**Session 16 Assignment 2**

1. Pen down the limitations of MapReduce.

* There are certain cases where MapReduce is not a suitable choice :
* **Real-time** processing. It's not **always** very easy to implement each and everything as a MR program. It does not process streamed data, and if we want real-time options on top of it, we’ll have to use platforms like Storm and Impala, Giraph (for graph processing). MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.
* When your intermediate processes need to talk to each other (jobs run in isolation). It is not so efficient for iterative processing, as Hadoop does not support cyclic data flow (i.e. a chain of stages in which each output of the previous stage is the input to the next stage) and needs a sequence of MR jobs to run iterative tasks.
* When you out processing requires lot of data to be **shuffled** over the network. When you need to handle streaming data. MR is best suited to **batch process** huge amounts of data which you already have with you.
* When you can get the desired result with a standalone system. It's obviously less painful to configure and manage a standalone system as compared to a distributed system. When you have **OLTP** needs. MR is not suitable for a large number of short on-line transactions.
* When you have **OLTP** needs. MR is not suitable for a large number of short on-line transactions.
* MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In MapReduce, Map takes a set of data and converts it into another set of data, where individual elements are broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.
* It is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.
* Doesn't work well with iterative computation because it doesn't store any data in memory to be worked on. The developer must create multiple MapReduce jobs to accomplish one iterative process; just the number of IO operations will significantly contribute to slowness and burdensome on the network.
* The process of testing code is not convenient, as is also the process of using user-defined functions.
* Many Big Data problems don't fit the MapReduce format, which means other big data tools must be used to gather with it to solve most problems. Also others professionals have to learn new (or something similar to what they already know) programming language to be able to use it; otherwise, they will have to use another tool altogether to solve their problems.

1. What is RDD? Explain few features of RDD?

* **Resilient Distributed Dataset** also known as **RDD** is the primary data abstraction in Apache Spark and the core of Spark.
* **Resilient** fault-tolerant with the help of [RDD lineage graph](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd.html#lineage) and so able to re-compute missing or damaged partitions due to node failures.
* **Distributed** with data residing on multiple nodes in a [cluster](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-cluster.html).
* **Dataset** is a collection of [partitioned data](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd-partitions.html) with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

Below are the some of the features of the RDD:

* **In-Memory -** data inside RDD is stored in memory as much (size) and long (time) as possible.
* **Immutable or Read-Only**, It does not change once created and can only be transformed using transformations to new RDDs.
* **Lazy evaluated**, the data inside RDD is not available or transformed until an action is executed that triggers the execution.
* **Cacheable**, you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
* **Parallel**, process data in parallel.
* **Typed** — RDD records have types, e.g. Long in RDD [Long] or (Int, String) in RDD [(Int, String)].
* **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
* **Location-Stickiness** — RDD can define [placement preferences](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd.html#preferredLocations) to compute partitions (as close to the records as possible).

1. List down few Spark RDD operations and explain each of them.

* RDD in Apache Spark supports two types of operations:
* Transformation
* Actions

**Transformations**

Spark RDD Transformations are functions that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map (), filter (), reduceByKey() etc.

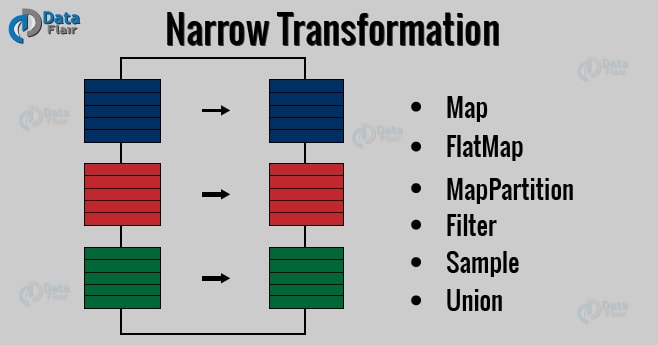
Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations.

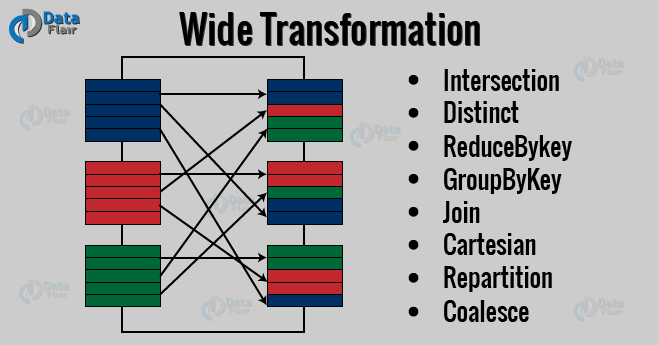
There are two kinds of transformations:

**Narrow transformation** ->it is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage known as pipelining.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/spark-narrow-transformation-1.jpg)

**Wide Transformations** -> It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as shuffle transformations because they may or may not depend on a shuffle.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/spark-wide-transformation.jpg)

**Actions**

An Action in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.

Using transformations, one can create RDD from the existing one. But when we want to work with the actual dataset, at that point we use Action. When the Action occurs it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.