## limitations of MapReduce

1. When you need a response fast. e.g. say < few seconds
2. Processing graphs
3. Iterations
4. When map phase generate too many keys. Then sorting takes for ever
5. Machine Learning
6. Slow as it stores data back to disk after each MR job
7. Difficult to code in Java
8. Cannot process real time

## RDD and its features

**Definition**:

RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. RDD is, essentially, the Spark representation of a set of data, spread across multiple machines, with APIs to act on it. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

**Features:**

**Resilient**, i.e.fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures

**Distributed** with data residing on multiple nodes in a cluster.

**Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other

**In -Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible..

**Immutable** or **Read -Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs

**Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.

**Cacheable** , i.e. you can hold all the data in a persistent "storage" memory (default and the most preferred) or disk (least preferred due to access speed).

**IParallel**, i.e. 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 to compute partitions (as close to the records as possible).

## Basic RDD Operations

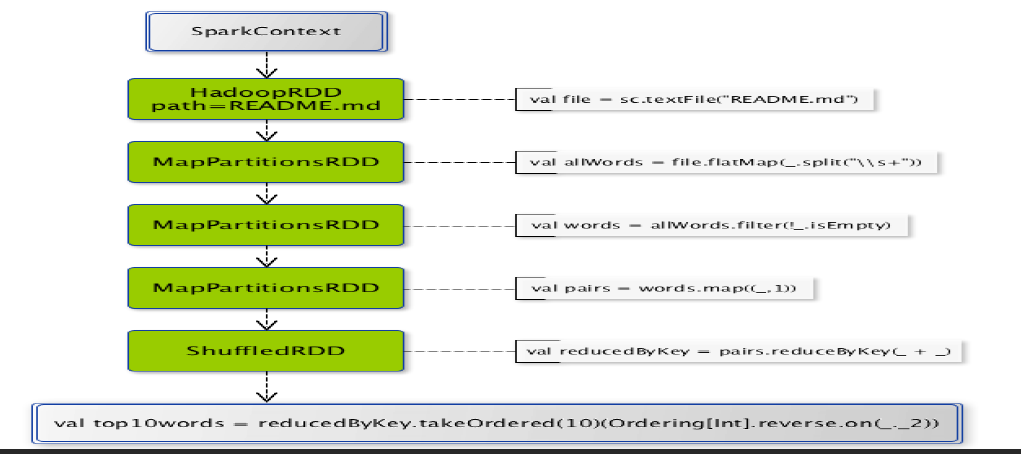
Two types of Apache Spark RDD operations are- **Transformations and Actions**. A Transformation is a function that produces new RDD from the existing RDDs but when we want to work with the actual dataset, at that point Action is performed. When the action is triggered after the result, new RDD is not formed like transformation.

### RDD Transformation

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more new RDD as output with changing the input RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().

After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).



## RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. E.g. count(), collect(), etc.