Spark

Apache Spark is a **Unified Analytics Engine** that provides great performance increases over Hadoop MapReduce. From a features perspective, it has been engineered to be **fast** (up to 100x faster than Hadoop), **easy to use** and **consolidated** (supports SQL queries, stream processing, machine learning, and graph processing).

[**Spark Philosophy**](#_5x0gsotgmfyi) **2**

[Unified Platform](#_h2dm3xkrumq0) 2

[Compute Engine](#_fvur5rcilite) 2

[Libraries](#_4wzgh4fckr9j) 2

[Ecosystem](#_flp2lxqu12mc) 3

[**Spark Architecture**](#_6okieu4kfmlb) **4**

[Overview](#_tjiappdp96jk) 4

[Driver](#_lo6t2fg38ers) 4

[Executor](#_wn768p3tsma0) 5

[Clusters and Resource Management](#_4ft37jtpg6fe) 5

[**Running Spark**](#_gemiguj27zme) **6**

[Resilient Distributed Dataset (RDD)](#_wv9tbzlcx81k) 7

[DataFrames](#_z5174rk3es3v) 7

[Partition](#_voqsup39mce7) 8

[Lazy Evaluation](#_k8pu74bors8a) 8

[Transformations](#_a8tntp4xpyxu) 9

[Narrow Transformations](#_rh7fd2x2tzsa) 9

[Wide Transformations](#_erlgc3ea2o3i) 10

[Useful Transformations](#_38lg1i191z1x) 11

[Actions](#_rms0qiz4h9ob) 12

[SQL](#_3qpyuegbvm3v) 14

[**Hadoop, Spark and Limitations**](#_wr6i9k2kr3u9) **14**

[Hadoop vs Spark](#_pfk4dwk0cs01) 15

[Spark Strengths and Limitations](#_fn7rbdh7fqrg) 16

# 

# 

# Spark Philosophy

Apache Spark is a unified computing engine and set of libraries for big data; to that end it adheres to three foundational principles as key components:

* Unified Platform
* Compute Engine
* Libraries

## Unified Platform

When we say “unified platform” what we mean is that Spark was designed to deliver support for a wide range of data analytics tasks, from data loading to SQL querying to machine learning modeling. More importantly, Spark supports the ability to mix and match these various capabilities into a single scan of data, allowing developers to for example load and filter a dataset and then apply a machine learning model over it all in the same job. Prior to Spark, in order to perform these types of tasks, one would have to stitch together various libraries and paradigms, sacrificing performance along the way.

## Compute Engine

Spark **only** handles loading data from storage systems and performing work on it. As a consequence, Spark can load data from a variety of inputs, such as distributed file systems (HDFS), cloud storage solutions (AWS S3), key value pair storage (Apache Cassandra or Redis) and append only message stores (Apache Kafka or AWS Kinesis). Spark **does not** store data long term (only in memory for computations) and generally is data storage agnostic (it doesn’t prefer data from S3 vs HDFS, and so on).

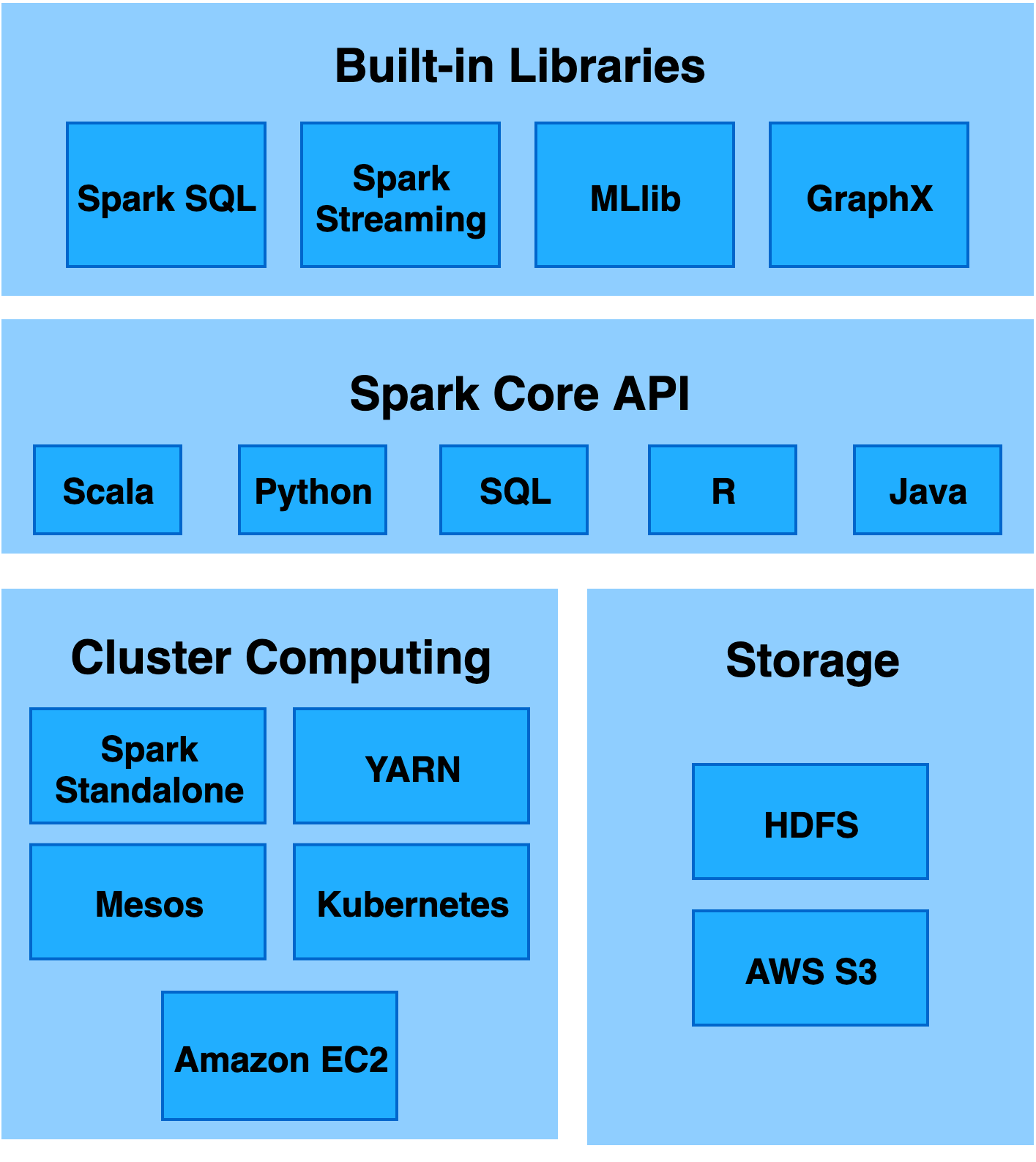
## Libraries

Spark publishes libraries that provide a common API to the core engine in a variety of different languages. This builds on the unified engine design as the methods needed to perform Spark tasks are more or less the same across different languages that connect to Spark’s core. Spark comes “built in” with libraries for SQL and structured data (Spark SQL), machine learning (MLlib), stream processing (Spark Streaming / Structured Streaming) and graph analytics (GraphX). However there are hundreds of additional external libraries as well that are maintained by a very active open source community. Check out the index on spark packages [**here**](https://spark-packages.org/)**.**

# 

## Ecosystem

The core Spark engine powers the functionality across all the libraries that are integrated with it. When a user writes Spark code in an integrated library (say PySpark), the code is translated by Spark into JVM instructions that it can manage and understand. The end user never has to explicitly write any JVM specific code. Additionally, the data sources that Spark understands can be almost anything. Overall, the ecosystem looks something like this:

[](https://www.draw.io/?page-id=s9RA_xU1StFlzUuubKDz&scale=auto#G1oXcFUbFNR3PwrfkvrjkWyOBmlUN03m5w)

**Spark Core** — Spark Core is the base engine for large-scale parallel and distributed data processing. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems. It provides in-memory computing capabilities to deliver speed and a generalized execution model to support a wide variety of applications.

**Cluster management** — A cluster manager is used to acquire cluster resources for executing jobs. Spark core runs over diverse cluster managers including YARN, Apache Mesos, and Amazon EC2. The cluster manager handles resource sharing between Spark applications.

**Storage** — Spark can access data in HDFS or any Hadoop data source, S3, Cassandra, Kafka, Redis, etc.

**Spark SQL + DataFrames** — Spark SQL integrates structured, relational processing with Spark’s functional programming API. Spark SQL is a Spark module for structured data processing that provides an interactive SQL querying shell for exploring data. To that end, it also provides a programming abstraction (or class) called a **DataFrame** (not to be confused with **pandas.DataFrame**) that allows high level interaction with the underlying data.

**Spark Streaming** — Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark’s ease of use and fault tolerance characteristics.

**MLlib (Machine Learning)** — Machine learning built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and speed (up to 100x faster than MapReduce).

**GraphX** — GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

# Spark Architecture

Like Hadoop, ideally Spark requires a “cluster” of machines to execute. These machines are managed by a **cluster manager**, specialized software that pools resources of multiple machines together allowing Spark to leverage the collective resources as a single logical unit. Spark comes with an out of the box cluster manager, called **Yarn** but also supports other, open source alternatives such as [**Apache Mesos**](http://mesos.apache.org/)**.**

## Overview

We call the process of managing, provisioning and executing work over a data set a Spark “Application”. Generally speaking, a Spark Application consists of a **driver** and a set of **executors**.

### **Driver**

The **Driver** runs the main code for the job; it will run on a single node in a cluster and manage the following three key tasks:

* Maintain metadata about the Spark Application
* Respond to the main code requirements or user input
* Schedule work across executors

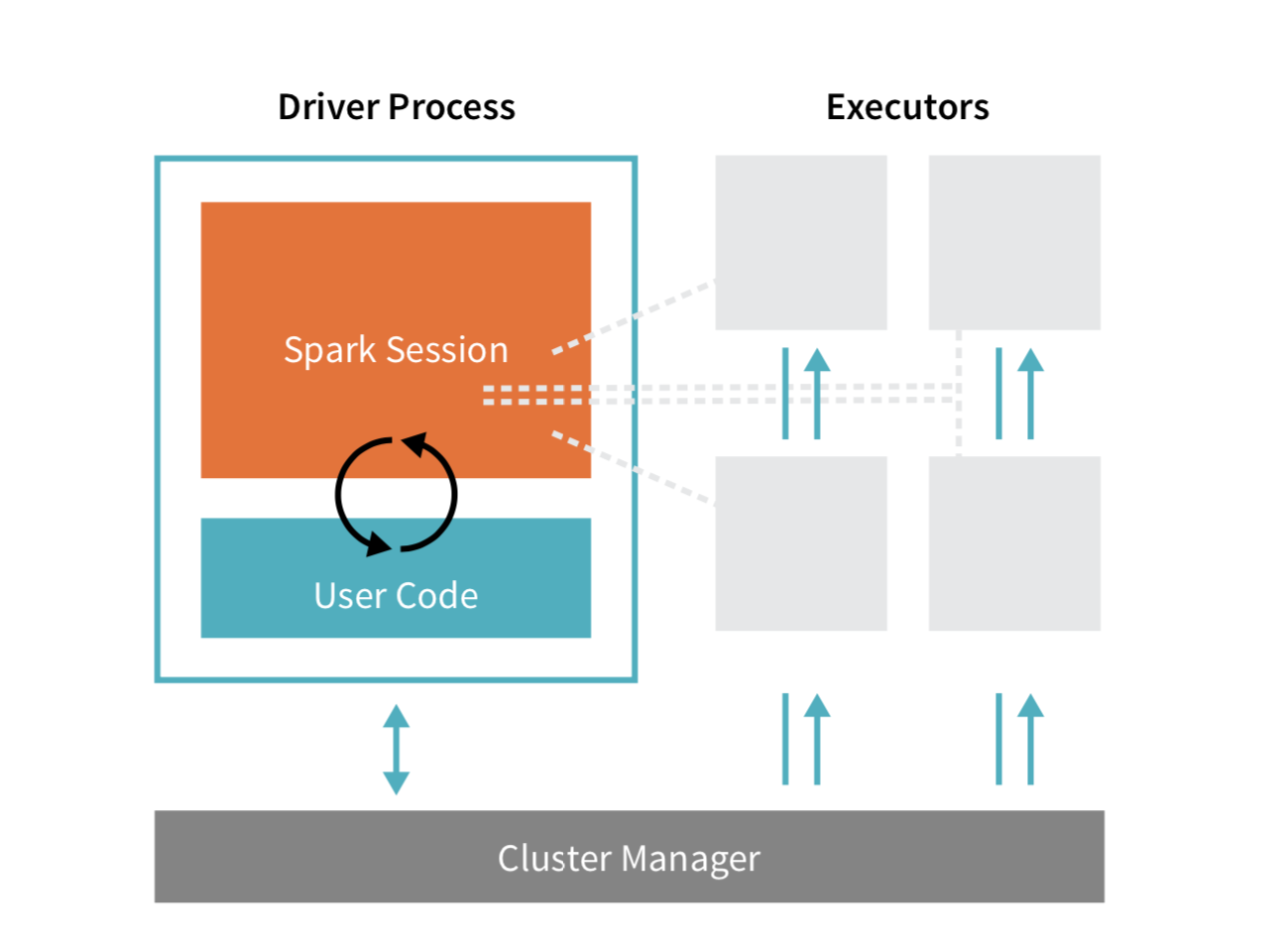
### **Executor**

The **Executor**(s) run the work (processing) that is assigned to them by the driver. Essentially, the executor is responsible for two main tasks:

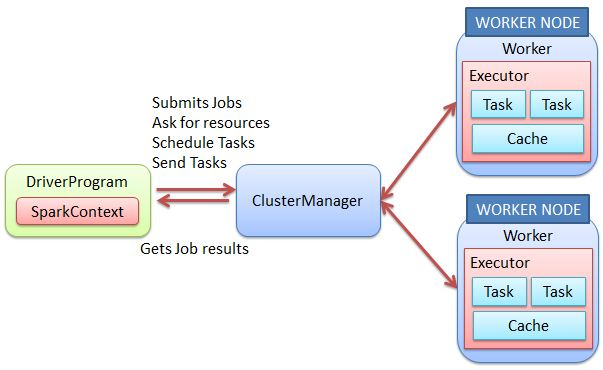
* Run application code assigned by the driver
* Report back the state of the data computation back to the driver.

### **Clusters and Resource Management**

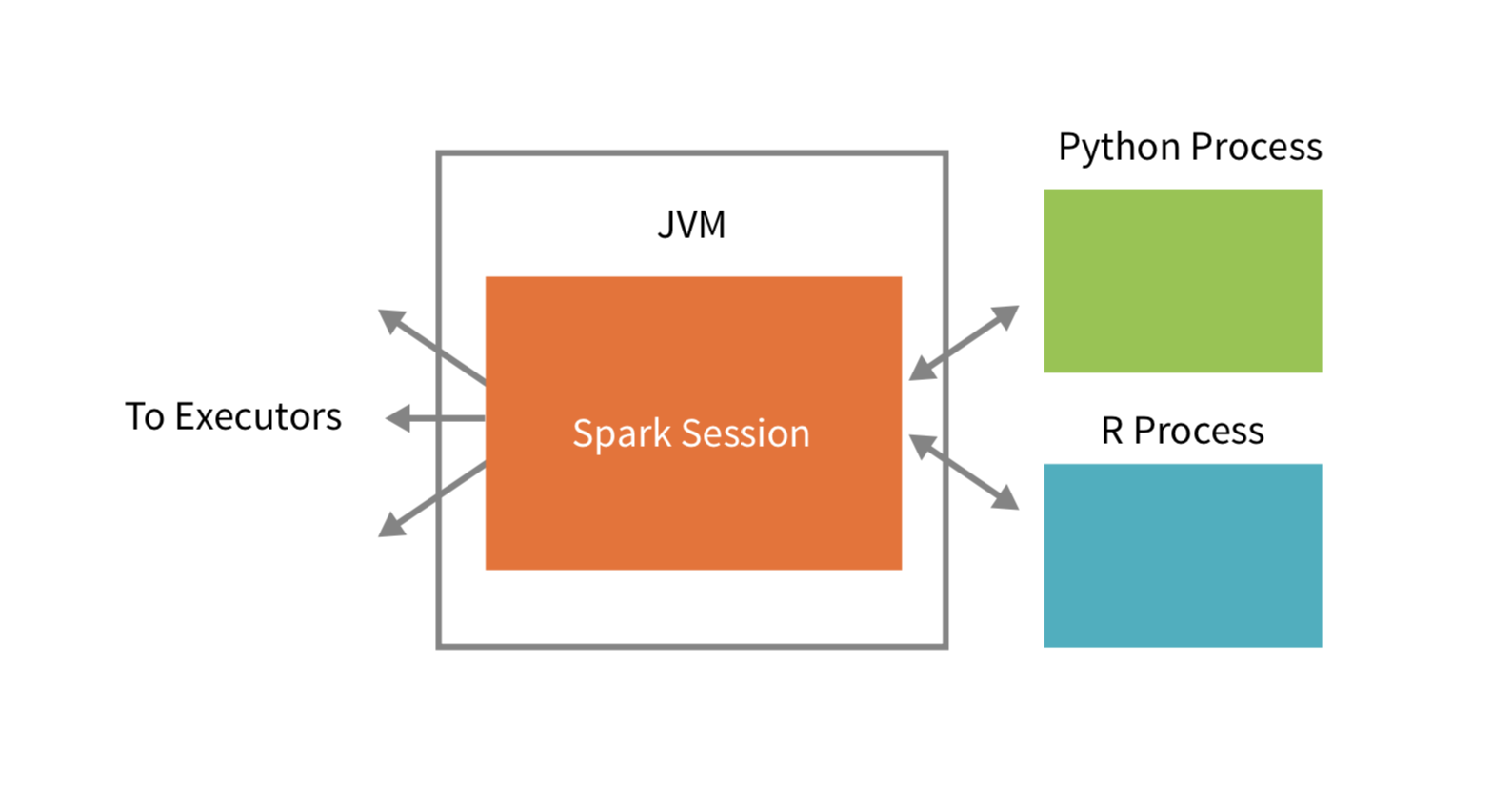
The cluster manager - Yarn or Mesors - actually manages the physical machines and allocates resources for Spark Applications.

****

From an architectural point of view, the Driver, or **DriverProgram**, lives inside a **master** node. **Executor**(s) are processes that run within **worker** nodes. Each executor runs “tasks” which are small subsets of the total work that is tracked by the driver (see diagram below).



When running Spark on a single machine, we can configure the system to run in **local** mode, where each of the components defined are run as threads on a single computer.



Generally speaking, there are multiple clients to the DriverProgram that are implemented in many languages (such as Python) whereas the executors are largely managed by the driver and typically run Spark code, written in scala.

# Running Spark

The **driver** runs the application’s main() program, regardless of language (assuming there is a library available). Note that the main() program *could* just be a command that is run in the spark shell.

The driver begins by creating the app’s **SparkSession**, which is the heart of the application. Generally, there is a 1:1 relationship between a SparkSession and a Spark Application A **SparkConf** object is available to configure properties that the driver program will send to SparkSession. Some key parameters set here deal with how the SparkSession object will request resources for the app from the Cluster Manager.

After reading the user’s program and data, SparkSession creates **Tasks**, or sets of parallelized jobs to be processed by the application. Based on these tasks, the driver program requests resources from the Cluster Manager, which in turn acquires **Worker Nodes** for the cluster per overall Spark resource availability. On each worker node, the Cluster Manager launches at least one **Executor**, a JVM process, which will complete tasks and store data in-memory. SparkSession determines which individual tasks to send all the executors. Meanwhile, the cluster manager keeps track of the status of all the worker nodes and their executors.

## Resilient Distributed Dataset (RDD)

The Resilient Distributed Dataset, or RDD, is the fundamental data abstraction provided to us by Spark. Essentially, it is a fault tolerant abstraction of elements that can be operated on in parallel. They are basically **objects** that represent a piece of data.

RDDs are **immutable**, meaning that they can never be updated - to apply changes, we must *derive* a new RDD from the old one.

There are two primary types of RDDs:

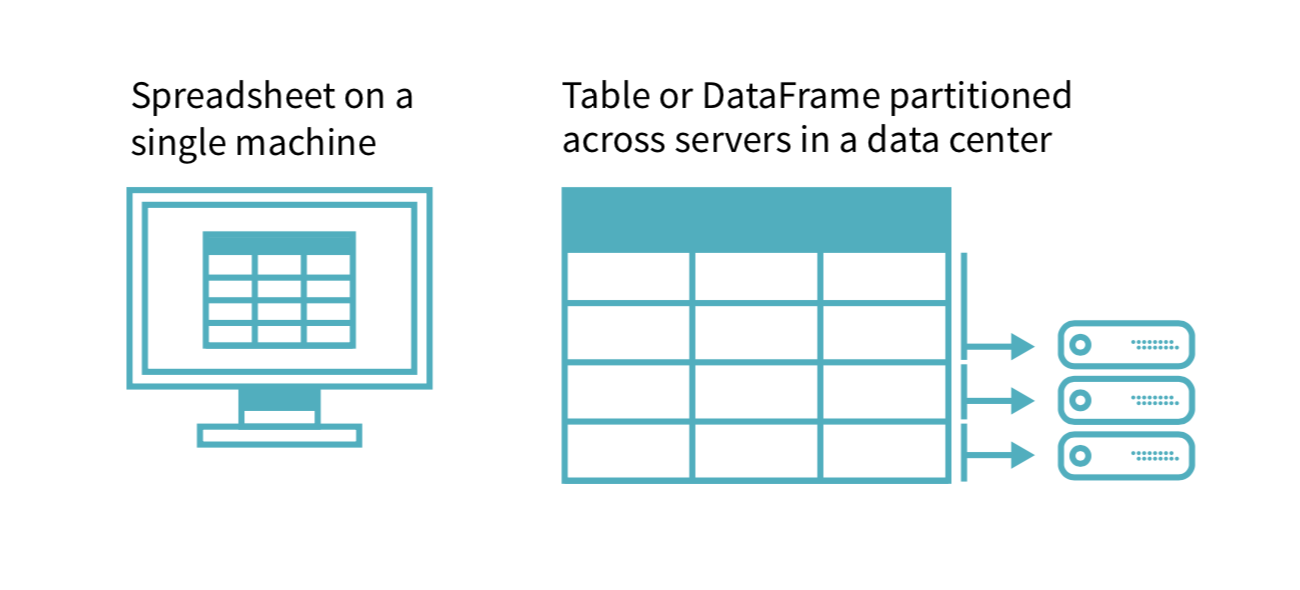
* **Parallelized Collections**: these run on **Scala** collections (basically lists) and run functions on them in parallel
* **Hadoop Datasets**: these run functions on each record in **HDFS** or any other Hadoop supported file storage system (ie: S3, etc)

While RDDs are useful, we will usually prefer to work with **DataFrames**, another fundamental data abstraction provided by Spark.

## DataFrames

The **DataFrame** represents essentially a two dimensional table of data featuring rows and columns. We define a **schema** to be the list of columns and the types of the data stored in those columns.

Simply put, a DataFrame is the Spark representation of a CSV file or the **pandas.DataFrame** object. Importantly, **Spark.DataFrame** is distributed, meaning that the data - although part of a logical “table” or spreadsheet, can live on 10s or 1000s of individual machines.



DataFrames were built to make it easier to work with Spark on non-JVM languages. Also, DataFrames support using a query language (such as SQL) to manipulate data.

### **Partition**

A partition is a chunk of data from the DataFrame that is passed along to an **executor** to be processed in parallel. Put another way, if we have 100 rows (**R)** of data in our DataFrame and a Spark cluster with 5 **worker nodes (WN)**, then:

* **P0** → **R1** ,..., **R20**, sitting in physical machine: **WN1**
* **P2** → **R21** ,..., **R40**: **WN2**
* …
* **P4** → **R81** ,..., **R100**: **WN5**

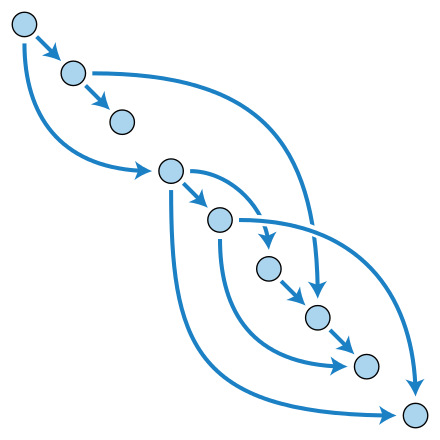
Basically, one fifth of the rows would be assigned to each partition, which represents a **physical** machine.

When interacting with DataFrames, we generally do not manually manage partitioning. Instead, we tell Spark how we would like our data to be transformed and let Spark figure out how to actually execute our transformations in the cluster. At some point, a DataFrame gets converted into an RDD by spark for execution - but by interacting with DataFrames directly and not RDDs, we do not have to worry about how this happens.

### **Lazy Evaluation**

When we operate over DataFrames and specify the actual work we want to be executed, Spark creates a list of operations and waits until the last possible moment before actually executing our instructions.

Essentially, Spark creates an **execution plan** from our instruction set and then generates a **physical plan** to run on the **executors** themselves that tries to run as efficiently as possible. Because Spark waits as long as it can before execution of the instructions, the dataflow and changes to the data can be optimized end to end. This is one of the reasons why Spark outperforms Hadoop. (**PS:** note where else we learned about lazy evaluators … with python generators!)



The instruction set we provide and the resultant execution plan that Spark derives takes an input DataFrame and then generates a new DataFrame that is the same as the previous frame but with data changes as specified by our code. This is called the **directed acyclic graph** (or DAG), it provides a good mental model for thinking about how we operate over data in Spark. Note from the figure that these types of graphs can by definition never “cycle” (as in, create a loop). So we are guaranteed to go from point A to point B no matter what. However, modeling our dataflow with this concept in mind allows Spark to implement performance optimizations.

## Transformations

In order to operate over DataFrames, we define **transformations** that take each row of our DataFrame and apply some sort of change or update to it. Transformations are the “arrows” in our DAG chart from above - each “arrow” generates a new DataFrame which is the next “node” in the chart.

Note also that when we actually define a transform we don’t actually get a result right away, the result is only available once we invoke an **action** (which begins the “processing” of our transforms).

When dealing with transformations, there are two types.

### **Narrow Transformations**

These are transforms where each input partition will result in a single output partition. Because partitions live on the same machine, this is more efficient. For example, if our DataFrame consisted only of 10 numbers and 2 partitions - we can imagine that:

**P0** → **1,2,3,4,5**

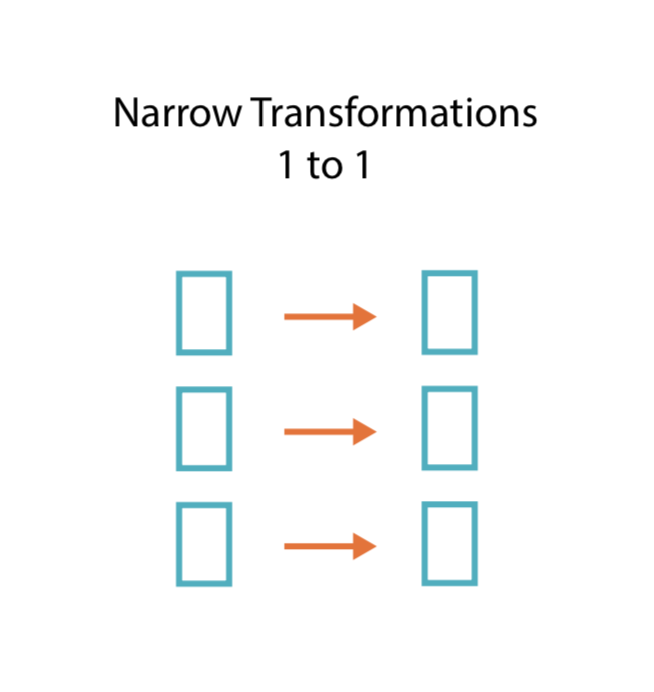
**P1** → **6,7,8,9,10**

Now, if we define a transformation that adds **one** to each number, we don’t need to “shuffle” our partitions - since they are simply being added to. We end up with:

**P0** → **2,3,4,5,6**

**P1** → **7,8,9,10,11**

Now is a “narrow transformation”:



### **Wide Transformations**

In contract, a wide transformation will require a value or row in the input partition “switching” to a different output partition. **Shuffling** is a good example here. For instance, if we wanted to transform out DataFrame such that the evens were in the first 5 rows and the odds in the last 5 rows:

**P0** → **1,2,3,4,5**

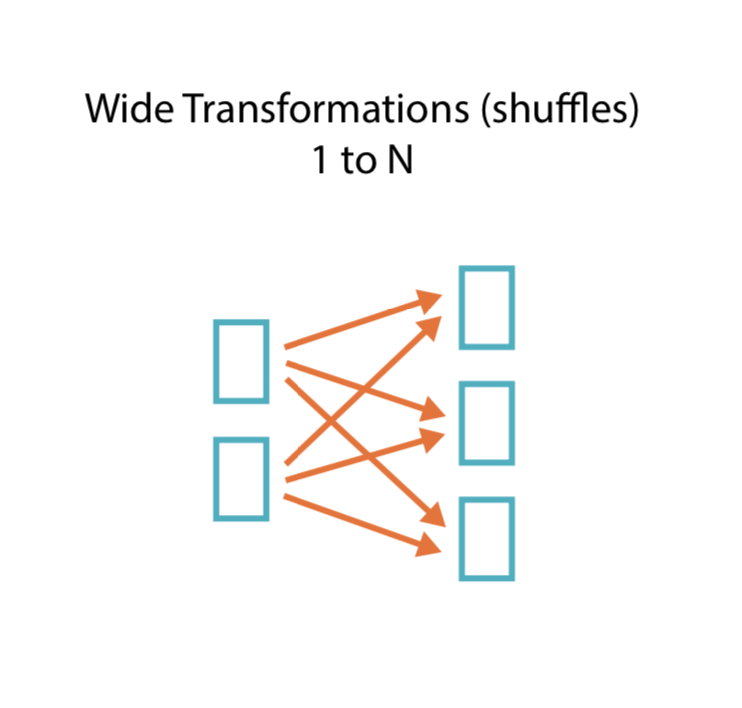
**P1** → **6,7,8,9,10**

To:

**P0** → **2,4,6,8,10**

**P1** → **1,3,5,7,9**

Then we end up with a “shuffle” in that the values are changing positions. When this happens, Spark must exchange data from partitions across the cluster. To accomplish this, Spark will have to write the results of the transformation to disk whereas for narrow transformations - the entire operation can be accomplished in memory. As such, wide transformations are less performant.



### **Useful Transformations**

|  |  |
| --- | --- |
| **transformation** | **description** |
| **map(func)** | return a new distributed dataset formed by passing each element of the source through a function func |
| **filter(func)** | return a new dataset formed by selecting those elements of the source on which func returns true |
| **flatMap(func)** | similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item) |
| **sample(withReplacement, fraction, seed)** | sample a fraction fraction of the data, with or without replacement, using a given random number generator seed |
| **union(otherDataset)** | return a new dataset that contains the union of the elements in the source dataset and the argument |
| **distinct([numTasks]))** | return a new dataset that contains the distinct elements of the source dataset |
| **groupByKey([numTasks])** | when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs |
| **reduceByKey(func, [numTasks])** | when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function |
| **sortByKey([ascending], [numTasks])** | when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument |
| **join(otherDataset, [numTasks])** | when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key |
| **cogroup(otherDataset, [numTasks])** | when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith |
| **cartesian(otherDataset)** | when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements) |

## Actions

Actions allow us to run our DAG and apply the transformations we have queued up to our dataset. There are three types of actions:

* Display data in console
* Convert data to native objects in the client language (ie: Python, R)
* Write to output data sources

For example, going back to our partitions from above:

**P0** → **1,2,3,4,5**

**P1** → **6,7,8,9,10**

If we applied a **map** transform to add **1** to each number (a **narrow** transform) and then applied a **count** action, the map would only run once we specified the **count** action to run.

|  |  |
| --- | --- |
| **action** | **description** |
| **reduce(func)** | aggregate the elements of the dataset using a function func (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel |
| **collect()** | return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data |
| **count()** | return the number of elements in the dataset |
| **first()** | return the first element of the dataset – similar to take(1) |
| **take(n) generator seed** | return an array with the first n elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements |
| **takeSample(withReplacement, fraction, seed)** | return an array with a random sample of num elements of the dataset, with or without replacement, using the given random number generator seed |
| **saveAsTextFile(path)** | write the elements of the dataset as a text file (or set  of text files) in a given directory in the local filesystem,  HDFS or any other Hadoop-supported file system.  Spark will call toString on each element to convert  it to a line of text in the file |
| **saveAsSequenceFile(path)** | write the elements of the dataset as a Hadoop  SequenceFile in a given path in the local filesystem,  HDFS or any other Hadoop-supported file system.  Only available on RDDs of key-value pairs that either  implement Hadoop's Writable interface or are  implicitly convertible to Writable (Spark includes  conversions for basic types like Int, Double, String,  etc). |
| **countByKey()** | only available on RDDs of type (K, V). Returns a  `Map` of (K, Int) pairs with the count of each key |
| **foreach(func)** | run a function func on each element of the dataset –  usually done for side effects such as updating an  accumulator variable or interacting with external  storage systems |

## SQL

Because Spark simply converts our instructions into DAG style transformations, we can operate on a dataset with DataFrames or SQL and achieve the same effect. For example, given the following dataset:

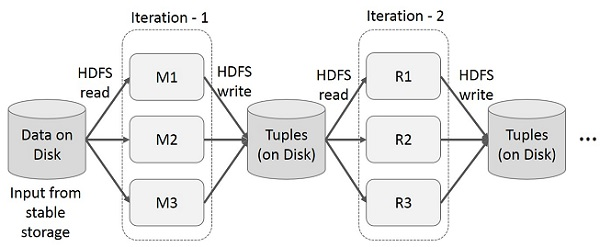
|  |
| --- |
| $ head /mnt/defg/flight-data/csv/2015-summary.csv   DEST\_COUNTRY\_NAME,ORIGIN\_COUNTRY\_NAME,count United States,Romania,15 United States,Croatia,1 United States,Ireland,344 |

We can run our Spark analysis either using the DataFrame approach *or* as a SQL query:

|  |
| --- |
| # create table flightData2015.createOrReplaceTempView("flight\_data\_2015")  # run query on table sqlWay = spark.sql(""" SELECT DEST\_COUNTRY\_NAME, count(1) FROM flight\_data\_2015 GROUP BY DEST\_COUNTRY\_NAME """) sqlWay.explain()   # create dataframe dataFrameWay = flightData2015.  # aggregate on dataframe dataFrameWay.groupBy("DEST\_COUNTRY\_NAME").count() dataFrameWay.explain() |

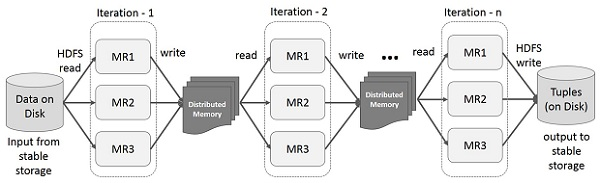
As it turns out, the **explain** method, which displays the DAG plan for execution, will be the same.

# Hadoop, Spark and Limitations

Although both technologies are written with the JVM (Scala compiles to JVM), Spark supports multiple popular programming languages, including Python, SQL, R, and Java. Because of Spark’s single, flexible core engine and consistent set of APIs, it is generally more versatile than Hadoop MapReduce. Its ability to unify contrasting techniques and processes into a well integrated platform is why Spark is generally regarded by many as the de facto tool for processing and analyzing Big Data.

There are a few key drawbacks to Hadoop Map Reduce when compared to Spark. The figure above demonstrates how we might be able to run iterative data analysis on Map Reduce (ie: compound transforms). Essentially, we would run 2 or N mapreduce jobs from init to completion. This would incur a good deal of overheads due to data replication, disk I/O, and serialization, making the system very slow.

Compare this to Spark:



Here, **N** transforms are supported “out of the box” thanks to DAG and additionally, since Spark waits as long as it possibly can before executing, it has the ability to optimize end to end. Plus, Spark attempts to store partitions in memory and writes to IO only for wide transformations - meaning inherently Spark will be a lot faster.

## Hadoop vs Spark

|  |  |  |
| --- | --- | --- |
| **Factors** | **Hadoop MapReduce** | **Spark** |
| **Written in** | Java | Scala |
| **Ease of Use** | Long & intricate; requires dense manual code for even simple tasks | More concise and straightforward |
| **Speed** | Faster than traditional system | 100x faster than Hadoop MapReduce |
| **Handling large datasets** | Batch processing | Batch, iterative, interactive, near real-time streaming, graph computation |
| **Caching** | Data caching not supported; reads and writes from disk | Distributed, in-memory data caching to improve system performance |
|  | Runs as heavier weight JVM processes | Runs multi-threaded tasks inside of JVM processes for faster startup, better parallelism, and better CPU utilization |

One area where Hadoop MapReduce out-performs Spark is handling *extremely* large datasets. If the data is so large that insufficient RAM becomes an issue, it might be better to opt for traditional Hadoop MapReduce. That said, Spark can integrate with Hadoop’s HDFS and MapReduce modules to leverage their capacity for handling such large data.

## Spark Strengths and Limitations

|  |  |
| --- | --- |
| **Strengths** | **Limitations** |
| Speedy processing using a “state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine” | Spark Streaming isn’t *really* “real time” processing - at the end of the day even streamed data is still treated as a small RDD batch |
| Dynamic/versatile | High memory consumption/more expensive |
| Use with interactive shells in Scala, Python, R, and SQL | Does not have its own file management system |

## 