The evolution of data processing frameworks in the context of big data can be summarized as follows:

1. \*\*MapReduce\*\*: The MapReduce programming model and framework were popularized by Google's research paper in 2004 and became the foundation of Apache Hadoop. The initial Hadoop implementation used the MapReduce framework for distributed data processing. While effective, as discussed earlier, it had certain limitations that spurred the development of more advanced frameworks.

2. \*\*YARN (Yet Another Resource Negotiator)\*\*: YARN was introduced in Hadoop 2.0, released in 2013, as a response to the limitations of the original MapReduce framework. YARN separated the resource management and job scheduling aspects from the MapReduce processing model, making the Hadoop ecosystem more versatile and allowing for the integration of other data processing frameworks. YARN aimed to improve cluster resource utilization and enable a broader range of applications beyond MapReduce.

3. \*\*Spark\*\*: Apache Spark, developed at the University of California, Berkeley, was introduced in 2014 as an alternative to both the original MapReduce framework and YARN. Spark introduced in-memory data processing, which dramatically improved the processing speed compared to Hadoop's disk-based processing. It also provided a more flexible programming model, support for multiple data sources, and libraries for various data processing tasks like machine learning and graph processing.

So, the historical sequence is indeed MapReduce, followed by YARN, and then Spark. Apache Spark's success and rapid adoption were partly due to its ability to address many of the limitations of the original MapReduce framework and YARN, offering a faster, more flexible, and more capable platform for large-scale data processing.

In Apache Spark, a distributed data processing framework, the concepts of Cluster Manager, Driver, and Executor are fundamental components that work together to execute Spark applications across a cluster of machines. Let's break down each of these components:

1. \*\*Cluster Manager\*\*:

The Cluster Manager is responsible for managing the resources and coordination of the cluster where a Spark application is going to run. It's responsible for allocating resources (CPU, memory) to various Spark applications and ensuring their efficient execution. Spark supports different cluster managers, including:

- \*\*Standalone\*\*: Spark comes with its standalone cluster manager, which can be used to manage clusters exclusively for Spark applications.

- \*\*Apache Hadoop YARN\*\*: YARN can also be used as a cluster manager for Spark applications, allowing Spark to share cluster resources with other applications in the Hadoop ecosystem.

- \*\*Apache Mesos\*\*: Mesos is another cluster manager that Spark can use. It provides more fine-grained resource sharing and is suitable for multi-framework clusters.

2. \*\*Driver\*\*:

The Driver is the central control and coordination component of a Spark application. It runs the main function and coordinates the execution of tasks across the cluster. The Driver contains the SparkContext, which is the entry point to Spark's functionality. It interacts with the Cluster Manager to acquire resources and schedule tasks. The Driver is responsible for:

- Dividing the application into tasks and stages.

- Creating and managing the execution plan.

- Communicating with Executors to launch tasks and gather results.

- Monitoring and managing the application's progress.

3. \*\*Executor\*\*:

Executors are worker processes that run on each node in the cluster. They are responsible for executing tasks assigned to them by the Driver. Executors manage the actual data and computation. Each application has its own set of Executors running on different nodes. Executors perform the following tasks:

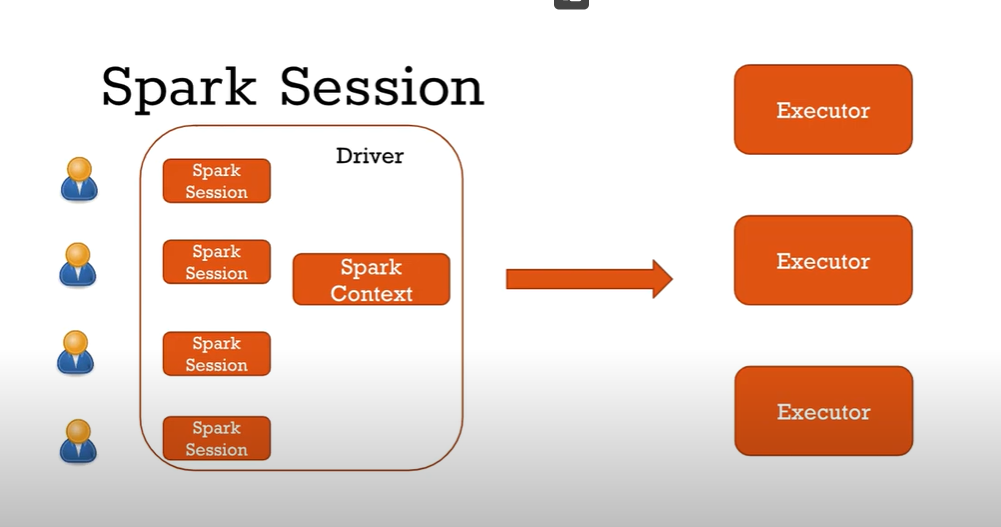
- Running tasks as directed by the Driver.

- Storing data in memory or on disk, depending on Spark's caching mechanism.

- Managing the local computation and data storage for tasks.

- Sending task status updates and results back to the Driver.

In summary, in an Apache Spark application, the Cluster Manager handles resource allocation, the Driver coordinates the execution of tasks, and Executors perform the actual computations on the data. This combination of components allows Spark to efficiently process data in parallel across a distributed cluster of machines.



For an application there will be one sparkContext but can have multiple sparkSessions



what is spark context?

In Apache Spark, the SparkContext (short for Spark Context) is one of the most important and fundamental components. It serves as the entry point to the Spark functionality and provides the essential connection between a Spark application and the Spark cluster.

The SparkContext is created by the Driver program (the main program of your Spark application) and represents the interaction point with the cluster manager. It is responsible for coordinating the execution of tasks across the cluster and managing the various resources required by the application.

Here are some key responsibilities and functions of the SparkContext:

1. \*\*Cluster Connection\*\*: The SparkContext establishes a connection to the cluster manager (Standalone, YARN, Mesos) and requests resources to run tasks.

2. \*\*Resource Management\*\*: It manages the allocation of resources such as CPU cores and memory for executing tasks on the cluster's worker nodes.

3. \*\*Distributed Data\*\*: SparkContext provides methods to create RDDs (Resilient Distributed Datasets) and DataFrames, which are the fundamental data abstractions in Spark for distributed data processing.

4. \*\*Broadcasting\*\*: The SparkContext allows for the broadcasting of read-only variables to be efficiently shared across the cluster, reducing the need to transfer data multiple times.

5. \*\*Accumulators\*\*: It supports the creation and management of accumulators, which are variables that can be used to accumulate values across multiple tasks in parallel.

6. \*\*Job and Task Management\*\*: The SparkContext orchestrates the execution of Spark jobs by dividing them into stages and tasks, and it handles fault tolerance and recovery mechanisms.

7. \*\*Communication\*\*: SparkContext communicates with Executors on the worker nodes to coordinate the execution of tasks and retrieve results.

The SparkContext was the primary entry point in earlier versions of Spark. However, starting from Spark 2.0, the recommended entry point is the SparkSession, which encapsulates the SparkContext along with additional features for working with structured data using DataFrames and Datasets. The SparkSession also simplifies the management of configuration settings and other aspects of Spark applications.

Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These abstractions all represent distributed collections of data however they have different interfaces for working with that data. The easiest and most efficient are DataFrames, which are available in all languages. We cover Datasets at the end of Part II and RDDs in Part III of this book. The following concepts apply to all of the core abstractions.