**Spark Fundamentals and Architecture:**

**What is Apache Spark?**

Apache Spark is an Open Source Big Data framework. It is faster and more general purpose data processing engine and is basically designed for fast computation. It covers a wide range of workloads such as batch, interactive, iterative and streaming.

**What is Hadoop?**

Hadoop is an Open Source framework for writing applications which processes structured and unstructured data that are stored in HDFS. Hadoop MapReduce is designed in a way to process a large volume of data on a cluster of commodity hardware. MapReduce can process data in batch mode.

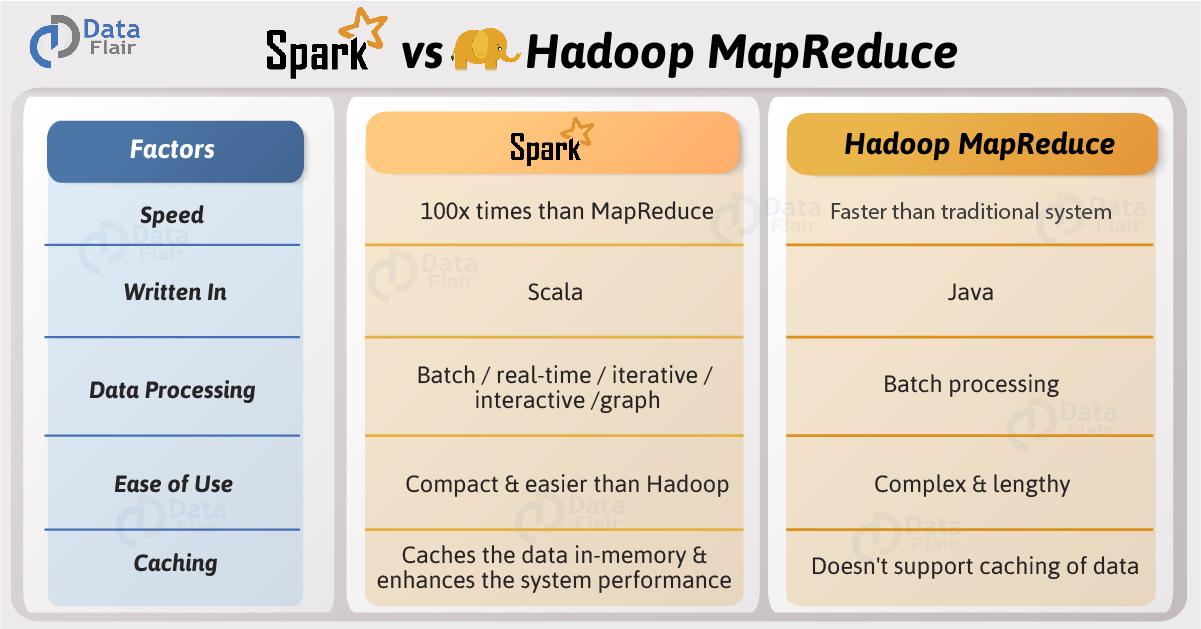
**How is Apache Spark better than Hadoop ?**

Apache Spark is a lightning fast cluster computing tool. Spark runs applications in Hadoop clusters up to 100x faster in memory and 10x faster on disk.

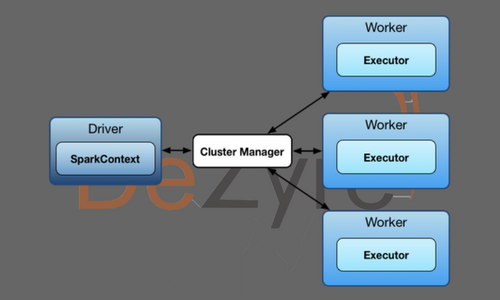
Spark makes it possible by reducing the number of read/write cycle to disk and storing intermediate data in-memory.

But, MapReduce reads and writes from disk and that slows down the processing speed.

**Spark Vs Mapreduce:** <https://data-flair.training/blogs/spark-vs-hadoop-mapreduce/>



**Spark Architecture:**



Apache Spark has a well-defined and layered architecture where all the spark components and layers are loosely coupled and integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions-

* *Resilient Distributed Datasets (RDD)*
* *Directed Acyclic Graph (DAG)*

### **Resilient Distributed Datasets (RDD)**

RDD’s are collection of data items that are split into partitions and can be stored in-memory on workers nodes of the spark cluster. In terms of datasets, apache spark supports two types of RDD’s – Hadoop Datasets which are created from the files stored on HDFS and parallelized collections which are based on existing Scala collections. Spark RDD’s support two different types of operations – Transformations and Actions.

### Directed Acyclic Graph (DAG)

Direct - Transformation is an action which transitions data partition state from A to B.

Acyclic -Transformation cannot return to the older partition

DAG is a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data. The DAG abstraction helps eliminate the Hadoop MapReduce multi stage execution model and provides performance enhancements over Hadoop.

## Spark Architecture Overview

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager –

1. Master Daemon – (Master/Driver Process)
2. Worker Daemon –(Slave Process)

A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal spark cluster or on separate machines i.e. in a vertical spark cluster or in mixed machine configuration.

### Role of Driver in Spark Architecture

Spark Driver – Master Node of a Spark Application

It is the central point and the entry point of the Spark Shell (Scala, Python, and R). The driver program runs the main () function of the application and is the place where the Spark Context is created. Spark Driver contains various components – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager responsible for the translation of spark user code into actual spark jobs executed on the cluster.

* The driver program that runs on the master node of the spark cluster schedules the job execution and negotiates with the cluster manager.
* It translates the RDD’s into the execution graph and splits the graph into multiple stages.
* Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
* Cockpits of Jobs and Tasks Execution -Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
* Driver exposes the information about the running spark application through a Web UI at port 4040.

### Role of Executor in Spark Architecture

Executor is a distributed agent responsible for the execution of tasks. Every spark application has its own executor process. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as “Static Allocation of Executors”. However, users can also opt for dynamic allocations of executors wherein they can add or remove spark executors dynamically to match with the overall workload.

* Executor performs all the data processing.
* Reads from and Writes data to external sources.
* Executor stores the computation results data in-memory, cache or on hard disk drives.
* Interacts with the storage systems.

# How to enable Dynamic Resource Allocation in Spark

Dynamic allocation allows Spark to, dynamically scale the cluster resources allocated for the Spark application. When dynamic allocation is enabled, if there are backlog of pending tasks for a Spark application, it can request for new executors. When the application becomes idle, its executors are released and the same can be acquired by other spark applications. To enable Dynamic allocation for Spark, following steps could be used:

1. In Ambari Spark-Configs, edit the Custom spark-defaults section and add the following parameters:

spark.dynamicAllocation.enabled = true spark.shuffle.service.enabled = true spark.dynamicAllocation.initialExecutors = 3 (Initial number of executors to run if dynamic allocation is enabled, this is same as "spark.dynamicAllocation.minExecutors") spark.dynamicAllocation.minExecutors = 3 (executors number will come to this number if executors are not in use, after 60 sec(default), controlled by "spark.dynamicAllocation. executorIdleTimeout") spark.dynamicAllocation.maxExecutors = 30 (maximum executors that job can request)

2. Restart Spark services using Ambari

3. Validate Dynamic allocation by running a sample job, for example,

pyspark --master yarn

Since no executor specification is specified at run time, the job would start with the settings completed in Ambari. As defined in our example setting, it would start with 3 executors. If the job needs more executor, it would request for the same and the following messages would be seen on the console:

6/05/23 09:39:47 INFO ExecutorAllocationManager: Requesting 2 new executors because tasks are backlogged (new desired total will be 4) 16/05/23 09:39:48 INFO ExecutorAllocationManager: Requesting 1 new executor because tasks are backlogged (new desired total will be 5)

If executors are specified while running the job, Dynamic allocation would be disabled. For example,

pyspark --master yarn --num-executors 50 --executor-memory 3G

The above program would display the following warning on the console:

16/05/23 09:18:54 WARN SparkContext: Dynamic Allocation and num executors both set, thus dynamic allocation disabled.

### Role of Cluster Manager in Spark Architecture

An external service responsible for acquiring resources on the spark cluster and allocating them to a spark job. There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager either of them can be launched on-premise or in the cloud for a spark application to run.

Choosing a cluster manager for any spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities. To get started with apache spark, the standalone cluster manager is the easiest one to use when developing a new spark application.

## **Understanding the Run Time Architecture of a Spark Application**

### What happens when a Spark Job is submitted?

When a client submits a spark user application code, the driver implicitly converts the code containing transformations and actions into a logical directed acyclic graph (DAG). At this stage, the driver program also performs certain optimizations like pipelining transformations and then it converts the logical DAG into physical execution plan with set of stages. After creating the physical execution plan, it creates small physical execution units referred to as tasks under each stage. Then tasks are bundled to be sent to the Spark Cluster.

The driver program then talks to the cluster manager and negotiates for resources. **The cluster manager then launches executors on the worker nodes on behalf of the driver**. At this point the driver sends tasks to the cluster manager based on data placement. Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors. Now executors start executing the various tasks assigned by the driver program. At any point of time when the spark application is running, the driver program will monitor the set of executors that run. Driver program in the spark architecture also schedules future tasks based on data placement by tracking the location of cached data. When driver programs main () method exits or when it call the stop () method of the Spark Context, it will terminate all the executors and release the resources from the cluster manager.

The structure of a Spark program at higher level is - RDD's are created from the input data and new RDD's are derived from the existing RDD's using different transformations, after which an action is performed on the data. In any spark program, the DAG operations are created by default and whenever the driver runs the Spark DAG will be converted into a physical execution plan.

### Launching a Spark Program

spark-submit is the single script used to submit a spark program and launches the application on the cluster. There are multiple options through which spark-submit script can connect with different cluster managers and control on the number of resources the application gets. For few cluster managers, spark-submit can run the driver within the cluster like in YARN on worker node whilst for others it runs only on local machines.

**Further Resource for Architecture:**

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-architecture.html

# What is MapReduce? How it Works - Hadoop MapReduce Tutorial

MapReduce is a programming model suitable for processing of huge data. Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

MapReduce programs work in two phases:

1. Map phase
2. Reduce phase.

Input to each phase are key-value pairs. In addition, every programmer needs to specify two functions: map function and reduce function.

The whole process goes through three phase of execution namely,

## How MapReduce works

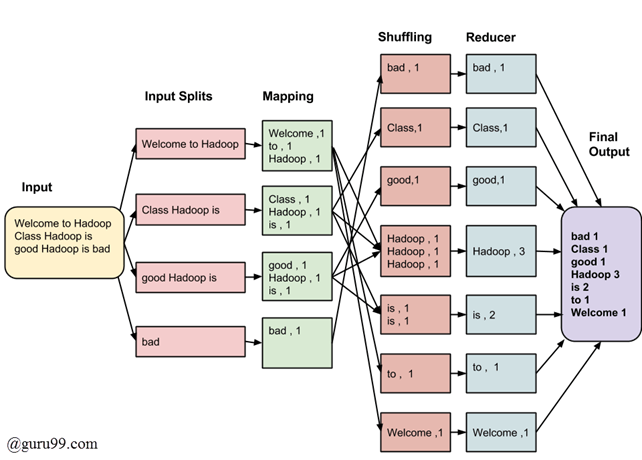
Lets understand this with an example –

Consider you have following input data for your MapReduce Program:

*Welcome to Hadoop Class*

*Hadoop is good*

*Hadoop is bad*



The data goes through following phases

Input Splits:

Input to a MapReduce job is divided into fixed-size pieces called input splits Input split is a chunk of the input that is consumed by a single map

Mapping

This is very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, job of mapping phase is to count number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling

This phase consumes output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, same words are clubed together along with their respective frequency.

Reducing

In this phase, output values from Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each words.

## The overall process in detail

* One map task is created for each split which then executes map function for each record in the split.
* It is always beneficial to have multiple splits, because time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better load balanced since we are processing the splits in parallel.
* However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
* For most jobs, it is better to make split size equal to the size of an HDFS block (which is 64 MB, by default).
* Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
* Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
* Map output is intermediate output which is processed by reduce tasks to produce the final output.
* Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
* In the event of node failure before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
* Reduce task don't work on the concept of data locality. Output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
* On this machine the output is merged and then passed to the user defined reduce function.
* Unlike to the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

## How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

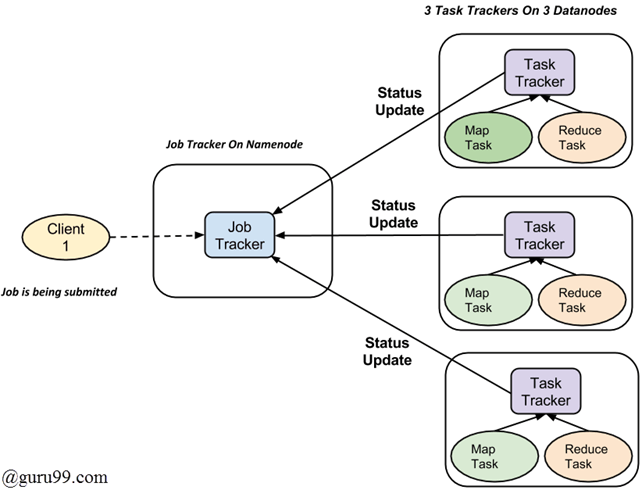
1. Map tasks (Spilts & Mapping)
2. Reduce tasks (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. Jobtracker : Acts like a master (responsible for complete execution of submitted job)
2. Multiple Task Trackers : Acts like slaves, each of them performing the job

For every job submitted for execution in the system, there is one Jobtracker that resides on Namenode and there are multiple tasktrackers which reside on Datanode.



* A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
* It is the responsibility of jobtracker to coordinate the activity by scheduling tasks to run on different data nodes.
* Execution of individual task is then look after by tasktracker, which resides on every data node executing part of the job.
* Tasktracker's responsibility is to send the progress report to the jobtracker.
* In addition, tasktracker periodically sends 'heartbeat' signal to the Jobtracker so as to notify him of current state of the system.
* Thus jobtracker keeps track of overall progress of each job. In the event of task failure, the jobtracker can reschedule it on a different tasktracker.

**Spark Fundamental Concepts**

sparkContext vs sparkConf

Prior to spark 2.0.0

sparkContextwas used as a channel to access all spark functionality.  
The spark driver program uses spark context to connect to the cluster through a resource manager (YARN orMesos..).

sparkConfis required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node

In order to use APIs of SQL,HIVE , and Streaming, separate contexts need to be created.

like

conf = SparkConf()

sc = SparkContext(conf)

hc = hiveContext(sc)

ssc = streamingContext(sc).

SPARK 2.0.0 onwards

SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with Dataframe and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession.

In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.

Once the SparkSession is instantiated, we can configure Spark’s run-time config properties

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('abc').getOrCreate()

Reasons behind Immutability of Spark RDD

1. It is easy to share the Immutable data safely among several process. Basically, due to updates from multiple threads at once, Immutability rules out a big set of potential problems.
2. Immutable data can as easily live on memory as on disk. This makes it easy move operations from the that hit disk to instead use data in memory. adding memory is much easier then adding i/o bandwidth.
3. Basically, RDDs are not just immutable but also deterministic function of their input. That means RDD can be recreated at any time. It helps in leverages the advantage of caching, sharing and replication. It isn’t really a collection of data but also a way of making data from other data.
4. If the computation is time-consuming, in that we can cache the RDD which result in performance improvement.

**RDD Properties:**

RDDs have the following properties –

1. Immutability and partitioning: RDDs composed of collection of records which are partitioned. Partition is basic unit of parallelism in a RDD, and each partition is one logical division of data which is immutable and created through some transformations on existing partitions.Immutability helps to achieve consistency in computations.

Users can define their own criteria for partitioning based on keys on which they want to join multiple datasets if needed.

1. Coarse grained operations: Coarse grained operations are operations which are applied to all elements in datasets. For example – a map, or filter or groupBy operation which will be performed on all elements in a partition of RDD.

3. Fault tolerance: Since RDDs are created over a set of transformations , it logs those transformations, rather than actual data.Graph of these transformations to produce one RDD is called as Lineage Graph.

For example –

firstRDD=sc.textFile("hdfs://...")

secondRDD=firstRDD.filter(someFunction);

thirdRDD = secondRDD.map(someFunction);

result = thirdRDD.count()

In case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation, rather than doing data replication across multiple nodes.This characteristic is biggest benefit of RDD , because it saves a lot of efforts in data management and replication and thus achieves faster computations.

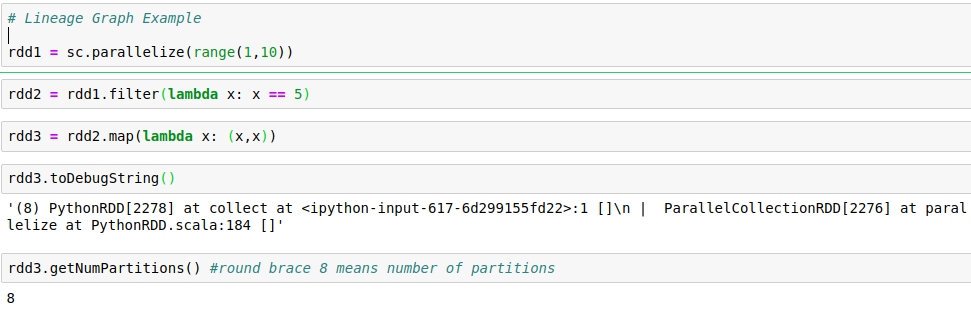
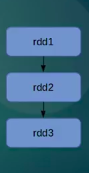
1. Lazy evaluations: Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations. So , in above example RDD will be evaluated only when count() action is invoked.
2. Persistence: Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk etc.)

These properties of RDDs make them useful for fast computations.

**What is lineage graph in Apache Spark:**

Spark lineage is set of steps that are required to generate RDD. In simple words spark lineage is set of transformation will apply on different RDD to create the final RDD.

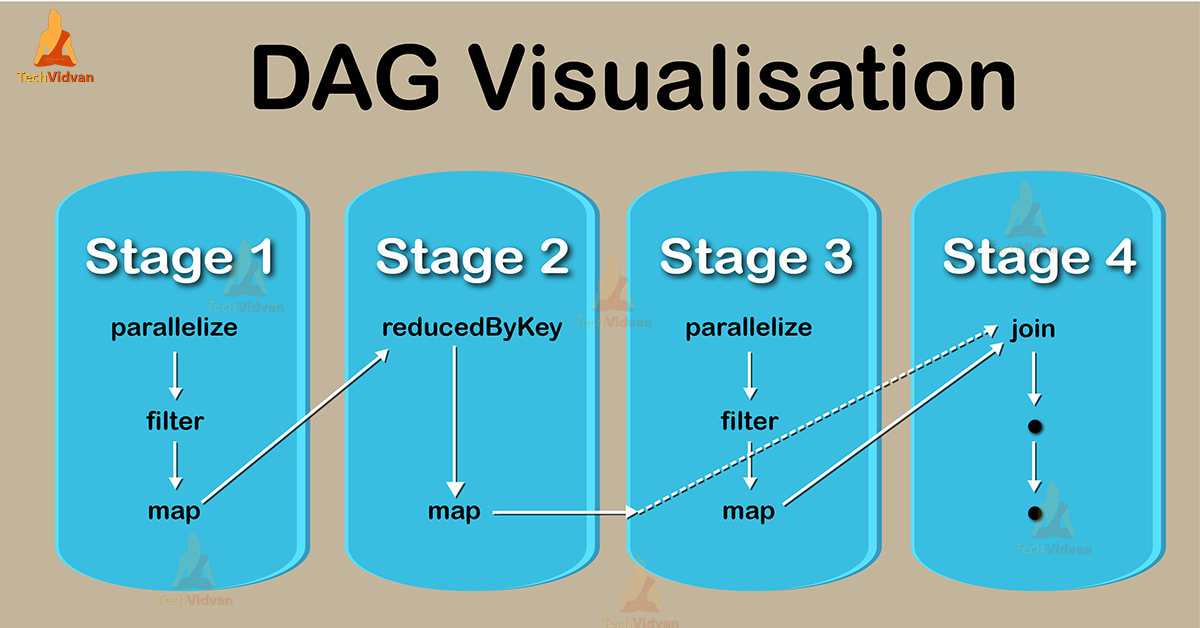
Lineage graph is the logical plan will tell how the RDD is going to create. Eg.



By using toDebugString method we will get to know the logical planes of the rdd.

**What is DAG?**

It is a Direct Acyclic graph of stages. We will apply multiple transformations on RDD and then we create the final RDD. When we call the action, then that logical plan(lineage graph) will submitted to catalyst optimizer. This catalyst optimizer will generate the final plan then this plan is given to DAG schedular. Dag scheduler will create the physical plan. Finally it will divide whole execution into stages. As part of stage it will identify what are the tasks can happen on the same machine and what are the tasks for which the shuffle is required and which can be run in parallel, etc. Based on this information it will create DAG of stages, and then DAG will execute.

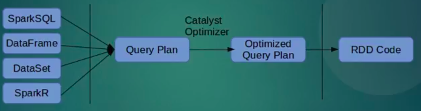


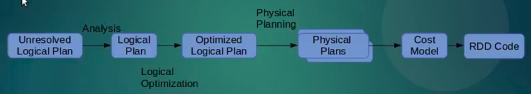
**Difference of DAG and Lineage Graph**

Lineage graph is the logical plan, DAG is the physical plan. DAG has more information and it knows how the stages will execute. It is more from the implementation and scheduling point of view. Lineage graph is more from dependency point of view. It will tell if we want to recreate RDD from which RDD you can create and what are the transformation need to apply on RDD

**Catelyst Optimizer:**

Catelyst Optimizer is one of the tree tansformation frame work. For our code tree will be created and this tree will goes to various phases and finally new tree will be created as the output, and from that code java code will be generated for our business logic.





**Catelyst Optimizer phases:**

1) Analysing Logical Plan to resolve references: rule based.

2) Logical Plan Optimization: rule based

3) Physical planning(it may generate many plans): Cost based

4) Code Generation: Rule bases

Each phase uses different types of tree nodes

Hence, Catelyst library is a library of nodes

1) Expression nodes

2) Node for Data types

3) Node for logical operator

4) Node for physical operation(mul, sub, add, etc)

**Analysis phase: Rule Based**

It will do Syntax check, written column is present are not by using catalog object. If anything is missing or wrong will throw an error.

So from unresoved plan tree to transform resolved plan tree, if the code is correct.

**Logical Plan Optimization: Rule Based**

After creating correct tree the code needs to be optimized. In this phase optimization will happen.

**Physical Plan: Cost based optimizer**

It will take optimized logical plan, and it will find best way to get the optimized solution. Based on cost based model. Example using broadcast variable to optimize join.

**Code Generation:**

Generate Java code to run on JVM. Finally it will create optimized RDD.

**ORC and Parquet similiarity:**

ORC and Parquet are column oriented format.

In column oriented format columns of same types reside adjacent to each other that results in better compression. Also on querying for specific columns processing is fast as you can directly fetch that column data with out having to fetch the whole row in memory.

**ORC vs Parquet:**

* Parquet is good for nested data(tree based). Because it stores the data in that fashion.
* ORC supports **predicate pushdown** efficiency and **ACID** properties. Predicate pushdown means while running the query it will easily optimise to check whether the particular data is in particular column or not. So that it can skip that partition. Due to ACID advantage provides good support for streaming data.
* ORC Compresses sizes upto 75%.

**Lazy Evaluation:**

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.

Transformations are lazy in nature meaning when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called(Through [DAG](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)).We can think Spark RDD as the data, that we built up through transformation. Since transformations are lazy in nature, so we can execute operation any time by calling an action on data. Hence, in lazy evaluation data is not loaded until it is necessary.

## Advantages of Lazy Evaluation in Spark Transformation

There are some benefits of Lazy evaluation in Apache Spark-

### a. Increases Manageability

By lazy evaluation, users can organize their Apache Spark Program. It reduces the number of passes on data by grouping operations.

### b. Saves Computation and increases Speed

Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get compute. It saves the trip between driver and cluster, thus speeds up the process.

### c. Reduces Complexities

The two main complexities of any operation are time and space complexity. Using Apache Spark lazy evaluation we can overcome both. Since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.

### d. Optimization

It provides optimization by reducing the number of queries. Learn more about[Apache Spark Optimization.](http://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/)

## 4. Conclusion

Hence, Lazy evaluation enhances the power of Apache Spark by reducing the execution time of the [RDD operations](http://data-flair.training/blogs/apache-spark-rdd-transformations-actions/). It maintains the lineage graph to remember the operations on RDD. As a result, it Optimizes the performance and achieves [rance](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/).

**What is partition in respect to file system in Spark ? Default value ? How to change it and why ?**

When Spark reads a file from HDFS, it creates a single partition for a single input split. Input split is set by the Hadoop InputFormat used to read this file. For instance, if you use textFile() it would be TextInputFormat in Hadoop, which would return you a single partition for a single block of HDFS (but the split between partitions would be done on line split, not the exact block split), unless you have a compressed text file. In case of compressed file you would get a single partition for a single file (as compressed text files are not splittable).

When you call rdd.repartition(x) it would perform a shuffle of the data from N partititons you have in rdd to x partitions you want to have, partitioning would be done on round robin basis.

When you call rdd.coalesce(x). It will help us reduce the number of partition without full suffeling. Efficient for reducing partition size.

If you have a 30GB uncompressed text file stored on HDFS, then with the default HDFS block size setting (128MB) it would be stored in 235 blocks, which means that the RDD you read from this file would have 235 partitions. When you call repartition(1000) your RDD would be marked as to be repartitioned, but in fact it would be shuffled to 1000 partitions only when you will execute an action on top of this RDD (lazy execution concept)

1. **Broadcast variable and accumulator**

**Broadcast variable:**

Broadcast variables are pretty simple in concept. They're variables that we want to share throughout our cluster. However there are a couple of caveats that are important to understand. Broadcast variables have to be able to fit in memory on one machine. That means that they definitely should NOT be anything super large, like a large table or massive vector. Secondly, broadcast variables are immutable, meaning that they cannot be changed later on. This may seem inconvenient but it truly suits their use case. If you need something that can change, I'd certainly point you to accumulators which will be covered in another post. So now

**we know that broadcast variables are:**

1. Immutable
2. Distributed to the cluster
3. Fit in memory

val hoods = Seq((1, "Mission"), (2, "SOMA"), (3, "Sunset"), (4, "Haight Ashbury"))

val checkins = Seq((234, 1),(567, 2), (234, 3), (532, 2), (234, 4))

val hoodsRdd = sc.parallelize(hoods)

val checkRdd = sc.parallelize(checkins)

Now that we've set those up, we need to broadcast the first table.

val broadcastedHoods = sc.broadcast(hoodsRdd.collectAsMap())

Now that that's our there across our cluster, let's go ahead and join the two!

val checkinsWithHoods = checkRdd.mapPartitions({row =>

row.map(x => (x.\_1, x.\_2, broadcastedHoods.value.getOrElse(x.\_2, -1)))

}, preservesPartitioning = true)

checkinsWithHoods.take(5)

// res3: Array[(Int, Int, Any)] = // Array((234,1,Mission), (567,2,SOMA), (234,3,Sunset), (532,2,SOMA), (234,4,Haight Ashbury))

You may have noticed that whole "preserve partitioning argument and that's to prevent the shuffle of data!

preservesPartitioning indicates whether the input function preserves the partitioner, which should be false unless this is a pair RDD and the input function doesn't modify the keys.

**Accumulator:**

As you might assume from the name, Accumulators are variables which may be added to through associated operations. There are many uses for accumulators including implementing counters or sums. Spark supports the accumulation of numeric types easily, but programmers can add support for other types. If there is a particular name for an accumulator in code, it is usually displayed in the Spark UI, which will be useful in understanding the running stage progress.

Accumulators are created from an initial value v; i.e. SparkContext.accumulator(v). Then the tasks running in the cluster can be added to it using the known “add method” or += operator in Scala. They cannot, however, read the value of it. The driver program has the ability to read the value of the accumulator, using the value method as shown below

scala> val accum = sc.accumulator(0, "Accumulator Example")

accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3)).foreach(x => accum += x)

scala> accum.value

res4: Int = 6

**How to Spark Submit with Yarn or Mesos as resource manager**

*# Run on a YARN cluster*

export HADOOP\_CONF\_DIR=XXX

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master yarn **\**

--deploy-mode cluster **\**  *# can be client for client mode*

--executor-memory 20G **\**

--num-executors 50 **\**

/path/to/examples.jar **\**

1000

*# Run on a Mesos cluster in cluster deploy mode with supervise*

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master mesos://207.184.161.138:7077 **\**

--deploy-mode cluster **\**

--supervise **\**

--executor-memory 20G **\**

--total-executor-cores 100 **\**

http://path/to/examples.jar **\**

1000

*# Run a Python application on a Spark standalone cluster*

./bin/spark-submit **\**

--master spark://207.184.161.138:7077 **\**

examples/src/main/python/pi.py **\**

1000