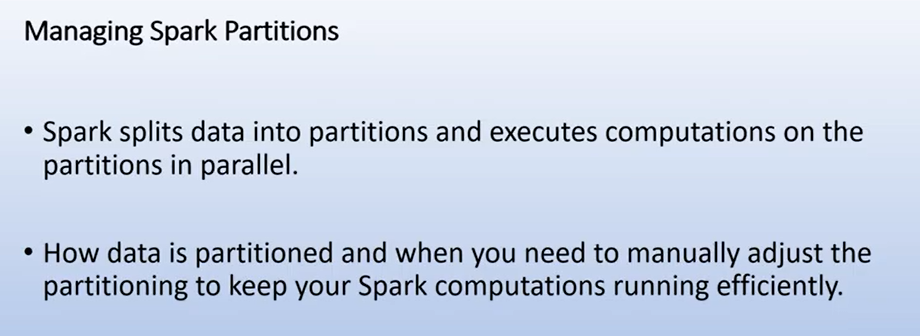
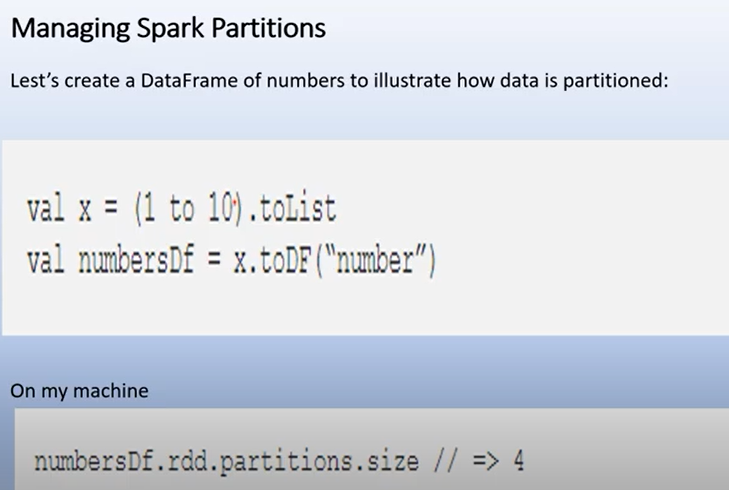
[Resilient Distributed Datasets](https://www.dezyre.com/article/working-with-spark-rdd-for-fast-data-processing/273" \o "Spark RDD" \t "_blank) are collection of various data items that are so huge in size, that they cannot fit into a single node and have to be partitioned across various nodes. Spark automatically partitions RDDs and distributes the partitions across different nodes. A partition in spark is an atomic chunk of data (logical division of data) stored on a node in the cluster. Partitions are basic units of parallelism in Apache Spark. RDDs in Apache Spark are collection of partitions.

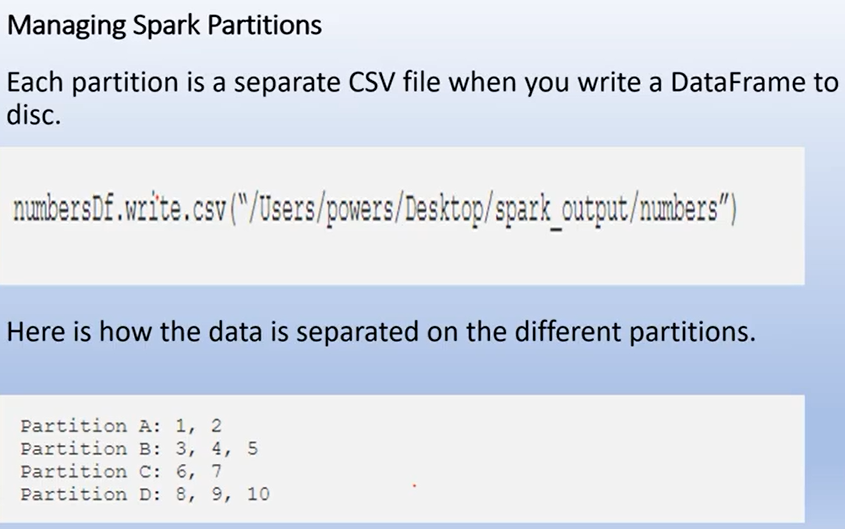
Managing spark partitions:



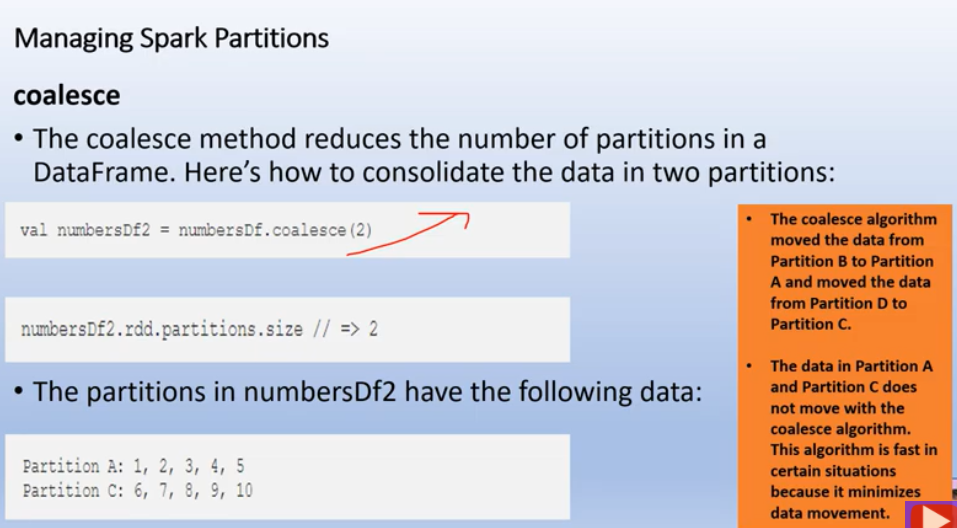
Sparks splits the data into partitions that’s how data is distributed across nodes and cluster and then each transformation or computation is performed on individual partitions that’s how we achieve the parallelism and we achieve parallel processing



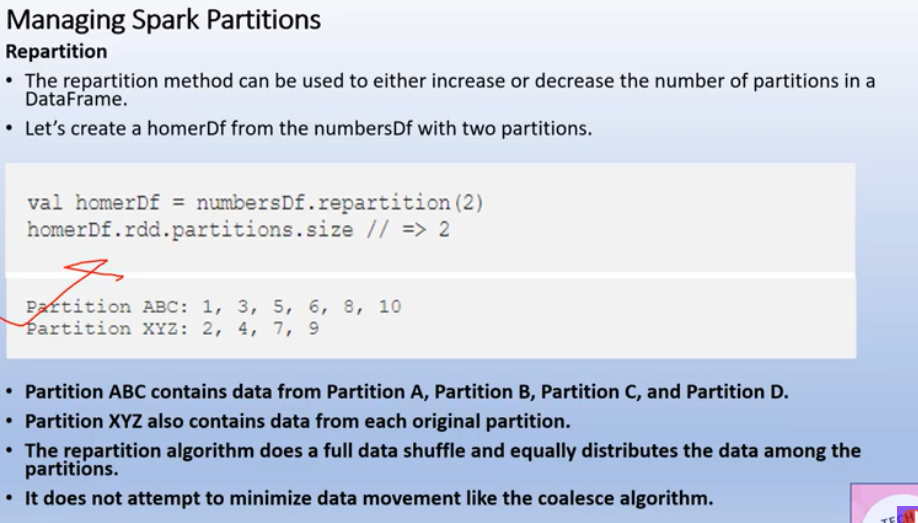
The default partition is 4 and data is sliced in robin fashion.



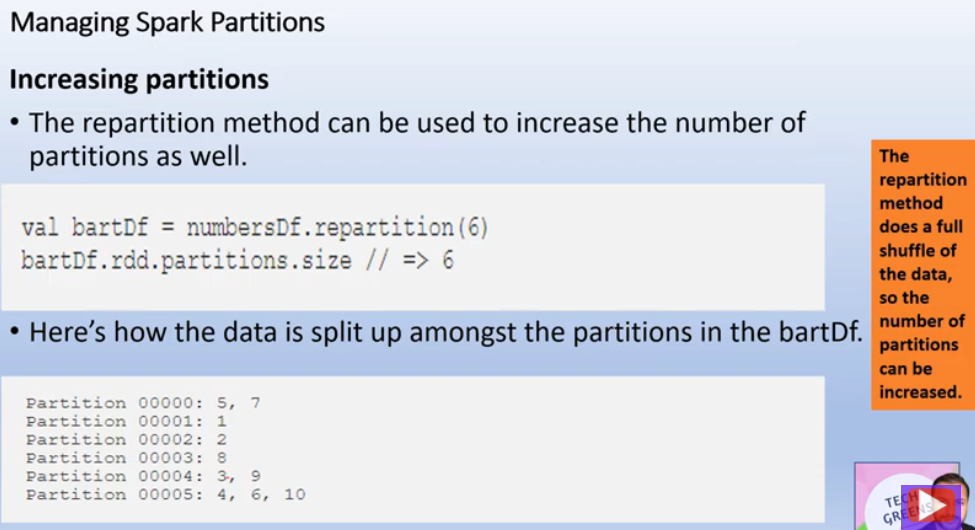
With the help of coalesce, you can reduce the number of partitions. if you see below the shuffling is done in two partitions as partition B move to partition A and partition D move to partition c ,only 50% shuffling is done. In coalesce the two partition move to existing two partitions.



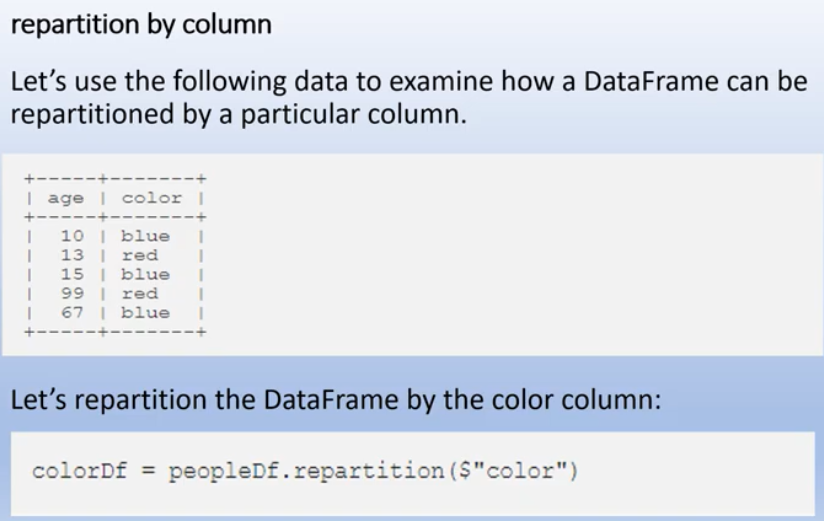
In case of repartition, new partition is created and all 4 existing partition is shuffled to create new partition and no equal distribution. The data move across the network is double as we compare with the coalesce



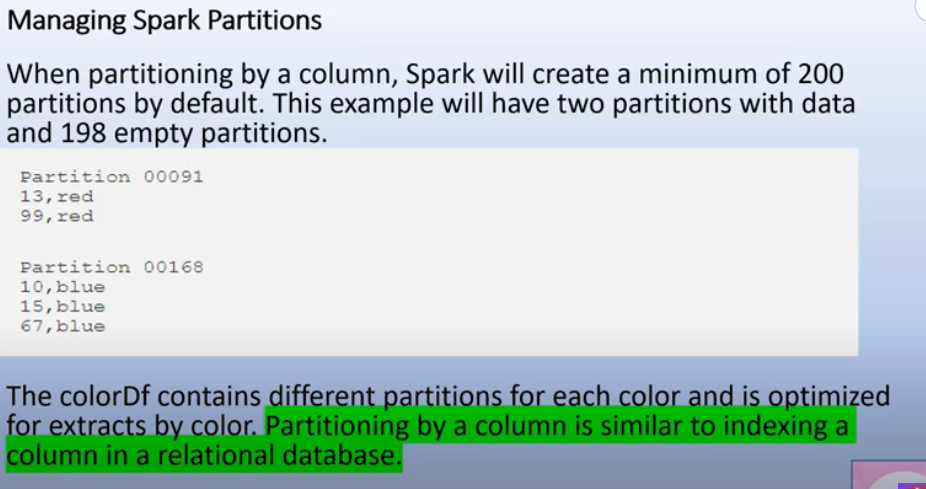
But in case of coalesce you cannot increase no. of partitions, as in repartition you can increase no. of partitions



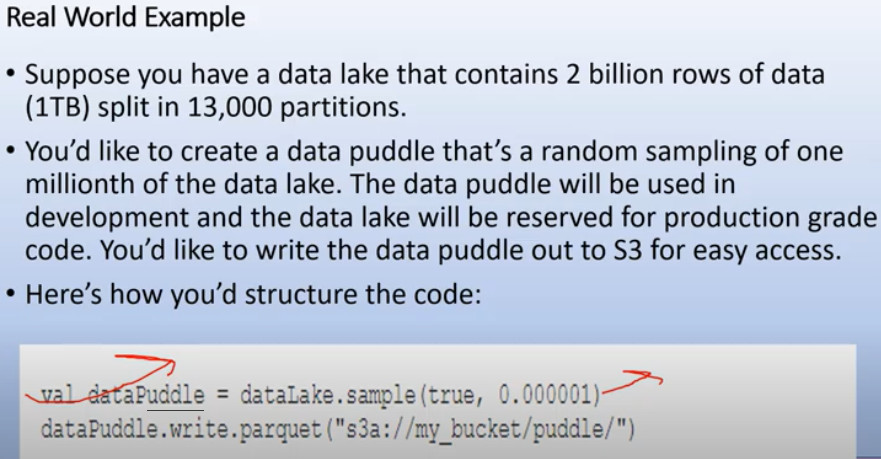
We can also do repartition on the basis of particular column



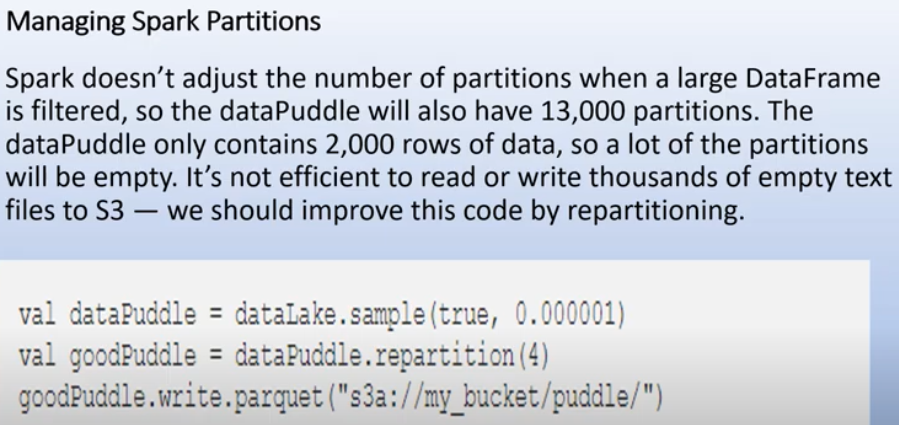
You can see the partition done on the basis of color below:

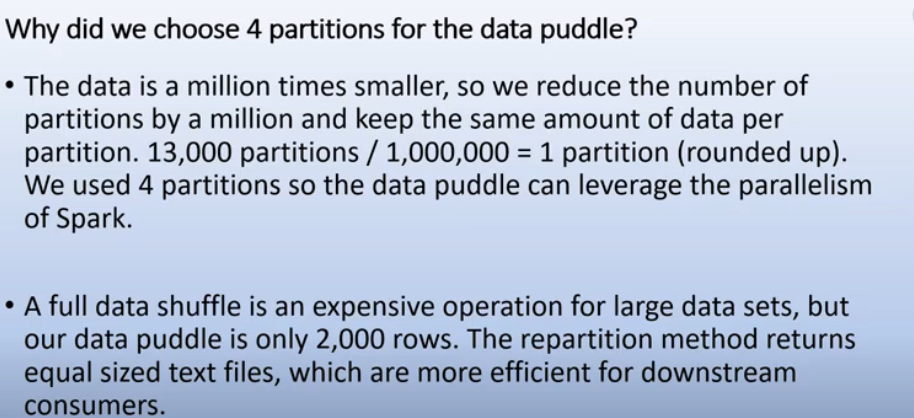


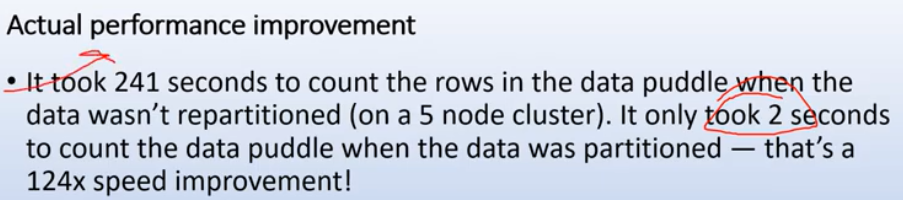
Lets take a real world example, where we have 2 billion(1 tb) of data split in 13000 partitions . suppose we take only 0.000001 percent of that data and store , then still it create 13000 of partition on that data also in which lots of partitions are empty and it also takes lots of time also.



To resolve this, we use repartition



s



One important way to increase parallelism of spark processing is to increase the number of executors on the cluster. However, knowing how the data should be distributed, so that the cluster can process data efficiently is extremely important. The secret to achieve this is partitioning in Spark. Apache Spark manages data through RDDs using partitions which help parallelize distributed data processing with negligible network traffic for sending data between executors. By default, Apache Spark reads data into an RDD from the nodes that are close to it.

Communication is very expensive in distributed programming, thus laying out data to minimize network traffic greatly helps improve performance. Just like how a single node program should choose the right data structure for a collection of records, a spark program can control RDD partitioning to reduce communications. Partitioning in Spark might not be helpful for all applications, for instance, if a RDD is scanned only once, then portioning data within the RDD might not be helpful but if a dataset is reused multiple times in various key oriented operations like joins, then partitioning data will be helpful.

Partitioning is an important concept in apache spark as it determines how the entire hardware resources are accessed when executing any job. In apache spark, by default a partition is created for every [HDFS](https://www.dezyre.com/hadoop-course/hdfs)partition of size 64MB. RDDs are automatically partitioned in spark without human intervention, however, at times the programmers would like to change the partitioning scheme by changing the size of the partitions and number of partitions based on the requirements of the application. For custom partitioning developers have to check the number of slots in the hardware and how many tasks an executor can handle to optimize performance and achieve parallelism.

 Having too large a number of partitions or too few - is not an ideal solution. The number of partitions in spark should be decided thoughtfully based on the cluster configuration and requirements of the application. Increasing the number of partitions will make each partition have less data or no data at all. Apache Spark can run a single concurrent task for every partition of an RDD, up to the total number of cores in the cluster. If a cluster has 30 cores then programmers want their RDDs to have 30 cores at the very least or maybe 2 or 3 times of that.

As already mentioned above, one partition is created for each block of the file in HDFS which is of size 64MB.However, when creating a RDD a second argument can be passed that defines the number of partitions to be created for an RDD.

val rdd= sc.textFile (“file.txt”, 5)

The above line of code will create an RDD named textFile with 5 partitions. Suppose that you have a cluster with four cores and assume that each partition needs to process for 5 minutes. In case of the above RDD with 5 partitions, 4 partition processes will run in parallel as there are four cores and the 5th partition process will process after 5 minutes when one of the 4 cores, is free. The entire processing will be completed in 10 minutes and during the 5th partition process, the resources (remaining 3 cores) will remain idle. The best way to decide on the number of partitions in an RDD is to make the number of partitions equal to the number of cores in the cluster so that all the partitions will process in parallel and the resources will be utilized in an optimal way.

The number of partitions in a Spark RDD can always be found by using the partitions method of RDD. For the RDD that we created the partitions method will show an output of 5 partitions

Scala> rdd.partitions.size

Output = 5

If an RDD has too many partitions, then task scheduling may take more time than the actual execution time. To the contrary, having too less partitions is also not beneficial as some of the worker nodes could just be sitting idle resulting in less concurrency. This could lead to improper resource utilization and data skewing i.e. data might be skewed on a single partition and a worker node might be doing more than other worker nodes. Thus, there is always a trade off when it comes to deciding on the number of partitions.

**IMPORTANT LINKS:**

[**https://www.dezyre.com/article/how-data-partitioning-in-spark-helps-achieve-more-parallelism/297#:~:text=A%20partition%20in%20spark%20is,and%20their%20salaries%2D%20CLICK%20HERE**](https://www.dezyre.com/article/how-data-partitioning-in-spark-helps-achieve-more-parallelism/297#:~:text=A%20partition%20in%20spark%20is,and%20their%20salaries%2D%20CLICK%20HERE)

<https://acadgild.com/blog/partitioning-in-spark>