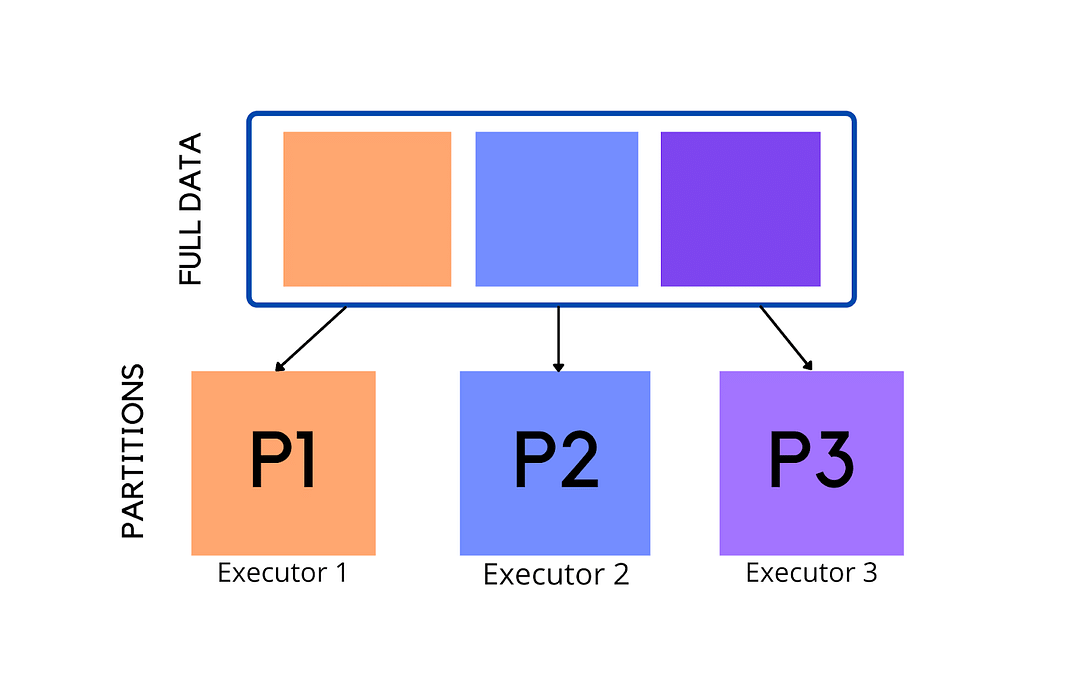
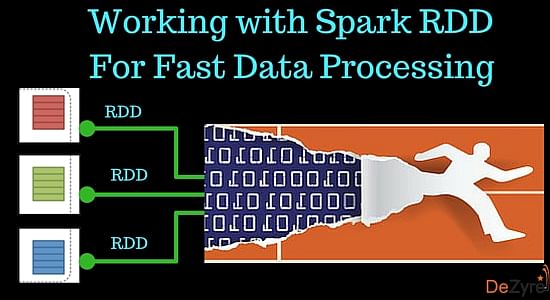
What is a partition in Spark?

Spark is a cluster processing engine that allows data to be processed in parallel. Apache Spark's parallelism will enable developers to run tasks parallelly and independently on hundreds of computers in a cluster. All thanks to Apache Spark's fundamental idea, RDD.



Resilient Distributed Datasets are collections 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 collections of partitions.

What are Resilient Distributed Datasets (RDDs)?



According to the original paper -Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, RDDs in Spark can be defined as follows-

*Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.*

According to the scaladoc of org.apache.spark.rdd.RDD:

*A Resilient Distributed Dataset (RDD), the basic abstraction in Spark, represents an immutable, partitioned collection of elements that can be operated on in parallel.*

*If you would like more information about Big Data careers, please click the orange "Request Info" button on top of this page.*

Let’s understand RDDs in Spark through their full form-Resilient Distributed Datasets:

Resilient

Meaning it provides fault tolerance through lineage graphs. A lineage graph keeps a track of transformations to be executed after an action has been called. RDD lineage graph helps recompute any missing or damaged partitions because of node failures.

Distributed

RDDs are distributed - meaning the data is present on multiple nodes in a cluster.

Datasets

Collection of partitioned data with primitive values.

There Exists a Synchronization Barrier between Map and Reduce Tasks

In Hadoop MapReduce, there always exists a synchronization barrier between the map and reduce tasks, thus the data should be persisted to the disc. Though this kind of design helps the job to recover whenever there is a failure, it does not leverage the memory of the Hadoop cluster in a complete form. Spark RDDs let the users transparently store data in memory and persevere it to the disc only if it is needed. So, there does not exist any synchronization barrier that can slow down the data processing speed making the Spark execution engine really fast.

Data Sharing is slow with Hadoop MapReduce - Spark RDDs solve this.

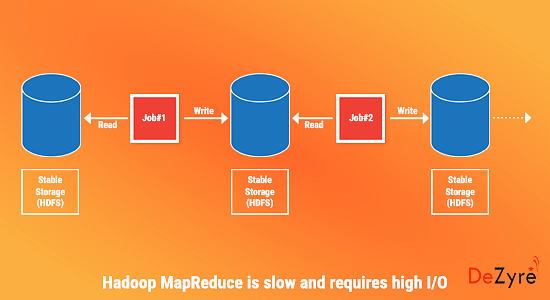
Hadoop MapReduce cannot handle iterative (Machine learning and graph processing tasks) and interactive (running ad-hoc queries on the same subset of data) applications. If data is kept in-memory for both these applications then the performance can be improved to a great extent.  In iterative distributed data processing where data needs to be processed over multiple jobs in machine learning algorithms like K-Means Clustering or Logistic Regression or Page Rank, it is very common that data is shared or reused between multiple jobs.

Iterative Operations on Hadoop MapReduce

Hadoop MapReduce is fault tolerant but it is very difficult to reuse intermediate results across multiple jobs. Data reuse or sharing is very slow in Hadoop MapReduce as the data needs to be stored into intermediate stable storage like Amazon S3 or HDFS. This requires multiple IO operations, serializations and data replication slowing down the overall computation of jobs.

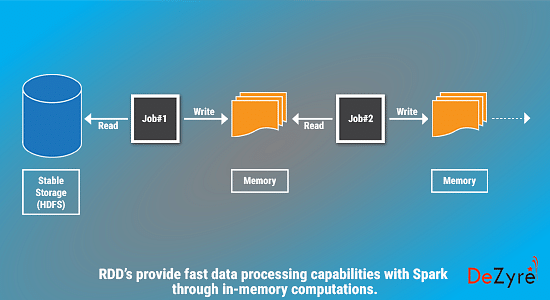
Suppose that there is a web server log that needs to be analyzed for a specific error code. You write Hadoop MapReduce code using regular expressions to look for the specific error code using grep. On executing the Hadoop MapReduce code on the server a set of files containing the grepped error\_code will be returned, the cluster will be closed and all the retrieved files will be stored on an Amazon S3 location mentioned in the Hadoop MapReduce code. You look at the retrieved files and notice that you need to write another piece of Hadoop MapReduce code to retrieve a few more files. This will require additional time to bring those files, process them and return the desired results.

The below image demonstrates iterative processing in MapReduce, which shows the intermediate stable storage required in Hadoop MapReduce for processing incurring an additional overhead -



Iterative Operations on Spark RDDs

Resilient Distributed Datasets solve this problem through fault tolerant distributed in-memory processing by storing the data in-memory so that the user can re-query the subset of data. The below image depicts how RDDs store the intermediate outputs in distributed memory instead of stable storage, making the execution faster than Hadoop MapReduce -



RRDs in Spark outperform Hadoop MapReduce by up to 20 times in iterative applications that require querying large volumes of data. The below graph shows the performance evaluation for iterative machine learning algorithms using Hadoop and Spark-

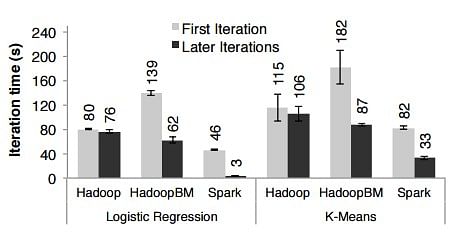
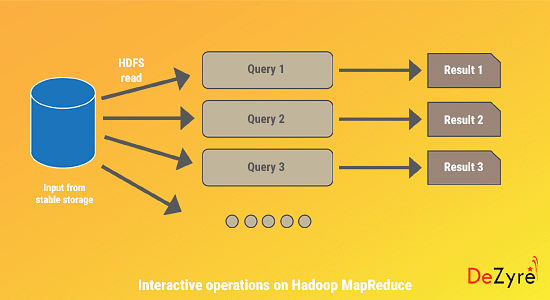


Image Credit : Original Paper on RDD's by Matei Zaharia

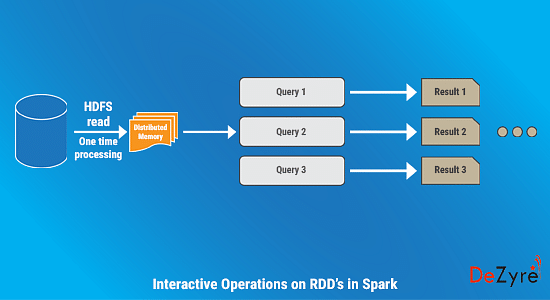
Interactive Operations on MapReduce

In interactive data mining applications users need to run multiple ad-hoc queries on the same subset of data. In Hadoop MapReduce, it is not efficient to run interactive ad-hoc queries because every query will perform disk I/O on the stable storage which is likely to surpass the overall execution time of the interactive application.



Interactive Operation on Spark RDDs

Apache Spark has been evaluated for interactive use by querying 1 TB of dataset with latency of 5 to 7 seconds. Users can run different queries on the same subset of data continuously by keeping it in-memory for improved execution times. The below image depicts how interactive operations are performed in Spark in-memory.



The below graph shows the performance evaluation for interactive applications using Hadoop and Spark-

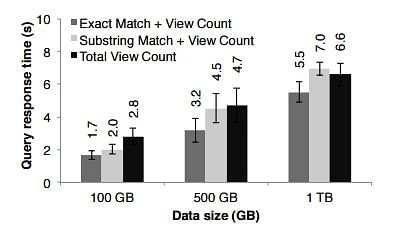


Image Credit : Original Paper on RDD's by Matei Zaharia

      Summary of the Limitations of Hadoop MapReduce over Spark RDDs

* Hadoop MapReduce uses coarse grained operations for processing that are just too heavy for processing iterative algorithms.
* Hadoop MapReduce cannot cache intermediate data in-memory but instead flushes intermediate data to disk after every step.

Types of RDDs in Spark

* A resultant RDD obtained by calling operations like map (), flatMap () is known as MapPartitions RDD.
* An RDD that provides functionality for reading data stored in HDFS is known as HadoopRDD.
* A resultant RDD obtained by calling operations like coalesce and repartition is known as a Coalesced RDD.

There are many other interesting types of RDDs in Spark like SequenceFileRDD, PipedRDD, CoGroupedRDD, and ShuffledRDD.

RDDs bring in many benefits to the Spark ecosystem and are best suited for batch analytic applications as they have all the required information in the lineage graph to reconstruct in parallel on different nodes after a failure. RDDs do have many advantages but cannot be used for all kinds of applications. RDDs might not be well-suited for applications like – a storage system for a web application or an incremental web crawler, that make asynchronous fine grained updates to shared state.

Characteristics of Partitions in Apache Spark

* Every machine in a spark cluster contains one or more partitions.
* The number of partitions in spark are configurable and having too few or too many partitions is not good.
* Partitions in Spark do not span multiple machines.

Partitioning in Apache Spark

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 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.

How many partitions should a Spark RDD have?

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 the 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 few 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.

Some acclaimed guidelines for the number of partitions in Spark are as follows-

When the number of partitions is between 100 and 10K partitions based on the size of the cluster and data, the lower and upper bound should be determined.

* The lower bound for spark partitions is determined by 2 X number of cores in the cluster available to application.
* Determining the upper bound for partitions in Spark, the task should take 100+ ms time to execute. If it takes less time, then the partitioned data might be too small or the application might be spending extra time in scheduling tasks.

Types of Partitioning in Apache Spark

1. Hash Partitioning in Spark
2. Range Partitioning in Spark

Hash Partitioning in Spark

Hash Partitioning attempts to spread the data evenly across various partitions based on the key. Object.hashCode method is used to determine the partition in Spark as partition = key.hashCode () % numPartitions.

Range Partitioning in Spark

Some Spark RDDs have keys that follow a particular ordering, for such RDDs, range partitioning is an efficient partitioning technique. In range partitioning method, tuples having keys within the same range will appear on the same machine. Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

Spark’s range partitioning and hash partitioning techniques are ideal for various spark use cases but spark does allow users to fine tune how their RDD is partitioned, by using custom partitioner objects. Custom Spark partitioning is available only for pair RDDs i.e. RDDs with key value pairs as the elements can be grouped based on a function of each key. Spark does not provide explicit control of which key will go to which worker node but it ensures that a set of keys will appear together on some node. For instance, you might range partition the RDD based on the sorted range of keys so that elements having keys within the same range will appear on the same node or you might want to hash partition the RDD into 100 partitions so that keys that have same hash value for modulo 100 will appear on the same node.

What is Spark repartition ?

Many times, spark developers will have to change the original partition. This can be achieved by changing the spark partition size and number of spark partitions. This can be done using the repartition() method.

df.repartition(numberOfPartitions)

repartition() shuffles the data and divides it into a number partitions. But a better way to spark partitions is to do it at the data source and save network traffic.

How to set partitioning for data in Apache Spark?

RDDs can be created with specific partitioning in two ways –

1. Providing explicit partitioner by calling partitionBy method on an RDD,
2. Applying transformations that return RDDs with specific partitioners. Some operation on RDDs that hold to and propagate a partitioner are-

* Join
* LeftOuterJoin
* RightOuterJoin
* groupByKey
* reduceByKey
* foldByKey
* sort
* partitionBy
* foldByKey

Best practices for Spark partitioning

* PySpark partitionBy() method

While writing DataFrame to the Disk/File system, PySpark partitionBy() is used to partition based on column values. PySpark divides the records depending on the partition column and puts each partition data into a subdirectory when you write DataFrame to Disk using partitionBy().

PySpark Partition divides a large dataset into smaller chunks using one or more partition keys. You can also use partitionBy() to build a partition on several columns; simply give the columns you wish to partition as an argument.

Let’s create a DataFrame by reading a CSV file.  You can find the dataset at this link [Cricket\_data\_set\_odi.csv](https://www.kaggle.com/cricketdataset/cricket-data-set-odi?select=Cricket_data_set_odi.csv)

# importing module

import pyspark

from pyspark.sql import SparkSession

from pyspark.context import SparkContext

# create sparksession and give an app name

spark = SparkSession.builder.appName(‘dezyreApp’).getOrCreate()

# create DataFrame

df=spark.read.option("header",True).csv("Cricket\_data\_set\_odi.csv")

* PySpark partitionBy() with One column:

For the following instances, we'll utilize team as a partition key from the DataFrame above:

df.write.option("header", True) \

partitionBy("Team") \

mode("overwrite") \

csv("Team")

We have a total of 9 different teams in our dataframe. Thus it produces nine directories. The partition column and its value (partition column=value) would be the name of the sub-directory.

FAQs

How to decide the number of partitions in Spark?

In Spark, one should carefully choose the number of partitions depending on the cluster design and application requirements. The best technique to determine the number of spark partitions in an RDD is to multiply the number of cores in the cluster with the number of partitions.

How do I create a partition in Spark?

In Spark, you can create partitions in two ways -

* By invoking partitionBy method on an RDD, you can provide an explicit partitioner,
* By applying Transformations to yield RDDs with specific partitioners.