Introducción al Almacenamiento Distribuido en RAM

Juan Mario Haut

Limitaciones de Hadoop

- Bajo Rendimiento en qué tareas?.
- Procesamiento en Disco duro?
- Flexibilidad?.
- Tiempo real?

Juan Mario Haut

Limitaciones de Hadoop

- Bajo Rendimiento para tareas iterativas.
- Necesidad de procesamiento en memoria RAM.
- Flexibilidad: Necesitamos hacer más cosas que map y reduce.
- Procesamiento en tiempo real.

SISTEMA DE FICHEROS

Apache Spark se originó en el AMPLab de la Universidad de California, Berkeley, en 2009, y se lanzó como un proyecto de código abierto en 2010. Se unió oficialmente a la Apache Software Foundation en 2013. Los investigadores del AMPLab crearon Spark para mejorar el rendimiento y las capacidades de Hadoop, y su diseño fue pensado para aprovechar las limitaciones de Hadoop MapReduce en la velocidad y eficiencia del procesamiento de datos grandes.

Juan Mario Haut

RDD

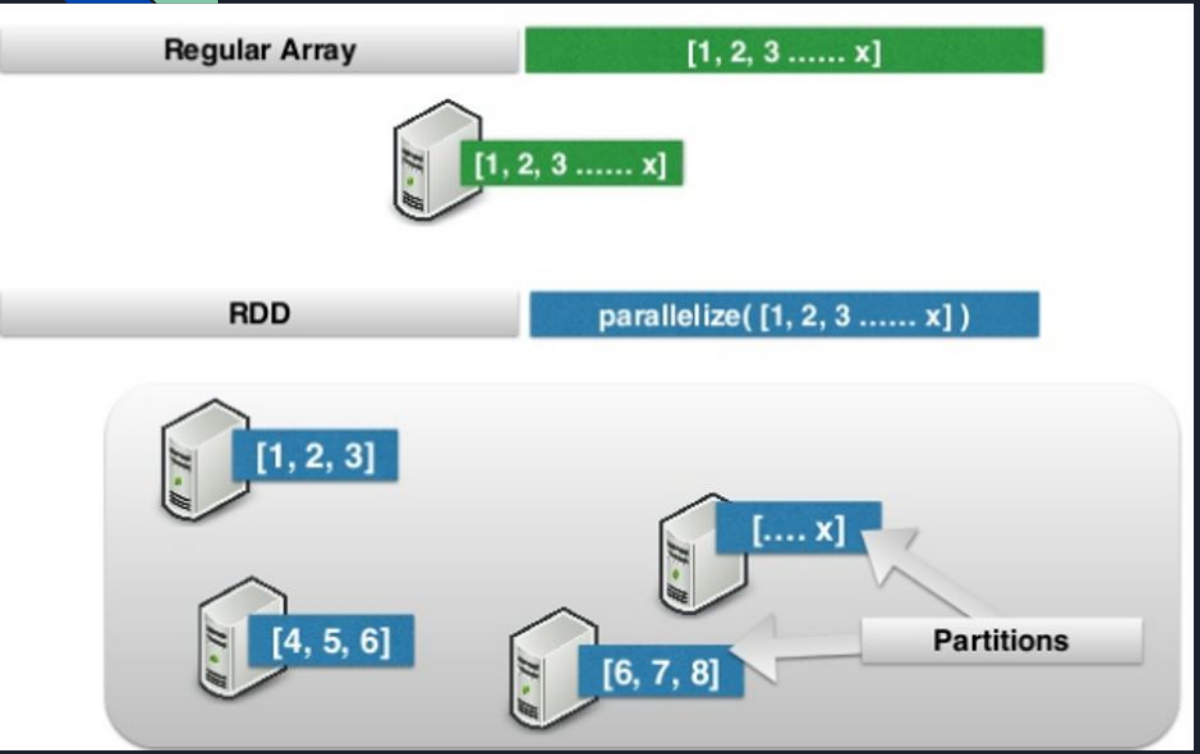
Un RDD (Resilient Distributed Dataset) es la abstracción fundamental en Apache Spark para manejar grandes volúmenes de datos de manera distribuida y resiliente. Un RDD representa un conjunto **inmutable** de datos distribuidos que pueden procesarse en paralelo a través de un clúster de computadoras. Esta estructura permite a Spark trabajar eficientemente con datos distribuidos y tolerar fallos, características esenciales para el procesamiento a gran escala.

Caraterísticas:
1.- Inmutabilidad: Una vez que el RDD es creado, no puede ser modificado.

2.- Distribución: Los RDDs se distribuyen a lo largo del cluster.

3.- Resiliencia: Los RDDs son tolerantes a fallo.

RDD

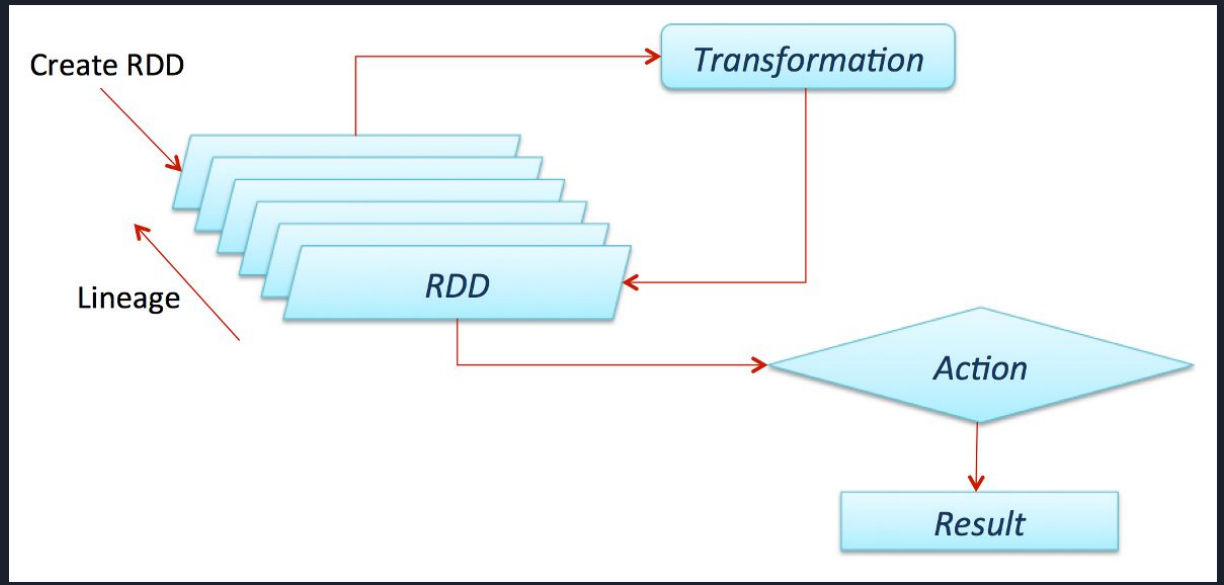



```
val  datosArchivo =
       sc.textFile("/Path/toFile")
```

RDD

Sobre los RDDs se realizan operaciones:

- Transformaciones: Realizan operaciones en el RDD
- Acciones: Da como resultado un dato.



RDD: Transformaciones.

Las transformaciones son operaciones que toman un RDD y devuelven un nuevo RDD como resultado. Estas operaciones son perezosas (o *lazy*), lo que significa que no se ejecutan inmediatamente cuando se definen, sino que Spark las registra internamente para ser ejecutadas más tarde. Esto permite a Spark optimizar el flujo de trabajo al agrupar múltiples transformaciones y ejecutarlas de manera más eficiente cuando finalmente se realiza una acción.

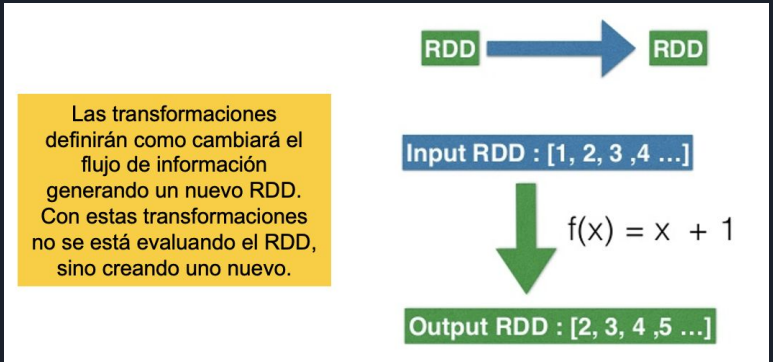
Al ser perezosas, las transformaciones no disparan el procesamiento inmediato, sino que crean un plan de ejecución o linaje (DAG, Directed Acyclic Graph), que representa la secuencia de operaciones a aplicar. Cuando se invoca una acción, Spark revisa todo este linaje y optimiza el cálculo, minimizando la cantidad de operaciones que realmente se ejecutan y eliminando las redundancias.

Ejemplo: rdd.map(lambda x: x+1)

Juan Mario Haut

Sistema de archivo distribuido: RDD

- Dan como resultado un nuevo RDD.
- Son ejecutados de forma *lazy*: ejecutadas sólo cuando son utilizadas en una acción.
- Muchas transformaciones son *element wise*, esto es, trabajan sobre un elemento a la vez.



Juan Mario Haut

Transformaciones

Juan Mario Haut

Transformaciones: MAP

La operación map() aplica una función a cada elemento de la colección (o RDD en el caso de Spark), transformando cada elemento de manera individual y devolviendo un nuevo conjunto con los resultados. Características principales:

- Elemento por elemento: La operación se aplica de forma independiente a cada elemento del RDD, por lo que no afecta la estructura general del conjunto.
- Transformación uno a uno: Para cada elemento en el RDD de entrada, el map() genera exactamente un elemento en el RDD de salida.
- Paralelismo: En Spark, map() es ejecutado de forma paralela en todas las particiones del RDD, lo que permite que se aplique la función a grandes volúmenes de datos de manera eficiente.

```
1. rdd = sc.parallelize([1, 2, 3, 4, 5])
2. rdd_mapped = rdd.map(lambda x: x * 2)
3. print(rdd_mapped.collect())
```

-

Juan Mario Haut

Transformaciones: FLAPMAP

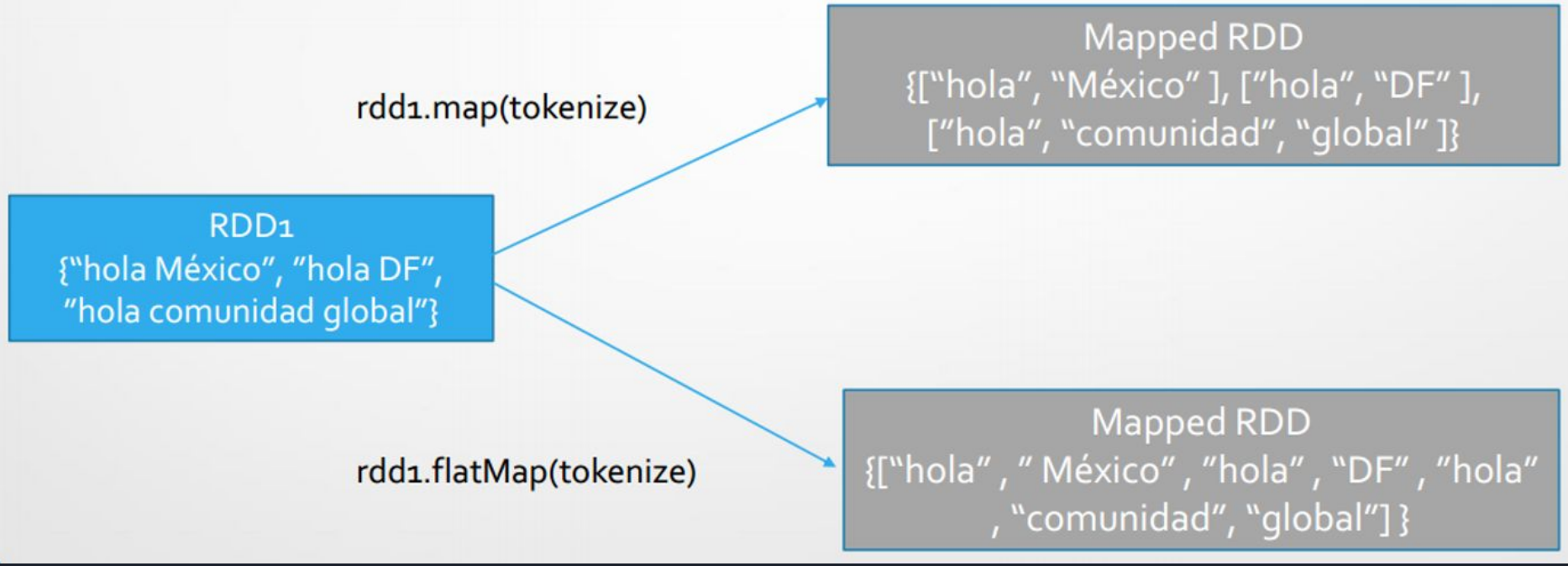
- La operación flatMap() es similar a map(), pero con una diferencia clave: mientras que map() transforma cada elemento de entrada en un único elemento de salida, flatMap() permite que cada elemento de entrada se transforme en una secuencia (cero, uno o más) de elementos de salida. Una vez que se aplica la función, Spark "aplana" las secuencias resultantes en una sola estructura de datos, creando un conjunto de salida sin niveles adicionales de anidación.

Características principales:

- Transformación uno a muchos: A diferencia de map(), que produce un elemento de salida por cada elemento de entrada, flatMap() puede producir múltiples elementos por cada entrada. Esto es útil cuando cada elemento en la colección de entrada debe expandirse en varios elementos de salida.
- Aplanado de secuencias: Después de aplicar la función a cada elemento de la colección, los resultados, que podrían ser listas u otras colecciones, se "aplanan" (flatten) en una única secuencia, eliminando los niveles de anidación.
- Paralelismo: Igual que map(), flatMap() es ejecutado de manera paralela sobre las particiones de un RDD, lo que asegura un procesamiento distribuido eficiente.

Juan Mario Haut

Transformaciones: FLAPMAP



Transformaciones: SAMPLE

La operación sample() en Apache Spark permite tomar una muestra aleatoria de los datos de un RDD. Esta operación selecciona una fracción de los elementos de forma aleatoria, lo que resulta útil para trabajar con grandes volúmenes de datos cuando solo se necesita procesar o analizar una parte representativa.

Características principales:

- Muestreo aleatorio: sample() selecciona elementos aleatorios del RDD utilizando una semilla generadora de números aleatorios, lo que garantiza que la muestra obtenida sea representativa.
- Control sobre el tamaño de la muestra: Permite definir qué proporción de los datos originales se desea incluir en la muestra, lo que se indica mediante un valor fraccionario (entre 0 y 1).
- Posibilidad de muestreo con o sin reemplazo: Se puede especificar si los elementos seleccionados se devuelven al conjunto original y pueden ser seleccionados nuevamente (con reemplazo) o si se seleccionan sin reemplazo (es decir, cada elemento se selecciona solo una vez).
- Reproducibilidad: Se puede usar una semilla específica para que el muestreo sea determinístico, es decir, para que al ejecutar el proceso varias veces con la misma semilla, se obtenga la misma muestra de datos.

Juan Mario Haut

Transformaciones: SAMPLE

sampled_rdd = rdd.sample(withReplacement=False, fraction=0.4, seed=42)

withReplacement →Indica si el elemento se puede seleccionar más de una vez o no.

fraction → Un valor de punto flotante que representa la fracción del RDD original que se debe muestrear

seed → Semilla aleatoria

Juan Mario Haut

Transformaciones: FILTER

La transformación filter se utiliza para filtrar un conjunto de datos, de modo que solo se conserven aquellos elementos que cumplen con una condición dada. Esta transformación es esencial para realizar la limpieza y el preprocesamiento de datos, ya que permite eliminar datos no deseados o irrelevantes antes de realizar análisis más profundos o entrenar modelos de machine learning.

Se define una condición que los elementos deben cumplir para ser incluidos en el resultado. Esta condición puede ser cualquier expresión que evalúe a True o False.

```
numbers = sc.parallelize(range(1, 11))

even_numbers = numbers.filter(lambda x: x % 2 == 0)

print("Números pares:", even_numbers.collect())
```

Juan Mario Haut

Transformaciones: GROUPBYKEY

Agrupa los valores de un RDD (Resilient Distributed Dataset) por sus claves. Esta operación es especialmente útil cuando deseas recopilar todos los valores asociados a una clave en un solo lugar, permitiendo realizar análisis adicionales sobre esos grupos de datos.

```
data = [("manzana", 3), ("naranja", 5), ("plátano", 2), ("manzana", 1), ("naranja", 3)]

fruit_rdd = sc.parallelize(data)

grouped_fruits = fruit_rdd.groupByKey()
```

Juan Mario Haut

Transformaciones: REDUCEBYKEY

Agregar (o reducir) los valores de un RDD de pares clave-valor utilizando una función de reducción. Esta operación se utiliza comúnmente para combinar valores que comparten la misma clave, realizando cálculos sobre esos valores y devolviendo un nuevo RDD que conserva las claves.

```
data = [("manzana", 3), ("naranja", 5), ("plátano", 2), ("manzana", 1), ("naranja", 3)]

fruit_rdd = sc.parallelize(data)

fruit_totals = fruit_rdd.reduceByKey(lambda a, b: a + b)

for fruit, total in fruit_totals.collect():

    print(f"{fruit}: {total}")
```

Juan Mario Haut

Transformaciones: SORTBYKEY

Permite ordenar un RDD de pares clave-valor según las claves. Esta operación devuelve un nuevo RDD que mantiene el mismo tipo de datos, pero con los elementos ordenados de acuerdo con las claves en orden ascendente o descendente.

```
data = [("manzana", 3), ("naranja", 5), ("plátano", 2), ("kiwi", 4), ("cereza", 1)]

fruit_rdd = sc.parallelize(data)

sorted_fruits = fruit_rdd.sortByKey()

for fruit, quantity in sorted_fruits.collect():

  print(f"{fruit}: {quantity}")
```

Juan Mario Haut

Transformaciones: DISTINCT

Se utiliza para eliminar duplicados de un RDD. Devuelve un nuevo RDD que contiene solo elementos únicos, eliminando todas las repeticiones en el conjunto de datos original. Esta operación es útil cuando necesitas trabajar con un conjunto de datos que solo contiene valores únicos.

```
data = ["manzana", "naranja", "plátano", "manzana", "kiwi", "naranja", "cereza", "kiwi"]

fruit_rdd = sc.parallelize(data)

distinct_fruits = fruit_rdd.distinct()

for fruit in distinct_fruits.collect():

    print(fruit)
```

Juan Mario Haut

Transformaciones: UNION

Esta operación devuelve un nuevo RDD que contiene todos los elementos de los RDDs de entrada, incluyendo duplicados. Es útil cuando deseas consolidar datos de diferentes fuentes o particiones en un solo conjunto de datos.

```
fruits1 = sc.parallelize(["manzana", "naranja", "plátano"])

fruits2 = sc.parallelize(["kiwi", "cereza", "naranja"])

combined_fruits = fruits1.union(fruits2)

for fruit in combined_fruits.collect():

    print(fruit)
```

Juan Mario Haut

Transformaciones: INTERSECTION

Devuelve un nuevo RDD que contiene solo aquellos elementos que están presentes en todos los RDDs de entrada. Esta operación es útil cuando necesitas identificar qué datos comparten diferentes conjuntos.

```
fruits1 = sc.parallelize(["manzana", "naranja", "plátano", "kiwi"])

fruits2 = sc.parallelize(["kiwi", "cereza", "naranja", "mango"])

common_fruits = fruits1.intersection(fruits2)

for fruit in common_fruits.collect():

    print(fruit)
```

Juan Mario Haut

Transformaciones: SUBTRACT

Devuelve un nuevo RDD que contiene solo los elementos que están en el primer RDD pero no en el segundo. Esta operación es útil cuando necesitas filtrar datos de un conjunto en función de otro conjunto.

```
fruits1 = sc.parallelize(["manzana", "naranja", "plátano", "kiwi"])

fruits2 = sc.parallelize(["kiwi", "cereza", "naranja"])

remaining_fruits = fruits1.subtract(fruits2)

for fruit in remaining_fruits.collect():

    print(fruit)
```

Juan Mario Haut

Acciones

Juan Mario Haut

Acciones: REDUCE

Aplica una función de reducción a los elementos de un RDD, combinando todos los elementos en un único resultado. Esta operación toma dos elementos a la vez y aplica una función que define cómo se deben combinar, continuando el proceso hasta que todos los elementos del RDD se han reducido a un único valor. Es especialmente útil para sumar, contar, o realizar operaciones similares sobre un conjunto de datos.

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

sum_result = numbers.reduce(lambda x, y: x + y)

print(f"La suma de los números es: {sum_result}")
```

Juan Mario Haut

Acciones: COLLECT

Devuelve todos los elementos del conjunto de datos como una matriz en el programa controlador. Esto suele ser útil después de un filtro u otra operación que devuelve un subconjunto suficientemente pequeño de los datos.

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

collected_numbers = numbers.collect()

print(f"Los números en el RDD son: {collected_numbers}")
```

Juan Mario Haut

Acciones: COUNT

Devuelve el número de elementos en el dataset..

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

count = numbers.count()

print(f"Los elementos en el RDD son: {count}")
```

Acciones: COUNTBYKEY

Se utiliza para contar el número de elementos para cada clave en un RDD de pares clave-valor (es decir, un RDD que tiene elementos en forma de (clave, valor)). Esta operación devuelve un diccionario (o un objeto similar a un mapa) donde las claves son las claves únicas del RDD y los valores son el conteo de las ocurrencias de cada clave.

```
fruits = sc.parallelize([("manzana", 1), ("naranja", 1), ("plátano", 1), ("manzana", 1), ("kiwi", 1), ("naranja", 1)])

fruit_counts = fruits.countByKey()

for fruit, count in fruit_counts.items():

    print(f"{fruit}: {count}")
```

Juan Mario Haut

Acciones: FIRST

Devuelve el primer elemento en el dataset..

```
fruits = sc.parallelize(["abe", "abby", "apple"])

fruit_counts = fruits.first()

“abe”
```

Juan Mario Haut

Acciones: SAVEASTEXTFILE

Se utiliza para guardar el contenido de un RDD en el sistema de archivos en formato de texto. Cada elemento del RDD se convierte en una línea en el archivo de salida. Esta acción es útil para exportar resultados procesados de un RDD a archivos que se puedan leer y analizar fácilmente, ya sea en HDFS, en el sistema de archivos local o en cualquier otro sistema compatible con Spark.

```
data = sc.parallelize(["manzana", "naranja", "plátano", "kiwi"])

data.saveAsTextFile("/ruta/de/tu/directorio/salida_fruits")
```

Juan Mario Haut

Acciones: FOREACH

Ejecuta una función sin parámetros para cada elemento de datos.

```
data = sc.parallelize(["manzana", "naranja", "plátano", "kiwi"])

data.foreach(lambda fruit: print(f"La fruta es: {fruit}"))
```

Juan Mario Haut