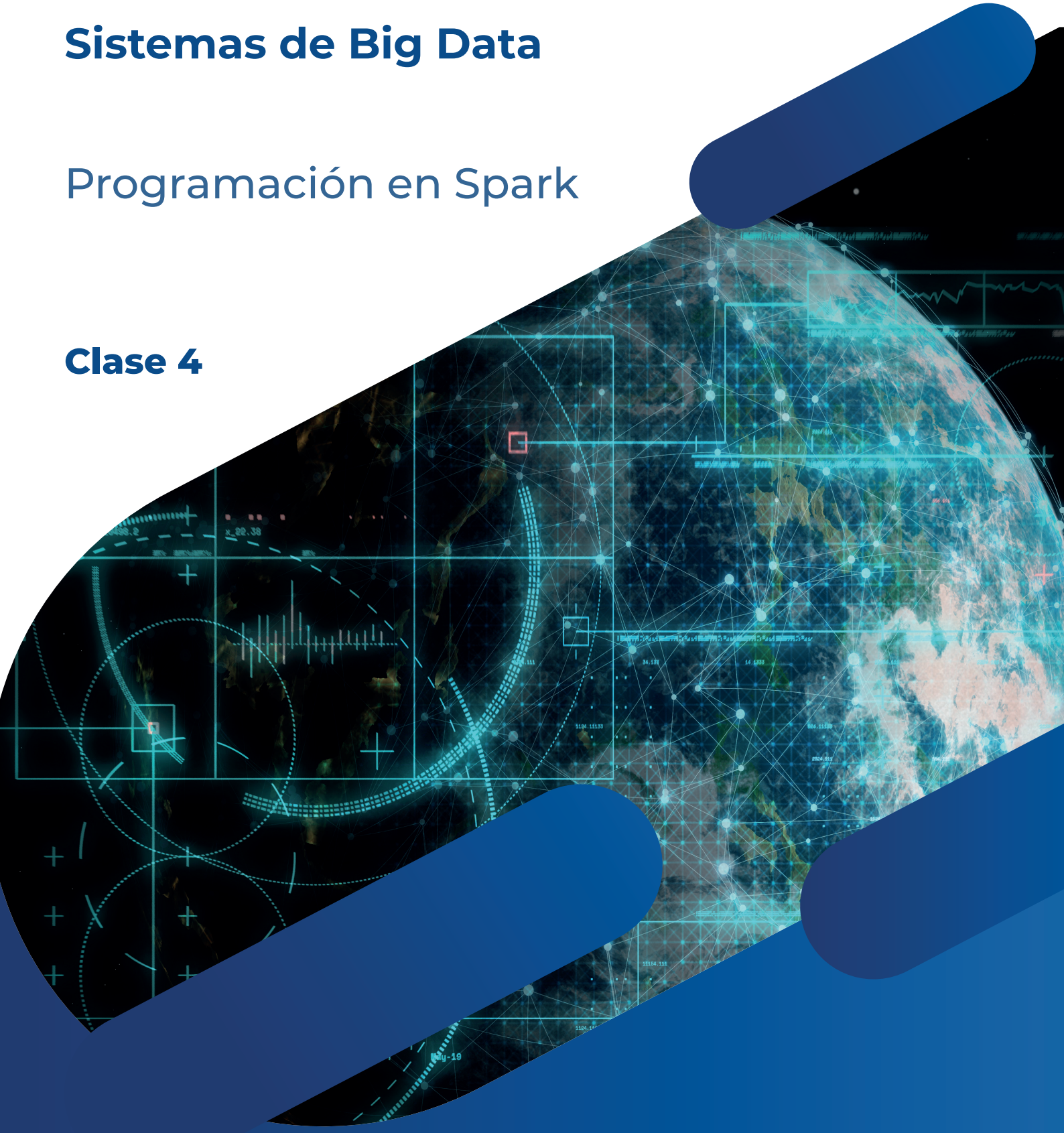
PUCE Virtual

Sistemas de Big Data

Programación en Spark

Clase 4



MAESTRÍA EN
SISTEMAS DE INFORMACIÓN
Mención Data Science

La excelencia no se improvisa

1. INTRODUCCIÓN DE LA CLASE

RDA 2: Aplicar técnicas de Big Data en grandes volúmenes de datos

Apache Spark es un motor de procesamiento distribuido diseñado para manejar grandes volúmenes de datos de manera eficiente y escalable. Su arquitectura permite la ejecución de tareas en memoria, reduciendo significativamente los tiempos de procesamiento en comparación con tecnologías tradicionales, como **Hadoop MapReduce**. Entre sus aplicaciones fundamentales se encuentra el algoritmo **WordCount**, el cual ilustra el uso de **Resilient Distributed Datasets (RDD)** para dividir, mapear y reducir datos en un clúster distribuido, demostrando la capacidad de Spark para realizar análisis de texto a gran escala de manera eficiente (Zaharia et al., 2016).

El ecosistema de Apache Spark se compone de módulos especializados que amplían sus capacidades de análisis y procesamiento de datos. Entre ellos se destacan **Spark SQL**, que permite consultas estructuradas mediante lenguaje **SQL**; **MLlib**, utilizado para aprendizaje automático escalable; **GraphX**, diseñado para el procesamiento de grafos; y **Spark Streaming**, que facilita el procesamiento de flujos de datos en tiempo real mediante la técnica de **microbatching**. Gracias a su compatibilidad con lenguajes como **Scala, Python, Java y R**, **Spark** se ha convertido en una herramienta esencial para la programación en el ámbito del Big Data, permitiendo la manipulación eficiente de datos estructurados y no estructurados (Karau et al., 2017).

Clase 4. Programación en Spark
4.1. Ejercicios con Spark

La clase anterior aplicamos el algoritmo WordCount utilizando MapReduce, en esta sección aplicaremos WordCount utilizando la API de Spark para Python llamada Pyspark.

Paso inicial, **arrancar** la máquina virtual.

Siguiente paso, **descargamos** el archivo *words.txt* desde la siguiente ruta.

Abrir la **terminal** e ir a la carpeta Downloads y localizar el archivo descargado usando los siguientes comandos:

- ***cd Downloads***
- ***ls***

Formatea el namenode con el siguiente comando ***hdfs namenode -format.***

Estar seguro de que **HDFS y YARN** esté levantado en la máquina virtual con el comando **jps,** si los servicios no están levantados, levantarlos utilizando los comandos que se muestran en la siguiente **figura**. Nota: YARN no es necesario para este ejercicio; sin embargo, por motivos ilustrativos lo levantamos.

```
u631013@T0003352549:~/Downloads$ jps
27150 Jps
u631013@T0003352549:~/Downloads$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [T0003352549]
u631013@T0003352549:~/Downloads$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
u631013@T0003352549:~/Downloads$ jps
27490 DataNode
29365 Jps
27735 SecondaryNameNode
28601 ResourceManager
28763 NodeManager
u631013@T0003352549:~/Downloads$
```

Verificar si tenemos el archivo *words.txt* ya cargado en **HDFS;** de lo contrario, subirlo. Lo puedes hacer por línea de comandos o por la interfaz gráfica. Los comandos son los siguientes:

- ***hdfs dfs -mkdir /user***
- ***hdfs dfs -mkdir /users/bigdata***
- ***hdfs dfs -copyFromLocal words.txt /users/bigdata***
- ***hdfs dfs -ls /users/bigdata***

Deberías tener una salida similar a esta:

```
Found 1 items
-rw-r--r--   1 u631013 supergroup    5458199 2025-03-06 18:41 /users/bigdata/words.txt
```

Una vez hayas **subido** el archivo a **HDFS,** inicia una sesión de Spark con la API de Spark para Python; **usa el comando *pyspark.***

```
Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
25/03/06 18:51:11 WARN Utils: Your hostname, T0003352549 resolves to a loopback address: 127
.0.1.1; using 192.168.193.48 instead (on interface wlo1)
25/03/06 18:51:11 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/03/06 18:51:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your platf
orm... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.5
      /_/

Using Python version 3.10.12 (main, Feb  4 2025 14:57:36)
Spark context Web UI available at http://192.168.193.48:4040
Spark context available as 'sc' (master = local[*], app id = local-1741297876028).
SparkSession available as 'spark'.
```

Dentro de la sesión de **Spark,** realiza los siguientes pasos:

Paso 1. Lee el archivo de Shakespeare **desde HDFS**. Podemos leer el archivo en HDFS en una variable llamada en este caso *lines*.

lines = sc.textFile("hdfs:/users/bigdata/words.txt")

El **SparkContext**, sc, es el principal punto de acceso a **Spark** con Python. El método textFile() lee el archivo en un Resilient Distributed Dataset (**RDD**), en donde cada línea en el archivo empieza un elemento en la colección de **RDD**. La URL *hdfs:/users/bigdata/words.txt* indica la ubicación del archivo en HDFS.

Podemos verificar que el archivo fue cargado exitosamente llamando al método count(), el mismo que imprime el número de elementos del RDD.

lines.count() ==> 124456

Paso 2. Divide cada línea en palabras. Ahora, dividiremos cada línea en un conjunto de palabras. Para dividir cada línea en palabras y guardarlas en un **RDD** llamado ***words*** ejecutamos lo siguiente:

words = lines.flatMap(lambda linea : linea.split(" "))

El método **flatMap**() itera sobre cada línea del RDD y ***lambda*** *linea : linea.split*(" ") es ejecutado en cada línea. La notación *lambda* es una función **anónima** en Python; por ejemplo, una función definida sin usar un nombre. En este caso, la función anónima toma un único **argumento**, *line,* y llama a **split**(" "), el mismo que divide la línea en un *array* de palabras.

Paso 3. Asignamos un valor inicial al contador de cada palabra. Ahora crearemos tuplas por cada palabra, con un valor inicial de contador de 1.

Nota: en el paso previo usamos la función **flatMap**, pero en este caso usamos la función **map.** En este paso queremos crear una tupla para cada palabra; por ejemplo, tenemos un mapeo de uno a uno entre las palabras de entrada y las tuplas de salida. En el paso previo, nosotros queríamos dividir cada línea en un conjunto de palabras. Hay un mapeo de uno a muchos entre las líneas de entrada y las palabras de salida. En general, se usa **map** cuando el número de *inputs* y el número de *outputs* tienen una relación uno a uno; y la función **flatMap,** para cuando tenemos una relación de uno a muchos.

tuples = words.map(lambda word : (word, 1))

Paso 4. Sumar todos los valores obtenidos por cada palabra: Podemos sumar todos los *counts* en las tuplas por cada palabra y lo guardamos en un nuevo RDD llamado *count.*

count = tuples.reduceByKey(lambda a,b : (a+b))

El método **reduceByKey**() llama a la expresión ***lambda*** para todas las tuplas con el mismo nombre. La expresión *lambda* lleva dos argumentos, a y b los cuales **representan** los valores a contar en cada tupla.

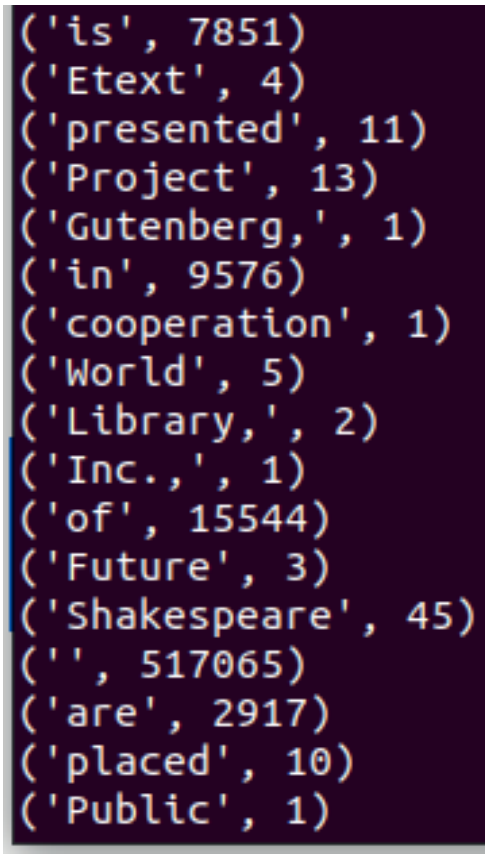
Paso 5. Escribir el contador de palabras en un archivo de **HDFS**. Podemos escribir el **RDD** *count* en **HDFS** usando el siguiente código:

count.coalesce(1).saveAsTextFile("hdfs:/users/bigdata/wordcount/outputDir")

El método ***coalesce()*** combina todas las particiones del RDD en una única partición dado que nosotros queremos un único archivo de salida y el método ***saveAsTextFile()*** escribe el **RDD** en una ubicación específica.

Paso 6. Visualizar los resultados. Podemos ver los resultados **copiando** el archivo de **HDFS** al sistema local de archivos y después ejecutar el comando ***more.*** Tal y como se indica a continuación:

- *hdfs dfs -copyToLocal /users/bigdata/wordcount/outputDir/part-00000 count.txt*
- *more count.txt*



Introducción a *Spark* [https://www.bigdata.uma.es/apache-spark-introduccion-que-es-y-como-funciona/]

4.2. Spark Core: programación en Spark

Si bien el ejercicio de **WordCount** es sencillo, ayuda a entender cómo trabajar con ***RDD.*** En esta sección se **explican**:

- Dos métodos para crear **RDD** en Spark.
- Qué significa que un objeto sea **inmutable.**
- Interpretar un programa de **Spark** como un *pipeline* de transformaciones y acciones.
- Listar cada uno de los pasos para crear un programa en ***S*park.**

Recordando el funcionamiento de **Spark**, tenemos un **Driver Program** que define el **Spark Context;** este es el punto de inicio de tu aplicación. Este **Driver** convierte todos los datos a RDD y todo lo que sigue a partir de este punto se maneja usando *RDD.* Los *RDD* pueden ser creados a base de archivos o de algún otro tipo de almacenamiento. También pueden ser construidos por estructuras de datos, colecciones o programas.

Todas las transacciones y acciones sobre estos **RDD** tienen lugar tanto localmente como o en los **Worker Nodes** manejados por un **Cluster Manager**. Cada una de estas transformaciones resulta en una nueva actualización de la versión del RDD. Al final, los RDD son convertidos y almacenados en un almacenamiento persistente como HDFS o algún directorio local.

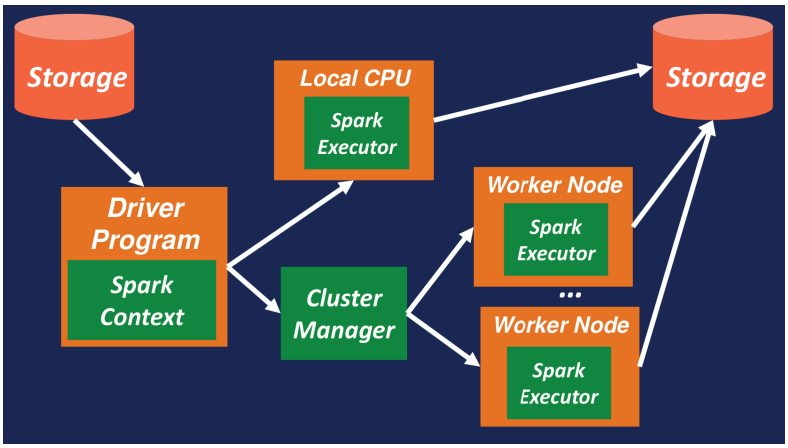


Figura N.º 1.**Creación de los RDD**

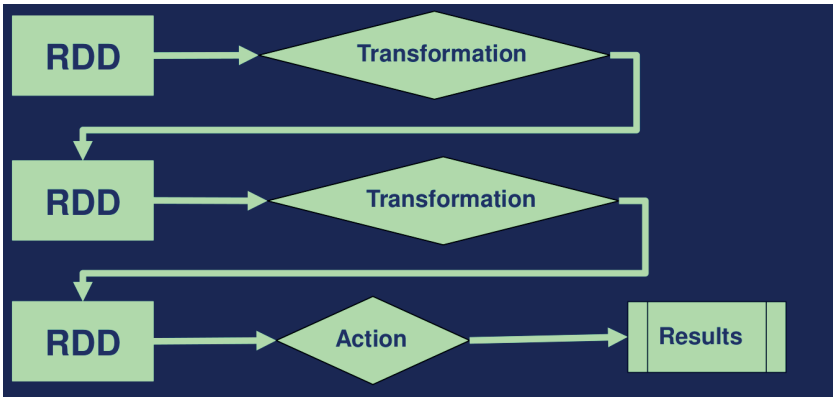
Los RDD son creados en el Driver Program. El desarrollador del Driver Program (en este caso tú) es el responsable de crearlos. Se los puede crear de dos maneras: leer un archivo a través del Spark Context; o, tal como se muestra en el siguiente ejemplo, se
puede crear a partir de una colección existente, como una lista que se convierte en una **colección distribuida,** como se indica en la imagen.
La función ***parallelize*** permite definir el número de particiones para distribuir los datos. En el caso anterior, se usa la función ***range*** para generar números del 0 a 9 y distribuirlos en tres particiones. Spark decidirá cómo asignar las particiones a cada uno de los ***executors*** y ***worker nodes.*** Los ***RDD*** pueden ser finalmente unificados en una única partición en el *driver,* usando la transformación ***collect.***

```
lines = sc.textFile("hdfs:/users/bigdata/words.txt")
```

```
lines = sc.parallelize(["big", "data"])
```

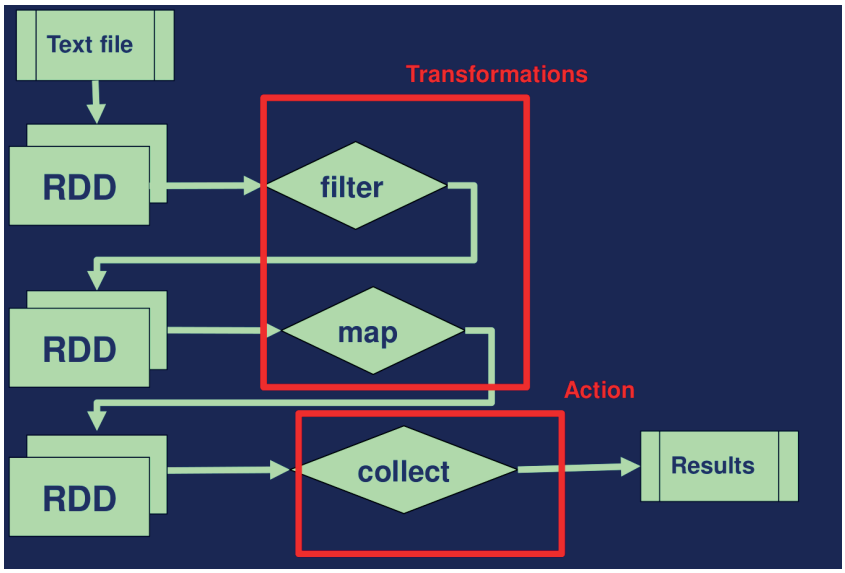
Procesamiento de *RDD:*

En este escenario vamos a empezar a procesar los ***RDD***, hay dos tipos de operaciones que sirven para procesar datos en **Spark,** las transformaciones y las acciones. Todas las particiones escritas en un **RDD** pasan a través de la misma transformación en el ***worker node*** cuando una transformación es aplicada a un RDD. **Spark** usa ***lazy evaluation*** para transformaciones, lo que significa que no son ejecutadas inmediatamente, sino que esperan una acción para ser ejecutada. Las transformaciones son computadas cuando la acción es **ejecutada**. Por esta razón, en muchos casos puedes ver errores que se muestran cuando el *pipeline* llega a la etapa de acción y no en las transformaciones.



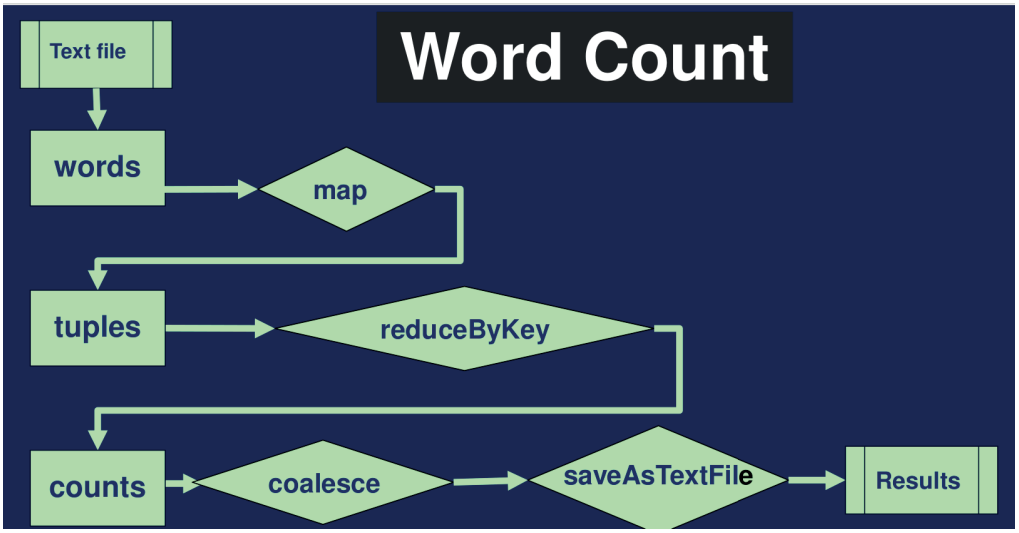
En el siguiente ejemplo tenemos un *pipeline* que convierte un archivo de texto en un **RDD** con dos particiones. Luego filtra algunos de los datos y luego les aplica una función ***map***. Al final se ejecuta la acción

collect sobre los **RDD** mapeados, con el objetivo de evaluar los resultados del *pipeline* y convertir las salidas en resultados. En el ejemplo, ***filter*** y ***map*** son transformaciones y ***collect*** es una acción.



Aunque los ***RDD*** se encuentran en memoria y no están persistidos en disco, se puede usar la función ***cash*** para persistirlos. Por ejemplo, cuando se precisa reusar el ***RDD*** creado a partir de una base de datos, que de otra forma sería costoso reejecutarlo, en lugar de eso podemos guardar los ***RDD*** en caché.

Retomando el ejemplo de **WordCount** primero que nada mapeamos el **RDD** de *words* para generar tuplas, luego aplicamos ***reduceByKey*** a las tuplas para generar *counts*. Al final, convertimos el número de particiones a una de tal forma que la salida es un solo archivo que se va a escribir en disco después, caso contrario, la salida va a estar regada en múltiples archivos del disco. Finalmente, ***saveAsTextFile*** es una acción que empieza la computación y escribe a disco



4.3. Spark Core: Transformaciones

Recordemos que los ***RDD*** son **inmutables**, esto quiere decir que no pueden ser cambiados por sí mismos. Se aplica una función de transformación sobre ellos para convertirlos en un nuevo RDD.

Este concepto es **esencial** para que **Spark** mantenga un *traqueo* de todas las operaciones que se realizaron sobre ese *pipeline*. Adicionalmente, como parte de un *pipeline* de Big Data, se empieza con un *RDD* y luego de realizan **múltiples transformaciones;** lo que genera ***RDD*** intermedios hasta tener un resultado final. También hay que considerar que todas estas transformaciones son *lazy* en Spark, lo que significa que estas operaciones **no son ejecutadas inmediatamente cuando son aplicadas a un *RDD***, así que cuando se aplica una transformación no pasa nada en ese momento.

Básicamente estamos preparando el *pipeline* de **Big Data** para que sea ejecutado después. Cuando se hayan definido todas las transformaciones y se ejecute una acción **Spark** buscará la mejor manera de ejecutar la computación asociada y empezar las tareas necesarias en los ***worker nodes***.

En esta sección se **explicará**:

- La diferencia entre *narrow transformation* y *wide transformation.*
- ***Map***, ***flatmap***, ***filter*** **y** ***coalesce*** como operaciones *narrow.*
- Al menos dos transformaciones.

Map: aplica una función a cada elemento del ***RDD.*** Esta es una transformación de ***uno a uno,*** también entra en la categoría de ***element-wise transformation,*** dado que transforma cada uno de los elementos del ***RDD*** de forma separada.

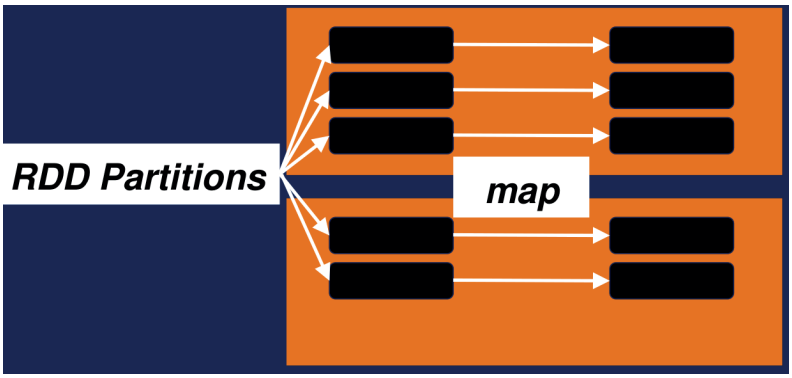


Figura N.º 5

En el siguiente ejemplo se define cómo se aplica la función llamada ***lower*** a todos los elementos del ***text_RDD***. La función ***lower*** convierte todos los caracteres de una línea en minúsculas. Entonces el *input* es una línea de texto y el *output* será la misma línea pero todo con minúsculas. En el ejemplo tenemos dos ***worker nodes*** (ver Figura N.º 5) representados por los **cuadros anaranjados** y los cuadros negros representan particiones del *dataset*. **Spark** trabaja por partición y no por elemento, esa es la diferencia principal entre **Spark** y **MapReduce**.

```
def lower(line):
    return line.lower()

lower_text_RDD = text_RDD.map(lower)
```

Figura N.º 6

La partición es solo una parte de nuestro ***dataset,*** con algunos elementos en él. La función ***map*** aplica la función a todos los elementos de esa partición en cada ***worker node***. Cada nodo aplica la función ***map*** a los datos.

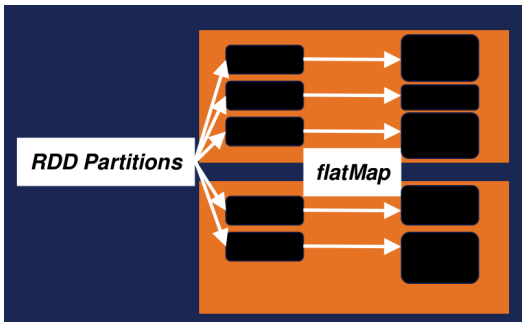
FlatMap: es similar a ***map***. Sin embargo, en lugar de devolver un elemento individual por cada mapeo, devuelve un ***RDD*** un promedio de todos los resultados para todos los elementos.

En el siguiente ejemplo, la función ***split_words*** toma una línea como *input*, que en este caso es un ele-

mento y su salida es cada palabra como un único elemento, en otras palabras, **divide** una línea en palabras, se hace lo mismo para cada línea.

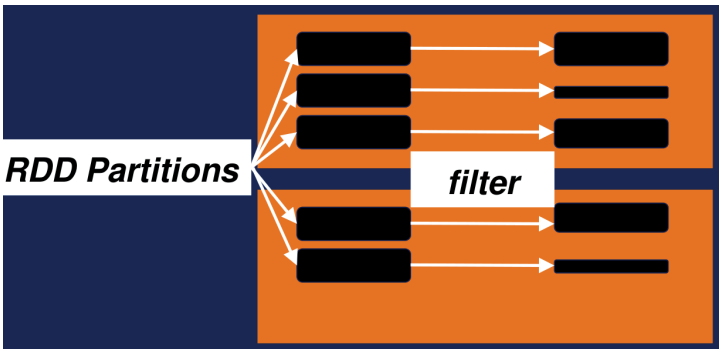
```
def split_words(line):
    return line.split()

words_RDD = text_RDD.flatMap(split_words)
words_RDD.collect()
```

Cuando la salida para todas las líneas es ***flattened***, tenemos una lista de palabras de una sola dimensión. Por lo tanto, tendremos todas las palabras de todas las líneas en una sola lista. **Dependiendo** de la longitud de la **línea** las particiones de salida **podrían** ser de diferente tamaño, esto se ve claramente en la siguiente imagen, la altura de los cuadros negros varía en función de en **función** de la longitud de las líneas. Para **Spark *map*** y ***flatMap*** son trasformaciones de tipo ***narrow***.



Las transformaciones ***narrow*** se refieren a que la lógica del procesamiento depende solo de que ya se encuentran en la partición y el ***data shuffling*** no es necesario.

Filter: conserva únicamente los **elementos** en donde la condición es verdadera. La **transformación *filter*** se ejecuta en cada elemento de la partición del ***RDD*** y devuelve solo los elementos que la transformación retorna como **positivos**.



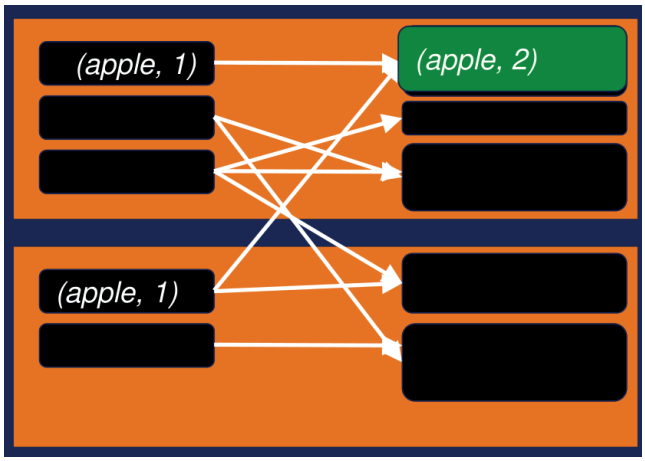
En el siguiente ejemplo, se aplica una función ***filter*** en las palabras que empiezan con la letra a. La **función** transforma el texto a minúsculas y se queda con los textos que empiezan con la letra 'a'. Finalmente, el *output* de la función será una lista con solo las palabras que empiezan con la letra 'a'. Esta también es una transformación ***narrow*** y solo se ejecuta de manera local, sin necesidad de ***shuffle*** *ninguna partición de RDD a través de los* ***worker nodes***. La salida de ***filter*** depende de los ***inputs*** y de la función de filtrado.

```
def starts_with_a(word):
    return word.lower().startswith("a")
words_RDD.filter(starts_with_a).collect()
```

Coalesce: permite **balancear** el número de **particiones**. Cuando se necesita reducir **significativamente** los datos iniciales después de realizar algún **filtrado** y otras **transformaciones,** tener un gran número de particiones podría ya **no ser necesario**. En este caso, se puede usar ***coalesce*** para reducir el número de particiones a una cantidad más **manejable**.

Wide Transformations

Una función que se usó en el ejemplo de **WordCount** es la función ***reduceByKey,*** la misma que permite combinar los valores usando la función ***reduce,*** que en el ejemplo de **WordCount** es una simple *sumarización*. En las transformaciones ***groupByKey*** y ***reduceByKey*** es necesario hacer un ***shuffle*** de los datos a través de los ***worker nodes***. A ese tipo de transformaciones las llamamos ***wide transformations***. En este tipo de transformaciones, el procesamiento depende de los datos que están en múltiples particiones distribuidas entre todos los ***worker nodes*** y esto requiere ***data*** sobre toda la red para obtener todos los *datasets* que se encuentran relacionados. En el siguiente gráfico se muestra el funcionamiento de la función ***reduceByKey.***



Transformaciones en Spark [https://spark.apache.org/docs/1.2.0/programming-guide.html#transformations]

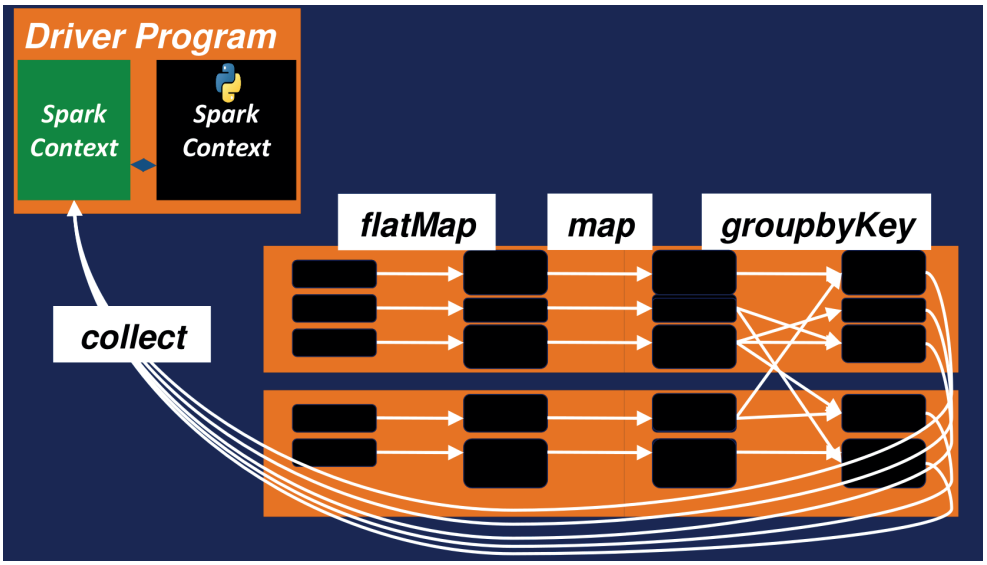
4.4. Spark Core: Acciones

En breves rasgos, se puede definir a las acciones como operaciones sobre los **RDD** que disparan la evaluación en el *pipeline* de transformaciones y devuelven el resultado final al ***driver program*** o guarda el resultado final en un esquema de almacenamiento persistente. También se lo puede definir como el último paso en el *pipeline* de **Spark**.

Los objetivos de esta sección se dividen en dos:

- **Poder** explicar los pasos de un *pipeline* de **Spark** que termina con una acción de *collect.*
- **Listar al menos** cuatro acciones comunes en **Spark.**

Para este ejemplo podemos imaginar que hemos **leído** un archivo creado por el **RDD** correspondiente en **Spark**, luego las particiones de ese RDD se mueven a través de los pasos de transformación en el *pipeline* que se muestra a continuación, definido por ***flatMap***, ***map*** o ***groupByKey***.



Cuando el paso final termina la acción ***collect*** es ejecutada y **Spark** manda todas las tareas para que sean ejecutadas por los **worker nodes**. La acción ***collect*** enviará todos los **RDD** resultantes desde los **workers** y los copiará a la Java Virtual Machine en el **driver program**. Finalmente, esto será añadido al Shell de **Python**. Mientras la acción ***collect*** copia todos los datos, la acción ***take*** copia los primeros 'n' resultados del driver.

Si los resultados son demasiado grandes para entrar en la memoria del **driver,** entonces hay una oportunidad para escribirlos directamente a **HDFS** en su lugar. Otra acción muy frecuente es la acción ***reduce;*** que toma dos elementos y retorna un resultado, como por ejemplo la suma de los **elementos**.

Pero en este caso no tenemos una **key**, solo tenemos una gran área de algunos valores y corremos esta función una y otra vez hasta reducir todo a un solo valor. Por ejemplo, la suma global de todo. Otra acción muy usada es ***saveAsText*** para guardar los resultados en el disco local o a **HDFS** y es muy útil cuando queremos guardar algo que se **procesó** antes y que es muy pesado volver a **generarlo.**

Algunas de las acciones más conocidas se listan en la siguiente tabla:

Acción	**Uso**
Collect()	Copia todos los elementos al driver.
Take(n)	Copia los primeros 'n' elementos.
Reduce(func)	Agrega elementos con func (toma 2 elementos y retorna 1).
saveAsTextFile(filename)	Guarda en un archivo local o en HDFS.

Referencias citadas en la Clase 4

- Apache Spark. (s.f.). *reduceByKey*. Recuperado el 7 de marzo de 2025, de https://spark.apache.org
- Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2017). *Learning Spark: Lightning-fast data analytics*. O'Reilly Media.
- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCaulay, M., ... & Stoica, I. (2016). *Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing*. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, 15(1), 1-14.

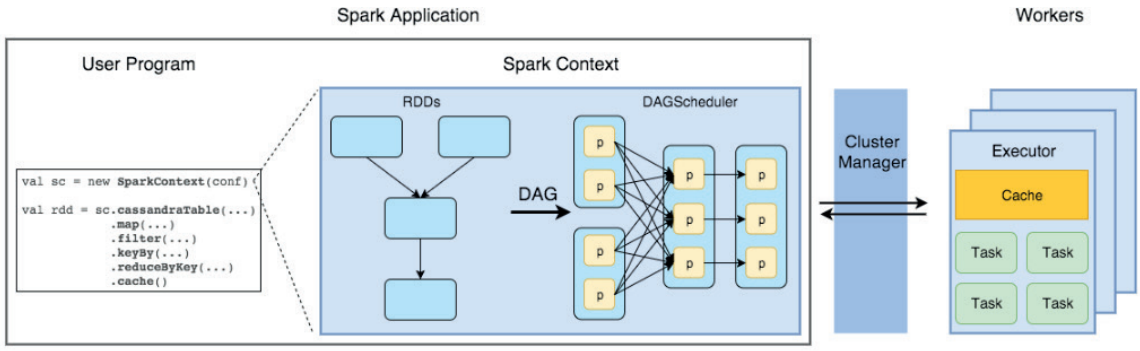
Definición de los términos citados en la Clase 4

Lazy Evaluation (evaluación perezosa). Estrategia de evaluación utilizada en algunos lenguajes de programación en la que una expresión no se evalúa inmediatamente cuando se define, sino hasta que su valor es realmente necesario. Esto permite **optimizar el uso de recursos,** evitar cálculos innecesarios y trabajar con estructuras de datos **potencialmente infinitas**. Es común en **lenguajes funcionales,** como **Haskell,** donde las expresiones se resuelven solo cuando se requieren en la ejecución del programa, lo que puede mejorar la eficiencia y reducir el **consumo de memoria** en ciertos casos.

ReduceByKey. Transformación en **Apache Spark utilizada en RDD (Resilient Distributed Datasets),** para agrupar valores por clave y aplicar una función de reducción sobre ellos. Esta operación es especialmente útil en el procesamiento distribuido de grandes volúmenes de datos, ya que combina los valores asociados a una misma clave en paralelo, optimizando el rendimiento al reducir la cantidad de datos transferidos entre nodos. Se usa comúnmente en escenarios como el conteo de palabras o la agregación de datos en **operaciones de análisis**.

Profundización Clase 4

El siguiente diagrama muestra cómo funciona una aplicación de Spark que consiste en el Spark Context con código definido por el usuario, la creación de RDD realizando transformaciones y obteniendo un resultado final.



El siguiente diagrama muestra cómo se cargan las particiones de un archivo de HDFS a particiones de tipo RDD en Spark.

PUCE Virtual

La excelencia no se improvisa

síguenos