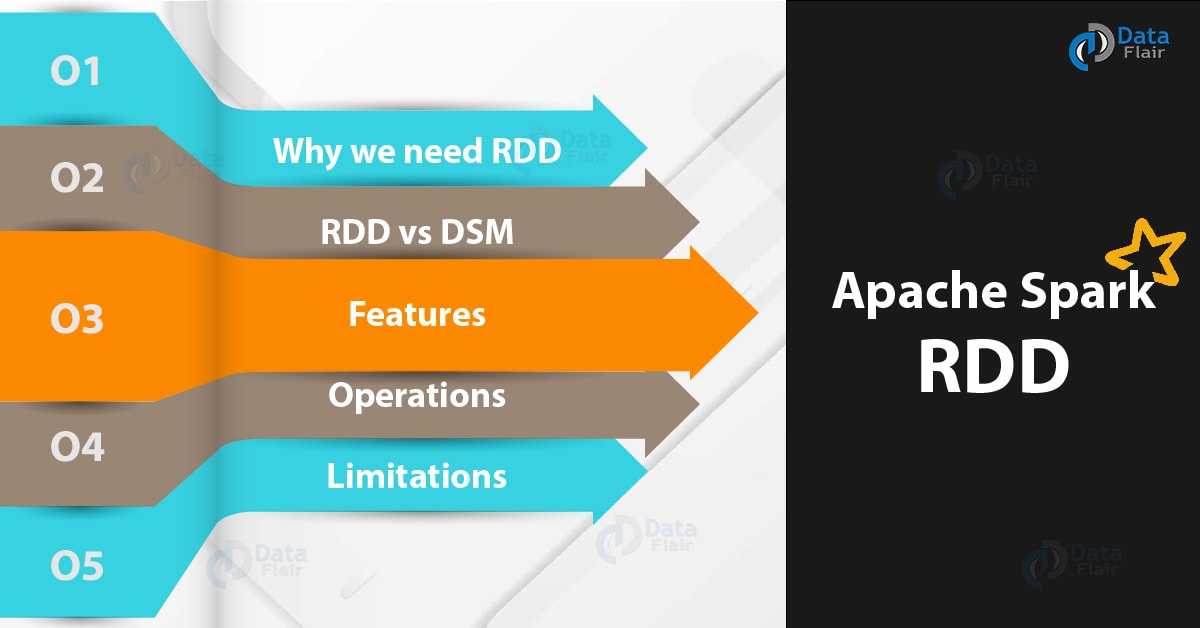
Spark RDD – Introduction, Features & Operations of RDD

1. Objective – Spark RDD

**RDD (Resilient Distributed Dataset)** is the fundamental data structure of [**Apache Spark**](http://data-flair.training/blogs/introduction-spark-tutorial-quickstart/) which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

In this blog, we are going to get to know about what is RDD in Apache Spark. What are the features of RDD, What is the motivation behind RDDs, RDD vs DSM? We will also cover Spark RDD operation i.e. transformations and actions, various limitations of RDD in Spark and how RDD make Spark feature rich in this Spark tutorial.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2016/07/Apache-Spark-RDD-01.jpg)

*Spark RDD – Introduction, Features & Operations of RDD*

2. What is Apache Spark RDD?

**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**](https://techvidvan.com/tutorials/ways-to-create-rdd-in-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.

3. Why do we need RDD in Spark?

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.

4. Spark RDD vs DSM (Distributed Shared Memory)

In this Spark RDD tutorial, we are going to get to know the difference between RDD and DSM which will take RDD in Apache Spark into the limelight.  
**i. Read**

* **RDD –**The read operation in RDD is either coarse grained or fine grained. Coarse-grained meaning, we can transform the whole dataset but not an individual element on the dataset. While fine-grained means we can transform individual element on the dataset.
* **DSM –** The read operation in Distributed shared memory is fine-grained.

**ii. Write**

* **RDD –**The write operation in RDD is coarse grained.
* **DSM –** The Write operation is fine grained in distributed shared system.

**iii. Consistency**

* **RDD –**The consistency of RDD is trivial meaning it is immutable in nature. Any changes on RDD is permanent i.e we can not realtor the content of RDD. So the level of consistency is high.
* **DSM –** In Distributed Shared Memory the system guarantees that if the programmer follows the rules, the memory will be consistent and the results of memory operations will be predictable.

**iv. Fault-Recovery Mechanism**

* **RDD –**The lost data can be easily recovered in Spark RDD using lineage graph at any moment. Since for each transformation, new RDD is formed and RDDs are immutable in nature so it is easy to recover.
* **DSM –** Fault tolerance is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting.

**v. Straggler Mitigation**

Stragglers, in general, are those that take more time to complete than their peers. This could happen due to many reasons such as load imbalance, I/O blocks, garbage collections, etc.  
The problem with stragglers is that when the parallel computation is followed by synchronizations such as reductions. This would cause all the parallel tasks to wait for others.

* **RDD –** In RDD it is possible to mitigate stragglers using backup task.
* **DSM –**It is quite difficult to achieve straggler mitigation.

**vi. Behavior if not enough RAM**

* **RDD –**If there is not enough space to store RDD in RAM then the RDDs are shifted to disk**.**
* **DSM –** In this type of system, the performance decreases if the RAM runs out of storage.

5. Features of Spark RDD

Several features of Apache Spark RDD are:

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/08/features-of-RDD-in-spark.jpg)

*Features of Spark RDD*

**i. In-memory Computation**

SparkRDDs 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).

**ii. 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/)

**iii. 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/)

**iv. 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.

**v. 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.

**vi. Persistence**

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

**vii. Coarse-grained Operations**

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

**viii. 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. Follow this guide to [learn What is DAG?](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)

6. Spark RDD Operations

RDD in Apache Spark supports two types of operations:

* Transformation
* Actions

i. Transformations

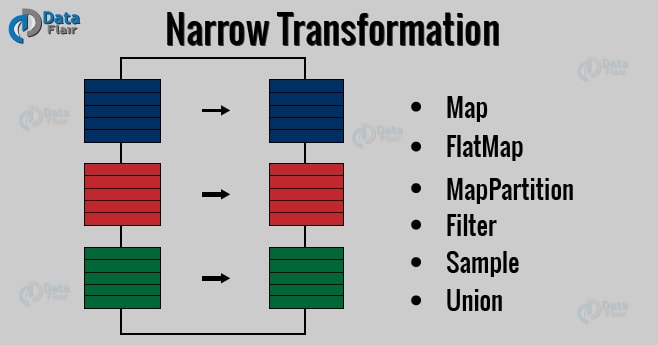
Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g.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

**a. 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.

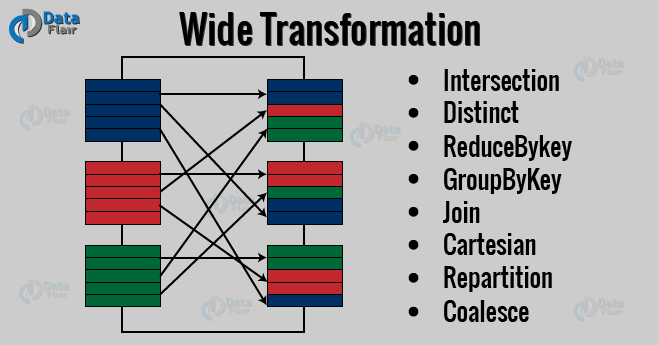
Spark groups narrow transformations as a stage known as **pipelining**.

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/08/spark-narrow-transformation-1.jpg)

*Spark RDD – Narrow Transformation*

b. 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/sites/2/2017/08/spark-wide-transformation.jpg)

*Wide Transformation*

ii. 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.

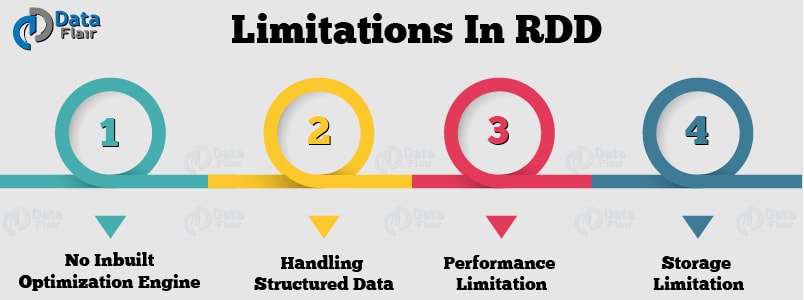
**Actions** are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.

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.

Follow this link to[**learn RDD Transformations and Actions APIs with examples.**](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/)

7. Limitation of Spark RDD

There is also some limitation of Apache Spark RDD. Let’s discuss them one by one-

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2017/08/limitations-of-rdd-1.jpg)

*Limitations of Apache Spark RDD*

**i. No inbuilt optimization engine**

When working with structured data, RDDs cannot take advantages of Spark’s advanced optimizers including [**catalyst optimizer**](http://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/) and**Tungsten execution engine**. Developers need to optimize each RDD based on its attributes.

**ii. Handling structured data**

Unlike **[Dataframe](http://data-flair.training/blogs/apache-spark-sql-dataframe-tutorial/)** and datasets, RDDs don’t infer the schema of the ingested data and requires the user to specify it.

**iii. Performance limitation**

Being in-memory JVM objects, RDDs involve the overhead of Garbage Collection and Java Serialization which are expensive when data grows.

**iv. Storage limitation**

RDDs degrade when there is not enough memory to store them. One can also store that partition of RDD on disk which does not fit in RAM. As a result, it will provide similar performance to current data-parallel systems.

So, this was all in Spark RDD Tutorial. Hope you like our explanation

8. Conclusion – Spark RDD

In conclusion to RDD, the shortcomings of Hadoop MapReduce was so high. Hence, it was overcome by Spark RDD by introducing in-memory processing, immutability etc. But there were some limitations of RDD. For example No inbuilt optimization, storage and performance limitation etc.  
Because of the above-stated limitations of RDD to make spark more versatile DataFrame and Dataset evolved.