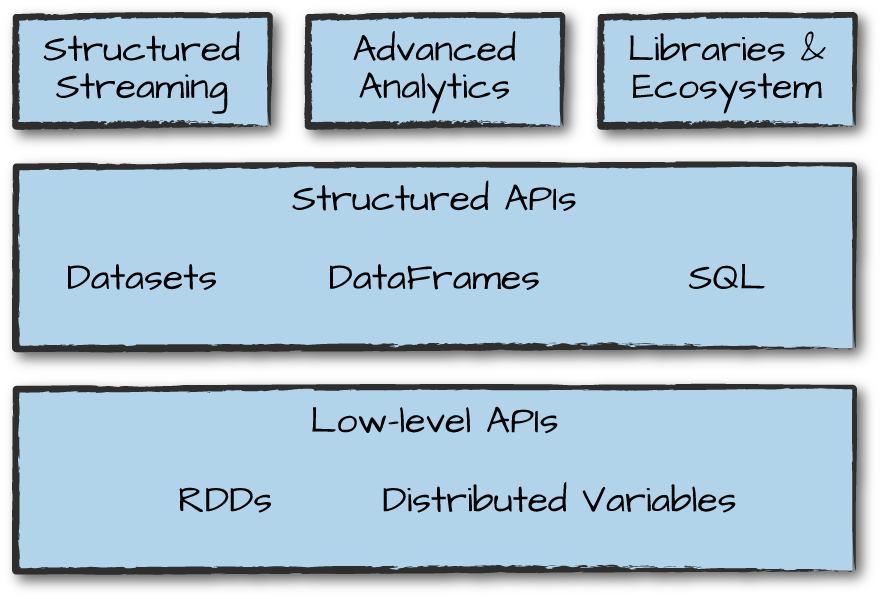
**What Is Apache Spark?**

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters. As of this writing, Spark is the most actively developed open source engine for this task, making it a standard tool for any developer or data scientist interested in big data



Spark’s focus on defining a unified platform is the same idea behind unified platforms in other areas of software. For example, data scientists benefit from a unified set of libraries (e.g., Python or R) when doing modeling, and web developers benefit from unified frameworks such as **Node.js or Django.**

**Computing engine**

 Spark handles loading data from storage systems and performing computation on it

You can use Spark with a wide variety of persistent storage systems, including cloud storage systems such as Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka.

Spark’s focus on computation makes it different from earlier big data software platforms such as Apache Hadoop

* Hadoop included both a storage system (the Hadoop file system, designed for low-cost storage over clusters of commodity servers) and a computing system (MapReduce), which were closely integrated together.
* Spark runs well on Hadoop storage, today it is also used broadly in environments for which the Hadoop architecture does not make sense, such as the public cloud  (where storage can be purchased separately from computing) or streaming applications.
* **Hadoop save HDFS storage and computing with Mapreduce.**
* **Spark maybe store many application like: Cloud and compute with API Spark**

The libraries have grown to provide more and more types of functionality. Spark includes libraries for SQL and structured data (**Spark SQL**), machine learning (**MLlib**), stream processing (**Spark Streaming and the newer Structured Streaming**), and graph analytics (**GraphX**)

# Spark’s Basic Architecture

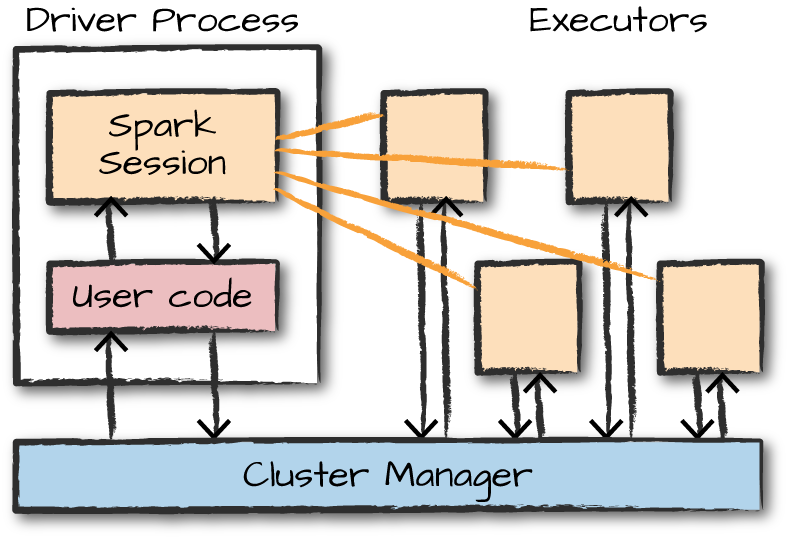
Typically, when you think of a “computer,” you think about one machine sitting on your desk at home or at work. However, as many users likely experience at some point, there are some things that your computer is not powerful enough to perform.  One particularly challenging area is data processing. A cluster, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer. Now, a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark’s standalone cluster manager , **YARN, or Mesos.**

## Spark Applications

**Spark Applications** consist of a **driver** process and a set of **executor** processes.The **driver** process runs your main() function, sits on a node in the cluster, and is responsible for **three things:** **maintaining information** about the Spark Application; **responding to a user’s program** or input; and analyzing, distributing, and **scheduling work.** The **driver** process is absolutely essential—it’s the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The executors are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for **only two things: executing code assigned to it by the driver**, and **reporting** the state of the computation on that executor back to the driver node.



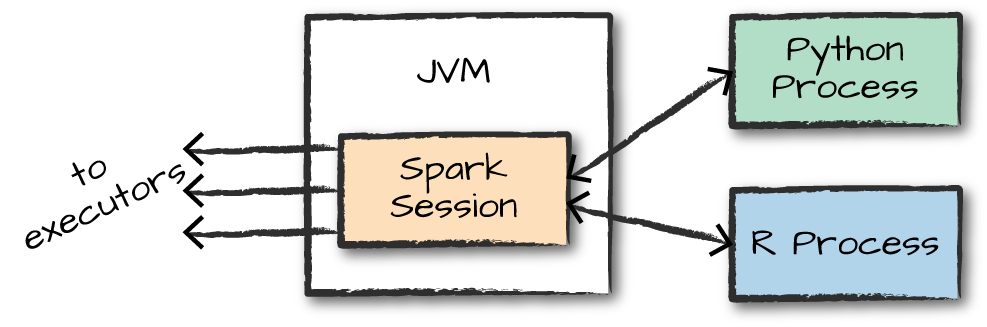
|  |
| --- |
| Spark, in addition to its cluster mode, also has a local mode. The driver and executors are simply processes, which means that they can live on the same machine or different machines. In local mode, the driver and executurs run (as threads) on your individual computer instead of a cluster |

Here are the key points to understand about Spark Applications at this point:

* Spark employs a cluster manager that keeps track of the resources available.
* The driver process is responsible for executing the driver program’s commands across the executors to complete a given task.

However, the driver can be “driven” from a number of different languages through Spark’s language APIs.

# Spark’s Language APIs



There is a SparkSession object available to the user, which is the entrance point to running Spark code. When using Spark from Python or R, you don’t write explicit JVM instructions; instead, you write Python and R code that Spark translates into code that it then can run on the executor JVMs.

Spark has two fundamental sets of APIs:

* The low-level “unstructured” APIs
* The higher-level structured APIs.

# The SparkSession

You control your Spark Application through a driver process called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.

# DataFrames

## image Partitions

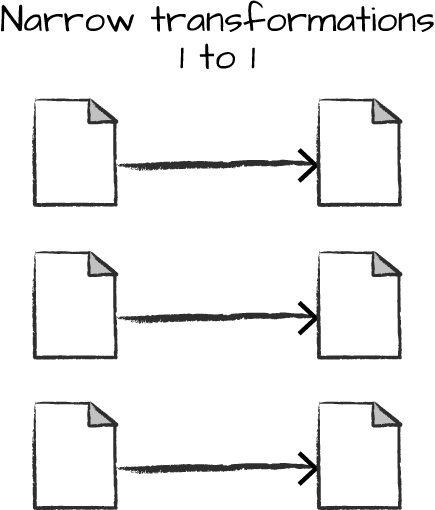
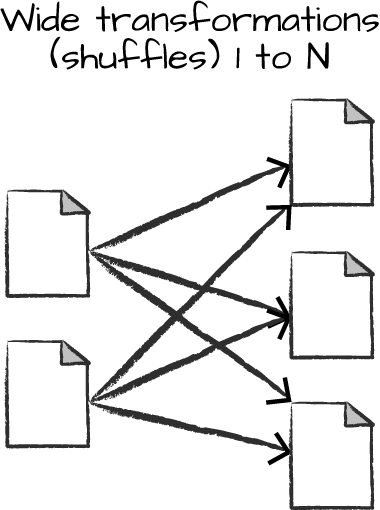
To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions. A partition is a collection of rows that sit on one physical machine in your cluster. A DataFrame’s partitions represent how the data is physically distributed across the cluster of machines during execution. If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

An important thing to note is that with DataFrames you do not (for the most part) manipulate partitions manually or individually. You simply specify high-level transformations of data in the physical partitions, and **Spark determines how this work will actually execute on the cluster.** Lower-level APIs do exist (via the RDD interface).

# Transformations( Don’t Understand)

Notice that these return no output. This is because we specified only an abstract transformation, and Spark will not act on transformations until we call an action. Transformations are the core of how you express your business logic using Spark

There are two types of transformations:

* those that specify narrow dependencies.
  + where only one partition contributes to at most one output partition
  + 
* those that specify wide dependencies.
  + input partitions contributing to many output partitions
  + 

You now can see how transformations are simply ways of specifying different series of data manipulation. This leads us to a topic called lazy evaluation.

## Lazy Evaluation

Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions

# Actions

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action. An action instructs Spark to compute a result from a series of transformations.  The simplest action is count, which gives us the total number of records in the DataFrame:

divisBy2.count()

The output of the preceding code should be 500. Of course, count is not the only action. There are three kinds of actions:

* Actions to view data in the console
* Actions to collect data to native objects in the respective language
* Actions to write to output data sources

In specifying this action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, and then a collect, which brings our result to a native object in the respective language.

This by inspecting the Spark UI, a tool included in Spark with which you can monitor the Spark jobs running on a cluster.

# Spark UI

You can monitor the progress of a job through the Spark web UI. . The Spark UI is available on port 4040 of the driver node The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state. It’s very useful, especially for **tuning** and **debugging**

