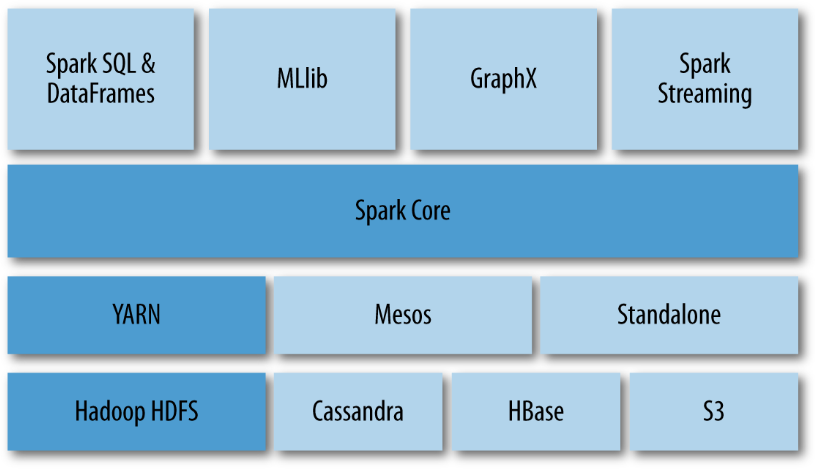
Apache Spark Core 2.3.0

# Spark Definition:

Spark is an open source parallel processing framework for running large-scale data analytics applications across clustered computers. It can handle both batch and real-time analytics and data processing workloads. It provides high-level APIs in Java, Scala, Python and R and an optimized engine that supports stream.

# Spark Libraries

1. **Spark Core:** Spark Core engine functions partly as an application programming interface (API) layer and underpins a set of related tools for managing and analyzing data.
2. **Spark SQL:** Spark SQL enables users to query data stored in disparate applications using the common SQL language.
3. **Spark Streaming:** Spark Streaming enables users to build applications that analyze and present data in real time.
4. **MLlib:** MLlib is a library of machine learning code that enables users to apply advanced statistical operations to data in their Spark cluster and to build applications around these analyses.
5. **GraphX:** A built-in library of algorithms for graph-parallel computation.



# Spark Features:

1. Swift Processing: 100x faster in memory and 10x faster on disk
2. Reusability
3. Dynamic in nature
4. In-memory computation
5. Near real-time stream processing
6. Fault tolerance
7. Lazy evaluation
8. Supports Multiple languages
9. Community support
10. Cost Effective
11. GraphX: simplifies the graph analytics tasks by the collection of graph algorithm and builders

# Spark Disadvantages:

1. No real-time processing
2. Problem with small files: In RDD, each file is a small partition, which means there are a number of tiny partitions within an RDD. Hence, if we want efficiency in our processing, the RDDs should be repartitioned into some manageable format. Basically, that demands extensive shuffling over the network.
3. No file management system
4. Less number of algorithms in MLlib
5. Latency: In comparison with Flink, Spark has higher latency
6. Manual Optimization: It is must that Spark job is manually optimized and is adequate to specific datasets. Moreover, to partition and cache in spark to be correct, it is must to control it manually.
7. Iterative processing: Data processing is in batches - each iteration is scheduled and processed separately.

# Spark Terminologies

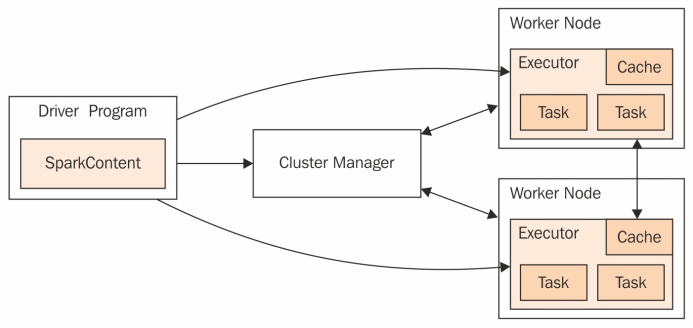
1. Spark Context
2. Spark Shell
3. Spark Application
4. Spark Driver
5. Spark Cluster Manager
6. Spark Executors
7. **Task:** Task is a unit of work that is sent to the executor. Each stage has some task, one task per partition. Same tasks are done over different partitions of RDD.
8. **Job:** Job is parallel computation consisting of multiple tasks that get spawned in response to [actions in Apache Spark.](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/)
9. **Stage:** Each job gets divided into smaller sets of tasks called **stages** that depend on each other. Stages are classified as computational boundaries. All computation cannot be done in single stage. It is achieved over many stages.

# SparkContext

SparkContext allows Spark application to access Spark Cluster with the help of the resource manager (i.e. Standalone, YARN or Mesos). SparkContext is created by passing SparkConf object which has configuration parameters that driver program passes to the context.

## Functions:

1. Gets current status of the application:
   1. **SpkEnv:** SparkEnv is the Spark's runtime environment for Spark's public services. It interacts with each other to establish a distributed computing platform for Spark Application. A SparkEnv object that holds the required runtime services for [running Spark application](http://data-flair.training/blogs/how-apache-spark-works-run-time-spark-architecture/) with the different environment for the driver and executor represents the Spark runtime environment.
   2. SparkConf
   3. Deployment Environment (local or cluster mode)
2. To set the configuration
3. To access various services: helps in accessing services like TaskScheduler, LiveListenBus, BlockManager, SchedulerBackend, ShuffelManager and the optional ContextCleaner.
4. To cancel a job: Refer DAG
5. To cancel a stage: request *DAGScheduler* to drop Stark stage
6. For closure cleaning in Spark
7. To register Spark listener
8. Programmable dynamic allocation
9. To access persistent RDD
10. To un-persist RDDs



# Resilient Distributed Datasets:

Spark is based on Resilient Distributed Datasets (RDDs) which is a fault tolerant collection of elements that can be operated on in parallel.

## Features of RDDs:

1. Immutability
2. Fault tolerance
3. Lazy evaluation

## RDD Lineage

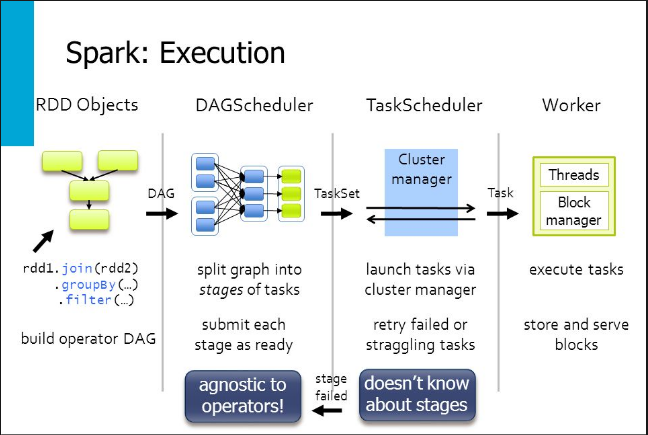
1. Evaluation of RDD is lazy in nature. It means a series of transformations are performed on an RDD, which is not evaluated immediately.
2. While [a new RDD](https://data-flair.training/blogs/create-rdds-in-apache-spark/) is created from an existing Spark RDD, that new RDD carries a pointer to the parent RDD. Thus, RDD dependency graph is formed which is called **lineage graph** or RDD operator graph or RDD dependency graph. This is an output of applying transformations to the spark.
3. Then a logical execution plan is created. Physical execution plan or execution [DAG](https://data-flair.training/blogs/dag-in-apache-spark/) is known as DAG of stages.

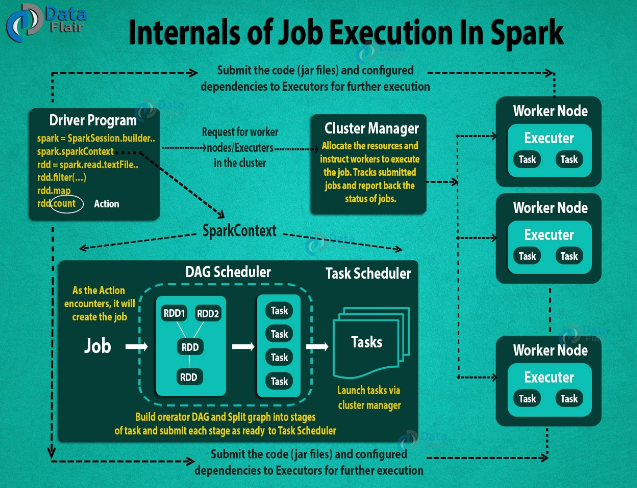
# Directed Acyclic Graph

DAG is a finite directed graph with no directed cycles. DAG in Spark is a set of Vertices and Edges, where vertices represents RDDs and edges represents the operation to be applied on the RDDs. In Spark DAG, every edge is directed from earlier to later in the sequence. On calling of Action, the created DAG is submitted to **DAGScheduler** which further splits the graph into the**stages**of the**task.**

## Working of DAG:

1. The interpreter is the first layer, using a Scala interpreter Spark interprets the code with some modifications.
2. Spark creates an operator graph.
3. When an **Action** is called on Spark RDD at a high level, Spark submits the operator graph to the **DAG Scheduler.**
4. Operators are divided into **stages** of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators are scheduled in a single stage.
5. The stages are passed on to the **Task Scheduler**. It launches task through [**cluster manager**](http://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/). The dependencies of stages are unknown to the task scheduler.
6. The **Workers** execute the task on the slave.





# Loading External Datasets

Notes on loading text files:

1. If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.
2. All of Spark’s file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/\*.txt"), and textFile("/my/directory/\*.gz").
3. The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks.

|  |  |
| --- | --- |
| **Spark Context Method** | **Description** |
| sc.textFile("data.txt") | Takes an URI for the file (either a local path or a HDFS URI) and reads it as a collection of lines (returns one record per line in the file) |
| sc.wholeTextFiles() | Reads a directory containing multiple small text files, and returns each of them as (filename, content) pairs |
| sc.sequenceFile[K, V] | K and V are the types of key and values in the file - which should be subclasses of Hadoop's [Writable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Writable.html) interface, like [IntWritable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/IntWritable.html) and [Text](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/Text.html). |
| sc.hadoopRDD(JobConf, InputFormat, K, V) | Used for other Hadoop input formats, |
| sc.newAPIHadoopRDD | For "new" MapReduce API (org.apache.hadoop.mapreduce) |
| sc.objectFile | Supports saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD. |

# Understanding Closures

In Spark, it is important to understand scope and life cycle of variables and methods when executing code across a cluster. RDD operations that modify variables outside of their scope can be a frequent source of confusion.

## Example: Local vs Cluster mode

int counter = 0;

JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!

rdd.foreach(x -> counter += x);

println("Counter value: " + counter);

The behavior of the above code is undefined, and may not work as intended. To execute jobs, Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor. Prior to execution, Spark computes the task’s **closure**. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in this case foreach()). This closure is serialized and sent to each executor.

The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node. There is still a **counter** in the memory of the driver node but this is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of **counter** will still be zero since all operations on **counter** were referencing the value within the serialized closure.

In local mode, in some circumstances the foreach function will actually execute within the same JVM as the driver and will reference the same original **counter**, and may actually update it.

To ensure well-defined behavior in these sorts of scenarios one should use an [**Accumulator**](http://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators). Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.

In general, closures - constructs like loops or locally defined methods, should not be used to mutate some global state. Spark does not define or guarantee the behavior of mutations to objects referenced from outside of closures. Some code that does this may work in local mode, but that’s just by accident and such code will not behave as expected in distributed mode. Use an Accumulator instead if some global aggregation is needed.

### Printing elements of an RDD

On a single machine, the following codewill generate the expected output and print all the RDD’s elements. However, in cluster mode, executors write to the executor’s stdout , not the one on the driver, so stdout on the driver won’t show these!

//Wrong: Don't do this!! *Works on Single machine, but not in cluster mode*

*rdd.foreach(println)*;

or

*rdd.map(println);*

To print all elements on the driver, one can use the **collect()** method to first bring the RDD to the driver node thus below code works; but this can cause the driver to run out of memory, because collect() fetches the entire RDD to a single machine.

//Wrong: Don't do this!! *In cluster mode, this collects all the elements to driver node and can cause OutOfMemory*

*rdd.collect().foreach(println);*

If we need to print a few elements of the RDD, safer approach is to use **take()**.

//This works

*rdd.take(100).foreach(println)*

# RDD Operations

At higher level, two type of RDD transformations can be applied: **narrow transformation**(e.g. map(), filter() etc.) and **wide transformation**(e.g. reduceByKey()). Narrow transformation does not require the shuffling of data across a partition, the narrow transformations will be grouped into single stage while in wide transformation the data is shuffled. Hence, wide transformation results in stage boundaries.

## Narrow Transformations

|  |  |
| --- | --- |
| **RDD Transformations** | **Description** |
| rdd.filter(func) | Returns new RDD with elements that meet the predicate |
| rdd.map(func) | Returns new RDD after converting each element as per the function provided. Returned RDD may or may not be of the same type. |
| rdd.flatMap (func) | It returns RDD with many more elements after applying the function provided. This is mainly used to flatten a structure. |
| rdd.mapPartitions(func) |  |
| rdd.mapPartitionsWithIndex(func) |  |
| rdd.sample(withReplacement, fraction, seed) |  |
| rdd.union(otherDataset) |  |

## Wide Transformations

|  |  |
| --- | --- |
| **RDD Transformations** | **Description** |
| rdd.intersection(otherDataset) |  |
| rdd.distinct([numPartitions]) |  |
| rdd.groupByKey([numPartitions]) |  |
| rdd.reduceByKey([numPartitions]) |  |
| rdd.aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions]) |  |
| rdd.sortByKey([asc], [numPartitions]) |  |
| rdd.join(otherDataset, [numPartitions]) |  |
| rdd.cogroup(otherDataset, [numPartitions]) |  |
| rdd.cartesian(otherDataset) |  |
| rdd.pipe(command, [envVars]) |  |
| rdd.coalesce([numPartitions]) |  |
| rdd.repartition([numPartitions]) |  |
| rdd.repartitionAndSortWithinPartitions( partitioner) |  |

## Actions

|  |  |
| --- | --- |
| **RDD Actions** | **Description** |
| rdd.reduce(func) |  |
| rdd.collect() |  |
| rdd.count () |  |
| rdd.first() |  |
| rdd.take(n) |  |
| rdd.takeSample(withReplacement, num, [seed]) |  |
| rdd.takeOrdered(n, [ordering]) |  |
| rdd.saveAsTextFile(path) |  |
| rdd.saveAsSequenceFile(path)  *(Java and Scala)* |  |
| rdd.saveAsObjectFile  *(Java and Scala)* | Supports saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD. |
| rdd.countByKey() |  |
| rdd.foreach(func) |  |

References

1. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia Learning Spark - Lightning Fast Big Data Analysis
2. http://spark.apache.org/docs/latest/rdd-programming-guide.html
3. https://data-flair.training/blogs/category/spark/