Comprehensive Guide to set up and schedule job using Apache Spark

Contents

Introduction 2
Pre-requisites 3
Hardware Requirement 3
Deployment Strategies 3
Network Configuration: 4
Application 4
Setting up an Apache Spark Multi-node cluster 5
Overview 5
Memory Distribution 5
Architecture 5
Network Diagram 5
Configuration 6
Steps to Submit Jobs 14
Templates to make a job 23

.

Introduction

Apache Spark, an open-source distributed computing system, offers a platform for programming entire clusters with implicit data parallelism and fault tolerance. Its capabilities encompass both batch processing, handling large data volumes in discrete batches, and stream processing, managing continuous data streams. Spark's versatility extends to Java, Scala, Python, R, and bash, ensuring accessibility for diverse developer communities. In this documentation, we delve into setting up a 3-node Apache Spark cluster, enabling users to distribute processing and memory demands for efficient parallel computing.

Pre-requisites

- **Java Development Kit (JDK)**: Install OpenJDK 8 or later on all nodes.
- **Operating System**: Apache Spark supports various Linux distributions such as CentOS, Ubuntu, and Debian. Ensure that your servers are running a supported version of Linux.
- **Network Configuration**: Ensure that the servers are connected to a reliable network with sufficient bandwidth and low latency.
- **Root Access**: You will need root access to the servers for installing and configuring the Apache Spark software.

Hardware Requirement

Component	**Minimum Requirement**
Core	1 ≥ cores
OS	WindowsXP ≥ or Ubuntu, CentOS and Debian
RAM	2GB ≥
Network Connectivity	Stable network connections between nodes
Network	Gigabit Ethernet or higher

Deployment Strategies

Nodes Roles:

- **Master Node:**

➢ Controls the cluster and schedules tasks.

➢ Manages resources and makes decisions.

- Only one needed.

- **Worker Nodes:**
  - Do the actual work (processing data).
  - Provide computing power.
  - You can have several.

- **Monitor Node :**
  - Keeps an eye on the cluster's health.
  - Shows performance stats.
  - Can also work as a worker.

Network Configuration:

- **Private Network:**
  - Connect all cluster nodes to a private network.
  - Use private IP addresses for safety.

- **Load Balancing :**
  - If using multiple master nodes, balance the load evenly among them.
  - Helps with performance and reliability.

- **DNS or Hosts:**
  - Make sure all nodes can talk to each other by name or IP.

Application

Spark is suitable for wide range of applications, including

- **Real-time Data Processing**: Spark efficiently handles live data by sharing resources across many machines, allowing for fast, simultaneous data processing.
- **Machine Learning Model Training**: Spark pools resources from various nodes, making it possible to train complex machine learning models faster and on a larger scale.
- **Data Aggregation and Analysis**: Spark's resource pooling allows for quick processing of large datasets, which is key in generating insights in sectors like finance and healthcare.
- **Graph Processing:** By sharing resources, Spark supports complex and large-scale graph processing, useful in areas like social network analysis and recommendation systems.

Setting up an Apache Spark Multi-node cluster

Overview

This guide will show you how to set up Apache Spark 3.1.1 on a multi-node cluster, specifically designed for Ubuntu 22.04. It includes detailed instructions for configuring both the Spark master and worker nodes, aimed at establishing a three-node cluster. By adhering to this tutorial, you will be prepared to effectively distribute job resources across all nodes, supporting jobs written in Python, R, Java, Scala, and Bash scripts.

Memory Distribution

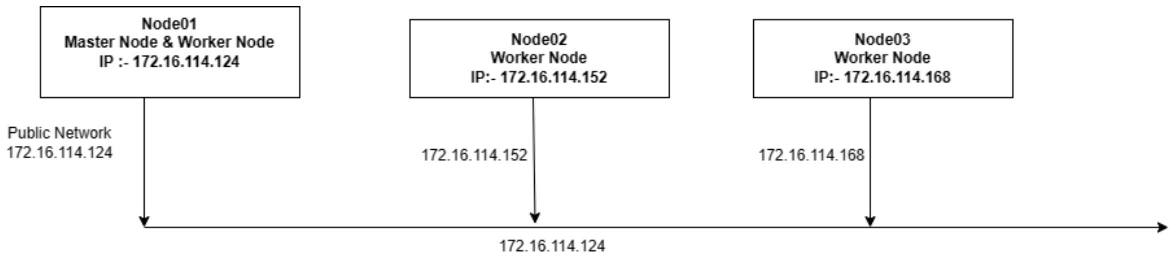
The discrepancy between the actual memory configurations of your machines and the memory displayed by Apache Spark can be explained through several key factors related to Spark and Java Virtual Machine (JVM) settings:

- Java Heap Settings: The JVM doesn't use all the physical RAM but operates within a set heap size limit defined by parameters like -Xms (initial size) and -Xmx (maximum size). These settings might be lower than the total RAM on your nodes.
- Reserved Memory: Spark reserves a portion of memory for system operations, which isn't available for tasks. This reduces the usable memory for Spark.
- System Overhead: Additional memory is used for JVM metadata and thread stacks, which isn't reflected in the memory available for Spark.
- OS Usage: The operating system and other processes also consume RAM, leaving less available for Spark.

Architecture

Node(Machine Name)	**Role**	**IP Address**	**Cores**	**RAM**	**Spark RAM**
node01	Master node	172.16.114.124	6	5.8 GB	4.8 GB
node02	Worker Node	172.16.114.152	2	3.8 GB	2.8 GB
node03	Worker Node	172.16.114.168	2	1.9 GB	1024.0 MB

Network Diagram



Configuration

Below are the steps to install and set up an Apache Spark Cluster with no prerequisites required.

Step 1: Set Hostnames

Log into each node and set their hostnames.

```
# On node01 (Master)
sudo nano /etc/hostname
# Enter "node01" in the file, save and exit

# On node02 (Worker)
sudo nano /etc/hostname
# Enter "node02" in the file, save and exit

# On node03 (Worker)
sudo nano /etc/hostname
# Enter "node03" in the file, save and exit
```

Explanation

- sudo nano /etc/hostname: Opens the hostname file in the nano editor with superuser privileges. This file contains the name of the machine (node).

Step 2: Configure Hosts File

Add the IP addresses and hostnames to the /etc/hosts file on each node.

```
# Run this on all nodes

sudo nano /etc/hosts

# Add the following lines:

172.16.114.124 node01

172.16.114.152 node02

172.16.114.168 node03

# Save and exit

sudo reboot
```

Explanation

- sudo nano /etc/hosts: Opens the hosts file, which maps IP addresses to hostnames. This helps in name resolution, especially when DNS is not available.
- sudo reboot: Reboots the machine to ensure all hostname changes take effect system-wide.

Step 3: Install Java

Java is required for Apache Spark, so install it on all nodes.

```
# Run this on all nodes

sudo apt-get update

sudo apt install openjdk-8-jdk -y

# Verify Java installation

java -version
```

Explanation

- sudo apt-get update: Updates the package lists for upgrades and new packages from the repositories.
- sudo apt install openjdk-8-jdk -y: Installs the Java Development Kit version 8, necessary for running Spark, without requiring interaction during installation.
- java -version: Checks and displays the installed Java version to verify successful installation.

Step 4: Install Scala

Scala is also required for Spark.

```
# Install Scala on all nodes

sudo apt-get install scala -y

# Check Scala version

scala -version
```

Explanation

- sudo apt-get install scala -y: Installs Scala, a programming language used for writing Spark applications. The -y flag allows the installation to proceed without interactive prompts.
- scala -version: Checks and displays the installed Scala version.

Step 5: Configure SSH

Setup SSH on the master and worker node as well as allow passwordless login to the worker nodes.

```
# On node01 (Master)

sudo apt-get install openssh-server openssh-client -y

sudo systemctl enable ssh

sudo ufw enable

sudo ufw allow ssh

sudo systemctl start ssh

sudo systemctl status ssh

ssh-keygen -t rsa -P ""

cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

ssh-copy-id node01@172.16.114.124

ssh-copy-id node02@172.16.114.152

ssh-copy-id node03@172.16.114.168

# Replace 'node01,node02,node03' with your actual username on the nodes
```

```
# On node02 and node03 (Worker Nodes)

sudo apt-get install openssh-server -y

sudo systemctl enable ssh

sudo ufw enable

sudo ufw allow ssh

sudo systemctl start ssh

sudo systemctl status ssh
```

Explanation:

- sudo apt-get install openssh-server openssh-client -y: Installs the OpenSSH server and client packages, enabling secure remote operations between the nodes.
- ssh-keygen -t rsa -P "": Generates a new RSA key pair for SSH authentication, with an empty passphrase for passwordless login.
- cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys: Adds the public key to the authorized_keys file to allow passwordless SSH access to the local machine.
- ssh-copy-id user@IP: Copies the public key to the remote hosts' authorized keys to enable passwordless SSH access to those machines. Replace user and IP with the actual user names and IP addresses.
- sudo apt-get install openssh-server -y: Installs the OpenSSH server package on Ubuntu, automatically confirming "yes" to all prompts.
- sudo systemctl enable ssh: Enables the SSH service to start automatically at boot time.
- sudo ufw enable: Activates the Uncomplicated Firewall (UFW) to start blocking and allowing connections based on its rules.
- sudo ufw allow ssh: Adds a rule to the firewall to allow inbound SSH connections on the default SSH port (22).
- sudo systemctl start ssh: Starts the SSH service immediately, allowing SSH connections to the server.
- sudo systemctl status ssh:Displays the current status of the SSH service, including whether it is active and running.

Step 6: Download and Install Apache Spark

This step should be performed on all nodes (master and workers) to ensure that each has the necessary binaries to execute their respective roles within the Spark cluster.

```
# On all nodes


#Download Apache Spark

wget https://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz


#command to untar the spark tar file

tar xvf spark-3.1.1-bin-hadoop3.2.tgz
```

Explanation

- wget https://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz: Downloads the Apache Spark software package (version 3.1.1 with Hadoop 3.2 integration) from the official Apache archive website.
- tar xvf spark-3.1.1-bin-hadoop3.2.tgz: Extracts the downloaded tar.gz file, revealing the Spark installation directory and files.
- sudo mv spark-3.1.1-bin-hadoop3.2 /usr/local/spark: Moves the extracted Spark directory to /usr/local/spark, a common location for software installations, making it accessible system-wide under a standardized path.
- sudo nano ~/.bashrc: Opens the Bash run commands file for the current user in the nano text editor, allowing you to add or edit environment variables or other startup commands.
- export PATH=$PATH:/usr/local/spark/bin: Adds the /usr/local/spark/bin directory to the PATH environment variable, enabling the execution of Spark's command-line tools from any terminal session.
- source ~/.bashrc: Applies the changes made in the .bashrc file immediately, updating the current shell session with the new PATH settings without needing to log out and back in.

Step 7: Configure Spark Master

In this step we will be setting up our master node ie node01

```
# On node01 (Master)
cd /usr/local/spark/conf
cp spark-env.sh.template spark-env.sh
sudo nano spark-env.sh
# Add the following lines
export SPARK_MASTER_HOST='172.16.114.124'
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
# Save and exit
# Configure the slaves file
sudo nano slaves
# Add the following lines
node02
node03
# Save and exit
```

Explanation

- cd /usr/local/spark/conf: Changes the current directory to the Spark configuration directory.
- cp spark-env.sh.template spark-env.sh: Copies the template file spark-env.sh.template to create a new configuration file named spark-env.sh.
- sudo nano spark-env.sh: Opens the spark-env.sh configuration file in the Nano text editor with superuser privileges for editing.
- export SPARK_MASTER_HOST='172.16.114.124': Sets the environment variable SPARK_MASTER_HOST to the IP address of the Spark Master node (which is also the current node).
- export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64: Sets the environment variable JAVA_HOME to the path where Java 8 OpenJDK is installed.
- sudo nano slaves: Opens the slaves file in the Nano text editor with superuser privileges for editing. This file lists the worker nodes in the Spark cluster.
- node02: Adds the hostname of the second worker node (node02) to the slaves file.
- node03: Adds the hostname of the third worker node (node03) to the slaves file.
- Save and exit: This is not a command but an instruction to save the changes made in the file and exit the Nano text editor. In Nano, you typically press Ctrl + O to save the file and Ctrl + X to exit the editor.

Step 8: Start Apache Spark Cluster

Our Apache Spark Cluster is ready.

```
# On all nodes

cd /usr/local/spark

./sbin/start-all.sh

# Check running services

jps
```

Explanation

- cd /usr/local/spark: Changes the current directory to the root directory of the Spark installation.
- ./sbin/start-all.sh: Executes the script start-all.sh located in the sbin directory to start all Spark services, including the Spark Master and Worker nodes, on the current node.
- jps: Invokes the Java Virtual Machine Process Status Tool (jps) to list all running Java processes.

Step 10: Access Spark UI

Access the Spark master's web UI by navigating to http://172.16.114.124:8080 in a web browser to view the cluster status and manage jobs. (Replace IP 172.16.114.124 with your master node IP).

Step 11: Add worker nodes.

Execute the following commands to add worker node. It's optional to include your monitor node in the cluster as a worker node to use its resources as well, the step remains the same. In this set up I have added my monitor node as my worker node.

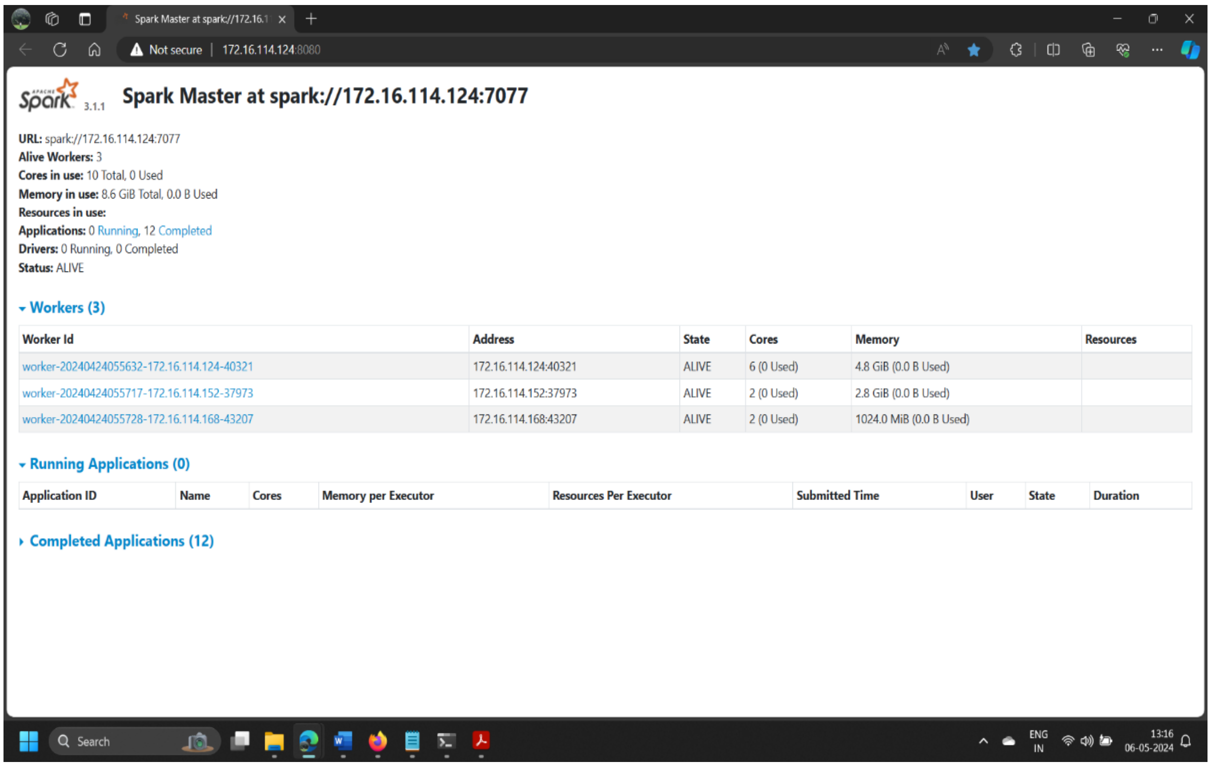
```
#On all worker nodes

# Navigate to the Spark sbin directory

cd /usr/local/spark/sbin/


# Start the worker node and connect it to the master node

./start-worker.sh spark://172.16.114.124:7077
```

Explanation

- cd /usr/local/spark/sbin/: Changes the current directory to the Spark sbin directory where Spark scripts, including those for managing worker nodes, are located.
- ./start-worker.sh spark://172.16.114.124:7077: Executes the start-worker.sh script, initiating a worker node on the current machine and connecting it to the Spark master node located at IP address 172.16.114.124 on port 7077. You can replace 172.16.114.124 with the IP of your master n

Congratulation if you have got a dashboard which is looking like this you have successfully set up your Spark Cluster!!

(You can monitor your jobs here)


Spark Master at spark://172.16.1
Not secure | 172.16.114.124:8080

Spark 3.1.1 **Spark Master at spark://172.16.114.124:7077**

URL: spark://172.16.114.124:7077
Alive Workers: 3
Cores in use: 10 Total, 0 Used
Memory in use: 8.6 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 12 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

▾ Workers (3)

Worker Id	Address	State	Cores	Memory	Resources
worker-20240424055632-172.16.114.124-40321	172.16.114.124:40321	ALIVE	6 (0 Used)	4.8 GiB (0.0 B Used)	
worker-20240424055717-172.16.114.152-37973	172.16.114.152:37973	ALIVE	2 (0 Used)	2.8 GiB (0.0 B Used)	
worker-20240424055728-172.16.114.168-43207	172.16.114.168:43207	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

▾ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

› Completed Applications (12)

Steps to Submit Jobs

Next, we'll explore how to submit jobs using Java, R, and Python.

Java:

Step 1: Writing Your Java Program

Before anything, you need to have a Java program that is designed to run on Apache Spark. Let's call this program ContinuousCalculatePi.java. This program calculates π continuously.

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.sql.SparkSession;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.IntStream;


public class ContinuousCalculatePi {

private static List<Integer> generateRange(int start, int end) {

 return IntStream.range(start, end)

                .boxed()
```

```
            .collect(Collectors.toList());
   }
           .collect(Collectors.toList());
   }
   public static void main(String[] args) {
       // Create a SparkSession with Fair Scheduler
       SparkSession spark = SparkSession.builder()
               .appName("ContinuousCalculatePi")
               .config("spark.scheduler.mode", "FAIR")
               .getOrCreate();
// Create a JavaSparkContext
       JavaSparkContext jsc = new JavaSparkContext(spark.sparkContext());




try {
           // Continuous processing loop
           while (true) {
               List<Integer> range = generateRange(0, 1000000);
               JavaRDD<Integer> iterations = jsc.parallelize(range);

               // Add your processing logic here

               Thread.sleep(5000); // Sleep for 5 seconds
           }
       } catch (InterruptedException e) {
           e.printStackTrace();
       } finally {
           jsc.stop();
       }
   }
}
```

Step 2: Navigating to Your Project Directory

Open a terminal and navigate to the directory where your Java file (ContinuousCalculatePi.java) is located. This is typically a directory within your Spark installation or a designated workspace for your Spark projects.

```
cd /path/to/your/project/directory


#In my case my ContinuousCalculatePi.java is located in in sbin

cd /usr/local/spark/sbin
```

Step 3: Compiling Your Java Program

Compile your Java program using the javac command. You'll need to include the Spark jars as part of the classpath. This is necessary because your Java program likely uses classes and methods from Spark's libraries.

```
javac -cp "/usr/local/spark/jars/*" ContinuousCalculatePi.java
```

Explanation

- The command compiles the Java source file `ContinuousCalculatePi.java`, including dependencies from all JAR files in the `/usr/local/spark/jars` directory, using the specified classpath.

Step 4: Creating a JAR File

After compiling your Java file, you'll have a .class file (e.g., ContinuousCalculatePi.class). You need to package this into a JAR file to submit it to Spark. Use the jar command to create this JAR file.

```
jar cf ContinuousCalculatePi.jar ContinuousCalculatePi.class
```

Explanation

- The command compiles the Java source file `ContinuousCalculatePi.java`, including dependencies from all JAR files in the `/usr/local/spark/jars` directory, using the specified classpath.

- A JAR file is like a suitcase for Java programs, where all the necessary files (like code and images) are packed together, making it easy to carry and run the program on different computers.

Step 5: Submitting the Job to Spark

Use the spark-submit command to run your job on the Spark cluster. You'll need to specify several parameters including the class name, the master node's URL, and the path to your JAR file. Adjust these parameters based on your specific setup and needs.

```
#Make sure you are in sbin directory if you have accidentally come out then go back to sbin

cd /usr/local/spark/sbin



#Command to submit the job

/usr/local/spark/bin/spark-submit --class ContinuousCalculatePi --master
spark://your.master.node.ip:7077 --executor-cores 4 ContinuousCalculatePi.jar
```

Explanation

- --class: Specifies the main class of your application.
- --master: The master URL for the cluster (e.g., spark://172.16.114.124:7077).
- --executor-cores: Number of cores to use on each executor. Modify this based on your cluster's configuration and requirements.
- ContinuousCalculatePi.jar: The path to the JAR file containing your compiled application.

R Programming:

Step 1: Install Necessary Software

Install R and Development Libraries ,R along with some development libraries that are often required for building R packages from source.

```
sudo apt install r-base

sudo apt-get install -y libcurl4-openssl-dev libssl-dev libxml2-dev libfontconfig1-dev libfreetype6-dev
```

Explanation

- r-base: This package installs the base R system, which includes the R language interpreter and essential libraries.
- libcurl4-openssl-dev: This package provides the development files and headers needed for compiling programs that use libcurl with OpenSSL support. libcurl is a client-side URL transfer library used for making HTTP requests and transferring data over various protocols.
- libssl-dev: This package provides the development files and headers for OpenSSL, which is a cryptographic library used to secure communications over computer networks.
- libxml2-dev: This package provides the development files and headers for libxml2, a library used for parsing and manipulating XML files.
- libfontconfig1-dev: This package provides the development files and headers for Fontconfig, a library used for configuring and customizing font access on Unix-like systems.
- libfreetype6-dev: This package provides the development files and headers for FreeType, a font rasterization engine used to render fonts on computer screens and printers.

Step 2: Prepare Your R Script

The R script initializes a connection to a Spark cluster, defines a function for parallel computing, continuously applies this function to a dataset in parallel, and prints the results. It demonstrates how to harness the distributed computing power of Spark from within R.

```
#I have named my R script as my_script.R which is located in /home/node02/nano
/home/node02/my_script.R

#Below is my R code

library(SparkR)

# Initialize Spark context

sc <- sparkR.session(master = "spark://172.16.114.124:7077", sparkHome = "/usr/local/spark")

# Define your parallel computing logic as a function

parallel_compute <- function() {

  # Create a Spark DataFrame from a local data frame or any other data source

  df <- as.DataFrame(data.frame(x = 1:100))

  # Perform parallel computing operations

  result <- spark.lapply(df, function(row) {

    # Your parallel computation logic here

    return(row * 2)

  }

# Collect and return the results

  collected_result <- collect(result)

  return(collected_result)

}
```

Explanation

- library(SparkR): This command loads the SparkR library into the R environment, making all its functions and capabilities available for use in the current R session. It essentially imports the SparkR package.
- sc <- sparkR.session(master = "spark://172.16.114.124:7077", sparkHome = "/usr/local/spark"): This command establishes a connection to a Spark cluster. It creates a Spark context (sc) by initializing a Spark session using the specified master URL ("spark://172.16.114.124:7077") and Spark installation directory ("/usr/local/spark"). This session allows R to communicate with the Spark cluster for distributed data processing.

Step 3: Run Your R Script Locally (for Testing)

Before deploying it to Spark, you might have tested your R script locally using Rscript, which is an efficient way to run R scripts from the command line.

```
#Change the directory to where your R program is located

cd /home/node02/

#Command to run your R program

Rscript my_script.R
```

Step 4: Submit Your R Script to Spark

Find the spark-submit Command, check the location of the spark-submit command, which is used to submit jobs to Spark

```
#Command to locate spark-submit

which spark-submit


#In my case I got the below out put

/usr/local/spark/bin/spark-submit


#Submit your job using spark-submit

/usr/local/spark/bin/spark-submit --master spark://172.16.114.124:7077
/home/node02/my_script.R
```

Python:

Step 1: Make sure you have python installed.

If you don’t have python installed in your system follow the below steps to install python.

```
#Install python

sudo apt install python3

#Install pip

sudo apt install python3-pip
```

Explanation

- pip is a package manager for Python, facilitating the installation, removal, and management of Python packages.

Step 2: Write your Python code.

I have written a python code that calculates the value of Pi continuously and named it is sripy3.py , I have created it /usr/local/spark/sbin

```
#Open your preferred text editor , write your code and save it  with a .py extension

nano sripy.py

#Write your code

from pyspark.sql import SparkSession

import random

# Create a SparkSession

spark = SparkSession.builder \

    .appName("CalculatePi") \

    .getOrCreate()

# Function to generate a random point and check if it falls within the unit circle

def inside_circle(p):

    x, y = random.random(), random.random()

    return 1 if x*x + y*y < 1 else 0

while True:

    # Create RDD with a large number of iterations

    iterations = spark.sparkContext.parallelize(range(1000000))

    # Perform the calculation

    count = iterations.map(inside_circle).reduce(lambda x, y: x + y)

    # Calculate the value of Pi

    pi_estimate = 4 * count / 1000000

    print("Estimated value of Pi:", pi_estimate)

# Stop the SparkSession (This will never be executed as the script runs indefinitely)

# spark.stop()
```

```
 # Run parallel computing continuously until manually stopped
repeat {
  # Perform parallel computing
  result <- parallel_compute()
  # Process or save the results as needed
  # For demonstration, we'll print the first few results
  cat("Results:\n")
  print(head(result))
  # Sleep for a while before starting the next iteration
  # Adjust the sleep duration as needed
  Sys.sleep(60)  # Sleep for 60 seconds
}
# Stop the Spark context
sparkR.stop()
```

Step 3: Test your code locally.

Before running your python code on Spark you need to check for errors locally in your system.

```
#command to run python locally
python3 sripy3.py
```

Step 4: Submit the job in Apache Spark Cluster.

Finally run submit your python job to your Apache Spark Cluster

```
#Go to the location of your spark-submit command
cd /usr/local/spark/bin/spark-submit



#Command to submit your python job
./spark-submit   --master spark://172.16.114.124:7077   --total-executor-cores 4
/usr/local/spark/sbin/sripy3.py
```

Explanation

- --master spark://172.16.114.124:7077: Specifies the address of the Spark master node where the application will be submitted. In this case, it indicates that the master node is located at the IP address 172.16.114.124 and is listening on port 7077.
- --total-executor-cores 4: Specifies the total number of CPU cores to allocate across all executor instances for the application. In this case, it allocates 4 CPU cores.
- /usr/local/spark/sbin/sripy3.py: Specifies the path to the Python script that contains the Spark application code to be executed. This script will be submitted to the Spark cluster for processing.

Templates to make a job

Below are code templates for running in an Apache Spark cluster, provided in Python, Java, R, and Scala.

Python:-

```
from pyspark import SparkContext, SparkConf
def main():
    conf = SparkConf().setAppName("ParallelCountJob")
    sc = SparkContext(conf=conf)

    # Create an RDD
    data = sc.parallelize(range(1, 100001), numSlices=10)
    # Perform a simple map and reduce job
    result = data.map(lambda x: x*2).reduce(lambda x, y: x + y)

    print("Total sum:", result)
    sc.stop()
if __name__ == "__main__":
    main()
```

Explanation :-

- Importing Libraries:
  - `from pyspark import SparkContext, SparkConf`: Imports SparkContext for establishing a connection to a Spark cluster and SparkConf for configuring Spark parameters.

- Defining the Main Function:
  - `def main()`: Defines the main function to organize the core functionality of the script, ensuring it is modular and reusable.

- Configuring and Initializing Spark:
  - `conf = SparkConf().setAppName("ParallelCountJob")`: Creates a configuration object for Spark and sets the application name to "ParallelCountJob".
  - `sc = SparkContext(conf=conf)`: Initializes the SparkContext with the specified configuration, connecting to the Spark cluster and acting as the coordination point for all Spark operations.

- Creating an RDD (Resilient Distributed Dataset):
  - `data = sc.parallelize(range(1, 100001), numSlices=10)`: Creates an RDD from a range of integers (1 to 100,000) and specifies that this dataset should be split into 10 partitions for parallel processing.

- Map and Reduce Operations:
  - `result = data.map(lambda x: x*2).reduce(lambda x, y: x + y)`:
  - Applies a map function to double each number in the RDD.
  - Uses a reduce function to sum all the doubled numbers, reducing the data to a single sum.

- Outputting the Result:
  - `print("Total sum:", result)`: Outputs the total sum calculated by the reduce operation to the console, displaying the result of the computation.

- Stopping the Spark Context:
  - `sc.stop()`: Stops the SparkContext to free up resources and prevent memory leaks, ensuring clean termination of the application.

- Main Guard for Execution:
  - `if __name__ == "__main__": main()`: Ensures that the `main` function is only called if the script is run directly, not when imported as a module. This is standard practice in Python to prevent certain code from running when scripts are imported.

Java:-

```
import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.SparkConf;


public class ParallelCountJob {

    public static void main(String[] args) {

        SparkConf conf = new SparkConf().setAppName("ParallelCountJob");

        JavaSparkContext sc = new JavaSparkContext(conf);


        JavaRDD<Integer> data = sc.parallelize(range(1, 100001), 10);

        Integer result = data.map(x -> x * 2).reduce((x, y) -> x + y);


        System.out.println("Total sum: " + result);

        sc.close();

    }


    private static List<Integer> range(int start, int end) {

        return IntStream.rangeClosed(start, end).boxed().collect(Collectors.toList());

    }
```

Explanation:-

- Importing Libraries:

- `import org.apache.spark.api.java.JavaRDD;`: Imports the JavaRDD class for resilient distributed dataset operations in Java.
- `import org.apache.spark.api.java.JavaSparkContext;`: Imports the JavaSparkContext class, which serves as the main entry point for Spark functionality in Java applications.
- `import org.apache.spark.SparkConf;`: Imports the SparkConf class to configure various Spark parameters.

- Class Definition:
- `public class ParallelCountJob`: Defines a public class named `ParallelCountJob` which contains the main method and other utility methods.

- Main Method:
- `public static void main(String[] args)`: The entry point of the application. This method will be executed when the program starts.

- Spark Configuration and Initialization:
- `SparkConf conf = new SparkConf().setAppName("ParallelCountJob");`: Creates a Spark configuration object and sets the application name to "ParallelCountJob".
- `JavaSparkContext sc = new JavaSparkContext(conf);`: Initializes a JavaSparkContext with the specified configuration, connecting to the Spark cluster and preparing the environment for distributed operations.

- Creating and Processing an RDD:
- `JavaRDD<Integer> data = sc.parallelize(range(1, 100001), 10);`: Creates a JavaRDD from a custom range of integers (1 to 100,000), divided into 10 partitions for parallel processing.
- The range of numbers is generated by the `range` method, which uses Java streams to create a list of integers from 1 to 100,000.

- Map and Reduce Operations:
- `Integer result = data.map(x -> x * 2).reduce((x, y) -> x + y);`:
- `map(x -> x * 2)`: Doubles each number in the RDD.
- `reduce((x, y) -> x + y)`: Summarizes all the elements of the RDD by adding them up, resulting in a single integer that represents the total sum of all doubled numbers.

- Outputting the Result:
  - `System.out.println("Total sum: " + result);`: Prints the total sum computed by the reduce operation to the console.

- Closing Spark Context:
  - `sc.close();`: Properly closes the JavaSparkContext, releasing the resources associated with it to prevent any potential resource leaks.

- Utility Method - range:
  - `private static List<Integer> range(int start, int end)`: A helper method that generates a list of integers from `start` to `end` using Java's IntStream, which is then collected into a List object.

R Programming: –

```
library(SparkR)


sparkR.session(appName="ParallelCountJob")


data <- parallelize(1:100000, numSlices=10)

result <- collect(reduce(map(data, function(x) x*2), function(x, y) x + y))


print(paste("Total sum:", result))


sparkR.stop()
```

Explanation:-

- Library Import:
  - The `SparkR` library is imported, allowing access to Spark functionality within R.

- Spark Session Initialization:
  - A Spark session is initialized using `sparkR.session(appName="ParallelCountJob")`.
  - The `appName` parameter sets the name of the Spark application to "ParallelCountJob".

- Data Generation:
  - A dataset containing numbers from 1 to 100,000 is generated using `parallelize(1:100000, numSlices=10)`.
  - The `numSlices` parameter specifies the number of partitions to divide the dataset into.

- Parallel Processing:
  - A map operation is applied to each element of the dataset, doubling its value using `map(data, function(x) x*2)`.
  - The `reduce` function is then used to aggregate the results of the map operation by adding them together.
  - The `collect` function gathers the results from the distributed computation and returns them to the driver program.
- Result Display:
  - The total sum of the doubled numbers is printed using `print(paste("Total sum:", result))`.
- Spark Session Termination:
  - The Spark session is stopped using `sparkR.stop()` to release resources and shut down the Spark application.

Scala: –

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf


object ParallelCountJob extends App {
  val conf = new SparkConf().setAppName("ParallelCountJob")
  val sc = new SparkContext(conf)
  val data = sc.parallelize(1 to 100000, 10)
  val result = data.map(_ * 2).reduce(_ + _)
  println(s"Total sum: $result")
  sc.stop()
```

Explanation :-

- Library Import:
  - ➢ The necessary Apache Spark libraries are imported:
  - ➢ `org.apache.spark.SparkContext`: Provides the main entry point for Spark functionality.
  - ➢ `org.apache.spark.SparkConf`: Represents the configuration settings for a Spark application.

- Spark Configuration:
  - ➢ A new `SparkConf` object named `conf` is created to configure the Spark application.
  - ➢ The application name is set to "ParallelCountJob" using `setAppName()`.

- Spark Context Initialization:
  - ➢ A new `SparkContext` object named `sc` is created using the configured `SparkConf`.
  - ➢ This establishes a connection to the Spark cluster.

- Data Generation:
  - ➢ A dataset containing numbers from 1 to 100,000 is generated using `sc.parallelize(1 to 100000, 10)`.
  - ➢ The `parallelize` method distributes the data across the Spark cluster into 10 partitions.

- Parallel Processing:
  - ➢ Each element in the dataset is multiplied by 2 using `data.map(_ * 2)`.
  - ➢ The `map` transformation applies the specified function (`_ * 2`) to each element in parallel.
  - ➢ The `reduce` action then aggregates the results by summing them together using `_ + _`.

- Result Display:
  - ➢ The total sum of the doubled numbers is printed using `println(s"Total sum: $result")`.

- Spark Context Termination:
  - ➢ The Spark context (`sc`) is stopped using `sc.stop()` to release resources and shut down the Spark application.