Tema 3: Procesamiento con Apache Spark

Máster en Ciencia de Datos (Universidad de Córdoba)

UCO ONLINE

Persistencia y Acumuladores

Tema 3: Procesamiento con Apache Spark

UCO ONLINE

Índice de la sección

- Introducción
- Persistencia
- Acumuladores

Introducción

- Dentro de Apache Spark se presentan un par de herramientas que nos pueden ser de mucha utilidad a la hora de trabajar con grandes cantidades de datos, y además donde queremos darle algo más de funcionalidad.
- Estos dos elementos de Apache Spark son:
  - Persistencia
  - Acumuladores

Persistencia (I)

- Una de las capacidades más importantes de Spark es conservar (o almacenar en caché) un conjunto de datos en la memoria entre operaciones.
- Cuando persiste un RDD, cada nodo almacena las particiones que calcula en la memoria y las reutiliza en otras acciones en ese conjunto de datos (o conjuntos de datos derivados de él).
- Esto permite que las acciones futuras sean mucho más rápidas (a menudo más de 10 veces). El almacenamiento en caché es una herramienta clave para los algoritmos iterativos y el uso interactivo rápido.

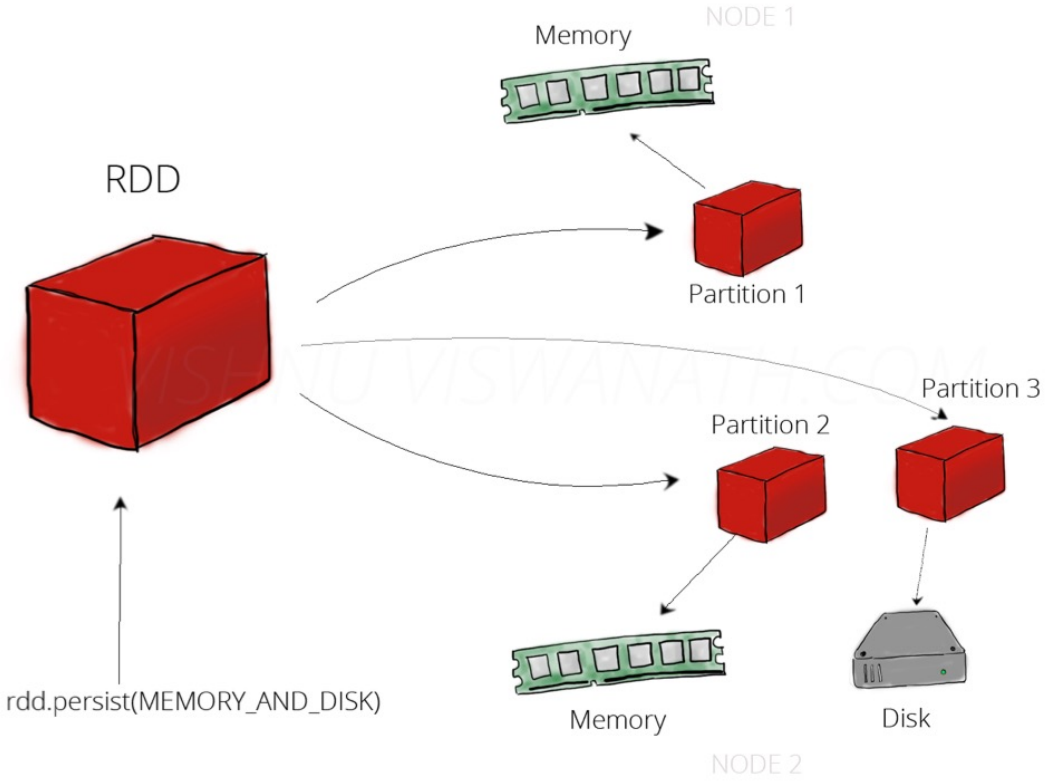
Persistencia (II)

- Se puede marcar un RDD para que se mantenga usando los métodos persist() o cache() en él.
- La primera vez que se calcula en una acción, se mantendrá en la memoria de los nodos.
- La caché de Spark es tolerante a fallos: si se pierde alguna partición de un RDD, se volverá a calcular automáticamente utilizando las transformaciones que la crearon originalmente.

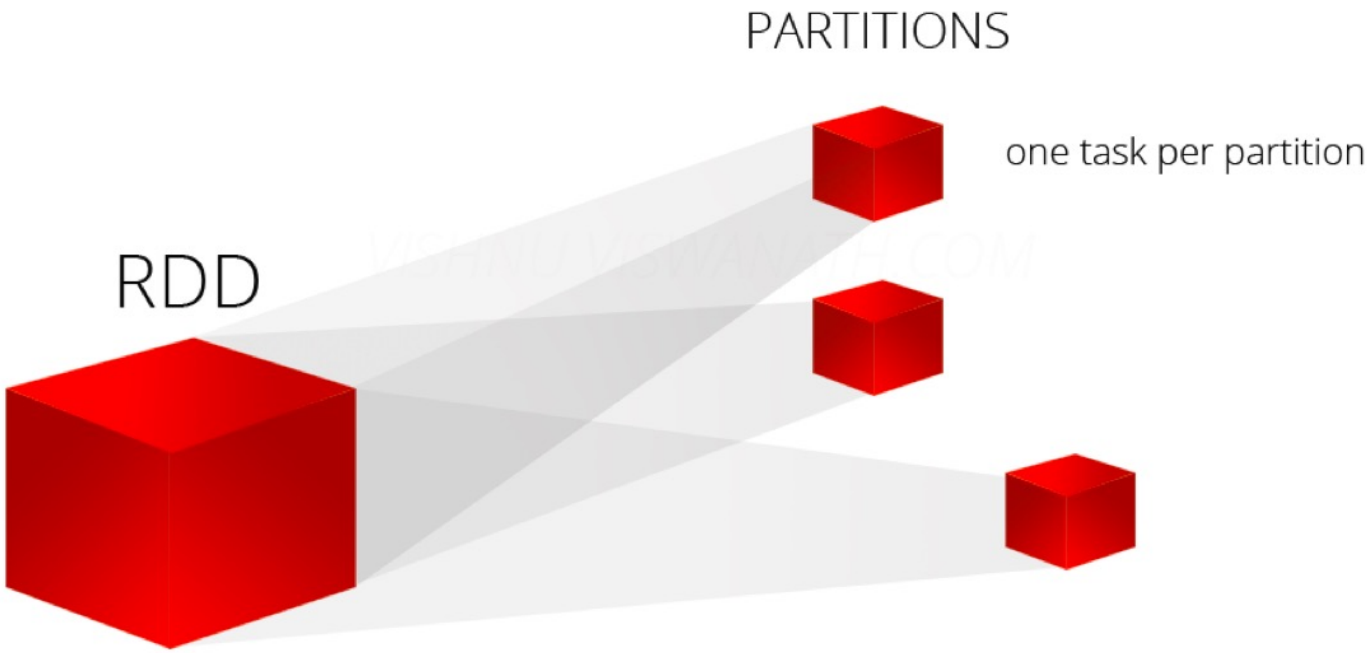
Persistencia (III)

- Además, cada RDD persistente se puede almacenar utilizando un nivel de almacenamiento diferente, lo que le permite, por ejemplo, conservar el conjunto de datos en el disco, conservarlo en la memoria pero como objetos Java serializados (para ahorrar espacio), replicarlo entre nodos.
- Estos niveles se establecen pasando un objeto StorageLevel (Scala, Java, Python) a persist().
- El método cache() es una forma abreviada de usar el nivel de almacenamiento predeterminado, que es StorageLevel.MEMORY_ONLY (almacenar objetos deserializados en la memoria).

Persistencia (IV)



Persistencia (V)



Persistencia (VI)

Forma de Almacenamiento - Storage Level	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, almacene las particiones que no caben en el disco y léalas desde allí cuando las necesite.
DISK_ONLY	Almacena las particiones RDD solo en el disco.

Persistencia (VII)

Forma de Almacenamiento - Storage Level	Significado
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Igual que los niveles de la diapositiva anterior, pero replica cada partición en dos nodos de clúster.
MEMORY_ONLY_SER (solo en Java y Scala)	Almacena RDD como objetos Java serializados (matriz de un byte por partición). Esto generalmente es más eficiente en espacio que los objetos deserializados, especialmente cuando se usa un serializador rápido, pero requiere más CPU para leer.
MEMORY_AND_DISK_SER (solo en Java y Scala)	Similar a MEMORY_ONLY_SER, pero vuelca las particiones que no caben en la memoria en el disco en lugar de volver a calcularlas sobre la marcha cada vez que se necesitan.

Persistencia (VIII)

- En Python, los objetos almacenados siempre se serializarán con la biblioteca **Pickle**, por lo que no importa si elige un nivel serializado.
- Los niveles de almacenamiento disponibles en Python incluyen MEMORY_ONLY, MEMORY_ONLY_2, MEMORY_AND_DISK, MEMORY_AND_DISK_2, DISK_ONLY, DISK_ONLY_2 y DISK_ONLY_3.

Persistencia (IX)

- Spark también conserva automáticamente algunos datos intermedios en las operaciones aleatorias (por ejemplo, reduceByKey), incluso sin que los usuarios llamen a persistir.
- Esto se hace para evitar volver a calcular toda la entrada si un nodo falla durante la reproducción aleatoria. Es conveniente utilizar la persistencia en el RDD resultante si está previsto reutilizarlo.

Persistencia (X)

- Los niveles de almacenamiento de Spark están destinados a proporcionar diferentes compensaciones entre el uso de la memoria y la eficiencia de la CPU. Algunos consejos para elegir el nivel de almacenamiento:
  - Si los RDD se ajustan cómodamente al nivel de almacenamiento predeterminado (MEMORY_ONLY), no se aconseja utilizar otro grado de persistencia. Esta es la opción más eficiente de la CPU, lo que permite que las operaciones en los RDD se ejecuten lo más rápido posible.
  - De lo contrario, se recomienda usar MEMORY_ONLY_SER y seleccionar una biblioteca de serialización rápida para hacer que los objetos sean mucho más eficientes en cuanto al espacio, pero aún razonablemente rápidos de acceder. (Java y Scala)

Persistencia (XI)

- No se debe realizar el volcado en el disco a menos que las funciones que calcularon sus conjuntos de datos sean costosas o filtren una gran cantidad de datos. De lo contrario, volver a calcular una partición puede ser tan rápido como leerla desde el disco.

- Se debe usar los niveles de almacenamiento replicados si desea una recuperación rápida de fallas (por ejemplo, si usa Spark para atender solicitudes de una aplicación web). Todos los niveles de almacenamiento brindan tolerancia total a fallas al volver a calcular los datos perdidos, pero los duplicados le permiten continuar ejecutando tareas en el RDD sin esperar a volver a calcular una partición perdida.

Persistencia (XI)

- Spark monitorea automáticamente el uso de la memoria caché en cada nodo y elimina las particiones de datos antiguas de forma menos usada recientemente (LRU).
- Si se desea eliminar manualmente un RDD en lugar de esperar a que desaparezca de la memoria caché, se debe utilizar el método RDD.unpersist(). Y tener en cuenta que este método no bloquea por defecto. Para bloquear hasta que se liberen los recursos, se debe especificar blocking=true al llamar a este método.

Ejemplo Persistencia (I)

- Se instala nuestro sistema, y creamos la SparkSession

```
[7]  !pip install pyspark
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.7/dist-packages (3.2.1)
Requirement already satisfied: py4j==0.10.9.3 in /usr/local/lib/python3.7/dist-packages (from pyspark) (0.10.9.3)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder\
.master("local")\
.appName("Colab")\
.config('spark.ui.port', '4050')\
.getOrCreate()
```

Ejemplo Persistencia (II)

- Se importan las librerías y se aplica la persistencia al RDD leído desde fichero

Ejemplo de Persistencia

```
[12] from pyspark import SparkConf
     from pyspark.context import SparkContext
     from pyspark import StorageLevel

     import time

     sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))
     fileMemory = sc.textFile("archivoPersistencia.txt")
     fileMemory.persist(StorageLevel.MEMORY_ONLY)
```

```
archivoPersistencia.txt MapPartitionsRDD[15] at textFile at NativeMethodAccessorImpl.java:0
```

Ejemplo Persistencia (III)

- Se realiza el cálculo de tiempo, sobre una transformación y una operación.

```
timestampInicio= time.time()* 1000
mayor6Memory=fileMemory.flatMap(lambda l:l.split(" ")).filter( lambda p: len(p)>6)
mayor6Memory.count()

timestampFin= time.time()* 1000

print("Tiempo : " + str(timestampFin - timestampInicio))
```

```
Tiempo : 179.011962890625
```

Ejemplo Persistencia (IV)

- Repetimos el proceso después de realizar unpersist()

```
[15] fileMemory.unpersist()
```

```
archivoPersistencia.txt MapPartitionsRDD[15] at textFile at NativeMethodAccessorImpl.java:0
```

```
timestampInicio= time.time()* 1000
mayorMemory=fileMemory.flatMap(lambda l:l.split(" ")).filter( lambda p: len(p)>6)
mayorMemory.count()

timestampFin= time.time()* 1000

print("Tiempo : " + str(timestampFin - timestampInicio))
```

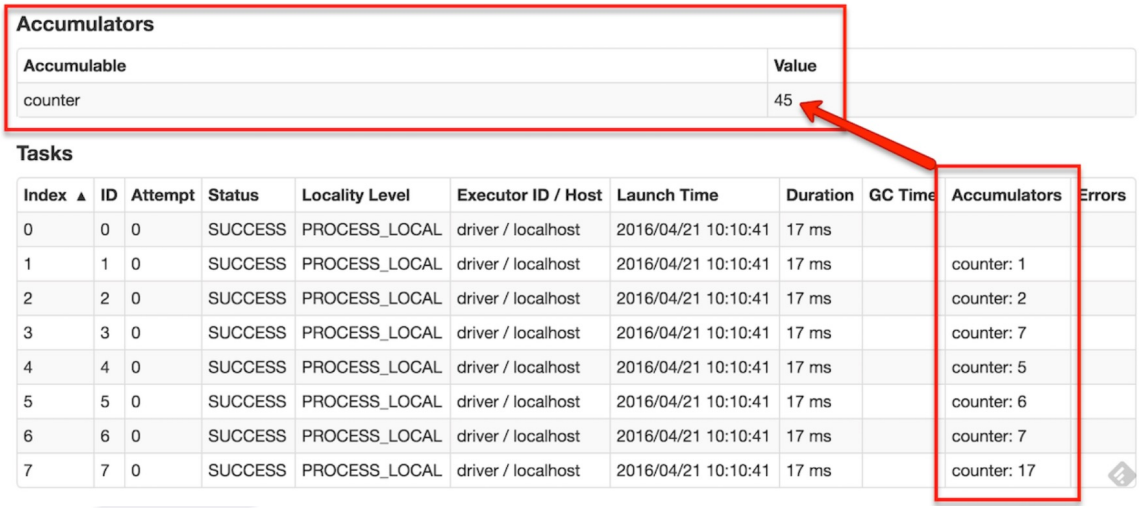
```
Tiempo : 152.7177734375
```

Acumuladores

- Los acumuladores son variables que solo se “suman” a través de una operación asociativa y conmutativa y, por lo tanto, pueden admitirse de manera eficiente en paralelo.
- Se pueden usar para implementar contadores (como en MapReduce) o sumas. Spark admite de forma nativa acumuladores de tipos numéricos y los programadores pueden agregar compatibilidad con nuevos tipos.

Acumuladores

- Se pueden crear acumuladores con o sin nombre. Como se ve en la imagen a continuación, se mostrará un acumulador con nombre (en este caso, contador) en la interfaz de usuario web para la etapa que modifica ese acumulador. Spark muestra el valor de cada acumulador modificado por una tarea en la tabla "Tareas".



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Acumuladores

- Resumiendo:
  - Los acumuladores son variable globales.
  - Nos sirven para tener una analítica de lo que hemos ido haciendo.
  - De forma nativa, Spark nos ofrece contadores tipo Long y Double.

Ejemplo Acumuladores

- Siguiendo con el ejemplo de los RDD a los que hemos aplicado persistencia obtenemos

Acumuladores

```
accum=sc.accumulator(0)

mayor6Memory=fileMemory.flatMap(lambda l:l.split(" ")).filter( lambda p: len(p)>6)

mayor6Memory.foreach(lambda p: accum.add(1))

accum.value
```

```
5140
```

Descarga de Ejemplos

- El ejemplo de cuadernos de Google Colab trabajado en esta unidad está disponible en la siguiente dirección:
- https://drive.google.com/file/d/1Q3mfvzx66i27YizZ8twuwnzvfcX72dHt/view?usp=sharing

- El archivo de carga para ejecutar la persistencia, puede descargarse de la siguiente dirección:
- https://drive.google.com/file/d/1KnsO5agyZuJ7nNiLDjmoJfS7wGD42KMF/view?usp=sharing

¡Gracias!

UCO ONLINE