

EDEM



Master Data Analytics

Apache Spark Sessions 2023-24

Introduction to Apache Spark

Pablo Pons Roger

0. Apache Spark Popularity

Expansion

Huge community: with over than 2k contributors, 39k commits and 28k forks, as you can see in the [Apache Spark Github repo](#)

Companies:

- Many companies using it in projects with different architectures (two tier, microservices, etc)
- Among some of them: Yahoo, NASA JPL - Deep Space Network, eBay Inc., Alibaba Taobao, etc
- And of course Databricks a company founded by some of the original Spark creators

The most widely-used engine for scalable computing

Thousands of companies, including 80% of the Fortune 500, use Apache Spark™.
Over 2,000 contributors to the open source project from industry and academia.

You can find out more about this on [Spark](#) and [Databricks](#) websites.

It should be noted that it is also a skill with **high employability** in the labor market. So you are on the right track ;)

Popular Spark Use Cases

Whether you are a data engineer, data scientist, or machine learning engineer, you'll find Spark useful for the following use cases:

- Processing in parallel large data sets distributed across a cluster
- Performing ad hoc or interactive queries to explore and visualize data sets
- Building, training, and evaluating machine learning models using Mllib
- Implementing end-to-end data pipelines from myriad streams of data
- Analyzing graph data sets and social networks

1. Before Apache Spark

Monolithic vs Distributed Architectures

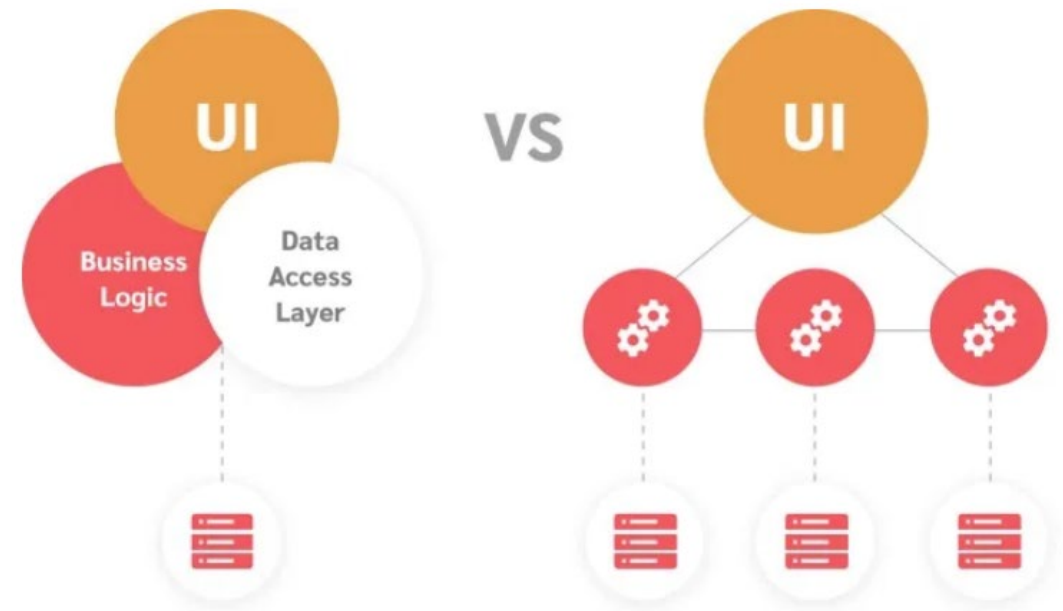
Monolithic Architecture:

“An architecture is monolithic when all the code is a single unit of deployment.” [1]

Distributed Architecture:

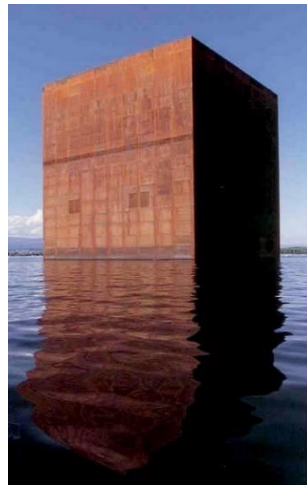
“A distributed architecture is a group of deployment units connected through remote access protocols.” [1]

[1] Richards M., Ford N.,(2020) Fundamentals of Software Architecture (An Engineering approach), edited by O'Reilly Media Inc. Chapter “Monolithic Versus Distributed Architecture.”



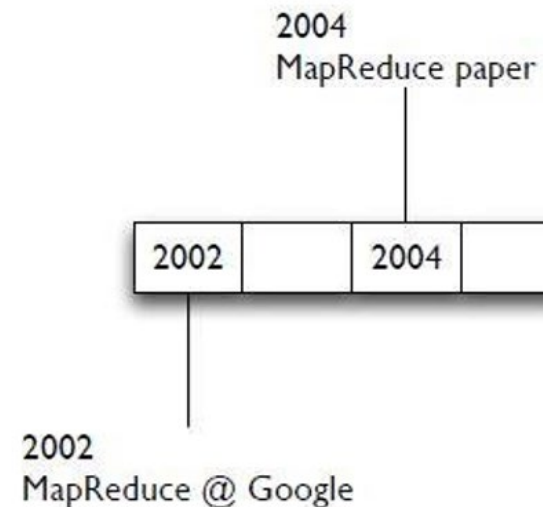
Monolithic vs Distributed Architectures

- For decades, the goal was a bigger, more powerful machine, but this approach has limitations
- As applications grow, characteristics like maintainability, agility, testability, and deployability are adversely affected
- Also it has a **high cost** and **limited scalability**
- These are the reasons why a monolithic architecture is not a good approach for a big data application



MapReduce

- Google set out to develop its search engine to index and search the world's data on the internet at lightning speed
- Neither traditional storage systems nor imperative ways of programming were able to handle the scale at which Google wanted to operate
- Need of new approaches → creation of the Google File System (GFS), MapReduce (MR), and Bigtable



MapReduce

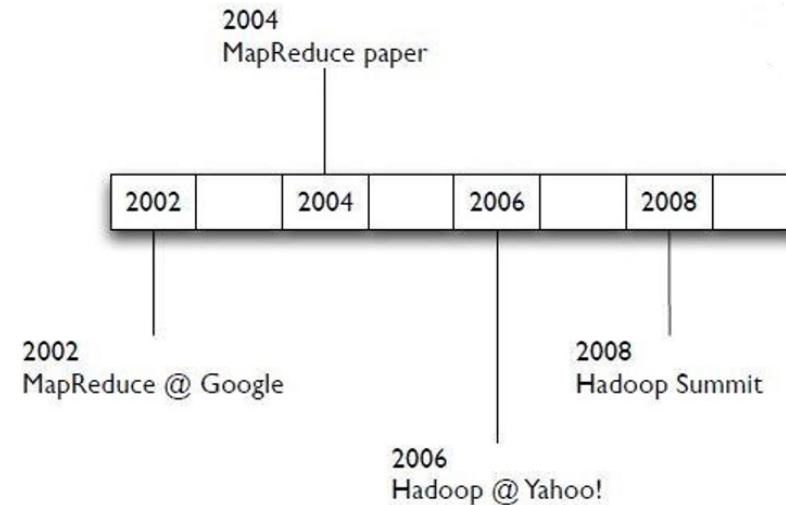
MapReduce (MR) introduced a new parallel programming paradigm.

- Based on functional programming, for large-scale processing of data distributed over GFS and Bigtable
- MR applications send computation code (map and reduce functions) to where the data resides instead of bringing data to your application → key in Big Data scenarios where your volume of data is huge
- Worker nodes in the cluster aggregate and reduce the intermediate computations and produce a final appended output from the reduce function → reduces network traffic and keeps most of the input/output (I/O) local to disk

HDFS

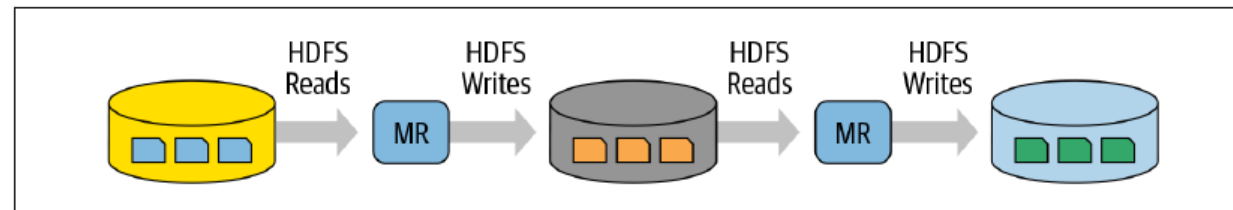
The computational challenges and solutions expressed in Google's GFS paper provided a blueprint for the Hadoop Distributed File System (HDFS).

- Donated to the Apache Software Foundation (ASF) in April 2006
- Became part of the Apache Hadoop framework of related modules: Hadoop Common, MapReduce, HDFS, and Apache Hadoop YARN
- Inspired a large open source community of contributors and two open source–based commercial companies: Cloudera and Hortonworks



MapReduce Frameworks on HDFS Shortcomings

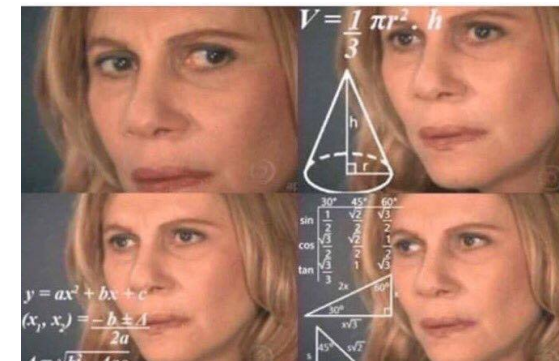
- It was hard to manage and administer, with cumbersome operational complexity
- Its general batch-processing MapReduce API was verbose
- With large batches of data jobs with many pairs of MR tasks → each intermediate computed result is written to the local disk for the subsequent stage of its operation
- This repeated performance of disk I/O took its toll: large MR jobs could run for hours on end, or even days



MapReduce Frameworks on HDFS Shortcomings

- Moreover, Hadoop MR fell short for combining other workloads such as machine learning, streaming, or interactive SQL-like queries
- To handle these new workloads, engineers developed bespoke systems (Apache Hive, Apache Storm, Apache Impala, Apache Mahout, etc.)
- Each with their own APIs and cluster configurations → more operational complexity

So, the question then became, was there a way to make Hadoop and MR simpler and faster?

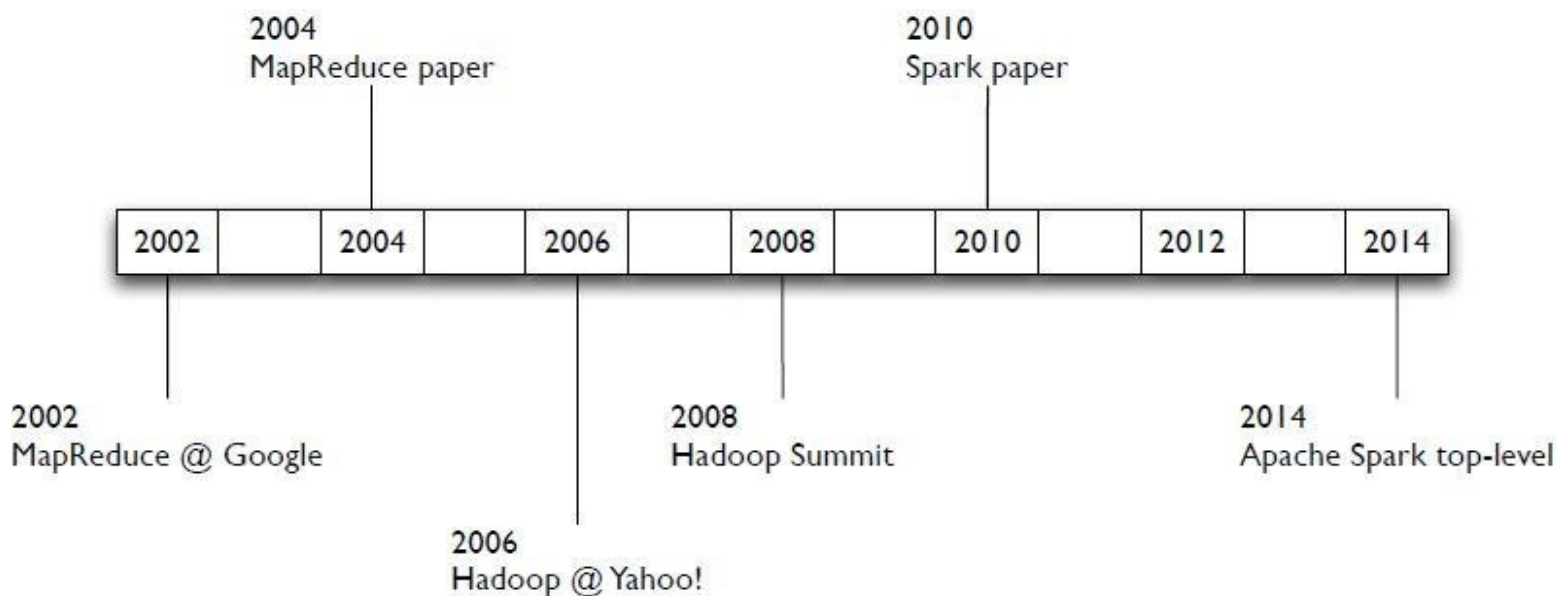


Spark's First Steps

- In 2009 Researchers at UC Berkeley who had previously worked on Hadoop MapReduce took on this challenge with a project they called *Spark*
- The aim was to bring in ideas from Hadoop MapReduce, but also to enhance the system:
 - Make it highly fault tolerant and embarrassingly parallel
 - Support in-memory storage for intermediate results of map and reduce computations
 - Offer easy and composable APIs in multiple languages as a programming model
 - And support other workloads in a unified manner

Spark's First Steps

- By 2013 Spark had gained widespread use, and some of its original creators and researchers donated the Spark project to the ASF and formed a company called **Databricks**
- Databricks and the community of open source developers worked to release Apache Spark 1.0 in May 2014, under the governance of the ASF

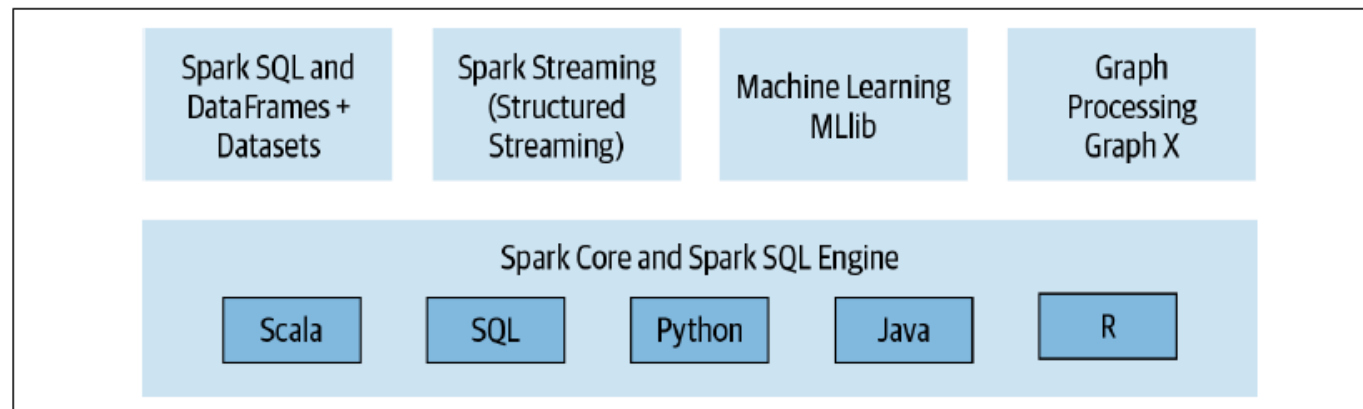


2. Introducing Apache Spark

What Is Apache Spark

*“Apache Spark is a unified engine designed for **large-scale distributed data processing**, on premises in data centers or in the cloud.”*

- Provides in-memory storage for intermediate computations, making it much faster than Hadoop MapReduce (up to **100 times** for certain jobs)
- It incorporates libraries with composable APIs for: Machine learning (MLlib), SQL for interactive queries (Spark SQL), Stream processing (Structured Streaming), Graph processing (GraphX)



Apache Spark Key Characteristics

Apache Spark provides the following major benefits:

1. Speed:

- Its internal implementation is designed to take advantage of the latest advances in the cost and performance of CPUs and memory → taking advantage of efficient multithreading and parallel processing
- Query computations as a directed acyclic graph (DAG) that can usually be decomposed into tasks that are executed in parallel
- Intermediate results retained in memory and its limited disk I/O → gives a huge performance boost
- Its physical execution engine, Tungsten, uses whole-stage code generation to generate compact code for execution

Apache Spark Key Characteristics

2. Ease Of Use:

- Provides a fundamental abstraction of a simple logical data structure called a RDD → all other higher-level structured data abstractions (DataFrames and Datasets) are constructed upon this
- Transformations and actions as operations → simple programming model and enhanced productivity as you can keep the focus on the content of computation
- Many supported programming languages: Scala, Java, Python, SQL, and R

3. Modularity:

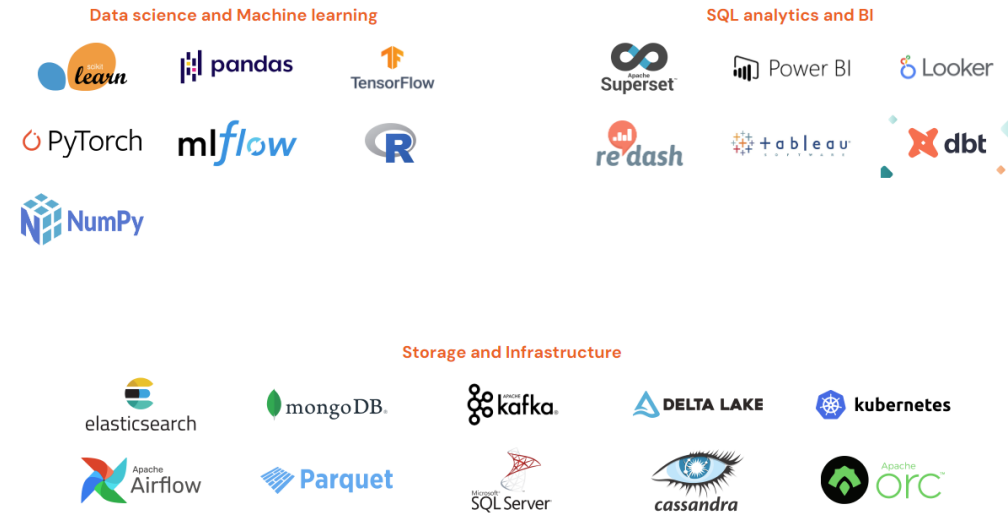
- Unified processing engine for many types of workloads (SQL, ML, Streaming, Graphs, etc)
- No need for distinct engines for disparate workloads
- Can write a single Spark application that can do it all

Apache Spark Key Characteristics

4. Extensibility:

- Spark decouples storage and computation → need of a **distributed storage system**
- Support for many sources: Hadoop, Cassandra, HBase, MongoDB, Hive, RDBMSs, etc
- DF readers and writers can also be extended to read from other sources such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3

The community of Spark developers maintains a list of third-party Spark packages including connectors for a variety of external data sources, performance monitors, and more.

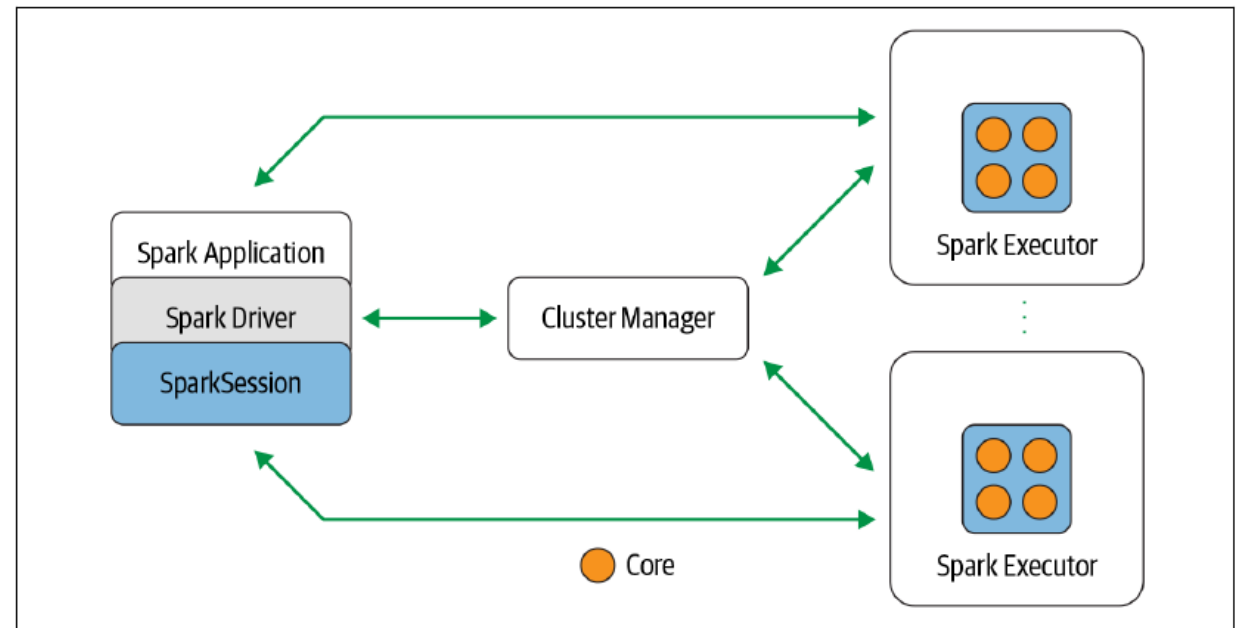


3. Apache Spark Architecture

Apache Spark Components

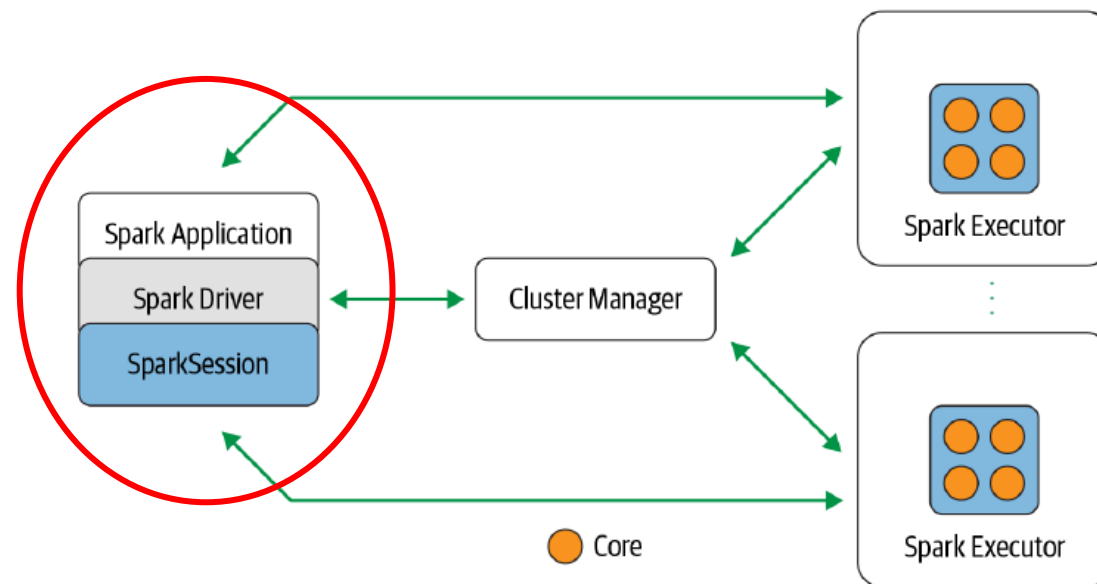
- As we have seen so far, Spark is a **distributed** data processing engine, so we will have a **cluster of machines** working collaboratively to get the job done
- At a high level in the Spark architecture, a **Spark application** consists of a **driver program** that is responsible for orchestrating parallel operations on the Spark cluster

- We can see the different components of the Spark architecture in this Figure. Let's look at each of them in more detail



Spark Driver

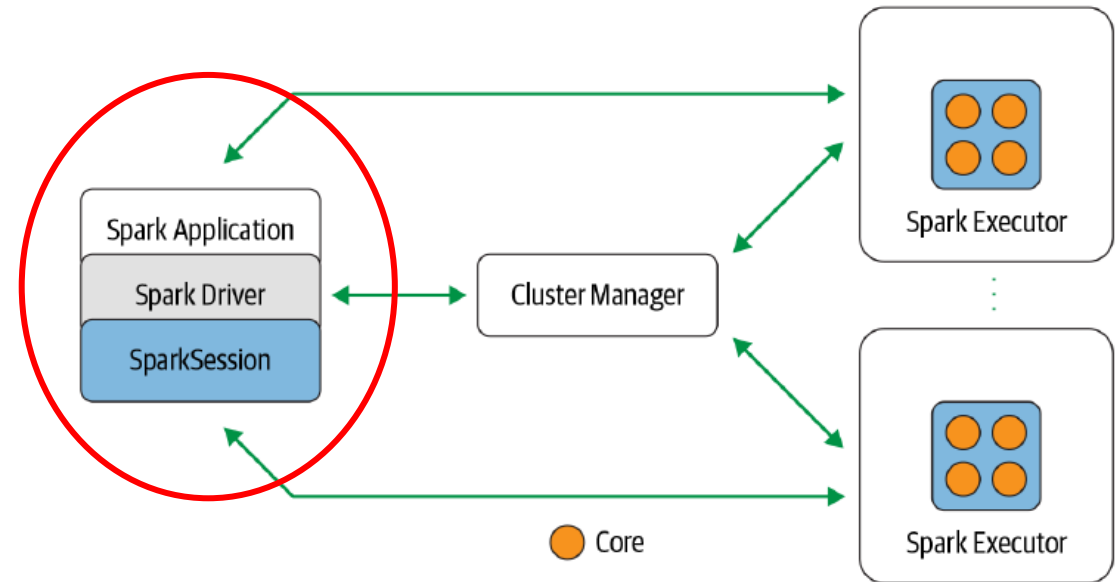
- Responsible for **instantiating a SparkSession**
- Requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs)
- Transforms all the Spark operations into DAG computations, schedules them and distributes their execution as tasks across the Spark executors



Spark Session

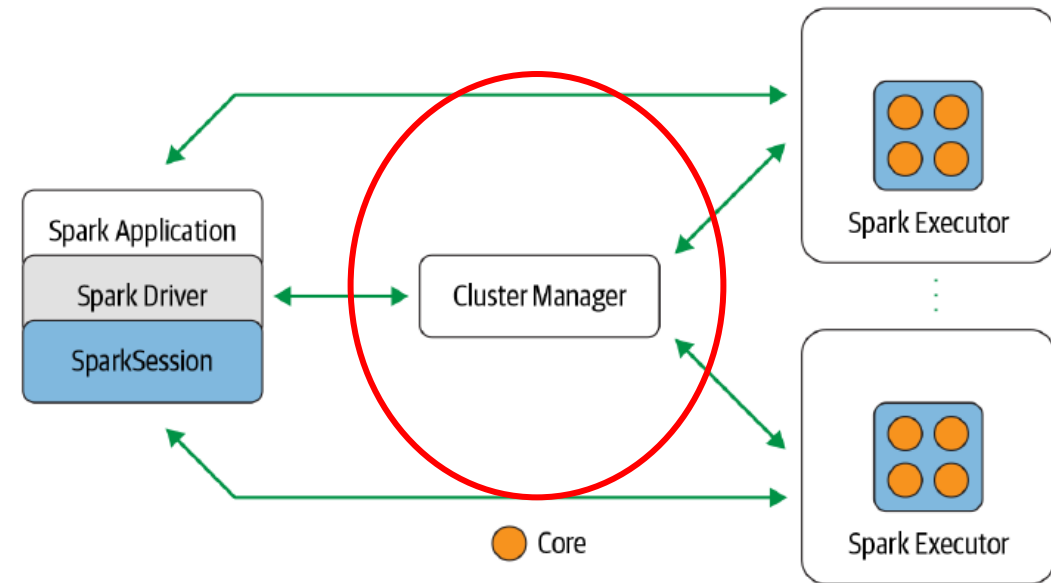
- A single unified entry point to all of Spark's functionality
- Through this conduit, you can for example:
 - Create JVM runtime parameters
 - Define DataFrames and DataSets
 - Read from data sources
 - Access catalog metadata
 - Issue Spark SQL queries

- Before Spark 2.0: more than one entry point (SparkContext, SQLContext, HiveContext, etc)
→ now unified in SparkSession



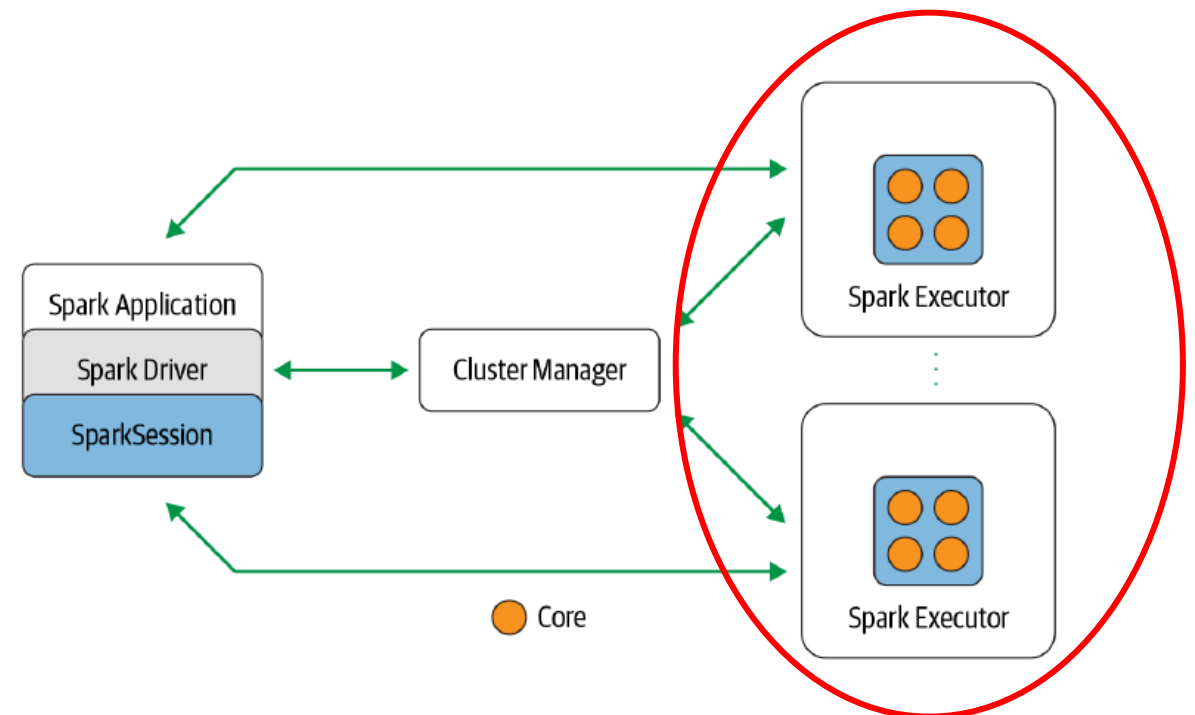
Cluster Manager

- Responsible for managing and allocating resources for the nodes on which your Spark application runs
- Spark supports four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes



Spark Executor

- A Spark executor runs on each worker node in the cluster
- Responsible for executing tasks on the workers
- In most deployments modes, only a single executor runs per node



Deployment Modes

The cluster manager is agnostic to where it runs → enables Spark to run in different configurations and environments

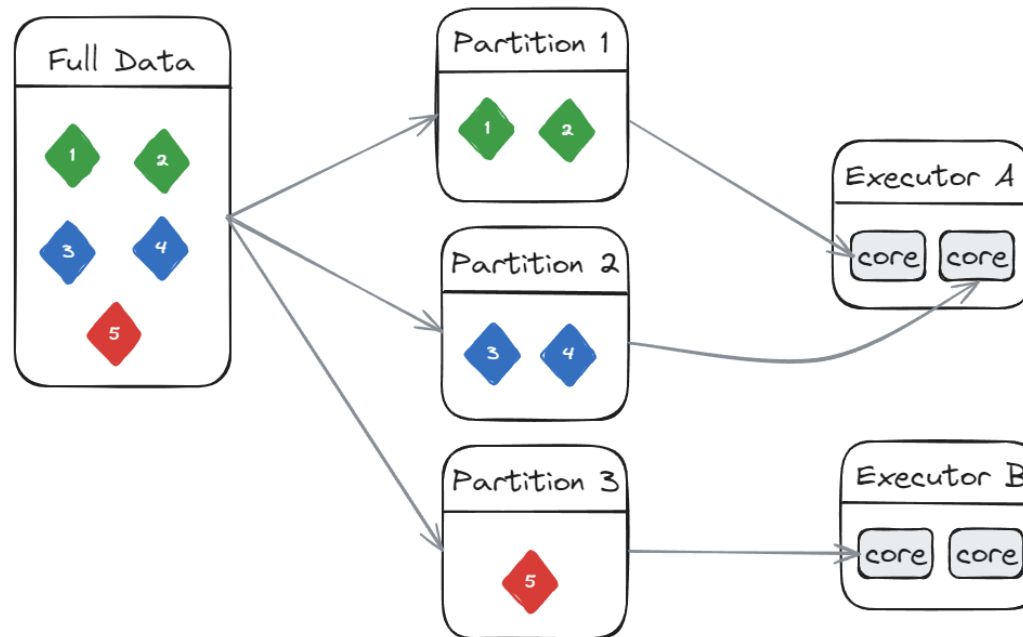
Below Table summarizes the available deployment modes:

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManager for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

4. Spark Parallel Processing

Distributed Data and Partitions

- Actual physical data is distributed across storage as partitions residing in the **distributed storage system** (HDFS, S3, etc)
- **Data locality:** Spark executors process only data that is close to the (whenever possible) → minimizes network bandwidth allowing efficient parallelism

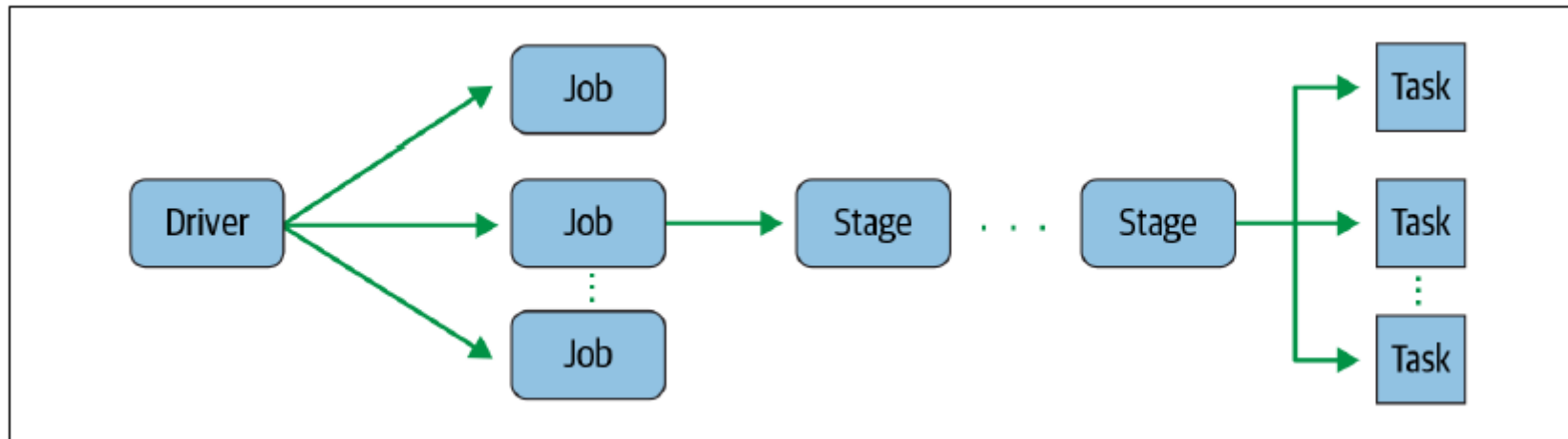


Jobs, Stages & Tasks

Spark Jobs: the driver converts your Spark application into one or more Spark jobs → then transforms each job into a DAG

Spark Stages: DAG could be a single or multiple Spark stages, created based on what operations should be performed serially or in parallel or depending on the operator's computation boundaries

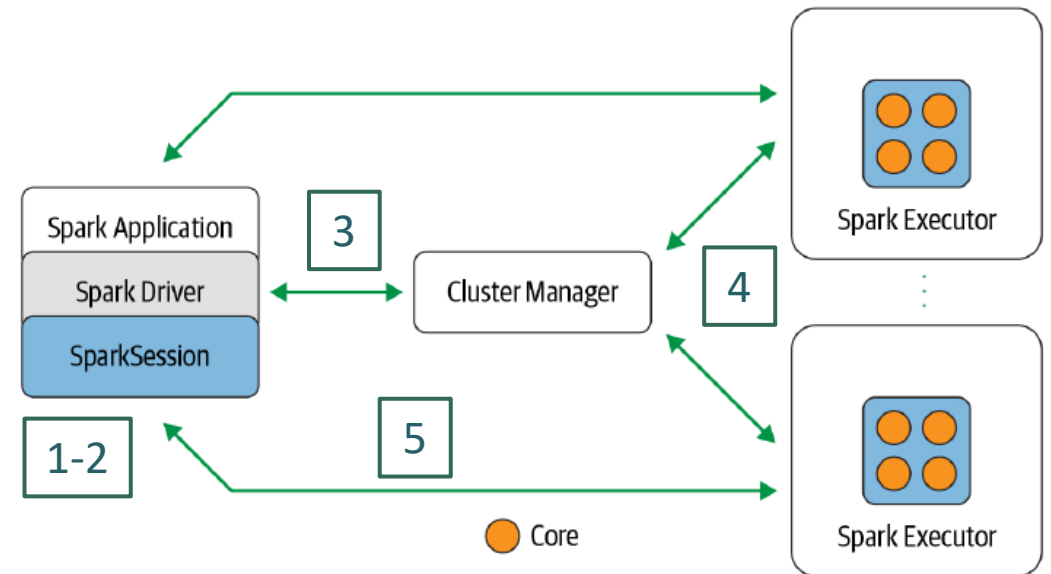
Spark Tasks: each stage is comprised of Spark tasks (a unit of execution) and each task maps to a single core and works on a single partition of data



Process Overview

Therefore, an overview of the process would be:

1. Driver instantiates a SparkSession
2. Transforms Spark operations into DAG, schedules and distributes them as tasks (jobs -> stages -> tasks)
3. Requests resources from the cluster manager for the executors
4. The resources are allocated
5. Driver communicates directly with the executors



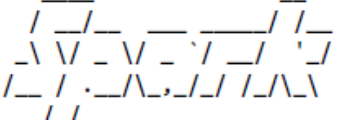
5. Spark Applications & Execution

Spark Shell

Spark provides interactive shells, very useful for exploratory analysis.

To start PySpark, cd to the bin directory and launch a shell by typing pyspark:

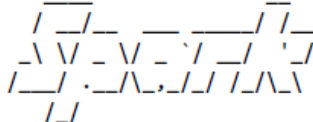
```
$ pyspark
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
20/02/16 19:28:48 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Welcome to
```

The Spark logo is a stylized representation of a star or a snowflake, composed of several small triangles pointing outwards from a central point. It is rendered in a light blue color.

version 3.0.0-preview2

To start a similar Spark shell with Scala, cd to the bin directory and type spark-shell:

```
$ spark-shell
20/05/07 19:30:26 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Spark context Web UI available at http://10.0.1.7:4040
Spark context available as 'sc' (master = local[*], app id = local-1581910231902)
Spark session available as 'spark'.
Welcome to
```

The Spark logo is a stylized representation of a star or a snowflake, composed of several small triangles pointing outwards from a central point. It is rendered in a light blue color.

version 3.0.0-preview2

Spark Submit

But to launch a spark job in general we must use the **spark-submit** command.

- We can pass many configurations to it, such as connection jars, reserved memory for the executors, etc
- For more information check the following links: [submit docs](#) + [config options](#)

```
$SPARK_HOME/bin/spark-submit mnmcount.py data/mnm_dataset.csv
```

```
+-----+-----+-----+  
|State|Color |Total|  
+-----+-----+-----+  
|CA   |Yellow|1807 |  
|WA   |Green |1779 |  
|OR   |Orange|1743 |
```