SPARK

SPARK

- Apache Spark is an open-source cluster computing framework. It was originally developed at UC Berkeley in 2009
- Its primary purpose is to handle the real-time generated data.
- Spark was built on the top of the Hadoop MapReduce.
- It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives.
- So, Spark process the data much quicker than other alternatives.

Spark Architecture

- The Spark follows the master-slave architecture.
- Its cluster consists of a single master and multiple slaves.
- The Spark architecture depends upon two abstractions:
  - Resilient Distributed Dataset (RDD)
  - Directed Acyclic Graph (DAG)

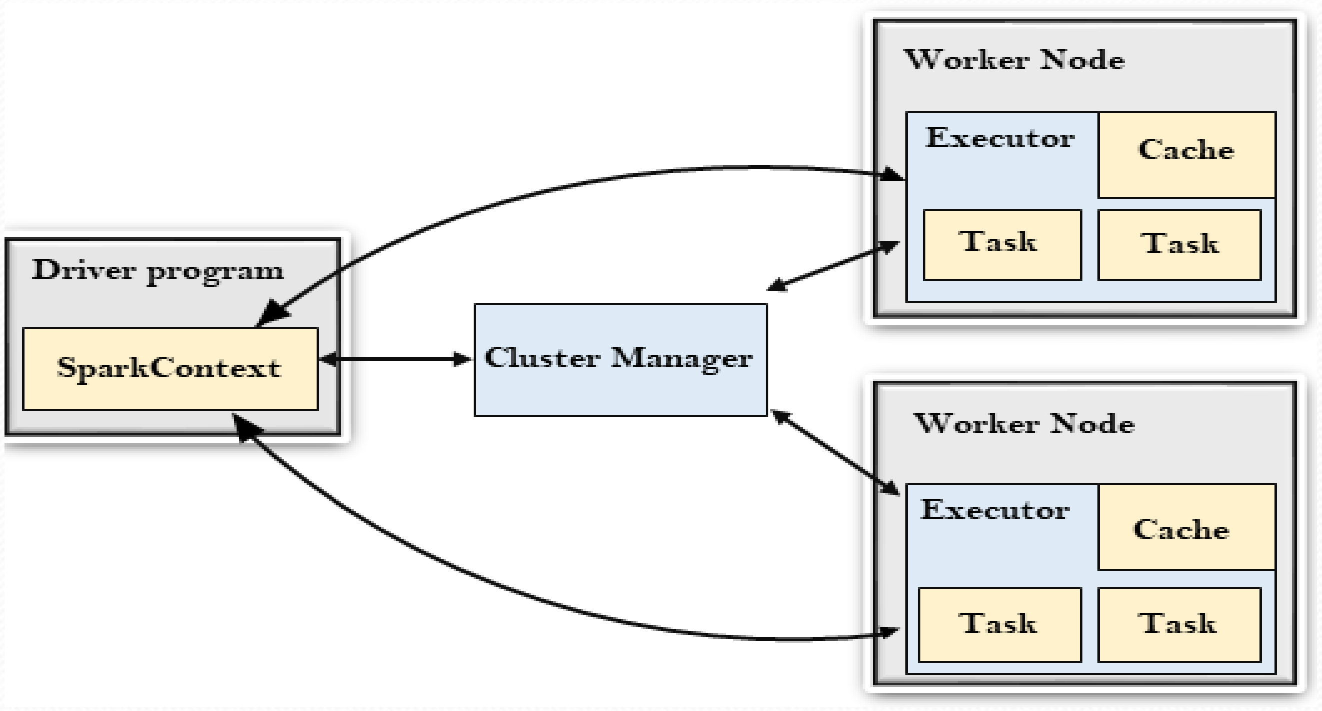
Resilient Distributed Datasets (RDD)

- The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,
- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

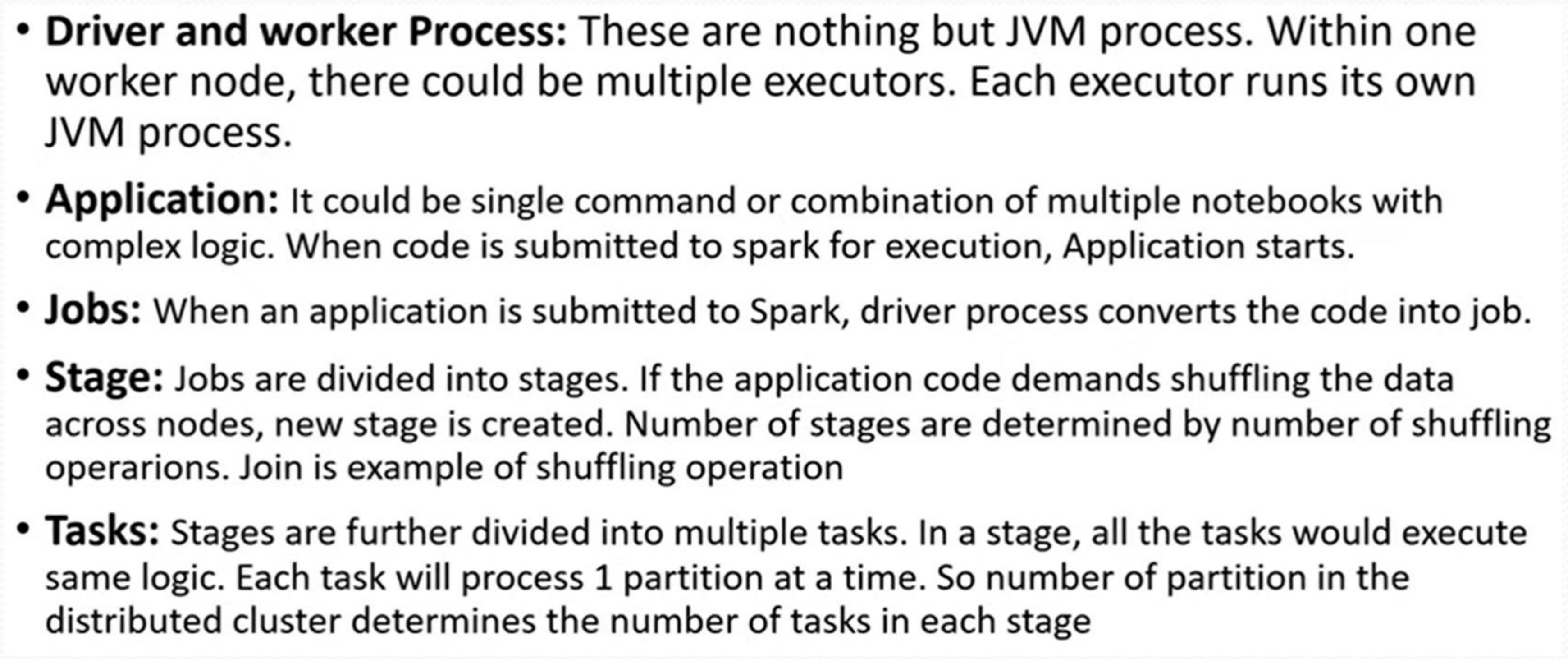
Directed Acyclic Graph (DAG)

- Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data.
- Each node is an RDD partition, and the edge is a transformation on top of data.
- Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.

Spark architecture.



Terminologies



- **Driver and worker Process:** These are nothing but JVM process. Within one worker node, there could be multiple executors. Each executor runs its own JVM process.
- **Application:** It could be single command or combination of multiple notebooks with complex logic. When code is submitted to spark for execution, Application starts.
- **Jobs:** When an application is submitted to Spark, driver process converts the code into job.
- **Stage:** Jobs are divided into stages. If the application code demands shuffling the data across nodes, new stage is created. Number of stages are determined by number of shuffling operarions. Join is example of shuffling operation
- **Tasks:** Stages are further divided into multiple tasks. In a stage, all the tasks would execute same logic. Each task will process 1 partition at a time. So number of partition in the distributed cluster determines the number of tasks in each stage

- Driver Program
  - The Driver Program is a process that runs the main() function of the application and creates the SparkContext object.
  - The purpose of SparkContext is to coordinate the spark applications, running as independent sets of processes on a cluster.
  - To run on a cluster, the SparkContext connects to a different type of cluster managers and then perform the following tasks: -
    - It acquires executors on nodes in the cluster.
    - Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
    - At last, the SparkContext sends tasks to the executors to run.

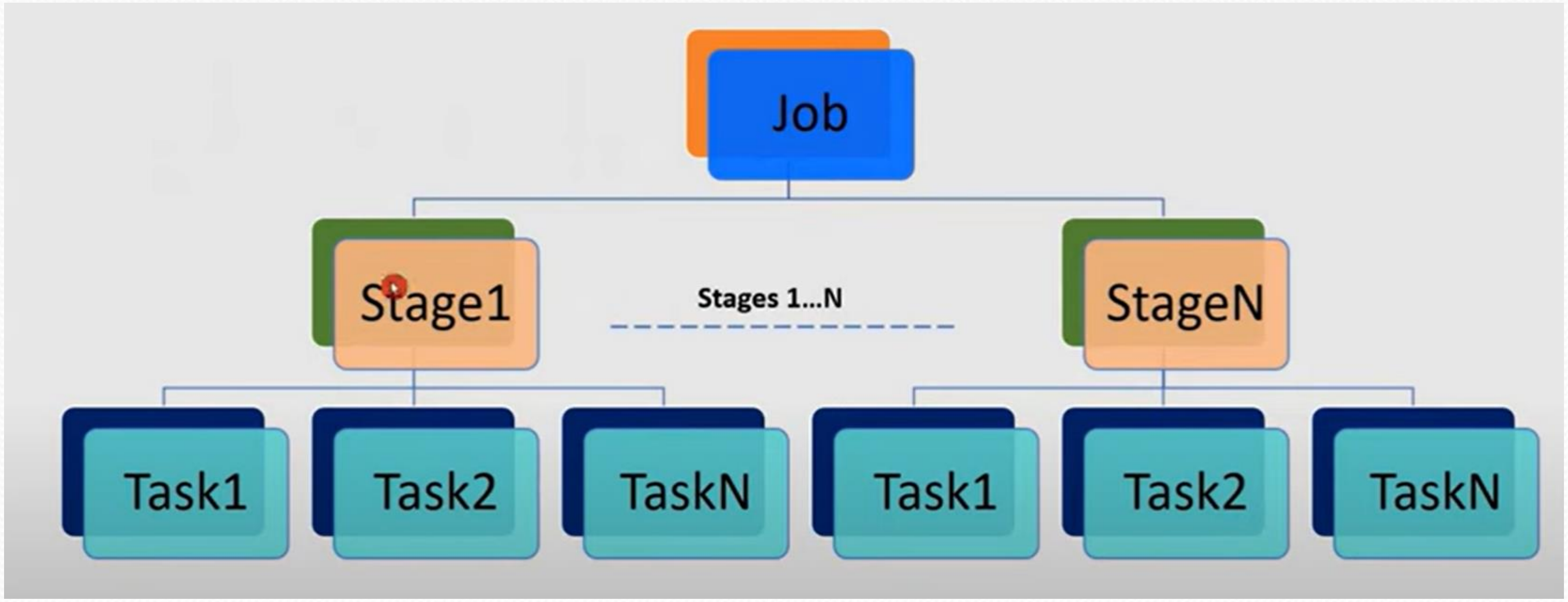
- Cluster Manager
    - The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
    - It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
    - Here, the <u>Standalone Scheduler</u> is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

- Worker Node
  - The worker node is a slave node
  - Its role is to run the application code in the cluster.
- Executor
  - An executor is a process launched for an application on a worker node.
  - It runs tasks and keeps data in memory or disk storage across them.
  - It read and write data to the external sources.
  - Every application contains its executor.
- Task
  - A unit of work that will be sent to one executor

Working:

- In your master node, you have the driver program, which drives your application. The code you are writing behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program.
- Inside the driver program, the first thing you do is, you create a Spark Context. Assume that the Spark context is a gateway to all the Spark functionalities. It is similar to your database connection. Any command you execute in your database goes through the database connection. Likewise, anything you do on Spark goes through Spark context.
- Now, this Spark context works with the cluster manager to manage various jobs. The driver program & Spark context takes care of the job execution within the cluster. A job is split into multiple tasks which are distributed over the worker node. Anytime an RDD is created in Spark context, it can be distributed across various nodes and can be cached there.

- Worker nodes are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned RDDs in the worker node and hence returns back the result to the Spark Context.
- Spark Context takes the job, breaks the job in tasks and distribute them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.
- If you increase the number of workers, then you can divide jobs into more partitions and execute them parallelly over multiple systems. It will be a lot faster.
- With the increase in the number of workers, memory size will also increase & you can cache the jobs to execute it faster.


Job
Stage1
Stages 1...N
StageN
Task1
Task2
TaskN
Task1
Task2
TaskN

SparkSession

- SparkSession is a unified entry point for Spark applications; it was introduced in Spark 2.0.
- It acts as a connector to all Spark’s underlying functionalities, including RDDs, DataFrames, and Datasets, providing a unified interface to work with structured data processing.
- It is one of the very first objects you create while developing a Spark SQL application.
- As a Spark developer, you create a SparkSession using the SparkSession.builder() method

- SparkSession consolidates several previously separate contexts, such as SQLContext, HiveContext, and StreamingContext, into one entry point, simplifying the interaction with Spark and its different APIs.
- It enables users to perform various operations like reading data from various sources, executing SQL queries, creating DataFrames and Datasets, and performing actions on distributed datasets efficiently.
- For those engaging with Spark through the spark-shell CLI, the 'spark' variable automatically provides a default Spark Session, eliminating the need for manual creation within this context.

How to Create SparkSession

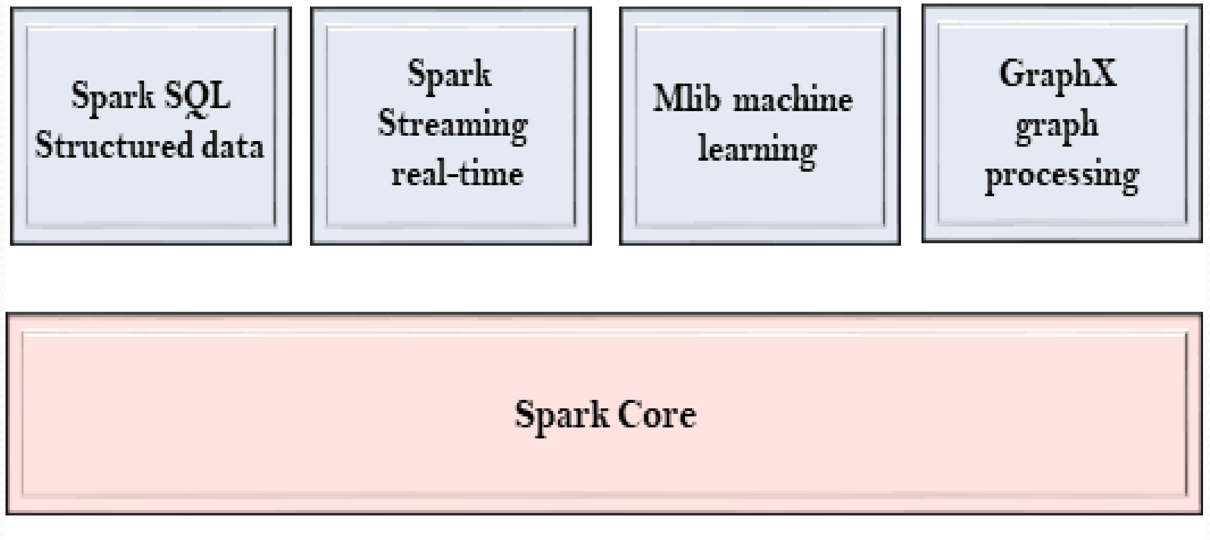
- Creating a SparkSession is fundamental as it initializes the environment required to leverage the capabilities of Apache Spark.
- To create SparkSession in Scala or Python, you need to use the builder pattern method builder() and calling getOrCreate() method.
- It returns a SparkSession that already exists; otherwise, it creates a new SparkSession. The example below creates a SparkSession in Scala.

```
import org.apache.spark.sql.SparkSession
object SparkSessionTest extends App {
 val spark = SparkSession.builder()
    .master("local[1]")
    .appName("SparkByExamples.com")
    .getOrCreate();
 println(spark)
 println("Spark Version : "+spark.version)
}
```

- SparkSession.builder() – Return SparkSession.Builder class. This is a builder for SparkSession. master(), appName(), and getOrCreate() are methods of SparkSession.Builder.
- master() – This allows Spark applications to connect and run in different modes (local, standalone cluster, Mesos, YARN), depending on the configuration.
- Use local[x] when running on your local laptop. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.
- For standalone use spark://master:7077
- appName() – Sets a name to the Spark application that shows in the Spark web UI. If no application name is set, it sets a random name.
- getOrCreate() – This returns a SparkSession object if it already exists. Creates a new one if it does not exist.

Spark Components

- The Spark project consists of different types of tightly integrated components.
- At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.



- Spark Core
  - The Spark Core is the heart of Spark and performs the core functionality.
  - It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.
- Spark SQL
  - The Spark SQL is built on the top of Spark Core. It provides support for structured data.
  - It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language).
  - It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.
  - It also supports various sources of data like Hive tables, Parquet, and JSON.

- Spark Streaming
  - Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
  - It uses Spark Core's fast scheduling capability to perform streaming analytics.
  - It accepts data in mini-batches and performs RDD transformations on that data.
  - Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
  - The log files generated by web servers can be considered as a real-time example of a data stream.

- MLlib
  - The MLlib is a Machine Learning library that contains various machine learning algorithms.
  - These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
  - It is nine times faster than the disk-based implementation used by Apache Mahout.
- GraphX
  - The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
  - It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
  - To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

•Resilient Distributed Dataset (RDD)

- Every Spark application is made-up of a Driver program which runs the primary function and is responsible for various parallel operations on the given cluster.
- The primary abstraction the Spark is the concept of RDD, which Spark uses to achieve Faster and efficient MapReduce operations.
- Resilient Distributed Dataset (RDD) is the fundamental data structure of Spark.
- They are immutable Distributed collections of objects of any type.
- As the name suggests is a Resilient (Fault-tolerant) records of data that resides on multiple nodes.

Features of Resilient Distributed Dataset (RDD)

- **Lazy Evaluation**
- All Transformations in the Apache Spark are lazy, which means that they do not compute the results as and when stated in Transformation statements.
- Instead, they Keep track of the Transformation tasks using the concept of DAG (Directed Acyclic Graphs).
- Spark computes these Transformations when an action requires a result for the driver program.

- **In-Memory Computation**
- Spark uses in-memory computation as a way to speed up the total processing time. In the in-memory computation, the data is kept in RAM (random access memory) instead of the slower disk drives.
- This is very helpful as it reduces the cost of memory and allows for pattern detection, analyzes large data more efficiently. Main methods that accompany this are *cache()* and *persist()* methods.
- **Fault Tolerance**
- The RDDs are fault-tolerant as they can track data lineage information to allow for rebuilding lost data automatically on failure.
- To achieve fault tolerance for the generated RDD's, the achieved data is replicated among various Spark executors in worker nodes in the cluster.

Ways to Create RDDs in Apache Spark

- Three ways to create an RDD in Apache Spark
- **Parallelizing collection (Parallelized)**
- We take an already existing collection in the program and pass it onto the SparkContext's parallelize() method.
- This is an original method which creates RDDs quickly in Spark-shell and also performs operations on them.
- It is very rarely used, as this requires the entire Dataset on one machine.
- Eg:
  - val data = Array(1, 2, 3, 4, 5)
  - val distData = sc.parallelize(data)

- **Referencing External Dataset**
- In Spark, the RDDs can be formed from any data source supported by the Hadoop, including local file systems, HDFS, Hbase, Cassandra, etc.
- Here, data is loaded from an external dataset.
- We can use SparkContext's textFile method to create text file RDD. It would URL of the file and read it as a collection of line. URL can be a local path on the machine itself.
- Eg:
- val distFile = sc.textFile("data.txt")

- Creating RDD from existing RDD
- Transformation mutates one RDD into another, and change is the way to create an RDD from an existing RDD.
- This creates a difference between Apache Spark and Hadoop MapReduce.
- Conversion works like one that intakes an RDD and produces one.
- The input RDD does not change, and as RDDs are immutable, it generates varying RDD by applying operations.
- Eg:
- val rdd3 = rdd.map(x=>x*2)

Spark RDD Operations

- RDD in Apache Spark supports two types of operations:
- Transformation
- Actions

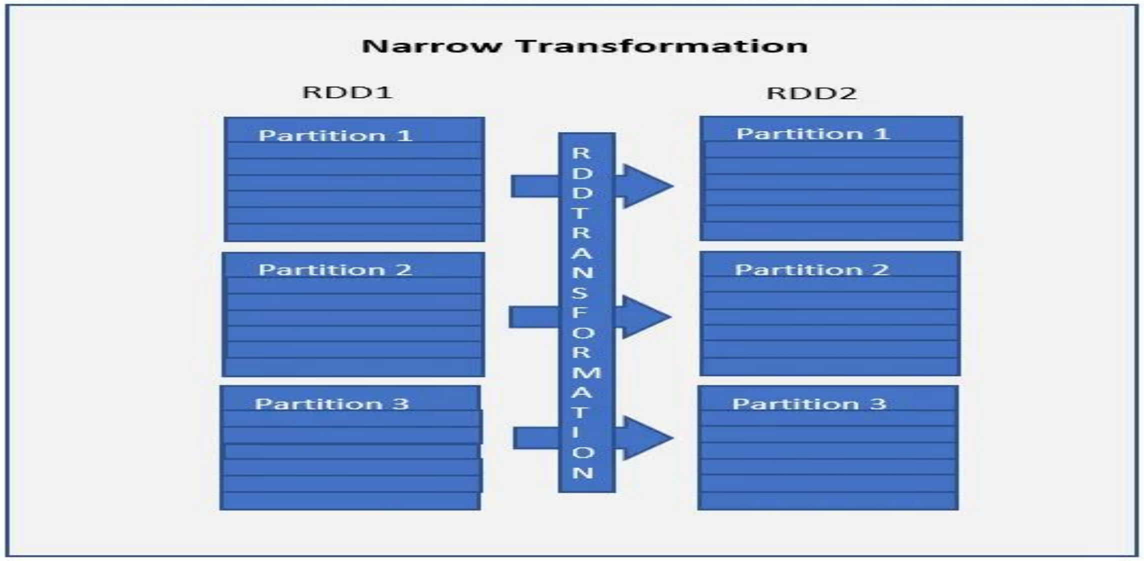
- i. Transformations
  - Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output.
  - They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations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

- 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.

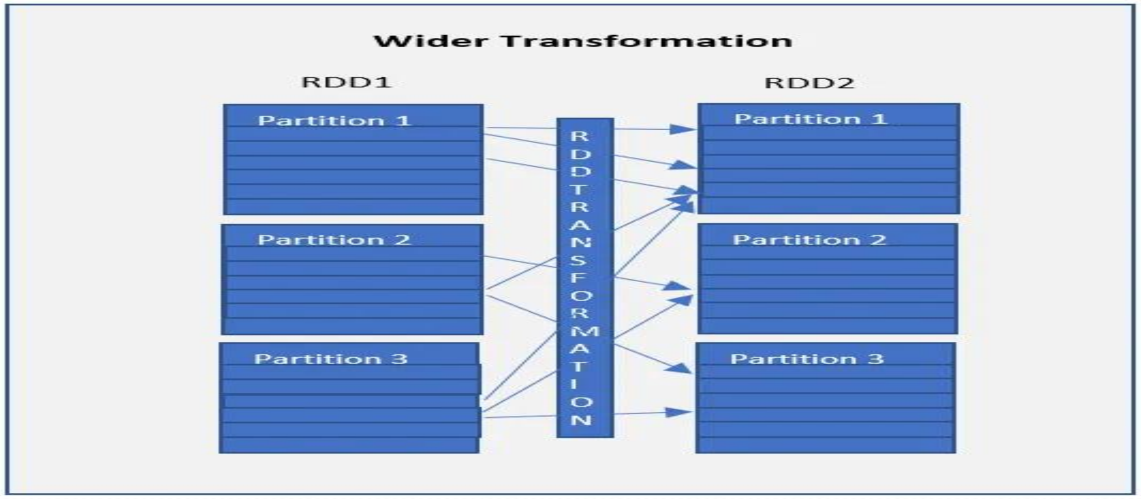
- At high level, there are two transformations that can be applied onto the RDDs, namely narrow transformation and wide transformation. Wide transformations basically result in stage boundaries.
- Narrow transformation — doesn’t require the data to be shuffled across the partitions. for example, Map, filter etc..
- wide transformation — requires the data to be shuffled for example, reduceByKey etc..

RDD Transformation Types

- There are two types are transformations.
- Narrow Transformation
  - Narrow transformations are the result of map() and filter() functions and these compute data that live on a single partition meaning there will not be any data movement between partitions to execute narrow transformations.



- Wider Transformation
  - Wider transformations are the result of groupByKey() and reduceByKey() functions and these compute data that live on many partitions meaning there will be data movements between partitions to execute wider transformations. Since these shuffles the data, they also called shuffle transformations.

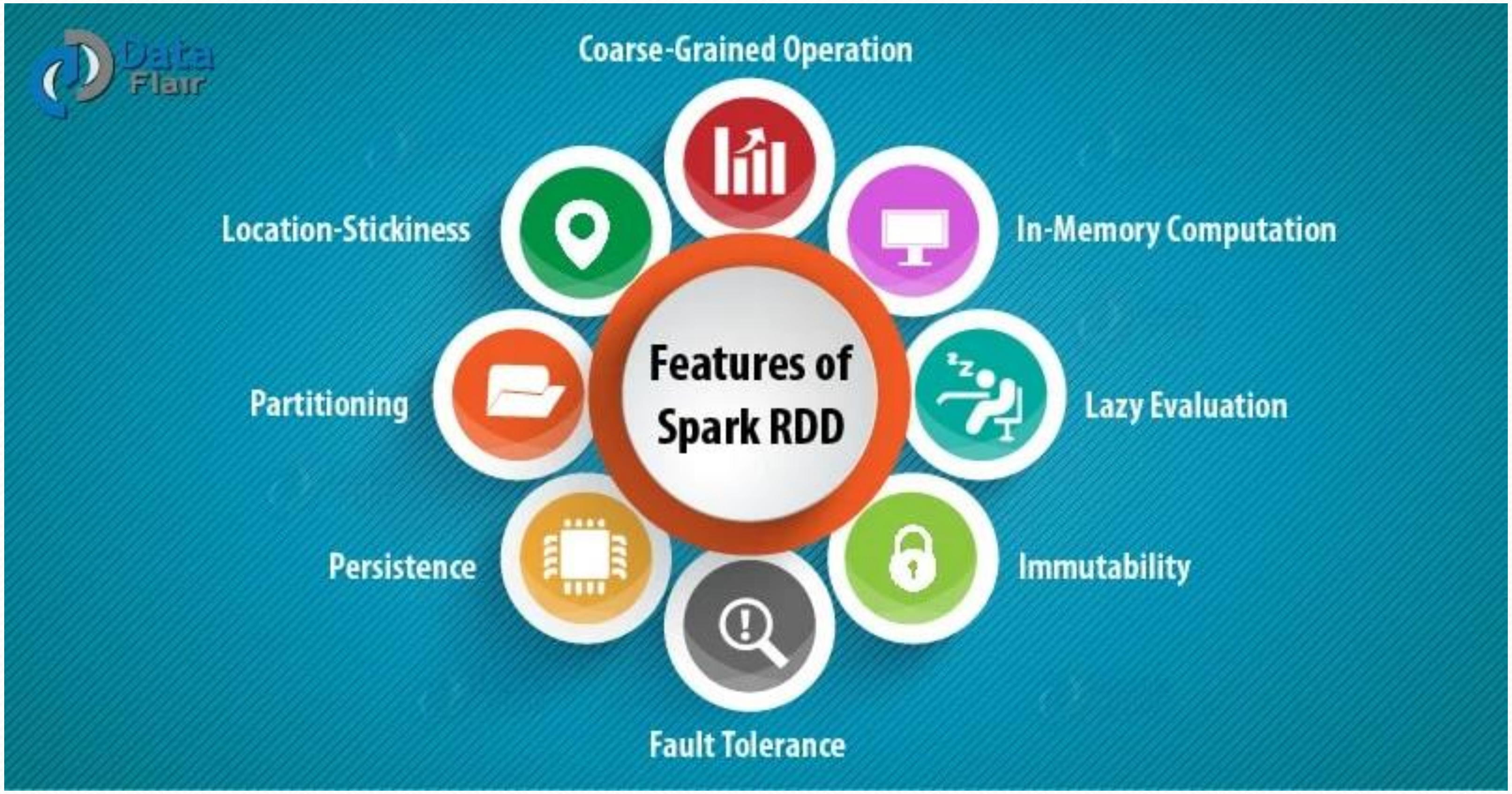


- Functions such as
  - map(),
  - mapPartition(),
  - flatMap(),
  - filter(),
  - union() are some examples of narrow transformation
- Functions such as
  - groupByKey(),
  - aggregateByKey(),
  - aggregate(),
  - join(),
  - repartition() are some examples of a wider transformations.

Example-RDD Operation

```
import org.apache.spark.sql.SparkSession
object CountLines {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder
      .appName("CountLines")
      .master("local")
      .getOrCreate()
    val sc = spark.sparkContext
    val lines = sc.textFile("data.txt")
    val pairs = lines.map(s => (s, 1))
    val counts = pairs.reduceByKey(_ + _)
    counts.collect().foreach(println)
    spark.stop()
  }
}
```

Features of Spark RDD



- i. 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).
- 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.

- 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.

- 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.

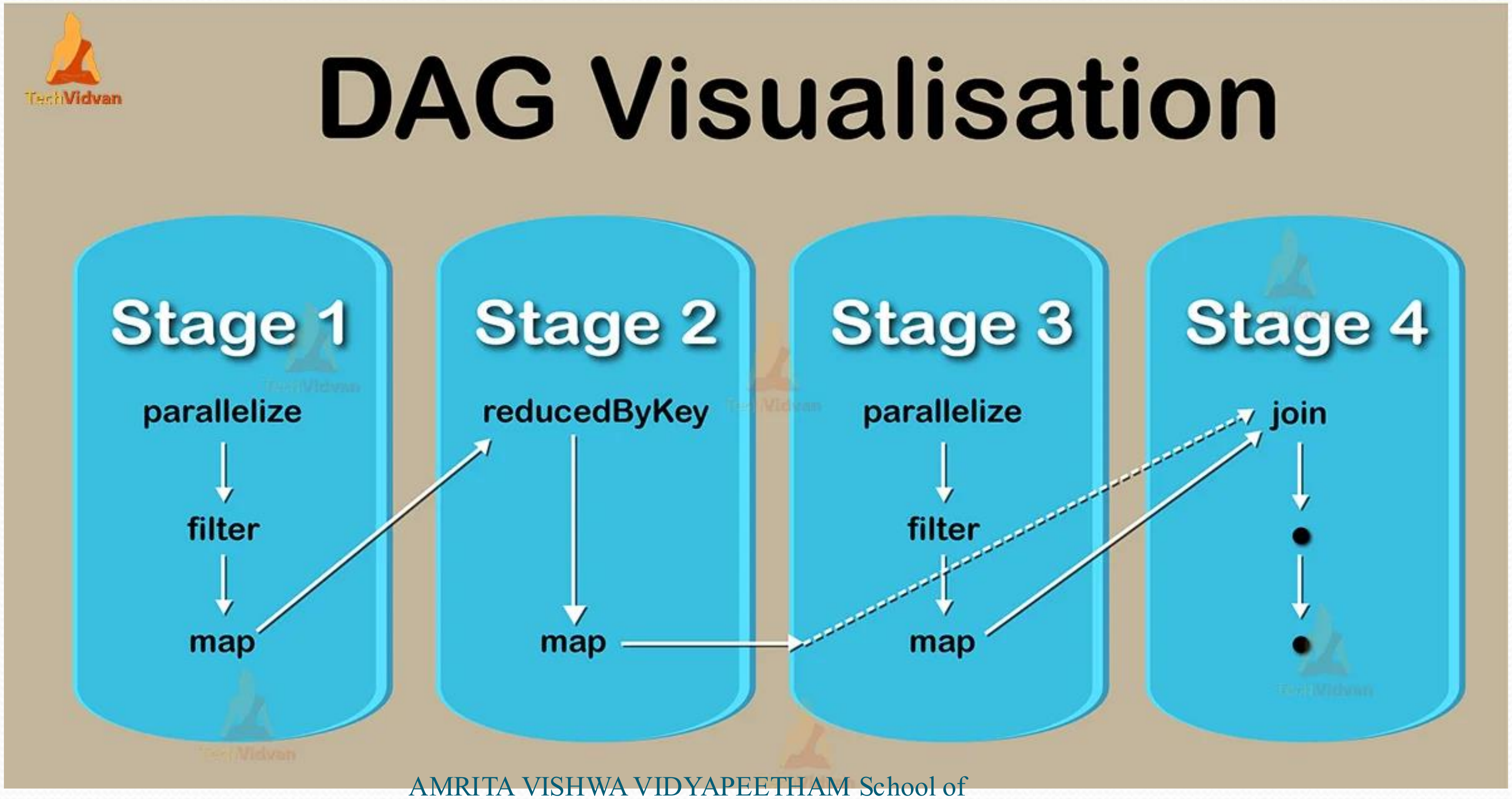
- v. Partitioning
- Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.
- vi. Persistence
- Users can state which RDDs they will reuse and choose 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.

Directed Acyclic Graph

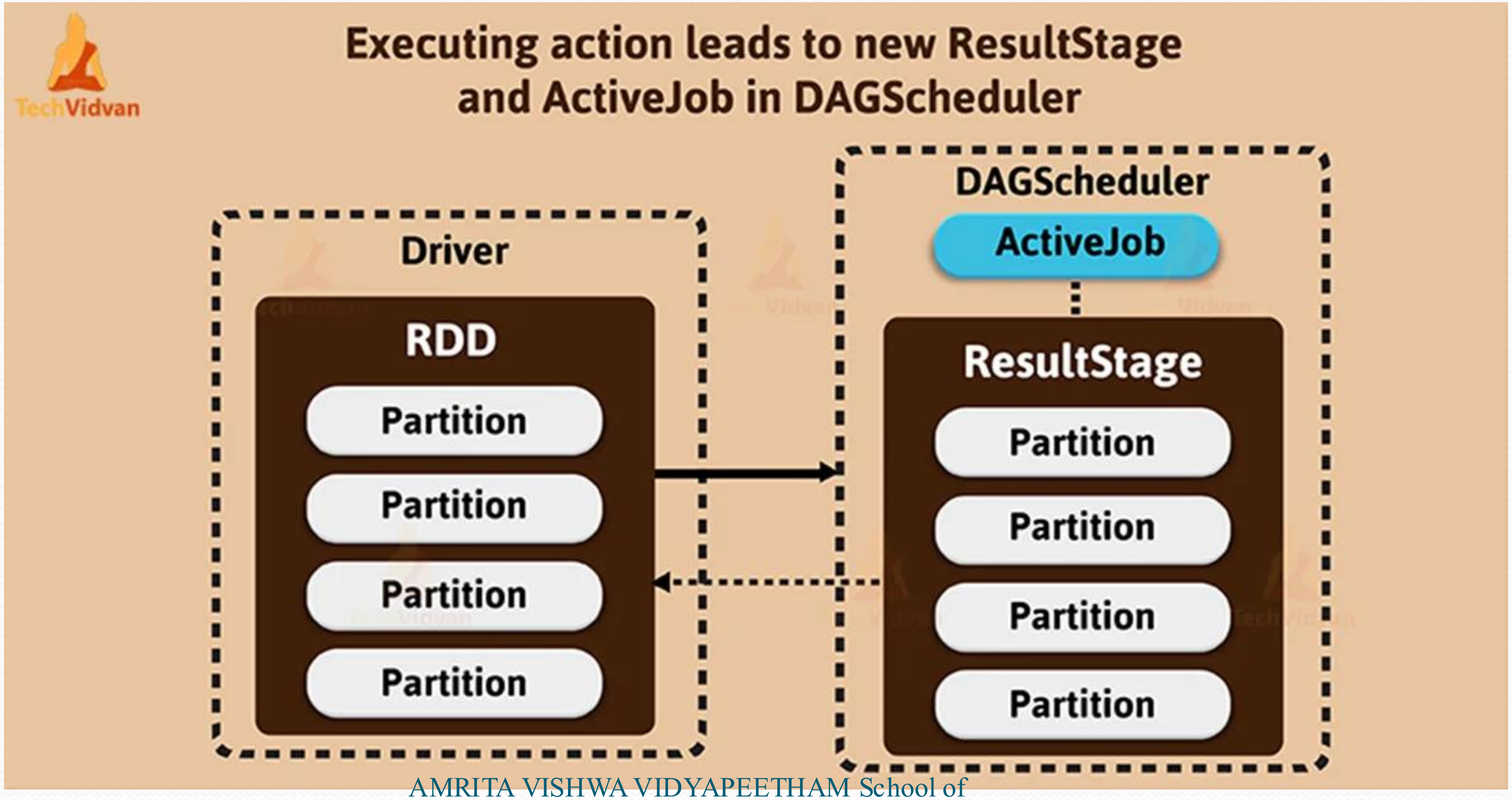
- **Directed Acyclic Graph DAG** in Apache Spark is a set of **Vertices** and **Edges**, where *vertices* represent the **RDDs** and the *edges* represent the **Operation to be applied on RDD**.
- In Spark DAG, every edge directs from earlier to later in the sequence.
- On the calling of *Action*, the created DAG submits to **DAG Scheduler** which further splits the graph into the **stages** of the **task**.
- Directed –
  - Means which is directly connected from one node to another. This creates a sequence i.e. each node is in linkage from earlier to later in the appropriate sequence.
- Acyclic –
  - Defines that there is no cycle or loop available. Once a transformation takes place it cannot returns to its earlier position.
- Graph –
  - From graph theory, it is a combination of vertices and edges. Those pattern of connections together in a sequence is the graph.

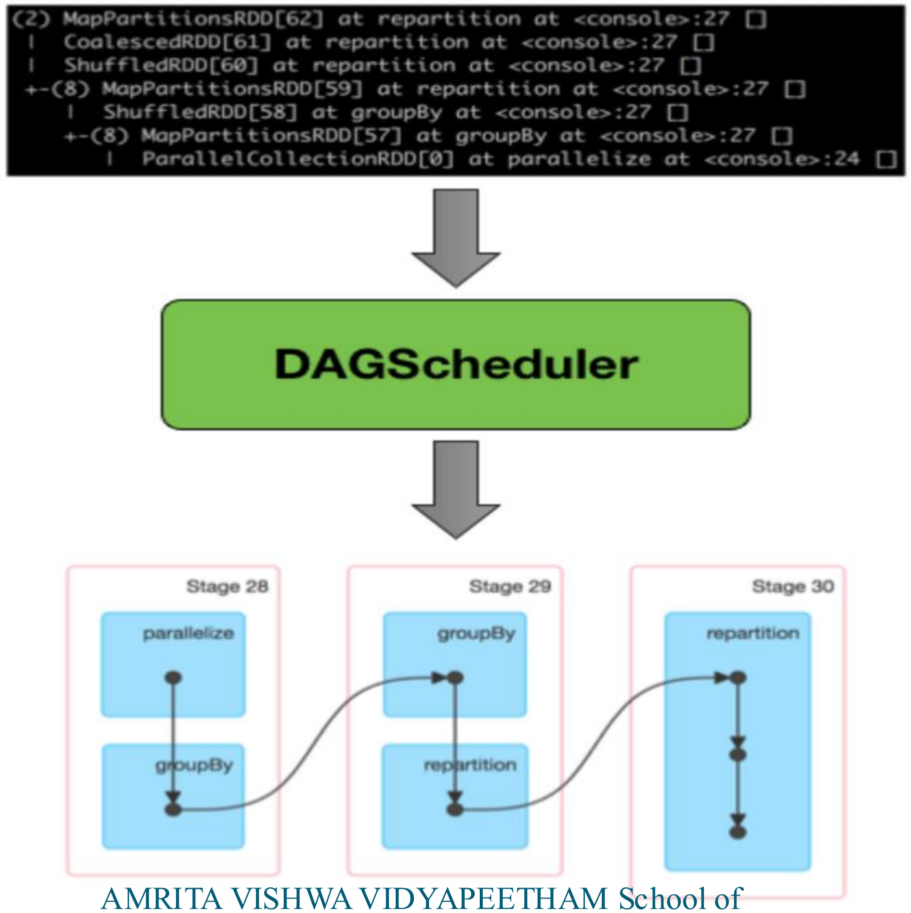
What is DAG in Apache Spark?

- **DAG** a finite direct graph with no directed cycles.
- There are finitely many vertices and edges, where each edge directed from one vertex to another.
- It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence.
- It is a strict generalization of MapReduce model.
- DAG operations can do better global optimization than other systems like MapReduce.
- The picture of DAG becomes clear in more complex jobs.
- Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage. In the stage view, the details of all RDDs belonging to that stage are expanded.
- The Scheduler splits the Spark RDD into **stages** based on various transformation applied.
- Each stage is comprised of **tasks**, based on the partitions of the RDD, which will perform same computation in parallel.
- The graph here refers to navigation, and directed and acyclic refers to how it is done.


TechVidvan
DAG Visualisation
Stage 1
parallelize
filter
map
Stage 2
reducedByKey
map
Stage 3
parallelize
filter
map
Stage 4
join

Working of DAG Scheduler

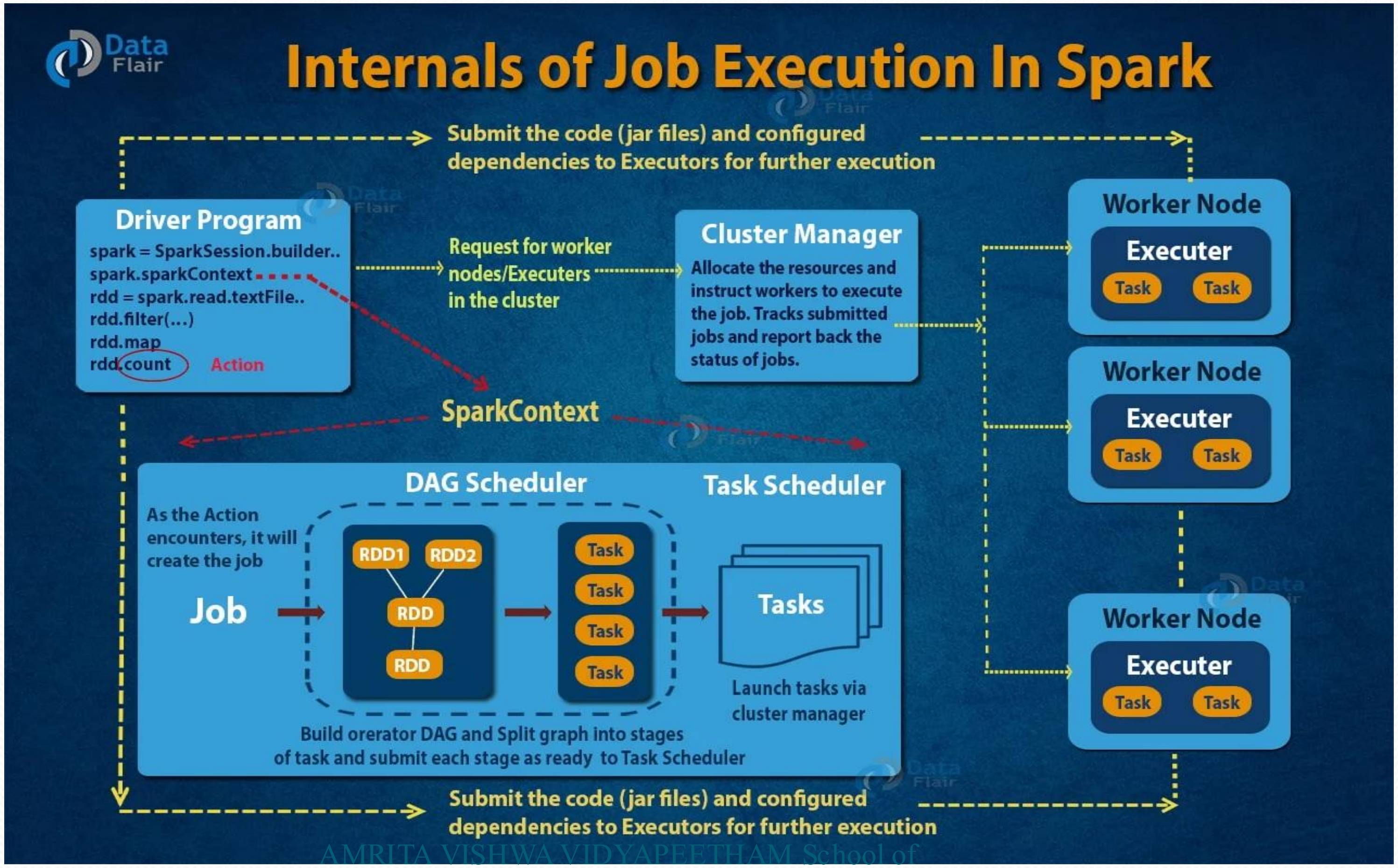




```
(2) MapPartitionsRDD[62] at repartition at <console>:27 []
 |  CoalescedRDD[61] at repartition at <console>:27 []
 |  ShuffledRDD[60] at repartition at <console>:27 []
 +-(8) MapPartitionsRDD[59] at repartition at <console>:27 []
    |  ShuffledRDD[58] at groupBy at <console>:27 []
    +-(8) MapPartitionsRDD[57] at groupBy at <console>:27 []
       |  ParallelCollectionRDD[0] at parallelize at <console>:24 []
```

- It is a scheduling layer in a spark which implements stage oriented scheduling. It converts logical execution plan to a physical execution plan. When an action is called, spark directly strikes to DAG scheduler. It executes the tasks those are submitted to the scheduler.

- Spark uses pipelining (lineage) operations to optimize its work, that process combines the transformations into a single stage. The basic concept of DAG scheduler is to maintain jobs and stages. It tracks through internal registries and counters.



Data Flair
Internals of Job Execution In Spark
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

How DAG works in Spark

- The interpreter is the first layer, using a Scala interpreter, Spark interprets the code with some modifications.
- Spark creates an operator graph when you enter your code in Spark console.
- When we call an **Action** on Spark RDD at a high level, Spark submits the operator graph to the **DAG Scheduler.**
- Divide the operators 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 schedule in a single stage.
- The stages pass on to the **Task Scheduler**. It launches task through cluster manager. The dependencies of stages are unknown to the task scheduler.
- The **Workers** execute the task on the slave.

There are following steps through DAG scheduler works:

- It completes the computation and execution of stages for a job.
- It also keeps track of RDDs and run jobs in minimum time and assigns jobs to the task scheduler. Task scheduler means submitting tasks for execution.
- It determines the preferred locations on which, we can run each task respectively.
- It is possible with the task scheduler. That gets the information of current cache status.
- It handles the track of RDDs, which are cached to devoid re-computing. In this way it also handles failure.
- As it remembers at what stages it already produced output files it heals the loss. Due to shuffle output files may lose so it helps to recover failure.

Word Count Problem

```
import org.apache.spark.{SparkConf, SparkContext}
object WordCount {
 def main(args: Array[String]): Unit = {
   // Step 1: Set up Spark configuration and context
   val conf = new
SparkConf().setAppName("WordCount").setMaster("
local[*]")
   val sc = new SparkContext(conf)
   // Step 2: Read the input file
   val inputFile = "path/to/your/input.txt" // Replace
with your file path
   val inputRDD = sc.textFile(inputFile)
   // Step 3: Perform the word count
   val wordCounts = inputRDD
     .flatMap(line => line.split("\\s+")) // Split lines
into words
map(word => word.replaceAll("[^a-zA-Z0-9]",
"").toLowerCase) // Clean words
     .filter(_.nonEmpty) // Remove empty strings
     .map(word => (word, 1)) // Map each word to
(word, 1)
     .reduceByKey(_ + _) // Reduce by key to count
occurrences
   // Step 4: Save or print the results
   wordCounts.collect().foreach { case (word, count)
=>
     println(s"$word: $count")
   }
   // Optional: Save to a file
   //
wordCounts.saveAsTextFile("path/to/output/directory
")
   // Step 5: Stop the Spark context
   sc.stop()
 }
}
```