Spark

What is Spark?

Apache Spark is an open-source data processing engine to store and process data in the real time across various clusters of computers using simple processing constructs. It is a powerful cluster computing platform in the Big Data ecosystem, designed to be both fast and general-purpose. It extends the popular Hadoop MapReduce model by efficiently supporting a broader range of computations, including interactive queries and stream processing. Unlike Hadoop, Spark can perform computations in memory, which makes it faster and more efficient.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools.

Spark is designed to be highly accessible, offering simple APIs in R, Python, Java, Scala, and SQL, and rich built-in libraries. So that developers and data scientists can incorporate Spark into their applications to rapidly query analyze, and transform data at scale.

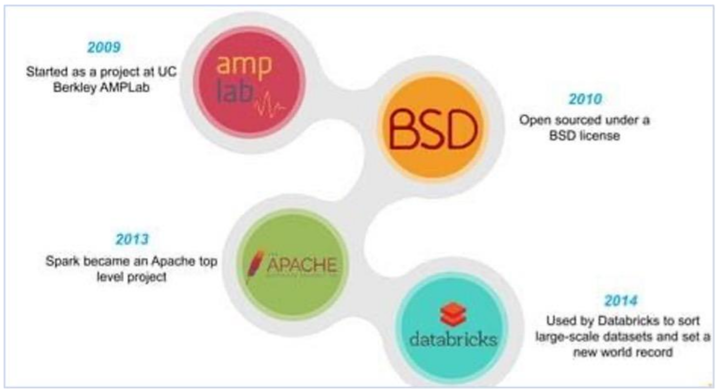
History of Spark:

Apache Spark began in 2009 as a research project at the University of California, Berkeley, in the RAD Lab, which later became known as the AMPLab. The goal was to create a faster and more flexible alternative to Hadoop's MapReduce, which was found to be inefficient for certain tasks like iterative machine learning and interactive data analysis. Spark was designed to overcome these limitations by supporting in-memory data processing and quicker data reuse. It was initially released under the BSD license, which allowed for open and flexible use by developers and organizations.

By 2010, Spark was already showing impressive results—being 10 to 20 times faster than MapReduce for specific workloads. In 2013, the project was donated to the Apache Software Foundation, where it quickly gained popularity in the big data community. It officially became a top-level Apache project in 2014.

In 2014, Databricks, a company founded by the original creators of Spark, played a major role in its development and adoption. Databricks contributed to the ongoing improvement of Spark and built a cloud-based platform that made it easier for organizations to use Spark for large-scale data analytics, machine learning, and artificial intelligence. Their work helped Spark grow into a widely used and trusted data processing engine.

Over time, Spark expanded to include several built-in libraries such as Spark SQL for structured data, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for real-time data. These made Spark a powerful unified engine for large-scale data processing. Today, it is widely used by companies and researchers around the world for big data and machine learning applications.


2009
Started as a project at UC Berkley AMPLab
amp lab
2010
Open sourced under a BSD license
BSD
2013
Spark became an Apache top level project
APACHE
2014
Used by Databricks to sort large-scale datasets and set a new world record
databricks

What are the differences between Hadoop and Spark?

- **Faster Processing:** Hadoop processes data using MapReduce, which is slower due to disk-based operations, while Spark processes data in-memory, making it up to 10 times faster.
- **Processing Type:** MapReduce supports only batch processing, whereas Spark supports both batch processing and real-time processing.
- **Code Efficiency:** Hadoop is written in Java and generally requires more lines of code, making development and execution slower. Spark, written in Scala, needs fewer lines of code and offers faster execution.
- **Security & Integration:** Hadoop uses Kerberos authentication, which is secure but difficult to manage. Spark supports easier authentication via a shared secret and can also run on YARN, leveraging Kerberos-based security when needed.

What are the features of Spark?

- **Fast Processing**: Spark processes data much faster than traditional systems like Hadoop MapReduce by reducing disk I/O and optimizing execution plans.
- **In-Memory Computing**: Spark stores intermediate data in memory (RAM) instead of writing to disk, which significantly speeds up processing—especially for iterative tasks like machine learning and graph computations.
- **Flexibility**: Spark supports multiple languages such as Scala, Python, Java, and R. It can run on various cluster managers like Hadoop YARN, Apache Mesos, Kubernetes, or as a standalone application.
- **Fault Tolerance**: Spark automatically recovers lost data using lineage information. If a node fails, Spark only recomputes the affected part of the data rather than restarting the entire job.
- **Better Analytics**: Spark offers a unified platform for a wide range of analytics tasks, supporting:
    1. **Spark SQL** – for processing structured data using SQL queries
    2. **MLlib** – for scalable machine learning algorithms
    3. **GraphX** – for graph analytics and computations
    4. **Spark Streaming** – for real-time data stream processing

- **Advanced Analytical Capabilities**: With its combination of speed, flexibility, and rich libraries, Spark enables complex data analysis, real-time insights, and intelligent applications—all in one ecosystem.

High-level Architecture of Spark

A Spark application involves five key entities: a driver program, a cluster manager, workers, executors, and tasks.

Workers: A worker provides CPU, memory, and storage resources to a Spark application. The workers run a Spark application as distributed processes on a cluster of nodes.

Cluster Managers: Spark uses a cluster manager to acquire cluster resources for executing a job. A cluster manager manages computing resources across a cluster of worker nodes. It provides low-level scheduling of cluster resources across applications. It enables multiple applications to share cluster resources and run on the same worker nodes.

Spark currently supports three cluster managers: standalone, Mesos, and YARN. Mesos and YARN allow to run Spark and Hadoop applications simultaneously on the same worker nodes.

Driver Programs: A driver program is an application that uses Spark as a library. It provides the data processing code that Spark executes on the worker nodes. A driver program can launch one or more jobs on a Spark cluster.

Executors: An executor is a JVM (Java virtual machine) process that Spark creates on each worker for an application. It executes application code concurrently in multiple threads. It can also cache data in memory or disk. An executor has the same lifespan as the application for which it is created. When a Spark application terminates, all the executors created for it also terminate.

Tasks: A task is the smallest unit of work that Spark sends to an executor. It is executed by a thread in an executor on a worker node. Each task performs some computations to either return a result to a driver program or partition its output for shuffle. Spark creates a task per data partition. An executor runs one or more tasks concurrently. The amount of parallelism is determined by the number of partitions.

What are the components of Apache Spark?

Spark Core

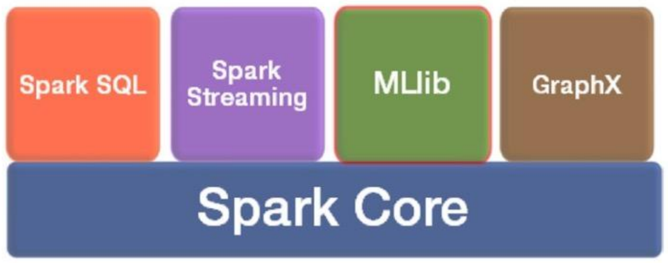
At the foundation of Apache Spark lies Spark Core, which is responsible for the essential functionalities of the system such as task scheduling, memory management, fault recovery, and interactions with storage systems. Spark Core provides the basic abstraction known as Resilient Distributed Datasets (RDDs), which are immutable, distributed collections of objects that can be processed in parallel. All other components in Spark are built on top of Spark Core, and it is this core engine that ensures efficient and fault-tolerant execution of distributed computing tasks across a cluster.

Spark SQL

Spark SQL is a module in Apache Spark designed for working with structured data. It allows users to run SQL-like queries using either SQL syntax or more modern DataFrame and Dataset APIs, offering both flexibility and high performance. Spark SQL is widely used for data transformation, integration, and analysis, and it can read data from a wide range of sources including Hive, Parquet, Avro, JSON, and JDBC databases. Because it optimizes queries through a cost-based optimizer called Catalyst and utilizes a columnar memory format called Tungsten, Spark SQL achieves performance comparable to traditional relational databases while operating at scale.

Spark Streaming

Spark Streaming extends the core capabilities of Apache Spark to support real-time data processing. It enables applications to process live data streams from sources such as Kafka, Flume, and socket connections. Spark Streaming operates by dividing the incoming data stream into small batches, which are then processed using the Spark engine. This micro-batching approach allows for scalable and fault-tolerant stream processing using the same APIs and execution model as batch jobs. Spark Streaming is ideal for use cases such as log processing, fraud detection, and real-time analytics dashboards.



MLlib (Machine Learning Library)

MLlib is Spark's scalable machine learning library designed to run common learning algorithms and statistical methods on distributed data. It includes algorithms for classification, regression, clustering, collaborative filtering, and dimensionality reduction. MLlib also provides tools for feature extraction, transformation, and constructing complex machine learning pipelines. Because it runs in-memory on distributed data, MLlib offers a significant speed advantage over traditional single-machine learning libraries, making it well-suited for building predictive models on large datasets.

GraphX

GraphX is a distributed graph analytics framework. It is a Spark library that extends Spark for large-scale graph processing. It provides a higher-level abstraction for graphs analytics than that provided by the Spark core API. GraphX provides both fundamental graph operators and advanced operators implementing graph algorithms such as PageRank, strongly connected components, and triangle count. It also provides an implementation of Google‘s Pregel API. These operators simplify graph analytics tasks.

GraphX allows the same data to be operated on as a distributed graph or distributed collections. It provides collection operators similar to those provided by the RDD API and graph operators similar to those provided by specialized graph analytics libraries. Thus, it unifies collections and graphs as first-class composable objects. A key benefit of using GraphX is that it provides an integrated platform for complete graph analytics workflow or pipeline. A graph analytics pipeline generally consists of the following steps:

a) Read raw data.

b) Preprocess data (e.g., cleanse data).

c) Extract vertices and edges to create a property graph.

d) Slice a subgraph.

e) Run graph algorithms.

f) Analyze the results.

g) Repeat steps e and f with another slice of the graph.

How does an Application Work in Spark?

When a Spark application is run, Spark connects to a cluster manager and acquires executors on the worker nodes. A Spark application submits a data processing algorithm as a job. Spark splits a job into a directed acyclic graph (DAG) of stages. It then schedules the execution of these stages on the executors using a low-level scheduler provided by a cluster manager. The executors run the tasks submitted by Spark in parallel.

- *Shuffle*. A shuffle redistributes data among a cluster of nodes. It is an expensive operation because it involves moving data across a network. Note that a shuffle does not randomly redistribute data; it groups data elements into buckets based on some criteria. Each bucket forms a new partition.
- *Job*. A job is a set of computations that Spark performs to return results to a driver program. Essentially, it is an execution of a data processing algorithm on a Spark cluster. An application can launch multiple jobs.
- *Stage*. A stage is a collection of tasks. Spark splits a job into a DAG of stages. A stage may depend on another stage. For example, a job may be split into two stages, stage 0 and stage 1, where stage 1 cannot begin until stage 0 is completed. Spark groups tasks into stages using shuffle boundaries. Tasks that do not require a shuffle are grouped into the same stage. A task that requires its input data to be shuffled begins a new stage.

Spark Cluster Manager

A cluster manager manages a cluster of computers. To be more specific, it manages resources such as CPU, memory, storage, ports, and other resources available on a cluster of nodes. It pools together the resources available on each cluster node and enables different applications to share these resources. Thus, it turns a cluster of commodity computers into a virtual super-computer that can be shared by multiple applications.

Distributed computing frameworks use either an embedded cluster manager or an external cluster manager. For example, prior to version 2.0, Hadoop MapReduce used an embedded cluster manager. In Hadoop 2.0, the cluster manager was separated from the compute engine.

Spark comes prepackaged with a cluster manager, but it can also be used with a few other open source cluster managers. Spark supports three cluster managers: Apache Mesos, Hadoop YARN, Kubernetes, and the Standalone cluster manager.

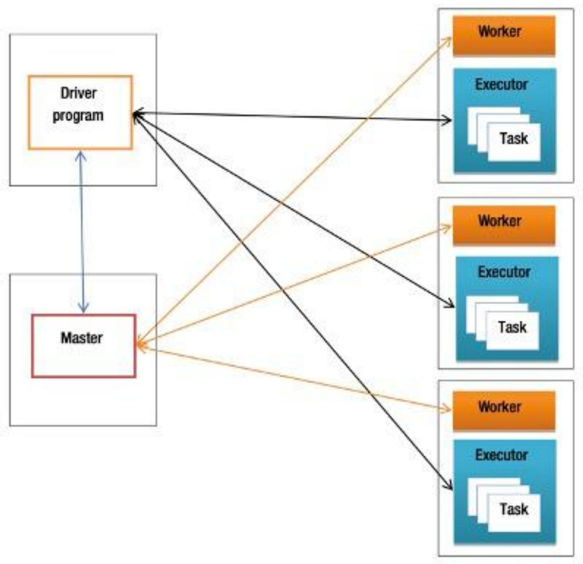
Standalone Cluster Manager

The Standalone cluster manager comes prepackaged with Spark. It offers the easiest way to set up a Spark cluster. The Standalone cluster manager consists of two key components: master and worker. The worker process manages the compute resources on a cluster node. The master process pools together the compute resources from all the workers and allocates them to applications. The master process can be run on a separate server or it can run on one of the worker nodes along with a worker process.

When a Spark application is deployed using the Standalone cluster manager, it consists of three main components: the **driver program**, **executors**, and **tasks**. The **driver program** is the core application that utilizes the Spark library and contains the logic for processing data. It is responsible for coordinating the execution of jobs across the cluster.

Executors are Java Virtual Machine (JVM) processes that run on worker nodes. These executors carry out the actual data processing tasks assigned by the driver. Each executor can run multiple tasks in parallel and also manages memory used for caching intermediate data.

To begin execution, the driver program establishes a connection with the cluster through a **SparkContext** object. This object serves as the main gateway to the Spark framework. It communicates with the **master node** to request computing resources from the cluster. Once resources are allocated, SparkContext initiates executors on the available worker nodes and sends them the application code. The driver then divides the overall job into smaller **tasks** and distributes them to the executors for parallel execution.



Apache Mesos

Apache Mesos is an open-source cluster manager that functions like a kernel for a cluster of machines. It aggregates compute resources from multiple servers and allows various applications to share these resources efficiently. By acting as a resource manager across the cluster, Mesos enables better coordination and utilization of computing power.

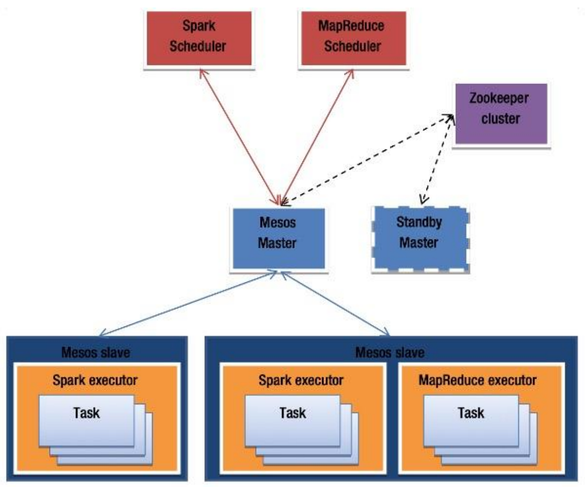
Mesos provides APIs in Java, C++, and Python, allowing developers to request cluster resources and schedule tasks programmatically. It also includes a user-friendly web interface for monitoring and managing the cluster in real time.

One of the major advantages of Mesos is its ability to allow multiple distributed computing frameworks to run simultaneously on the same cluster. This means that both Spark and non-Spark applications—such as Hadoop, MPI, Kafka, ElasticSearch, and Apache Storm can share the same pool of nodes without interfering with each other. As a result, Mesos helps avoid redundant data copies across clusters by allowing different frameworks to process the same data in place.

Compared to Spark's built-in Standalone cluster manager, Mesos offers a broader range of features and supports more complex use cases. It not only improves resource sharing among diverse applications but also simplifies the development of new distributed computing frameworks.

Key features of Apache Mesos include:

- **Scalability** to tens of thousands of nodes
- **Fault tolerance** for both master and slave nodes
- **Multi-resource scheduling**, including CPU, memory, disk, and ports
- **Support for Docker containers** for isolated, containerized environments
- **Native task isolation** to ensure secure and efficient execution



In Apache Mesos, every framework or application consists of two main components: a **scheduler** and **executors**. The **executors** run on the slave (worker) nodes and are responsible for executing the actual application tasks.

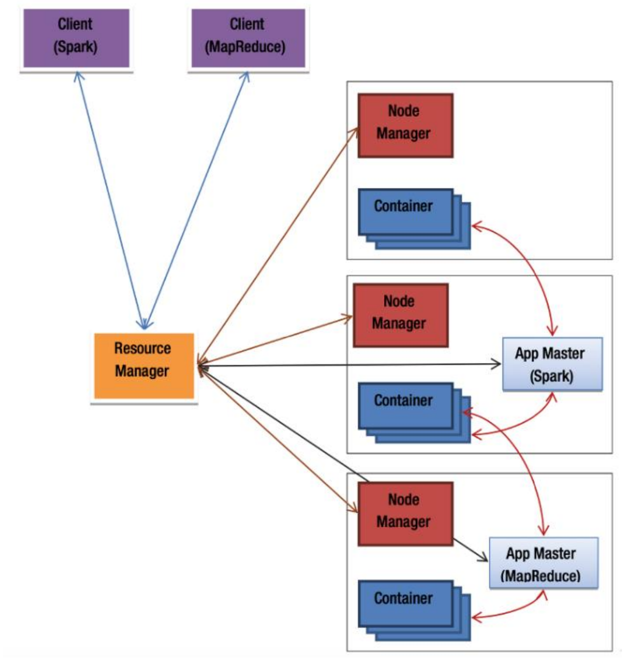
To ensure high availability and eliminate the risk of a single point of failure, Mesos supports **multiple master nodes**. At any given time, only one master is active, while the others remain in **standby mode**. If the active master fails, **Apache ZooKeeper** steps in to coordinate a seamless failover by electing a new active master.

When an application is deployed on a Mesos cluster, its **scheduler** connects to the **Mesos master**, which responds by sending a **resource offer**. This offer includes available compute resources (e.g., CPU cores, memory, disk space, and network ports) from the slave nodes. The amount and type of resources offered can be configured based on the needs of the application or framework.

The scheduler has the option to **accept or decline** a resource offer. If it accepts, the scheduler selects which specific resources it will use and sends the tasks to be executed to the Mesos master. The master then forwards these tasks to the relevant slave nodes. The slaves allocate the necessary resources to **executors**, which then carry out the tasks. This cycle continues—new offers are made as tasks complete and additional resources become available.

Hadoop YARN

YARN is a general-purpose, open-source cluster manager that was introduced with **Hadoop MapReduce version 2.0**, where the term "YARN" also became synonymous with this updated version of the Hadoop framework. YARN separates cluster resource management from job execution logic, making it a more flexible and extensible platform for running various distributed computing engines.



Key Features and Architecture

YARN allows not only **MapReduce** jobs but also other compute engines such as **Apache Spark**, **Apache Tez**, and more to run on the same cluster infrastructure. This unified cluster management is made possible by its modular architecture.

Core Components of YARN:

- **ResourceManager (RM)**:

Acts like the master node (similar to Mesos master). It manages and schedules the resources of the entire cluster by receiving resource reports from all NodeManagers.

- **NodeManager (NM)**:

Acts like the worker node (similar to Mesos slave). It manages the resources of a single machine and reports them to the ResourceManager. It also oversees the execution of tasks in containers on that node.

The ResourceManager consolidates resource information from all NodeManagers and allocates those resources to different applications. In essence, it functions as a global scheduler for the cluster.

YARN Application Components

A distributed application running on YARN typically consists of the following three components:

1. **Client Application**:
   This is the entry point for submitting jobs to the YARN cluster. For instance, when using Spark, the spark-submitcommand is the client application.

2. **ApplicationMaster (AM)**:
   Each application has its own ApplicationMaster, typically provided by the framework being used (e.g., Spark or MapReduce). The AM is responsible for negotiating resources with the ResourceManager and coordinating task execution with the NodeManagers. It also monitors the progress and health of the job. The AM runs on one of the nodes in the cluster.

3. **Containers**:
   A container represents a slice of resources (like CPU, memory) on a node. Once the ApplicationMaster secures resource allocations from the ResourceManager, it works with NodeManagers to launch containers across the cluster. These containers run the actual application tasks.

Advantages of Using YARN

- **Multi-framework Support**: Spark, MapReduce, and other engines can run side by side on the same cluster.
- **Resource Sharing**: Applications dynamically share resources for better utilization.
- **Hadoop Integration**: Existing Hadoop clusters can run Spark applications easily using YARN, without requiring separate infrastructure.

Kubernetes

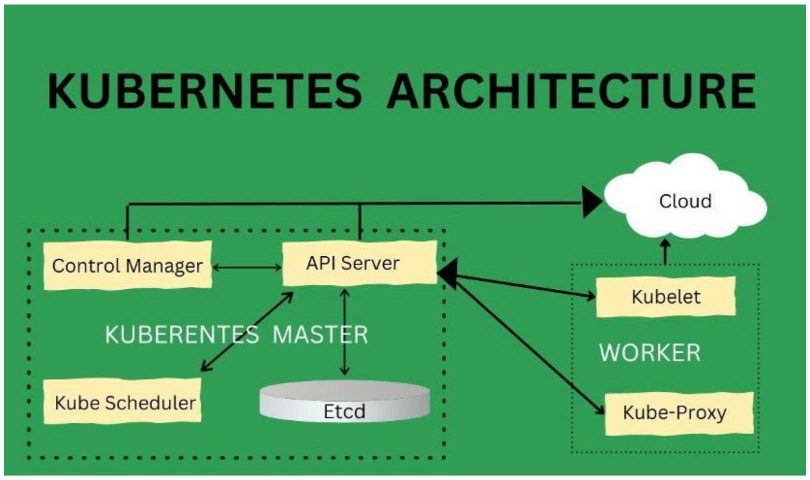
Kubernetes is a powerful, open-source container orchestration system that automates the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes is widely used in modern cloud-native architectures. Since Apache Spark 2.3, Kubernetes has been supported as a native **cluster manager** for running Spark workloads in containers.

Kubernetes as a Spark Cluster Manager

In a Spark-on-Kubernetes setup, Spark applications are packaged into **Docker containers** and run on a Kubernetes cluster. Kubernetes plays the role of a **cluster manager**, similar to YARN or Mesos, managing the life cycle of Spark driver and executor processes as pods.

A Spark application in Kubernetes consists of:

- A **driver pod**: This is where the Spark driver program runs. It handles the application logic and coordinates the execution of tasks.
- One or more **executor pods**: These pods execute tasks assigned by the driver and handle the actual data processing.



Key Components of Kubernetes for Spark:

- **Kube-API Server**: Acts as the central control plane component, receiving commands (e.g., from spark-submit) and managing application state.
- **Scheduler**: Places the driver and executor pods on appropriate nodes based on available resources.
- **Kubelet**: An agent running on each worker node that ensures containers (pods) are running as expected.
- **etcd**: Stores cluster configuration and state data.
- **Controller Manager**: Maintains the desired state of the cluster, such as ensuring a certain number of pods are running.

How Spark Submits a Job on Kubernetes

1. A user submits a Spark job using the spark-submit command with Kubernetes as the cluster manager.
2. The Spark driver is launched in a pod on the Kubernetes cluster.
3. The driver requests executor pods from the Kubernetes scheduler based on the application's needs.
4. Kubernetes launches the executor pods, which then connect back to the driver.
5. The driver coordinates task execution across executors until the job completes.

Benefits of Using Kubernetes for Spark

- **Containerized Deployment**: Spark runs inside Docker containers, ensuring consistency and easy deployment across environments.
- **Dynamic Scaling**: Kubernetes can automatically scale executors up or down based on workload.
- **High Availability and Fault Tolerance**: Built-in mechanisms like pod health checks and restarts enhance Spark application reliability.
- **Multi-tenancy and Resource Isolation**: Kubernetes namespaces and resource quotas allow for better sharing and isolation in multi-user environments.
- **Cloud-Native Integration**: Kubernetes works seamlessly with cloud platforms (AWS EKS, GCP GKE, Azure AKS), making it ideal for deploying Spark in cloud environments.

Installing Spark with R

- Download and install R and RStudio
- Download OpenJDK 17 and save it in the folder "C:\Program Files\openjdk"
- Set environmental variable for JDK:
- JAVA_HOME=C:\Program Files\openjdk
- Add new path: %JAVA_HOME%\bin
- Restart your PC and check in command prompt: java -version
- Run RStudio and run the commands:

```
install.packages("sparklyr")
library(sparklyr)
```

- To install Spark

```
spark_install(version = "3.4.1")
```

or

- Directly download Spark from https://spark.apache.org/downloads
- Unzip it into c:\Users\AppData\local\spark\spark-3.55-bin-hadoop3
- Set Environment Variable:
    - SPARK_HOME=c:\Users\AppData\local\spark\spark-3.55-bin-hadoop3
    - Add new path: %SPARK_HOME%\bin
- Restart your PC
- Connect R to the Installed Spark:

```
sc <- spark_connect(master = "local",
      version = "3.5.5",
      spark_home = " c:\Users\AppData\local\spark\spark-3.55-bin-hadoop3")
```

- To check Spark is working

```
spark_version(sc)
 library(sparklyr)
# Connect to local Spark
sc <- spark_connect(master = "local")
# Test Spark with some simple data
library(dplyr)
df <- sdf_copy_to(sc, iris, overwrite = TRUE)
```

Example 1: Load and View Data

```
# to see all data whos Sepal_Length is less than 5

df %>% filter(Sepal_Length > 5) %>% collect()
```

Example 2: Group and Summarize

```
 library(sparklyr)

# Connect to local Spark

sc <- spark_connect(master = "local")

# to see average mpg according to cylinder size

library(dplyr)

df <- sdf_copy_to(sc, mtcars, overwrite = TRUE)

df %>% group_by(cyl) %>%

  summarize(avg_mpg = mean(mpg)) %>% collect()
```

Example 3: Tokenize Text

```
text_data <- data.frame(

  id = 1:3,

  text = c("Spark is awesome", "R is powerful", "Big data rules")

)

text_tbl <- copy_to(sc, text_data, overwrite = TRUE)

tokenized <- text_tbl %>% ft_tokenizer(input_col = "text", output_col = "words")

tokenized %>% collect()
```

Example 4: Simple Machine Learning

```
# Predict mpg from weight

library(sparklyr)

# Connect to local Spark

sc <- spark_connect(master = "local")

# to construct a model dependant is mpg on independent is weight i.e. mpg= a + b*weight

library(dplyr)

df <- sdf_copy_to(sc, mtcars, overwrite = TRUE)

model <- ml_linear_regression(df, mpg ~ wt)

summary(model)

predictions <- ml_predict(model, df)

predictions %>% select(mpg, prediction) %>% collect()
```