DAG IN apache spark

In Apache Spark, DAG stands for **Directed Acyclic Graph**. It is a fundamental concept used by Spark's execution engine to ***represent and optimize*** the flow of operations in a data processing job.

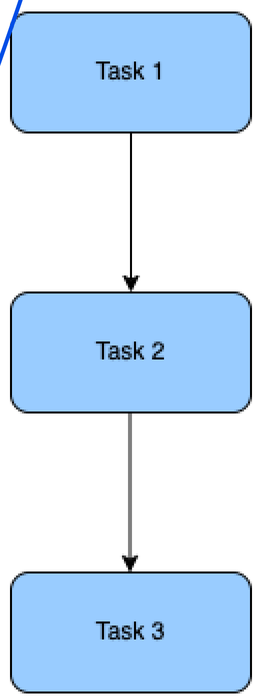
To understand DAG in Apache Spark, let's consider a real-life example of analyzing customer data in an e-commerce company.

Suppose we have a dataset containing information about customer orders, such as order IDs, customer IDs, order dates, and the total amount spent.

Let's say we want to perform the following tasks on this dataset:

1. Task 1: Filter out orders that were placed in the last 30 days.
2. Task 2: Group the filtered orders by customer ID.
3. Task 3: Calculate the total amount spent by each customer.

These tasks can be represented as a DAG. ***Each task is represented as a node in the DAG, and the dependencies between tasks are represented as directed edges.***



Directed Acyclic Graph (No Cycle)

In this example, **Task 1** filters the orders based on the order date and produces a filtered dataset. **Task 2** takes the output of Task 1 and groups the orders by customer ID. Finally, **Task 3** takes the output of Task 2 and calculates the total amount spent by each customer.

The DAG is ***acyclic***, meaning there are ***no cycles or loops in the graph***. This property allows Spark to optimize and schedule the execution of the operations effectively, as it can determine the dependencies and execute the stages in the most efficient order.

Stages in DAG

Stages are a key concept within the Directed Acyclic Graph (DAG) execution model. A stage represents a ***set of tasks that can be executed together*** in a single wave of computation, resulting in a more efficient execution of the Spark job.

Now, let's take an example. You have a text file of 1 GB and have created ten partitions of it. You also performed some transformations, and in the end, you requested to see how the first line looks. In this case, Spark will read the file only from the first partition and give you the results as your requested results do not require it to read the complete file.

Let's take a few practical examples to see how Spark performs lazy evaluation. In the first step, we created a list of 10 million numbers and made an RDD with four partitions below. And we can see the result in the below output image.

```
val data = (1 to 100000).toList val rdd = sc.parallelize(data,4)
println("Number of partitions is "+rdd.getNumPartitions)
```

```
Number of partitions is 4
data: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
2, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 1
5, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141,
61, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177,
197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213
233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249
269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285
305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321
341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393
413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429
449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465
485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501
521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537
557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573
```

Next, we will perform a fundamental transformation, like adding 4 to each number. Note that Spark, at this point, has not started any transformation. It only records a series of transformations in the form of RDD Lineage. You can see that RDD lineage using the function toDebugString

```
//Adding 5 to each value in rdd val rdd2 = rdd.map(x => x+5) //rdd2 objetc
println(rdd2) //getting rdd lineage rdd2.toDebugString
```

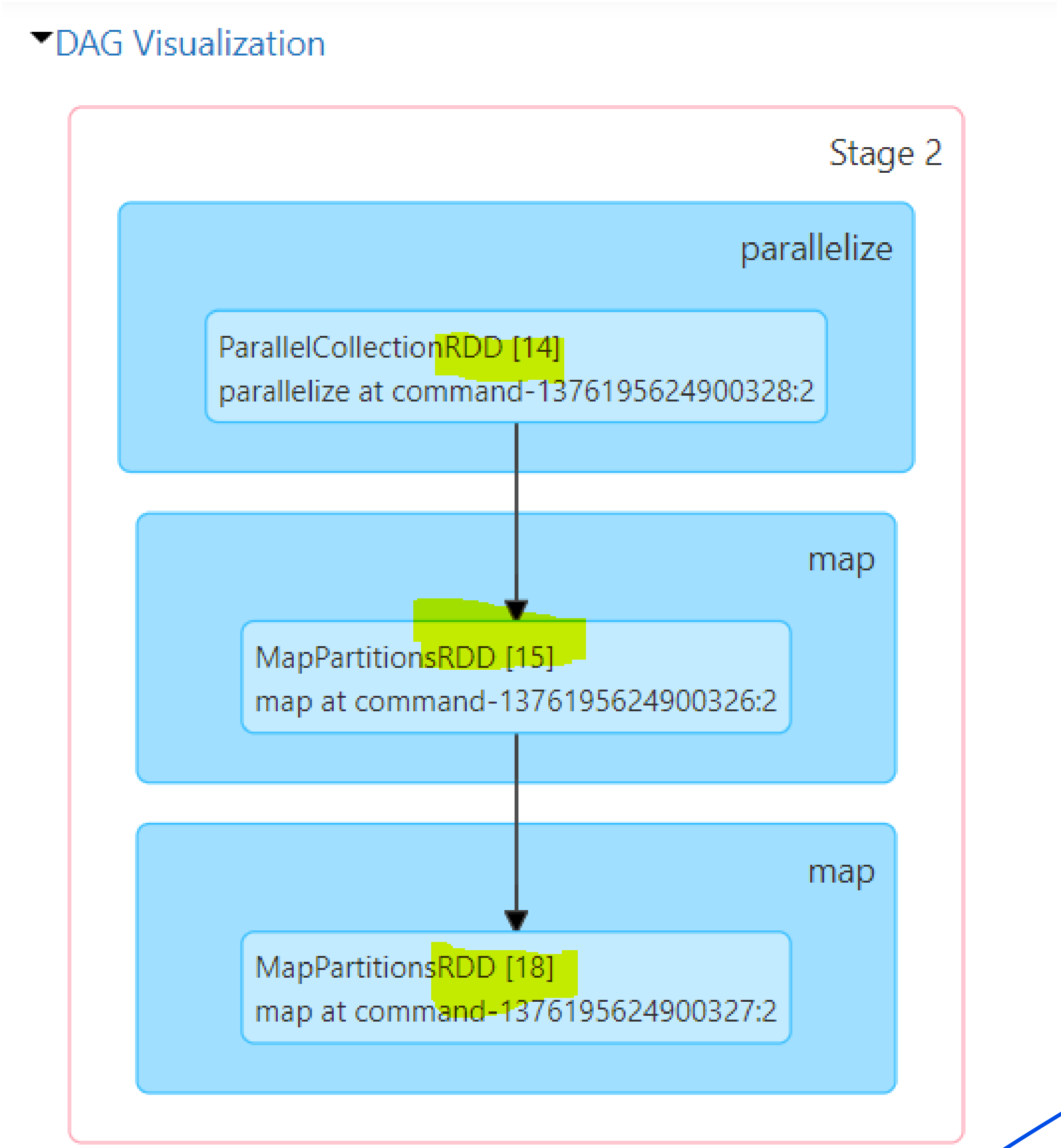
```
MapPartitionsRDD[15] at map at command-1376195624900326:2
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[15] at map at c
res1: String =
(4) MapPartitionsRDD[15] at map at command-1376195624900326:2 []
 |  ParallelCollectionRDD[14] at parallelize at command-13761956249003
```

Now if you observe MapPartitionsRDD[15] at map is dependent on ParallelCollectionRDD[14]. Now, let's go ahead and add one more transformation to add 20 to all the elements of the list.

```
//Adding 5 to each value in rdd val rdd3 = rdd2.map(x => x+20) //rdd2
objetc println(rdd3) //getting rdd lineage rdd3.toDebugString rdd3.collect
```

```
MapPartitionsRDD[18] at map at command-1376195624900327:2
rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[18] at map at c
res4: String =
(4) MapPartitionsRDD[18] at map at command-1376195624900327:2 []
 |  MapPartitionsRDD[15] at map at command-1376195624900326:2 []
 |  ParallelCollectionRDD[14] at parallelize at command-137619562490032
```

Now if you observe MapPartitionsRDD[18] at map is dependent on MapPartitionsRDD[15] and ParallelCollectionRDD[14]. Now, let's go ahead and add one more transformation to add 20 to all the elements of the list. After calling a action using collect we see that three stages of DAG lineage at ParallelCollectionRDD[14], MapPartitionsRDD[15] and MapPartitionsRDD[18].



From the above examples, we can able to understand that spark lineage is maintained using DAG.