Apache Spark got its start in 2009 at UC Berkeley’s AMPLab as a way to perform in-memory analytics on large data sets. At that time, Hadoop MapReduce was focused on large-scale data pipelines that were not iterative in nature. Building analytic models on MapReduce in 2009 was a very slow process, so AMPLab designed Spark to help developers perform interactive analysis of large data sets and to run iterative workloads, such as machine-learning algorithms, that repeatedly process the same data sets in RAM.

Spark doesn’t replace Hadoop. Rather, it offers an alternative processing engine for workloads that are highly iterative. By avoiding costly writes to disk, Spark jobs often run many orders of magnitude faster than Hadoop MapReduce. By "living" inside the Hadoop cluster, Spark uses the Hadoop data layer (HDFS, [HBase](http://www.infoworld.com/article/2610709/database/review--hbase-is-massively-scalable----and-hugely-complex.html), and so on) for the end points of the data pipeline, reading raw data and storing final results.

**Writing Spark applications**

Spark, written in [Scala](http://www.infoworld.com/article/2609013/java/scala-founder--language-due-for--fundamental-rethink-.html), provides a unified abstraction layer for data processing, making it a great environment for developing data applications. Spark comes with a choice of Scala, Java, and Python language bindings that are, for the most part, equivalent except at the bleeding edge, where only Scala implementations are available.

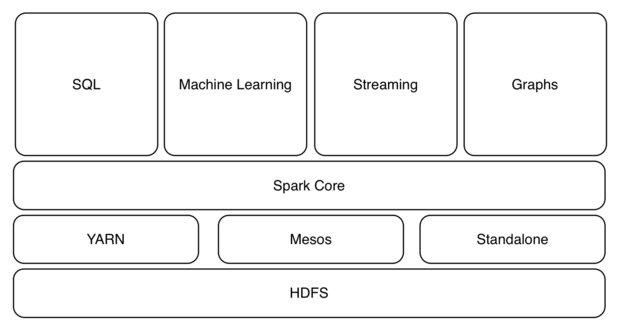
One of the nice features in Spark is the ability to work interactively from the Scala or Python console. This means you can try out code and immediately see the results of execution. This is handy both for debugging, where you can change a value and proceed again without going through a compile step, and for data exploration, where a typical process consists of tight loops of inspect-visualize-update.

Spark’s core data structure is a resilient distributed data (RDD) set. In Spark, driver programs are written as a series of transformations of RDDs, followed by actions on them. Transformations, as the name suggests, create new RDDs from existing ones by changing them in some way, such as by filtering the data according to some criteria. Actions work on the RDDs themselves. An action might be counting the number of instances of a data type or saving RDDs to a file.

Another nice benefit of Spark is the ease with which RDDs can be shared with other Spark projects. Because RDDs are used throughout the Spark stack you can freely mix SQL, machine learning, streams, and graphs in the same program.

Developers coming from other functional programming languages, such as LISP, Haskell, or F#, will have little difficulty adapting to Spark programming beyond learning the API. Thanks to the excellent collections system of Scala, applications written with Spark's Scala API can be remarkably clean and concise. Adjusting to Spark programming mostly involves keeping in mind the distributed nature of the system and knowing when object and functions need to be serializable.

Programmers coming from a background in procedural languages, like Java, may not find the functional paradigm so easy to pick up.



*Because Spark's RDDs can be shared across the system, you can freely mix SQL, machine learning, stream processing, and graph processing in the same program.*

Bottom of Form

**Resilient distributed data sets**

The consistent use of RDDs throughout the stack is one of the features that make Spark so powerful. Both conceptually and in implementation, RDDs are quite simple; most of the methods in the RDD class are less than 20 lines long. At the core, RDDs are a collection of distributed records, backed by some form of persistent storage, along with a lineage of transformations.

RDDs are immutable. You can’t modify an RDD, but you can easily create a new one with different values. Immutability is an important property of distributed data sets; it means that we don’t have to worry about another thread or process changing the value of an RDD when we’re not looking -- a traditional problem in multithreaded programming. It also means we can distribute RDDs throughout a cluster for execution without having to worry about synchronizing changes to the RDD across all nodes.

RDD immutability also plays a role in fault tolerance of Spark applications. Because each RDD keeps a history of computations to arrive at its current value and no other process could have made a change, recomputing an RDD that went down with a lost node is simply a matter of going back to the original persistent data partition and rerunning the computations on a different node. (Most partitions in Hadoop are persisted across multiple nodes.)

RDDs can be constructed from multiple types of data partitions. Most of the time data comes from HDFS, where the term "partition" is literal. However, RDDs can also be constructed from other persistent stores such as HBase, Cassandra, SQL databases (via JDBC), Hive ORC (Optimized Row Columnar) files, and other storage systems that expose a Hadoop InputFormat API. Regardless of the source of the RDD, the behavior is the same.

The final note on Spark transformations: They’re lazy, meaning that no computation is performed until an action requires a result to be returned to the driver. This is mostly relevant when working at the interactive Scala shell. As RDDs are incrementally transformed, there’s no cost -- until an action is performed. It is then that all values are computed and the results returned to the user. Further, because RDDs can be cached in memory, frequently used results don’t need to be recomputed each time.



*Spark transformations are lazy, meaning that no results are computed until an action requires the results to be returned to the user.*

**Executing Spark applications**

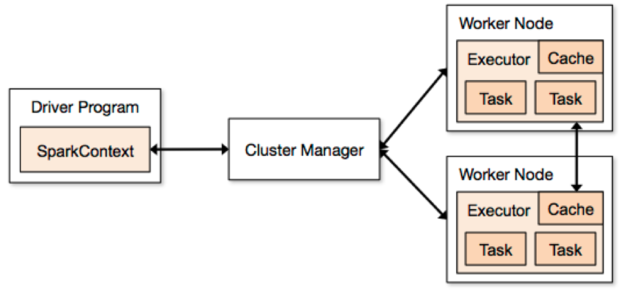
In order to submit a Spark job to the cluster, the developer needs to execute the driver program and connect it to a cluster manager (also known as a cluster master). The cluster manager presents a consistent interface to the driver, so the same application can be run on any supported cluster type.

Spark currently supports dedicated Spark (stand-alone), Mesos, and YARN clusters. Each driver program running in the cluster allocates resources and schedules tasks independently of every other. While providing application isolation, this architecture makes it difficult for the cluster to efficiently manage RAM, Spark’s most precious resource. Multiple high-memory jobs submitted simultaneously can be starved of resources. Although the stand-alone cluster manager implements a simple resource scheduler, it offers only FIFO scheduling across applications, and it is not resource aware.

Generally speaking, Spark developers must be much closer to the servers than data analysts using higher-level applications like Hive or Pig. For example, because the driver is scheduling tasks on the executors, it needs to run close to these worker nodes to avoid network latency.

Both driver and cluster manager HA are important. If the driver dies, the job will stop. If the cluster master dies, no new jobs can be submitted, but existing stages will continue to execute. As of Spark 1.1, master HA is only available with stand-alone Spark clusters via ZooKeeper. There is no driver HA.

Squeezing maximum performance from a Spark cluster can be somewhat of a black art, involving experimentation with various combinations of options for driver, executors, memory, and cores and measuring the results to optimize CPU and RAM usage for a specific cluster manager. Little documentation exists for these kinds of operational tasks, and you’ll likely have to resort to scouring mailing lists and reading source code.

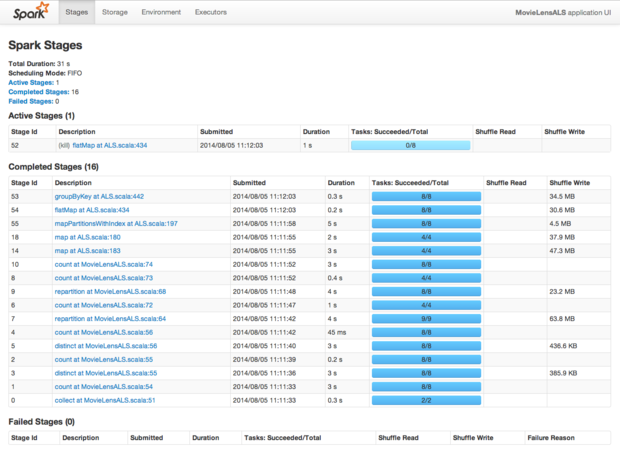


*The Spark application architecture. Spark currently can be deployed in a Spark stand-alone, YARN, or Mesos cluster. Note that each driver program running in the cluster allocates and schedules tasks independently of every other.*

**Monitoring and operations**

Each driver has a Web UI, typically on port 4040, that displays all sorts of useful information about running tasks, schedulers, executors, stages, memory and storage usage, RDDs, and so on. This UI is primarily an information tool and not one for managing the Spark application or cluster. Nevertheless, it's the go-to tool for debugging and performance tuning -- nearly everything you need to understand about what’s happening in your application is here.

Although a great start, the Web UI can be rough around the edges. For example, viewing historical jobs requires navigating to a separate history server, except when running a stand-alone-mode cluster manager. But the biggest shortcoming is the lack of any operational information or control. Starting and stopping nodes, viewing their health, and other cluster-level statistics aren’t available. Running a Spark cluster is still a command-line operation.



*Spark's Web UI provides a wealth of information about running tasks, but management of the cluster is done entirely from the command line.*

**An integrated analytics ecosystem**

The underlying RDD abstraction of Spark forms the core data structure for the entire Spark ecosystem. With modules for machine learning (MLlib), data querying (Spark SQL), graph analysis (GraphX), and streaming (Spark Streaming), a developer can seamlessly use libraries from each in a single application.

For example, a developer can create an RDD from a file in HDFS, transform that RDD into a SchemaRDD, query it with Spark SQL, then feed the results into an MLlib library. Finally, the resulting RDD can be plugged into Spark Streaming to apply predictive modeling to a message feed. Doing all of that outside of Spark would require using multiple libraries, marshaling and transforming data structures, and putting a whole lot of work into deploying it.

The integrated stack makes Spark uniquely valuable for interactive data exploration and repeated application of functions to the same data set. Machine learning is the sweet spot for Spark, and the ability to share RDDs across the ecosystem transparently greatly simplifies writing and deploying modern data analytics applications.