SPARK

# Overview

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-ware scheduling, fault tolerance, and load balancing.

Spark is a new framework which retains the scalability and fault tolerance of MapReduce. One of the main features Spark offers for speed is the ability to run computations in memory, and the system is also more efficient than MapReduce for complex applications running on disk.

To achieve these, Spark introduces an abstraction called resilient distributed datasets (RDDs).

# What Is Apache Spark?

Apache Spark is a cluster computing platform designed to be fast and general-purpose. Apache Spark consists of two parts: the first part is a programming model that creates a dependency graph. The second part is a runtime system which uses this graph to schedule work units on a cluster, and also transports code and data to worker nodes. At the core of Spark programming model is the Resilient Distributed Dataset or RDD.

# Spark VS MapReduce

Spark was purposely designed to support in-memory processing. The net benefit of keeping everything in memory is the ability to perform iterative computations at blazing fast speeds—something MapReduce is not designed to do, which makes Spark up to 20x faster than Hadoop for iterative applications and speeds up a real-world data analytics report by 40x.

Along with supporting simple “map” and “reduce” operations (which are supported by MapReduce), Spark also supports SQL queries, streaming data, and complex analytics such as graph algorithms and machine learning. Since Spark runs on existing Hadoop clusters and is compatible with HDFS, HBase and any Hadoop storage system, users can combine all capabilities into a single workflow while accessing and processing all data in the current Hadoop environment.

# Resilient Distributed Dataset (RDD)

An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. This partitioning function assigns elements of the collection RDDs can also be constructed from files on a local or remote filesystem like AmazonS3 or the Hadoop distributed filesystem. Once constructed, the programmer can perform transformations (map, filter, etc) and actions (toArray, length, etc) on the RDD.

## Transformations

Transformations are operations on RDDs that return new RDDs Transformations are lazily evaluated, which allow for optimizations to take place before execution. When the programmer performs an action on an RDD, all transformations are applied.

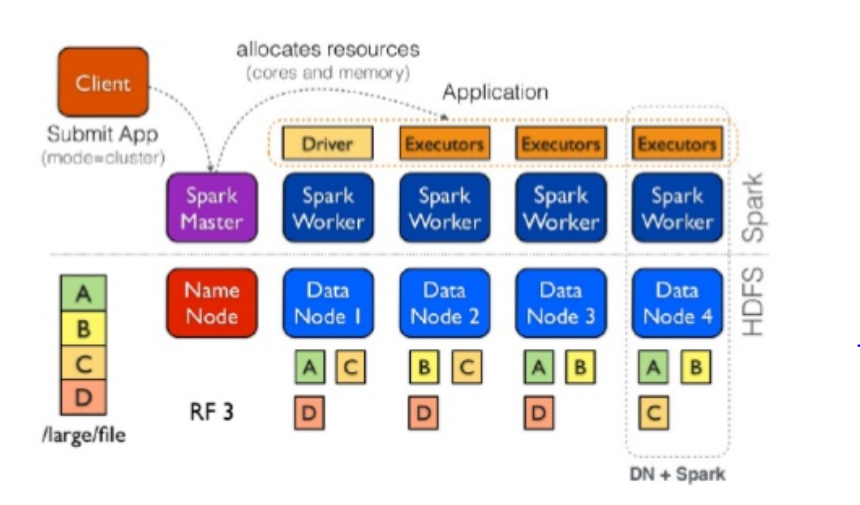
Examples: Map(), Filter(), groupByKey(), sortByKey(), Join(), Union()

## Actions

Actions are the operations on RDD which return a final value or write the data to an external storage system.

Examples: Reduce(), Collect(), First(), Take(), Count(), countByKey(), saveAsTextFile()

# Spark Runtime



A SparkContext initializes the application driver, the latter then registers the application to the cluster manager and gets a list of executors. The workers are long-lived processes that can store RDD partitions in RAM across operations.

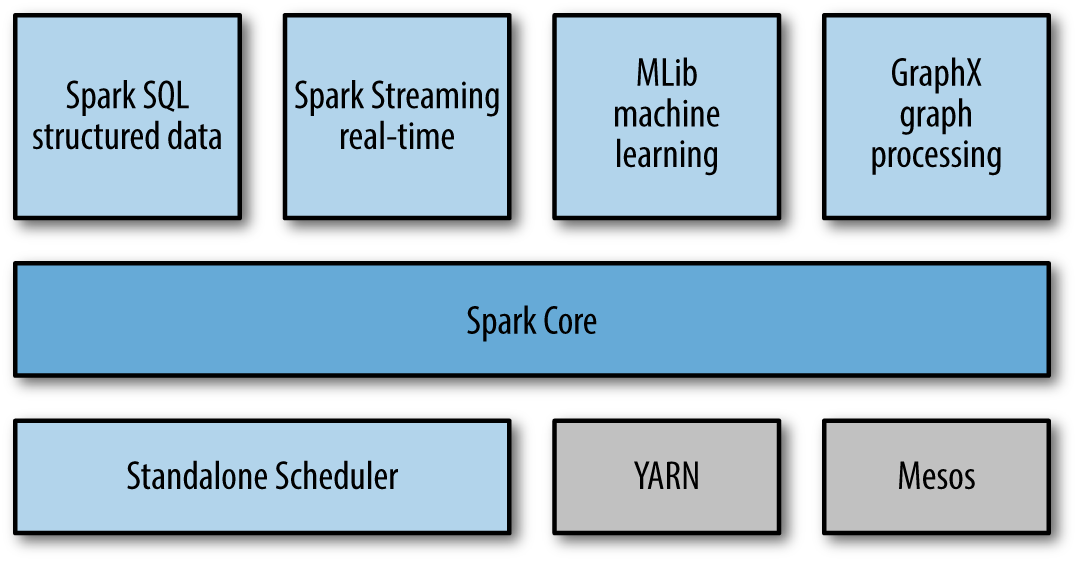
Then, the drivers takes full control of the Spark job. Spark code on the driver also tracks the

RDDs’ lineage. RDD lineage DAG is built on driver side with Data Source RDD(s) and Transformation RDD(s), which are created by transformations.

## Cluster Managers

Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler. If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; if you already have a Hadoop YARN or Mesos cluster, however, Spark’s support for these cluster managers allows your applications to also run on them.

# Spark’s components



# Fault tolerance

RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data.1 If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to re-compute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

# How Memory overflow is handled?

By default, Spark is allowed to discard partitions and re-compute them when memory is tight. Spark also gives the programmer control over the storage of intermediate RDDs. If an RDD is expensive to re-compute or if it is used several times, the programmer can indicate that the Spark runtime should persist it to a local cache. This caching ranges from in-memory caching (very fast but limited by memory) to on-disk caching (slow but plentiful). When a persisted RDD is no longer needed, the programmer can manually unpersist it or wait for it to be automatically freed when the Spark job ends.

# Advantages

A philosophy of tight integration has several benefits.

First, all libraries and higher-level components in the stack benefit from improvements at the lower layers. For example, when Spark’s core engine adds an optimization, SQL and machine learning libraries automatically speed up as well.

Second, the costs associated with running the stack are minimized, because instead of running 5–10 independent software systems, an organization needs to run only one. These costs include deployment, maintenance, testing, support, and others. This also means that each time a new component is added to the Spark stack, every organization that uses Spark will immediately be able to try this new component. This changes the cost of trying out a new type of data analysis from downloading, deploying, and learning a new software project to upgrading Spark.

Finally, one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously, analysts can query the resulting data, also in real time, via SQL (e.g., to join the data with unstructured logfiles). In addition, more sophisticated data engineers and data scientists can access the same data via the Python shell for ad hoc analysis. Others might access the data in standalone batch applications. All the while, the IT team has to maintain only one system.

# Who Uses Spark, and for What?

Because Spark is a general-purpose framework for cluster computing, it is used for a diverse range of applications. Two commonly used categories are data science and data applications.

## 1. Data Science Tasks

Spark supports the different tasks of data science with a number of components. The Spark shell makes it easy to do interactive data analysis using Python or Scala. Spark SQL also has a separate SQL shell that can be used to do data exploration using SQL, or Spark SQL can be used as part of a regular Spark program or in the Spark shell. Machine learning and data analysis is supported through the MLLib libraries. In addition, there is support for calling out to external programs in Matlab or R. Spark enables data scientists to tackle problems with larger data sizes than they could before with tools like R or Pandas.

## 2. Data Processing Applications

For engineers, Spark provides a simple way to parallelize these applications across clusters, and hides the complexity of distributed systems programming, network communication, and fault tolerance. The system gives them enough control to monitor, inspect, and tune applications while allowing them to implement common tasks quickly. The modular nature of the API (based on passing distributed collections of objects) makes it easy to factor work into reusable libraries and test it locally.

## Applications Not Suitable for RDDs

RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. RDDs would be less suitable for applications that make asynchronous fine grained updates to shared state, such as a storage system for a web application or an incremental web crawler.

# Future development

Each instance of Spark on a cluster currently has its own separate memory space. In future work, there is plan to investigate sharing RDDs across instances of Spark through a unified memory manager.

# Reference List

[1] “Spark: Cluster Computing with Working Sets” by Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker and Ion Stoica, University of California, Berkeley.

[2] “Learning Spark” by Holden Karau; Andy Konwinski; Patrick Wendell; Matei Zaharia Published by O'Reilly Media, Inc., 2015

[3] “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” by Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker and Ion Stoica, University of California, Berkeley, 2012

[4] “Scaling Machine Learning for Target Prediction in Drug Discovery using Apache Spark”. Harnie, D., Vapirev, A.E. Wegner, J.K., Gedich, A., more authors 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015.