1. **Limitations of Map Reduce:**

There are certain cases where mapreduce is not a suitable choice:

1. Real-time processing.
2. It's not always very easy to implement each and everything as a MR program.
3. When your intermediate processes need to talk to each other(jobs run in isolation).
4. When your processing requires lot of data to be shuffled over the network.
5. When you need to handle streaming data. MR is best suited to batch process huge amounts of data which you already have with you.
6. 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.
7. When you have OLTP needs. MR is not suitable for a large number of short on-line transactions.
8. When you need a response fast. e.g. say < few seconds (Use stream processing, CEP etc instead)
9. Processing graphs
10. Complex algorithms - some machine learning algorithms like SVM, and also see 13 drawfs ([The Landscape of Parallel Computing Research: A View From Berkeley](http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf))
11. Iterations - when you need to process data again and again. e.g. KMeans - use Spark
12. When map phase generate too many keys. Then sorting takes for ever
13. Joining two large data sets with complex conditions (equal case can be handled via hashing etc)
14. Stateful operations - e.g. evaluate a state machine
15. Cascading tasks one after the other - using Hive, Pig might help, but lot of overhead rereading and parsing data.
16. **RDD and its Features:**

RDD stands for “Resilient Distributed Dataset”. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

* Resilient, i.e. fault-tolerant with the help of RDD lineage graph([DAG](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)) and so able to recompute missing or damaged partitions due to node failures.
* Distributed, since Data resides on multiple nodes.
* Dataset represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.
* Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It posses self-recovery in the case of failure.
* There are three [ways to create RDDs in Spark](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) such as – Data in stable storage, other RDDs, and parallelizing already existing collection in driver program. One can also operate Spark RDDs in parallel with a low-level API that offers transformations and actions. We will study these Spark RDD Operations later in this section.
* Spark RDD can also be cached and manually partitioned. Caching is beneficial when we use RDD several times. And manual partitioning is important to correctly balance partitions. Generally, smaller partitions allow distributing RDD data more equally, among more executors. Hence, fewer partitions make the work easy.
* Programmers can also call a persist method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs [in memory](http://data-flair.training/blogs/apache-spark-in-memory-computing/) by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist.

The key motivations behind the concept of RDD are-

* Iterative algorithms.
* Interactive data mining tools.
* DSM (Distributed Shared Memory) is a very general abstraction, but this generality makes it harder to implement in an efficient and fault tolerant manner on commodity clusters. Here the need of RDD comes into the picture.
* In distributed computing system data is stored in intermediate stable distributed store such as [HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) or Amazon S3. This makes the computation of job slower since it involves many IO operations, replications, and serializations in the process.

In first two cases we keep data in-memory, it can improve performance by an order of magnitude.

The main challenge in designing RDD is defining a program interface that provides fault tolerance efficiently. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformation rather than fine-grained updates to shared state.

Spark exposes RDD through language integrated API. In integrated API each data set is represented as an object and transformation is involved using the method of these objects.

Apache Spark evaluates RDDs lazily. It is called when needed, which saves lots of time and improves efficiency. The first time they are used in an action so that it can pipeline the transformation. Also, the programmer can call a persist method to state which RDD they want to use in future operations.

Features of RDD:

1. In-memory Computation

Spark RDDs have a provision of [in-memory computation](http://data-flair.training/blogs/apache-spark-in-memory-computing/). It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program. Follow this guide for the deep study of[Spark Lazy Evaluation.](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/)

3. Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself. Follow this guide for the deep study of[RDD Fault Tolerance.](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/)

4. Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5. Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

6. Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

7. Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

8. Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation.

1. **Spark RDD operations:**

RDD in Apache Spark supports two types of operations:

Transformation

Actions

**3.1. 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, wide transformation.

**3.1.1. Narrow Transformations**

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**.

**3.1.2. 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.

**3.2. 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.