**Apache Spark**

**Overview:** Apache Spark is a fast and general-purpose cluster computing system.

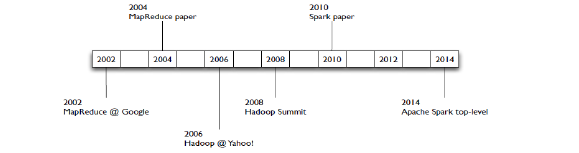
**Apache Spark is**

* Open Source
* lightning-fast cluster computing technology
* In-Memory Processing
* Specific pattern called DAG(Directed Acyclic Graph)

Apache Spark is an open-source ,light fastening execution engine builds around speed, ease of use and analysis.

[***Apache Spark***](http://spark.apache.org/)***is an open-source distributed general-purpose cluster computing framework with (mostly) in-memory data processing engine that can do ETL, analytics, machine learning and graph processing on large volumes of data at rest (batch processing) or in motion (streaming processing) with***[***rich concise high-level APIs***](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-overview.html#unified-api)***for the programming languages: Scala, Python, Java, R, and SQL.***

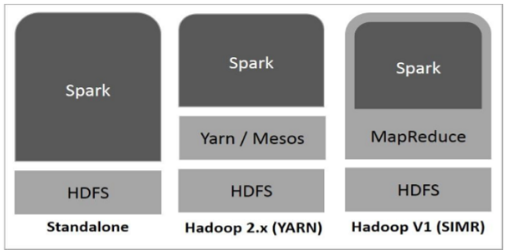
**Evolution:** Spark is one of Hadoop’s sub project developed in 2009 in UC Berkeley’s AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.



**Features of Spark:**

1. **Speed :**Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.
2. **Supports multiple languages** − Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
3. **Advanced Analytics** − Spark not only supports ‘Map’ and ‘reduce’. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

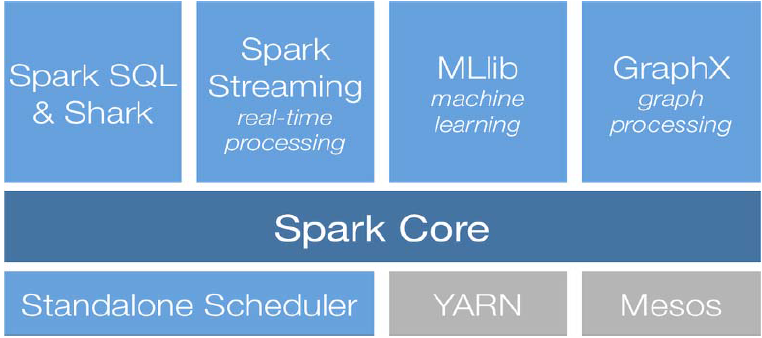
**Spark Built on Hadoop:** The following diagram shows three ways of how Spark can be built with Hadoop components.



There are three ways of Spark deployment as explained below.

* **Standalone** − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
* **Hadoop Yarn** − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
* **Spark in MapReduce (SIMR)** − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

**Spark Ecosystem:**



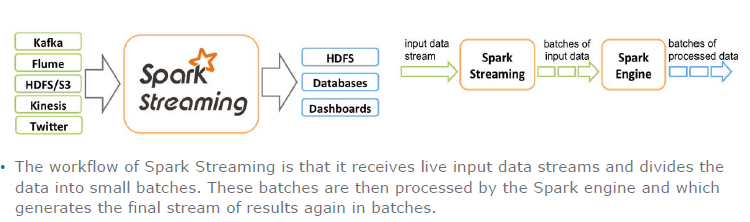
**Apache Spark Core:** Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

**Spark SQL:** Spark SQL is a component on top of Spark Core.

* Permits relational queries expressed in SQL, Hive SQL(Shark),Scala.
* Lets you query structured data inside Spark Program.
* Allows user to ETL their data available in different formats(JSOn, Parquet,Hive Table)

### Spark Streaming: Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics.

* Enable live streaming of data in scalable ,fault tolerant and high through put manner.
* Data can be ingested from many sources (Kafka, Flume, ZeroMQ, Kinesis or TCP sockets) and can be processed using complex algorithm with high-level function



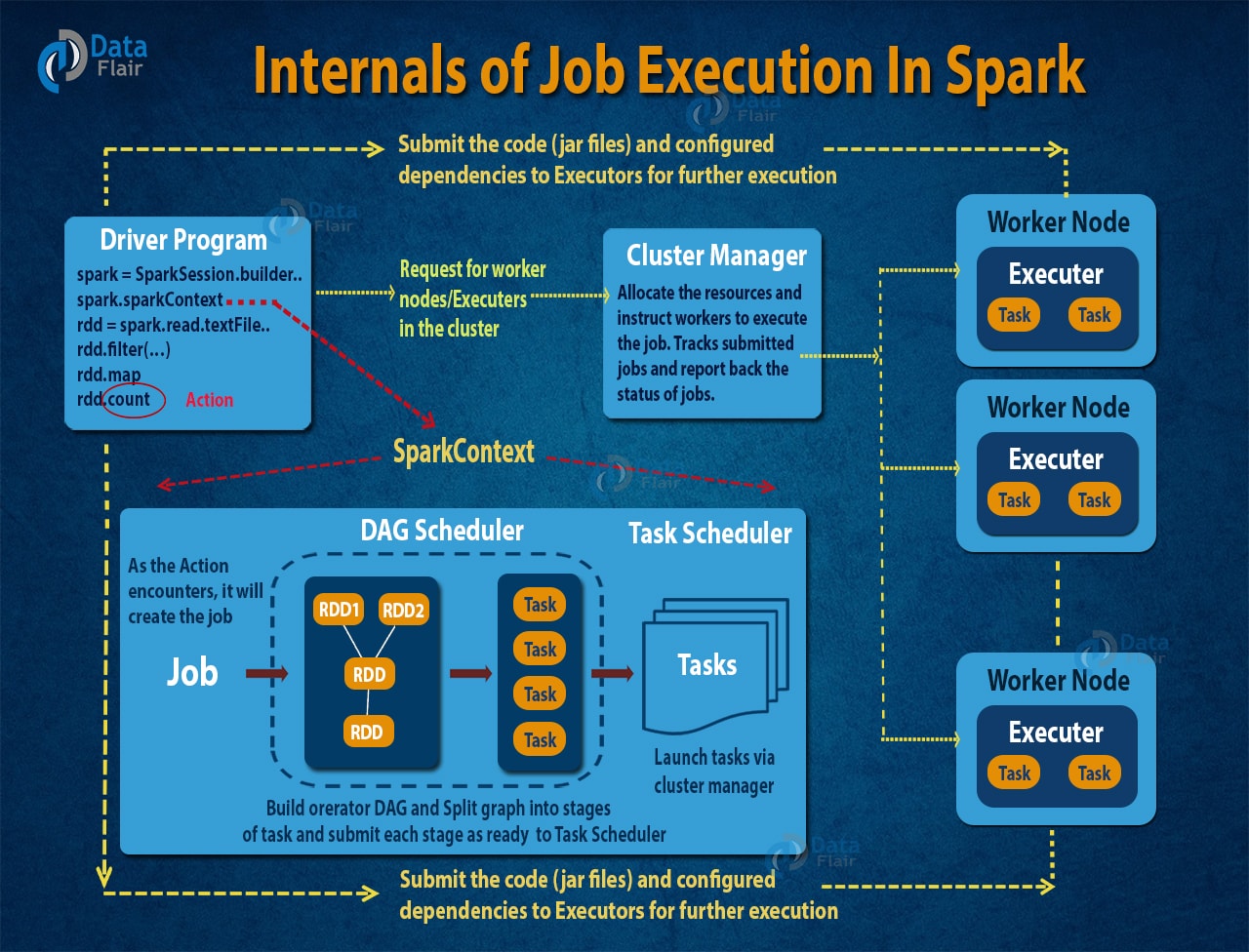
### MLlib (Machine Learning Library): MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture.

* Mlib uses linear Algebra Package called Breeze, which depend on netLib-java for optimized numerical processing.
* Spark MLlib is nine times as fast as the Hadoop disk-based version of Apache Mahout (before Mahout gained a Spark interface).

### GraphX: GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API.

**Spark and Its Processing:**

**DAG(Directed Acyclic Graph):**  is a finite [directed graph](https://en.wikipedia.org/wiki/Directed_graph) with no [directed cycles](https://en.wikipedia.org/wiki/Cycle_graph#Directed_cycle_graph). That is, it consists of finitely many [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) and [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)), with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again.



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.

Each RDD maintains a pointer to one or more parent along with metadata about what type of relationship it has with the parent. For example, if we call val b=a.map() on an RDD, the RDD b keeps a reference to its parent RDD a, that’s an RDD lineage.

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/scheduler/DAGScheduler.scala

**RDD(Resilient Distributed Database):** RDD is a major building block of Spark.Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. and is:

* Immutable
* Read only,Partitioned collection if record
* Fault Tolerant
* Resides in primary memory

**DataSet: Dataset** is a strongly-typed data structure in Spark SQL that represents a structured query with [encoders](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-sql-Encoder.html).

As of Spark 2.0, the main data abstraction of Spark SQL is [Dataset](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-sql-Dataset.html). It represents a **structured data** which are records with a known schema. This structured data representation Dataset enables [compact binary representation](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-sql-tungsten.html) using compressed columnar format that is stored in managed objects outside JVM’s heap. It is supposed to speed computations up by reducing memory usage and GCs.

**Spark Persistence:** One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).  Caching is a key tool for iterative algorithms and fast interactive use.

* You can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
* Spark’s cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.
* each persisted RDD can be stored using a different storage level

|  |  |
| --- | --- |
| **Storage Level** | **Meaning** |
| MEMORY\_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. |
| MEMORY\_AND\_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. |
| MEMORY\_ONLY\_SER  (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a [fast serializer](https://spark.apache.org/docs/latest/tuning.html), but more CPU-intensive to read. |
| MEMORY\_AND\_DISK\_SER  (Java and Scala) | Similar to MEMORY\_ONLY\_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. |
| DISK\_ONLY | Store the RDD partitions only on disk. |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes. |
| OFF\_HEAP (experimental) | Similar to MEMORY\_ONLY\_SER, but store the data in [off-heap memory](https://spark.apache.org/docs/latest/configuration.html#memory-management). This requires off-heap memory to be enabled. |

**Shared Variables:** Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function. These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program. Supporting general, read-write shared variables across tasks would be inefficient. However, Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators.

## Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

* The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

## Accumulators

Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

As a user, you can create named or unnamed accumulators.

* Accumulator are write only in nature
* Accumulators are like counter used in hadoop
* Spark automatically re-execute the failed task.
* In case of failure the data stored in accumulator are used to initiate the failed task.

Spark Submit

[Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is faster than [Apache Hadoop](http://data-flair.training/forums/topic/why-apache-spark-is-faster-than-hadoop) due to below reasons:

1) Apache Spark provides [in-Memory computating](http://data-flair.training/blogs/apache-spark-in-memory-computing/). Spark is designed to transform data In-memory and hence reduces time for disk I/O. While [MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) writes intermediate results back to Disk and reads it back.

2) Spark utilizes [Direct Acyclic Graph](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) that helps to do all the optimization and computation in a single stage rather than multiple stages in the MapReduce model

3) Apache Spark core is developed using [SCALA](http://data-flair.training/blogs/category/scala/)programming language which is faster than JAVA. SCALA provides inbuilt concurrent execution by providing immutable collections. While in JAVA we need to use Thread to achieve parallel execution.

The various disadvantages of [Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) are:

There is no support for real-time processing in Spark. It supports near real-time processing of live data. The real time data is divided into batches of the predefined interval. And also the result of the computation is returned in batches.

Problem with small file comes when we use Spark with a large number of small files. As [HDFS](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) allows a limited number of large files. Another place where Spark legs behind are we store the data gzipped in S3. This pattern is very nice except when there are lots of small gzipped files.

There is no dedicated file management system. It does not have its own file management system, so it relies on some other platform. For example, [Hadoop](http://data-flair.training/blogs/hadoop-introduction-tutorial-quick-guide/)or another cloud-based platform.

It is expensive. Because to keep data in-memory is quite expensive. Also, the memory consumption is very high, and it is not handled in a user-friendly manner. Apache Spark requires lots of RAM to run in-memory, thus the cost of Spark is quite high.

Apache Spark lags behind in a number of algorithms. MLlib legs behind in a number of an available algorithm like Tanimoto distance.

The job requires being manually optimized and adequate to specific datasets. The partitioning and caching are controlled manually for an authentic solution.

In Spark, the data iterates in batches. Also, scheduling and execution of each iteration take place separately.

High latency than Apache Flink.

Spark does not support record based window criteria. It only has time-based window criteria.

Back pressure Handling - Back pressure is buildup of data at an input-output when the buffer is full and not able to receive the additional incoming data. No data is transferred until the buffer is empty. Apache Spark is not capable of handling pressure implicitly rather it is done manually.