TASK 2:

Limitations Of Map Reduce:

Map Reduce is not a suitable choice:

1.) For real time processing.
2.) When intermediate processes need to interact with each other, since jobs run in isolation.
3.) When your processing requires lot of data to be **shuffled** over the network.
4.) When you need to handle streaming data. MR is best suited to batch process huge amounts of data which you already have with you.
5.) 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.
6.) When you have OLTP needs. MR is not suitable for a large number of short on-line transactions.

MapReduce does not work very well with:

1.) When you need a response fast. e.g. say < few seconds (Use stream processing, CEP etc instead)
2.) Processing graphs
3.) Complex algorithms e.g. some machine learning algorithms like SVM
4.) Iterations - when you need to process data again and again. e.g. KMeans - use Spark
5.) When map phase generate too many keys.
6.) Joining two large data sets with complex conditions (equal case can be handled via hashing etc)
7.) Stateful operations - e.g. evaluate a state machine Cascading tasks one after the other - using Hive, Big might help, but lot of overhead rereading and parsing data.

RDD & Features of 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) 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 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 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.

Features of RDD:

Several features of Apache Spark RDD are:

Features of Spark RDD

1. In-memory Computation

Spark RDDs have a provision of in-memory computation. 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.

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.

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.

Spark RDD Operations

RDD in Apache Spark supports two types of operations:

Transformation

Actions

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.

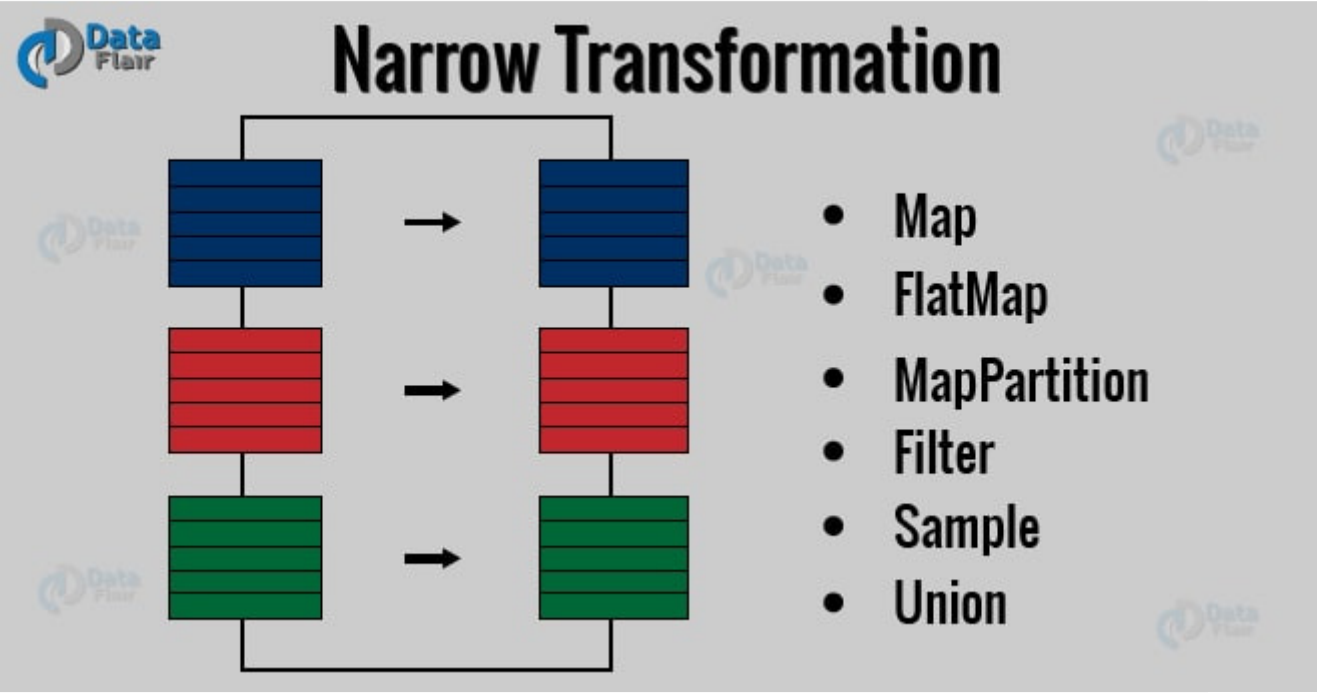
Transformations are lazy operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

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.

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.

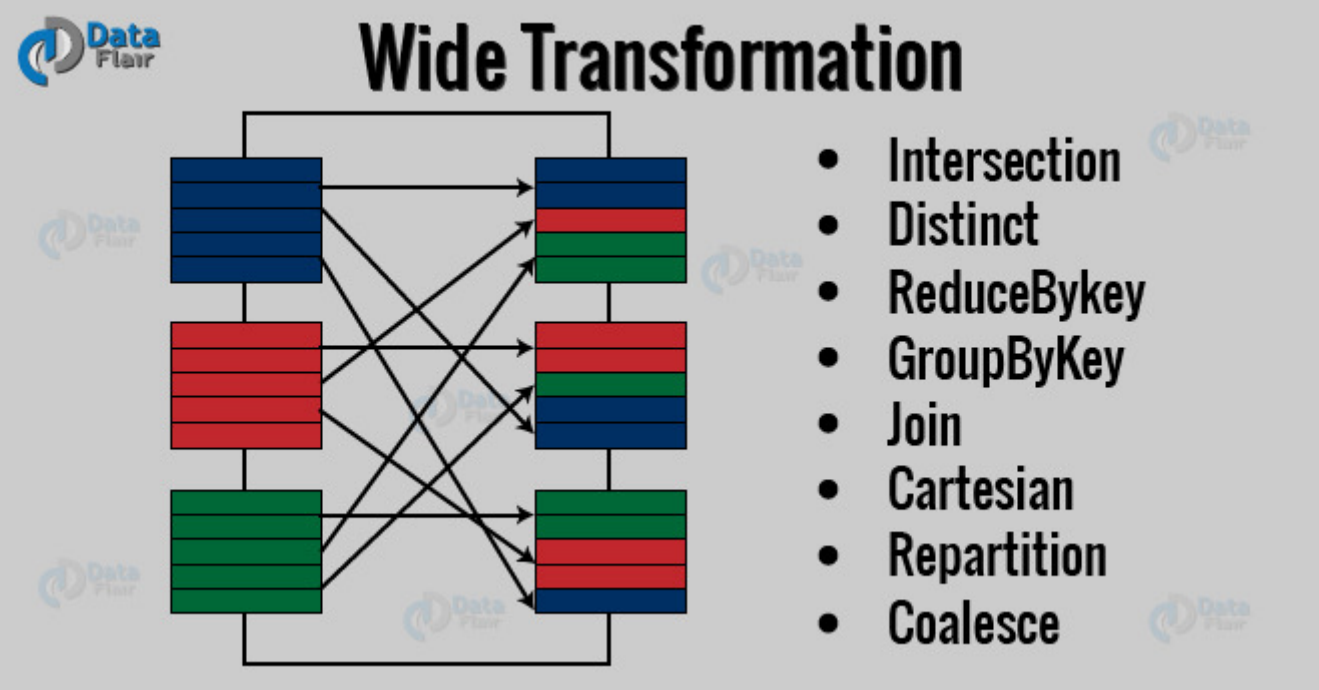
Spark groups narrow transformations as a stage known as pipelining.



Spark RDD – Narrow Transformation

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.



Wide Transformation

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.