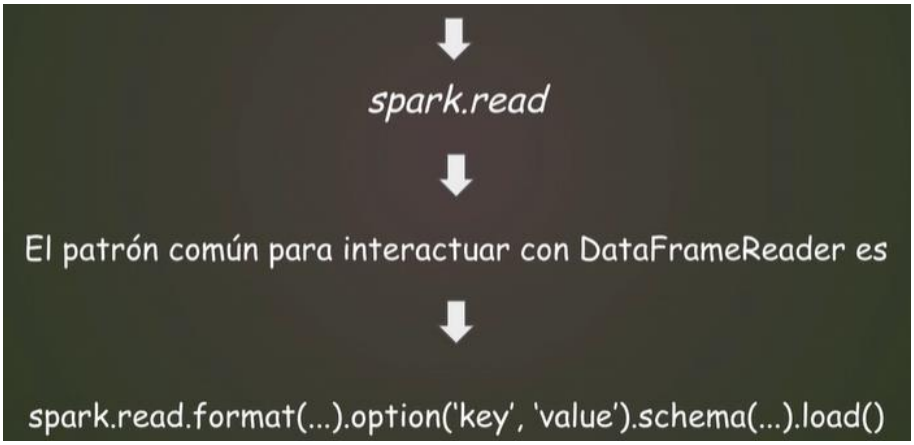
d. Con estas dos piezas de información, Spark puede realizar optimizaciones para acelerar las aplicaciones de procesamiento de datos.

e. Crear Dataframe a partir de un RDD.

f. Crear un DataFrame a partir de fuentes de datos.

i. Las 2 clases principales son **DataFrameReader** y **DataFrameWriter**.

ii. Una instancia de la clase DataFrameReader está disponible como read en la sesión de Spark.



iii. Elementos principales al leer datos.

Nombre	Opcional
format	No
option	Si
schema	Si

iv. Alternativa para leer datos.

1. Spark.read.json(“<path>”)
   Spark.read.format(“json”)
2. Spark.read.parquet(“<path”)
   Spark.read.format(“parquet”)
3. Spark.read.jdbc(“<path>”)
   Spark.read.format(“jdbc”)
4. Spark.read.orc(“<path>”)
   Spark.read.format(“orc”)

5. Spark.read.csv(“<path>”)
   Spark.read.format(“csv”)
6. Spark.read.text(“<path>”)
   Spark.read.format(“text”)

g. Crear un DataFrame a partir de fuentes de datos en la práctica.

h. Trabajar con columnas:

i. A diferencia de las operaciones con RDD, las operaciones estructuradas están diseñadas para ser más relacionales.
ii. Al igual que las operaciones con RDD, las operaciones estructuradas se dividen en dos categorías: transformación y acción.
iii. Los dataframes son inmutables y sus operaciones de transformación siempre devuelven un nuevo dataframe.

i. Transformaciones: funciones select y selectExpr: Seleccionar.

j. Transformaciones: Funciones filter y where: Filtra.

k. Transformaciones: funciones distinct y dropDuplicates: muestra los que nos son distintos.

l. Transformaciones: funciones sort y orderBy: Ordena.

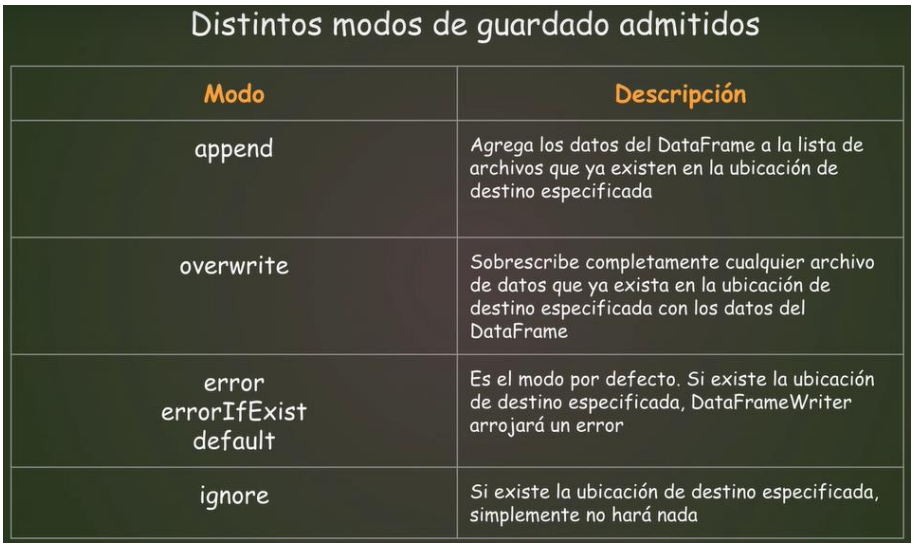
m. Transformaciones: funciones withColumn y withColumnRenamed: agrega una nueva columna y renombra columna.

n. Transformaciones: Funciones drop, sample y randomSplit: Elimina columnas, muestra cantidad de filas y divide los datos aleatorios.

o. Trabajar con datos incorrectos o faltantes.

p. Acciones sobre un DataFrame en Spark SQL.

q. Escritura de DataFrame:



r. Persistencia de DataFrames.

s. Ejercicios

8. Spark SQL Avanzado

a. Ver archivo "**SparkSQL_avanzado.ipynb**".

b. Agregaciones:

i. La realización de análisis interesantes sobre Big Data generalmente implica algún tipo de agregación para resumir los datos con el fin de extraer patrones o conocimientos o generar informes resumidos.

ii. Spark admite diferentes niveles de agrupación de filas:

1. Tratar un dataframe como un grupo.
2. Dividir un dataframe en varios grupos utilizando una o más columnas y realizar una o más agregaciones en cada uno de esos grupos.
3. Dividir un dataframe en varias ventanas y realizar una media móvil, una suma acumulativa o una clasificación.

iii. En spark, todas las agregaciones se realizan a través de funciones.

iv. Las funciones de agregación están diseñadas para realizar la agregación en un conjunto de filas, ya sea que ese conjunto de filas consista en todas las filas o en un subgrupo de filas en un dataframe.

c. Funciones count, countDistinct y aprox_count_distinct.

d. Función min y max.

e. Funciones sum, sumDistinct y avg.

f. Agregación con agrupación:

i. Las agregaciones generalmente se realizan en conjuntos de datos que contienen una o más columnas categóricas, que tienen una cardinalidad baja.

ii. Realizar agregación con agrupación es un proceso de dos pasos:

1. Paso 1: realizar la agrupación mediante la transformación groupBy(col1, col2,…), y es ahí donde se especifica por qué columnas agrupar las filas.
2. Paso 2: agregar las funciones de agregación deseadas.

g. Varias agregaciones por grupo.

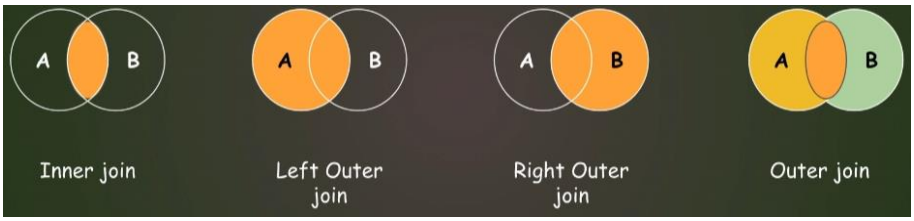
h. Agregación con pivote.

i. Join:

i. Realizar un join de dos conjuntos de datos requiere dos piezas de información:

1. Una expresión de join que especifica qué columnas de cada conjunto de datos deben usarse para determinar que filas de ambos conjuntos de datos se incluirán en el conjunto de datos combinado.
2. El tipo de join. Este que determina qué se debe incluir en el conjunto de datos combinado.

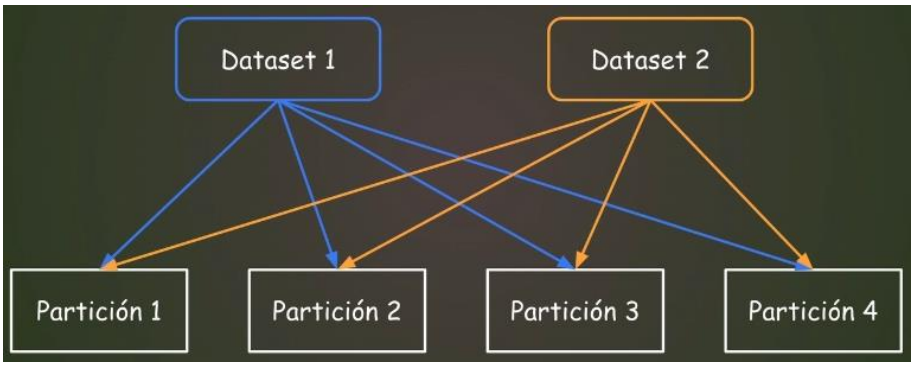
NOMBRE	DESCRIPCIÓN
Inner join	Devuelve filas de ambos conjuntos de datos cuando la expresión join se evalúa como verdadera.
Left outer join	Devuelve filas del conjunto de datos de la izquierda incluso cuando la expresión de join se evalúa como falsa.
Right outer join	Devuelve filas del conjunto de datos de la derecha incluso cuando la expresión de join se evalúa como falsa.
Outer join	Devuelve filas de ambos conjuntos de datos incluso cuando la expresión de join se evalúa como falsa.
Left Anti join	Devuelve filas solo del conjunto de datos de la izquierda cuando la expresión de join se evalúa como falsa.
Left Semi join	Devuelve filas solo del conjunto de datos de la izquierda cuando la expresión de join se evalúa como verdadero.
Cross	Devuelve filas combinando cada fila del conjunto de datos de la izquierda con cada fila del conjunto de la derecha.



j. Shuffle Hash Join:

i. El primer paso es calcular el valor hash de las columnas en la expresión de join de cada fila en cada conjunto de datos y luego mover esas filas con el mismo valor hash en la misma partición.

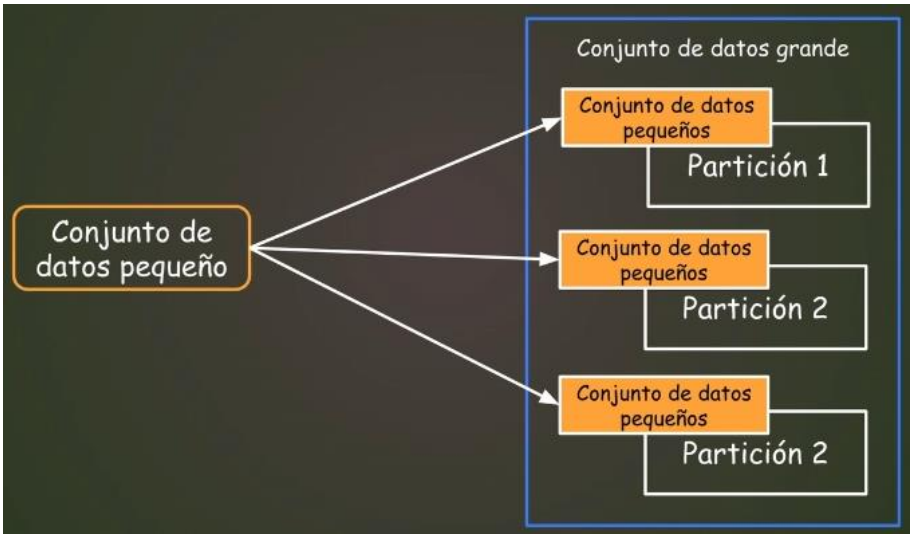
ii. El segundo paso combina las columnas de aquellas filas que tienen el mismo valor hash de columna.



k. Broadcast Hash Join:

i. El primero es transmitir una copia de todo el conjunto de datos más pequeño a cada una de las particiones del conjunto de datos más grande.

ii. El segundo paso es recorrer cada fila en el conjunto de datos más grande y busca las filas correspondientes en el conjunto de datos más pequeño con valores de columna coincidente.



9. Funciones en Spark SQL

- **a.** Ver archivo “**Spark_Funciones.ipynb**”.
- **b.** Funciones de fecha y hora.
- **c.** Funciones para trabajo con strings.
- **d.** Funciones para trabajo con colecciones.
- **e.** Funciones when, coalesce y lit.
- **f.** Funciones definidas por el usuario UDF.
- **g.** Funciones de ventana.
- **h.** Catalyst Optimizer.