Install Apache Spark 3.5.1 with Python API & Integrate with Jupiter Notebook on Windows 10

Author: Sri Adilakshmi M

Table of Contents

This document outlines the steps needed to install two modes – **Standalone cluster** and **Hadoop cluster** - of **Apache Spark 3.x** with **Python** application programming interface on Windows operating system and integrate with **Jupiter Notebook**.

1. Overview:

Apache Spark is a powerful open-source analytical and computing engine to process large scale data loads. It is heavily used in Data Engineering, Data Science and Machine learning projects since it offers easy programming interface in various languages such as **Python, SQL, Java, Scala** and **R**.

Spark provides heavy processing tools such as **Spark SQL** for structured data processing, **MLib** for machine learning, **Graphx** for graph processing**, Structured Streaming** for real time data processing and **pandas** API to handle pandas workloads.

The important features of Spark are:

- It unifies data processing with batch and or real-time streaming.
- Provides advanced data analytics even with ANSI SQL offering fast, distributed queries for dashboarding and reporting.
- Powerful engine that runs 100 times faster than the traditional Hadoop MapReduce.
- Spark can train machine learning algorithms on a single computer and deploy it on cluster with thousands of machines offering parallel and distributed computation.

The key components of Spark include **Spark Driver**, **Spark Executor**, **Cluster Manager** and **Spark History Server** (Web UI)**.** Spark also offers various Command Line Interfaces such as `spark-shell` (for Scala), `pyspark` (for Python), `sparkr` (for R) and `spark-sql` (for SQL) to execute the Spark code in different programming languages.

Spark supports 4 cluster managers:

1. **Standalone Cluster:** It is the simple cluster manager where Spark manages its cluster without any external dependencies. It is easy to setup and suitable for small-scale deployments.

2. **Apache Mesos:** It is a generic cluster manager that provides efficient resource allocation and scheduling, supporting dynamic resource sharing between applications.
   Note: This cluster manager support has been deprecated from Spark 3.2.0 version.

3. **Hadoop YARN:** A resource manager integrated with the Hadoop ecosystem. It provides fine grained resource allocation and scheduling, supporting multiple concurrent applications on the same cluster.

4. **Kubernetes**: An open source system for automating deployment, scaling and management of containerized applications.

The Spark applications can be run in three different modes:

- **Local Mode** – In local mode, Spark runs all processes with in a single JVM of the machine. It is simple mode of deployment which is mostly used for testing purposes. This mode is best suited for uses who are just learning spark or testing small-scale data processing workflows.

- **Client Mode** – This is the **default** deployment mode in which the Spark driver component runs on the client machine from where the job is submitted and connects to cluster manager for resource allocation. With this mode, if the machine or user session running the spark application terminates, the entire Spark application gets terminated with status fail. This mode is not suitable for Production deployments.

- **Cluster Mode –** In this mode, the Spark driver component gets launched on any of the available nodes in the Spark cluster, instead of launching on the client machine from where the job is submitted. Here, even if the local machine or user session is terminated, the spark application keeps running on the cluster. This mode is mostly used for processing large data sets and best suited for Production deployments.

Note:
Apache Spark downloads are pre-packaged with Hadoop libraries for users to install standalone Spark cluster. Spark also has "Hadoop free" download that allows to install Spark on top of existing Hadoop cluster.

2. Prerequisites:

The following prerequisites need to be installed before installing Spark.

1. **File Archiver:** Any file archiver such as **7zip** or **WinRAR** is needed to unzip the downloaded Spark binaries. 7zip can be downloaded from the 7zip Downloads website and WinRAR can be downloaded from the RAR lab Downloads website.

2. **JRE 8:** Spark 3.x requires Java 8 runtime environment.
   We can either download just JRE 8 (Java Runtime Environment) for Windows offline installation from the official Java Download for Windows Offline website or download the whole JDK 8 (Java Development Kit) directly from Oracle Java Downloads website. For the complete JDK installation steps, look at here.

3. **Python:** To run Apache Spark with Python API, we should have Python software running in the system. If Python is not already available, you can install it from Python

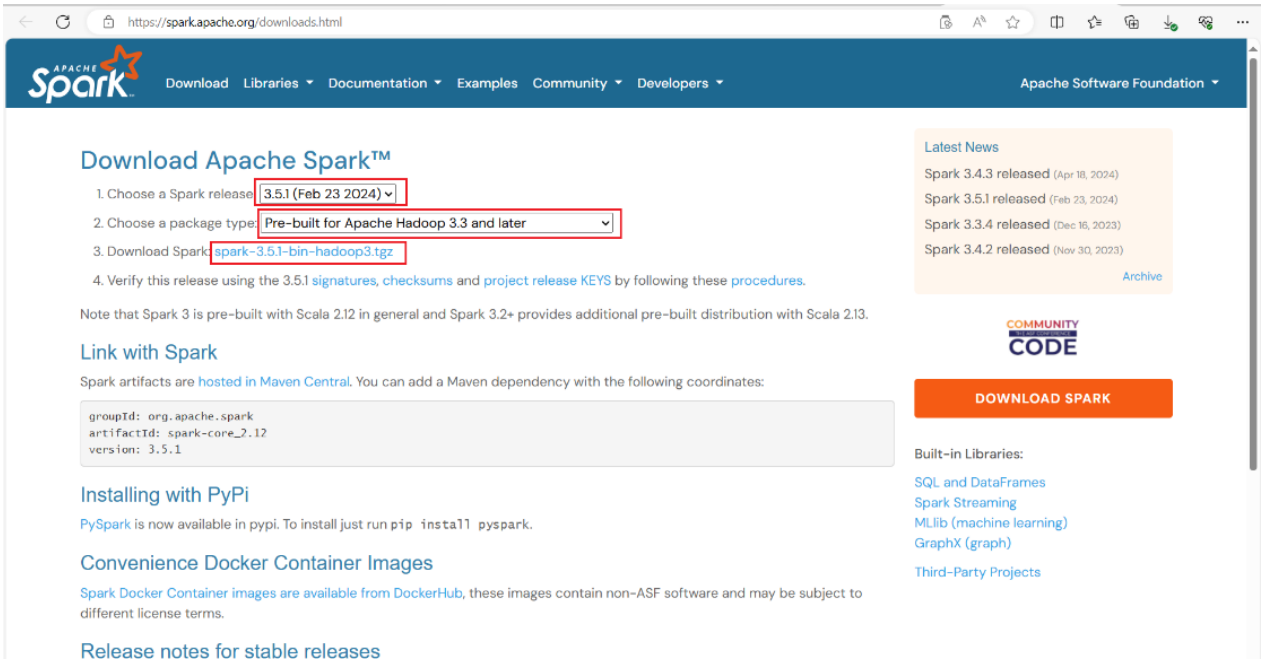
Downloads website but it is recommended to install **Anaconda Distribution** which is an open source software built for **Python and R** programming languages that are heavily used in Data Science, Data Engineer and Data Analytics fields. Refer to Official Anaconda Installation Guide on how to install it.

3. Install Standalone Spark:

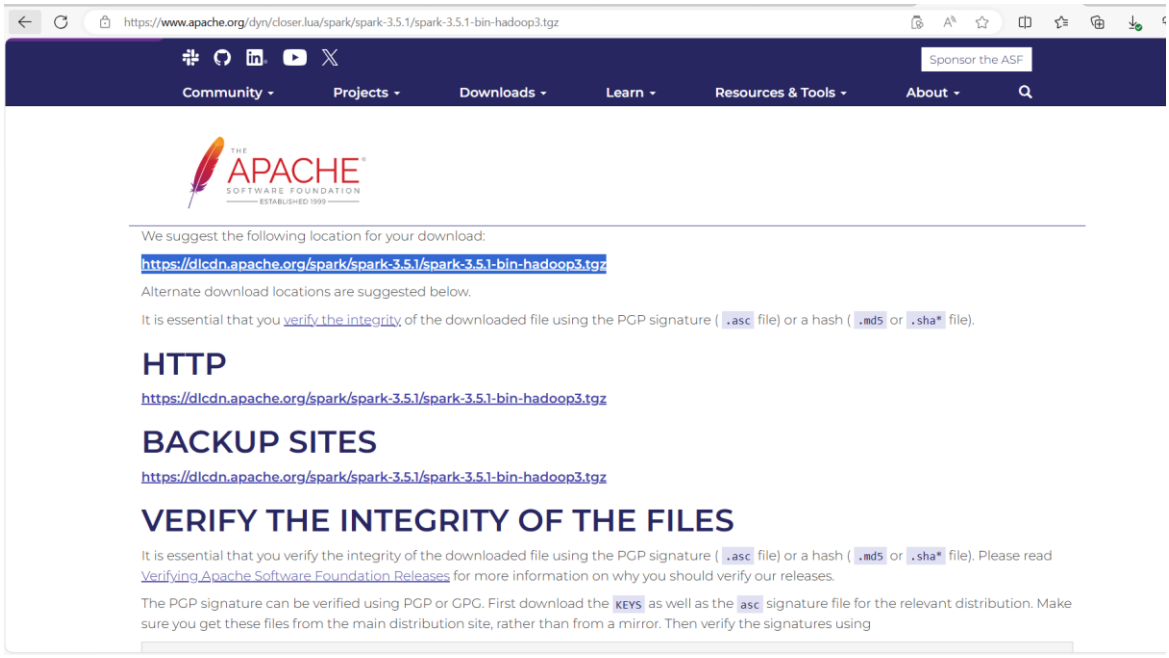
In this section, we will see how to setup Standalone Spark cluster with one master server and one worker node running locally.

3.1. Download Spark Binaries:

Download the latest **Spark 3.5.1** version from the Apache Spark Downloads page. In this page, choose the **package type** as **Pre-built for Apache Hadoop 3.3** and click on `spark-3.5.1-bin-hadoop3.tgz` binary file.

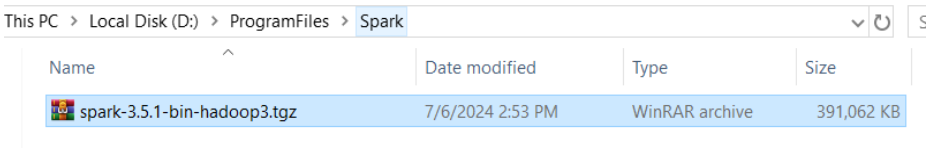


You will be navigated to spark-3.5.1 mirror website where click on the suggested location for spark-3.5.1-bin-hadoop3.tgz file that gets downloaded to your **Downloads** folder in your machine.

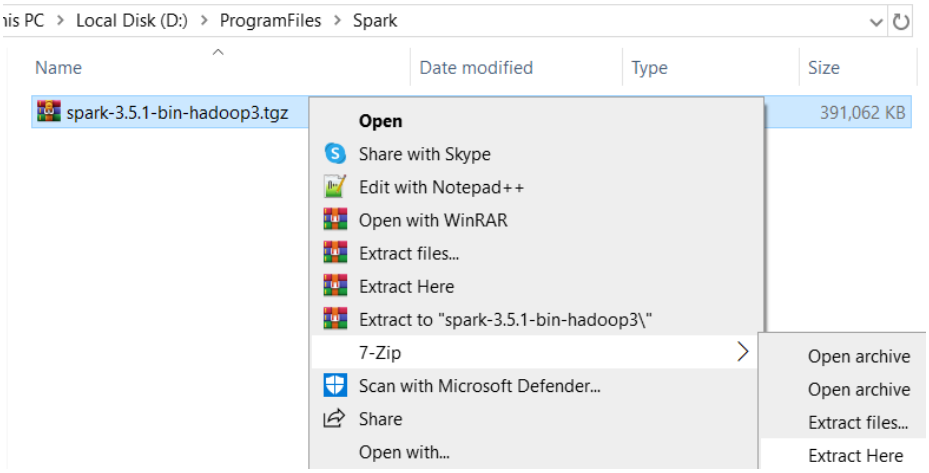


After the binary file is downloaded, unpack it using any file archiver (**7zip** or **WinRAR**) utility as below:

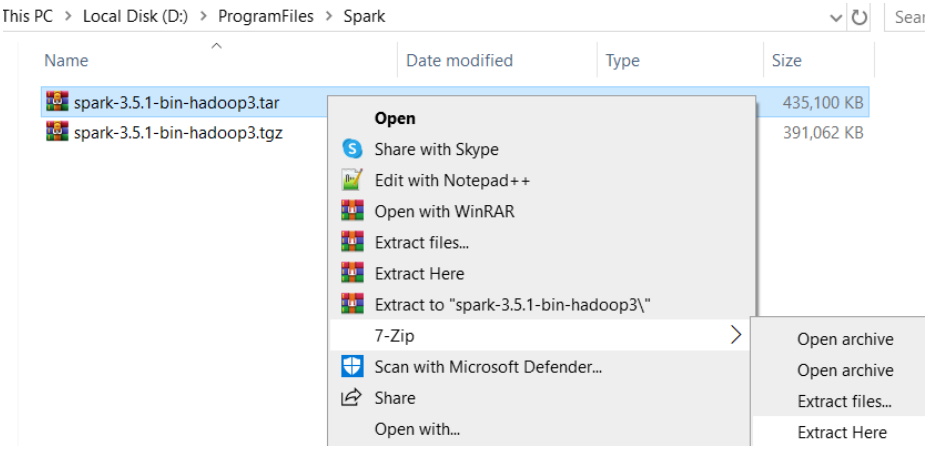
- Choose the installation directory in your machine and copy `spark-3.5.1-bin-hadoop3.tgz` file to that directory. Here, we are choosing Spark installation directory as `D:\ProgramFiles\Spark`.



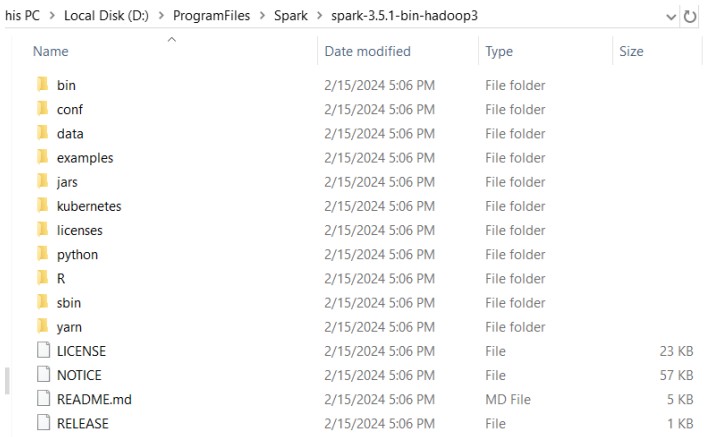
- Right click on `spark-3.5.1-bin-hadoop3.tgz` and choose **7-Zip -> Extract Here** option which extracts a new packed file `spark-3.5.1-bin-hadoop-3.tar`.



- Next, unpack `spark-3.5.1-bin-hadoop3.tar` file using 7zip utility.



- The tar file extraction may take few minutes to finish. After finishing, you see a folder named `spark-3.5.1-bin-hadoop3` which consists of Spark binaries and libraries.

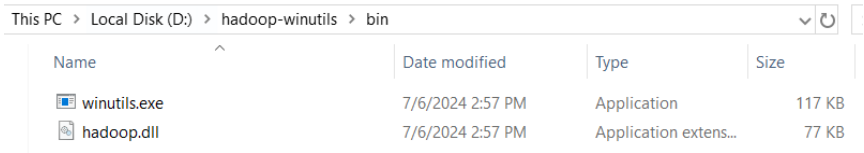


Note:
Although Spark has provided Hadoop libraries, they are not compatible to run on Windows systems by default. So, we need to get Hadoop's native IO utilities for Windows from cdarlint GitHub repository which are available for different versions of Hadoop.

- Since we installed the latest Spark 3.5.1 version, download utilities such as `winutils.exe` and `hadoop.dll` of the latest Hadoop 3.3.6 version from here.

- Create a folder named `hadoop-winutils` in **D drive** and create a sub-folder named `bin` inside it.

- Move `winutils.exe` and `hadoop.dll` files from your downloaded location to `D:\hadoop-winutils\bin` directory.



3.2. Setup Environment Variables:

After installing pre-requisites, Spark binaries and Hadoop utilities for Windows, we should configure the below environment variables defining Java, Hadoop and Spark default paths.

- **JAVA_HOME**: This is the JDK installation directory path in the machine *(in my machine, it is `D:\ProgramFiles\Java\jdk-1.8`)*. Ignore it if this is already done.

- **HADOOP_HOME**: This is the Hadoop's WinUtils path in the machine *(in our case, it is `D:\hadoop-winutils`)*

- **SPARK_HOME**: This is the Spark installation directory path in the machine *(in our case, it is `D:\ProgramFiles\Spark\spark-3.5.1-bin-hadoop3`)*

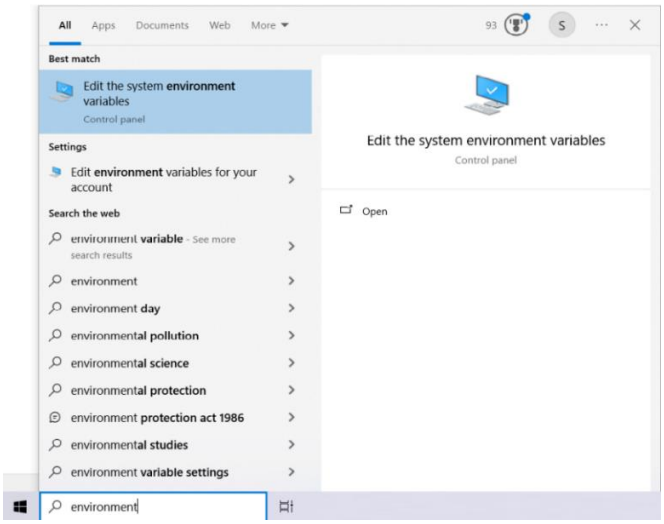
- **PYTHONPATH**: This is the Spark python directory path which should be *`%SPARK_HOME%\python`*

- **PYSPARK_PYTHON**: Set this variable to the location where *`python.exe`* file is available *(in my machine, it is `D:\ProgramFiles\Anaconda\anaconda3\python`)* which is needed for Spark to execute the Python code.

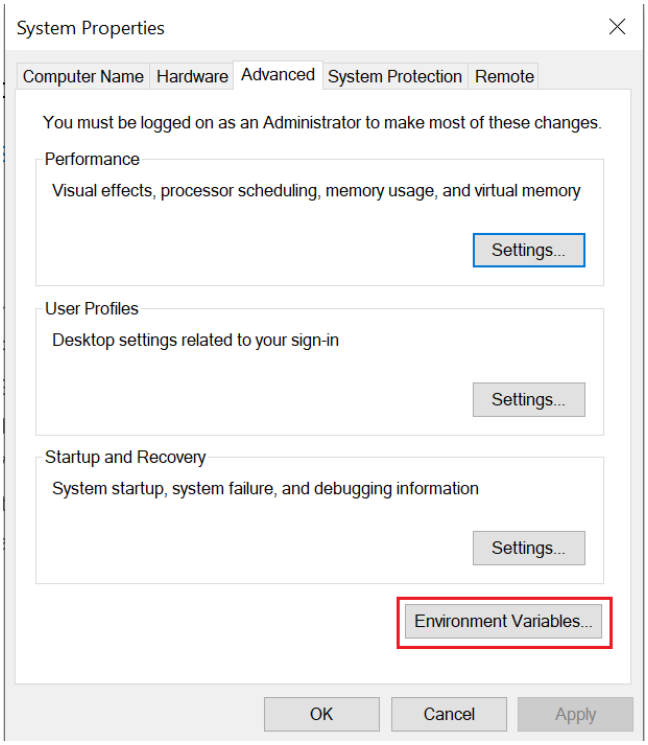
These variables need to be added to either **User environment variables** or **System environment variables** depending on Spark configuration needed **for a single user** or **for multiple users**.

In this tutorial, we will add User environment variables since we are configuring Spark for a single user. If you would like to configure Spark for multiple users, then define System environment variables.

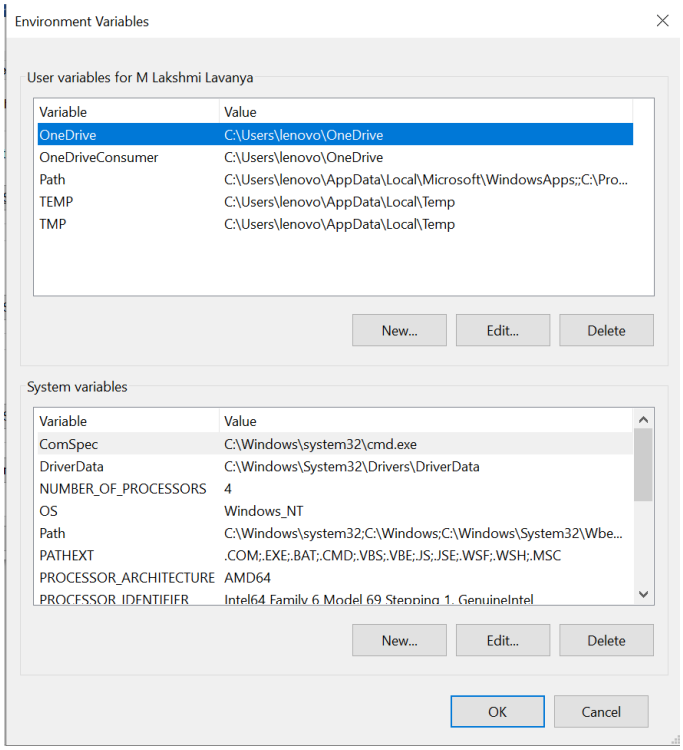
In the Windows search bar, start typing “environment variables” and select the first match which opens up **System Properties** dialog.



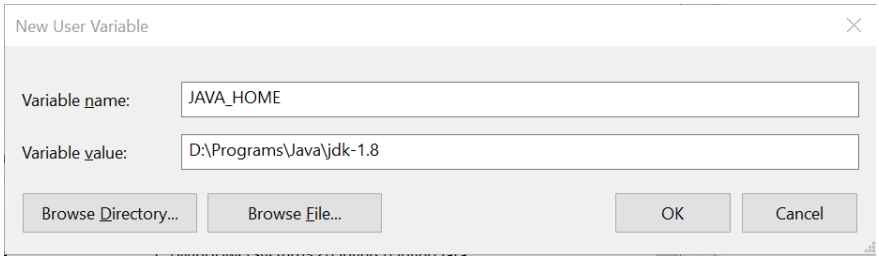
On the **System Properties** window, press **Environment Variables** button.



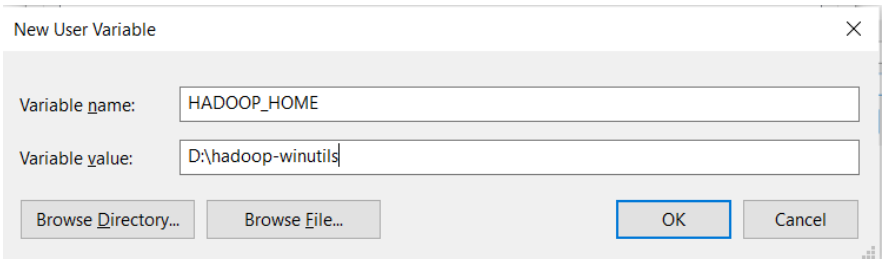
In the **Environment Variables** dialog, click on **New** under **User variables** section.



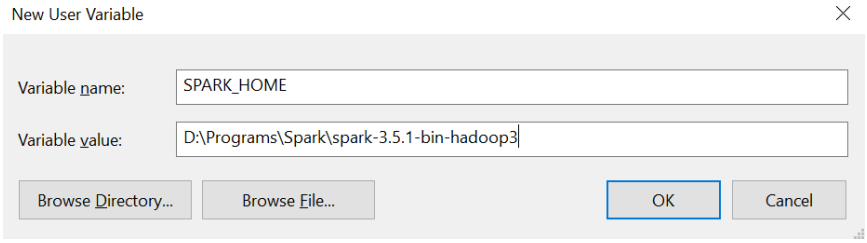
Add `JAVA_HOME` variable and press OK.



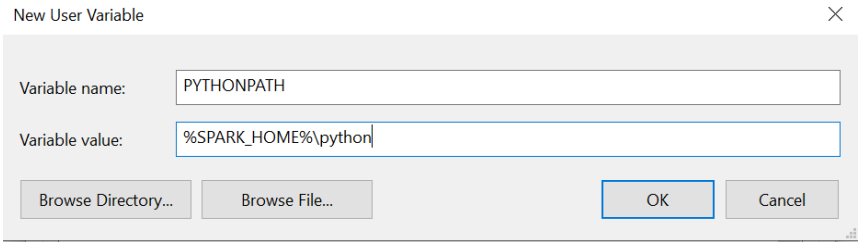
Click on **New** again and add `HADOOP_HOME` variable and press OK.



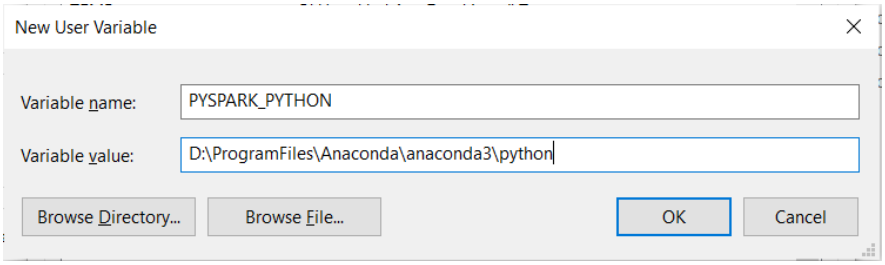
Click on **New** again and add `SPARK_HOME` variable and press OK.



Click on **New** again and add `PYTHONPATH` variable and press OK.

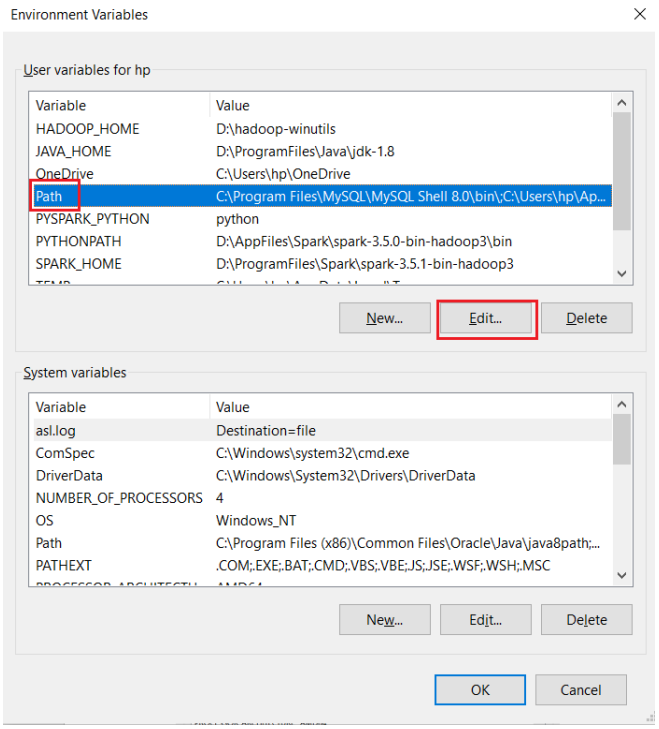


Click on **New** again and add `PYSPARK_PYTHON` variable and press OK.



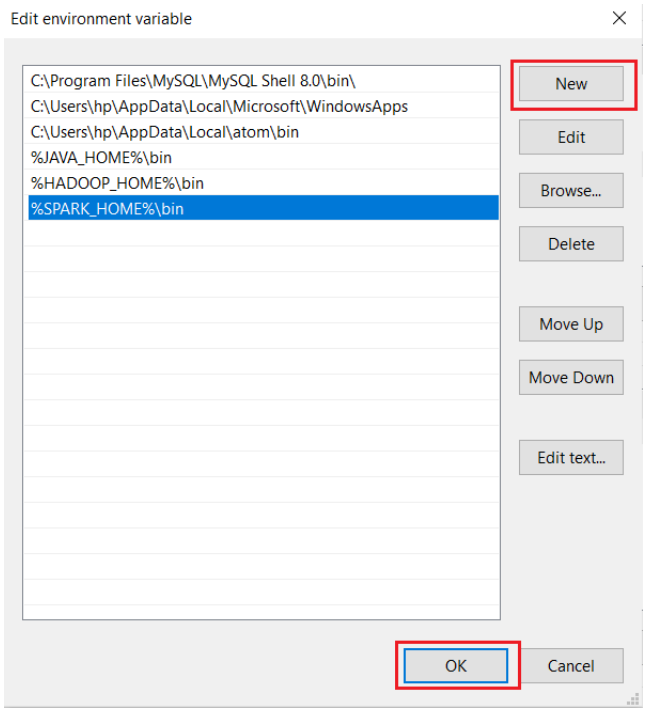
Now, we will update `PATH` variable to add Java, Hadoop and Spark binary paths.

Select `PATH` variable under **User Variables** and press **Edit** button.

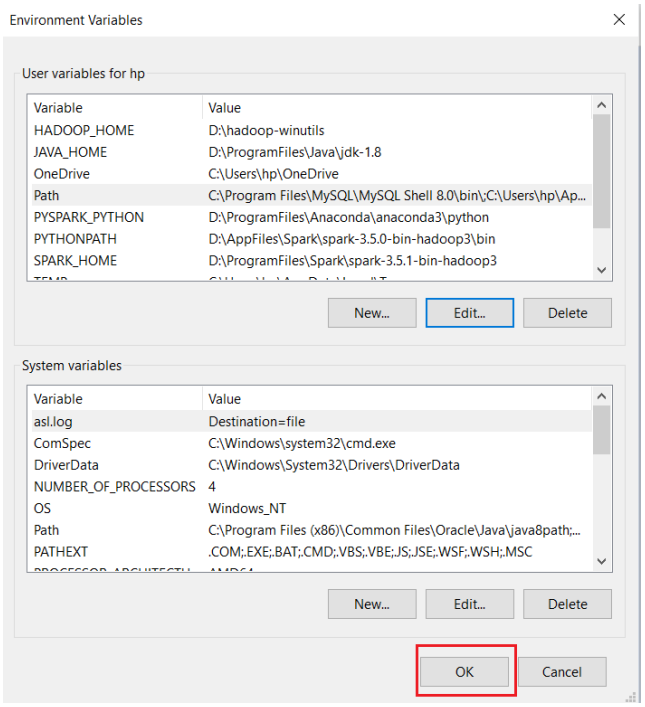


Press **New** and add the following values and press OK.

```
%JAVA_HOME%\bin
%HADOOP_HOME%\bin
%SPARK_HOME%\bin
```



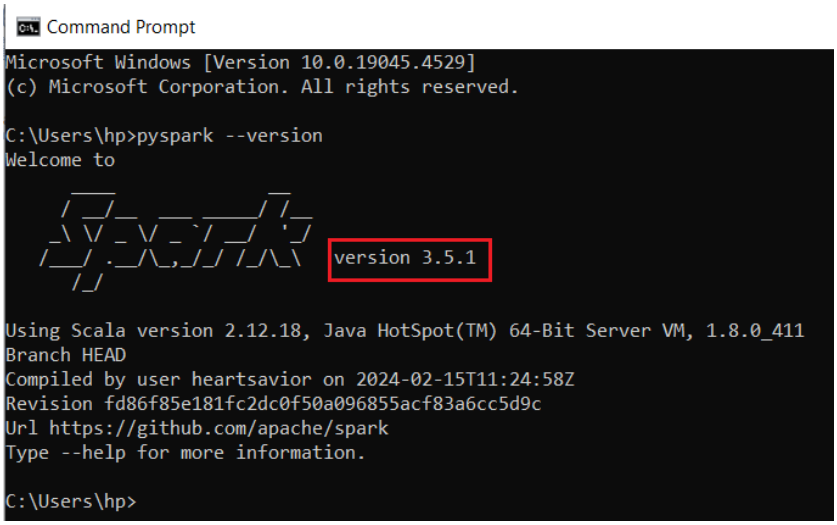
Press OK again to apply environment variable changes and close window.



3.3. Verify Spark Installation:

Open either **Command Prompt** or **Windows PowerShell** prompt and run the following command:

```
pyspark --version
```



After the above command is executed, you can see the installed **Spark version 3.5.1**.

3.4. Start Spark Services:

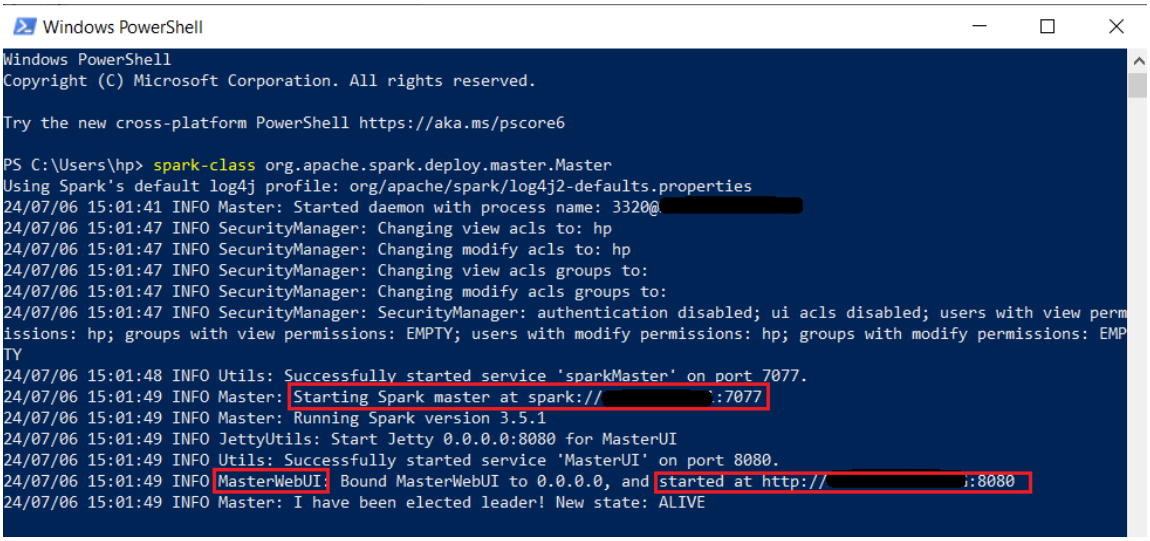
Now, we will start the Spark Master and Worker nodes in the local system.

3.4.1. Start Spark Master:

The master node is responsible for coordinating the work among the worker nodes in the Spark cluster.

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the Spark Master service:

```
spark-class org.apache.spark.deploy.master.Master
```

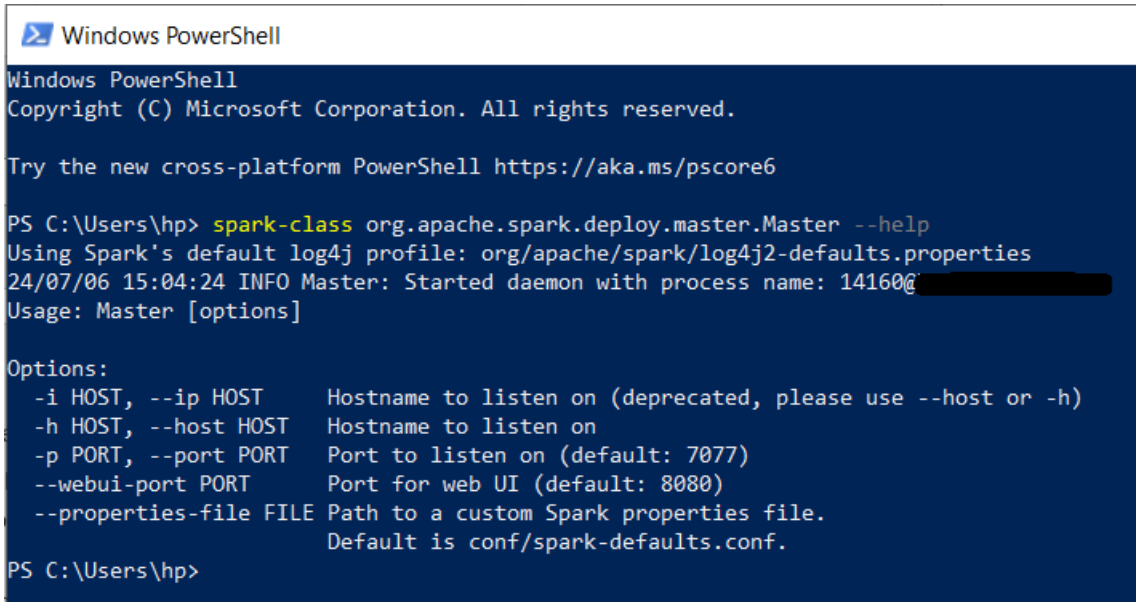


Here, we can see that Spark master is started at spark://<ipaddress>:7077 i.e listening on default port 7077 and **MasterWebUI** is started at http://<ipaddress>:8080 by default.

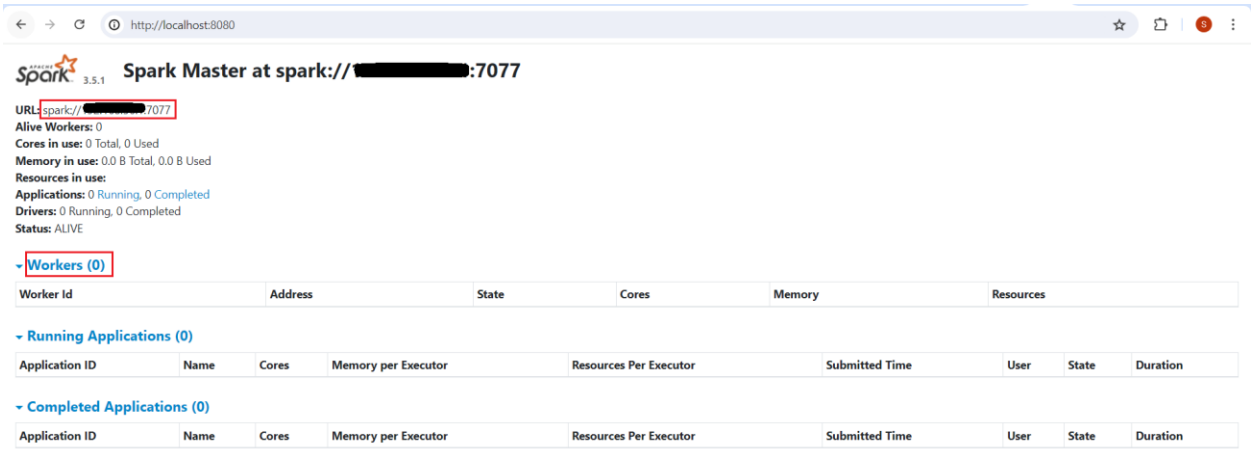
Note that we can change the Master node to listen, port to listen and web UI port by specifying additional options while starting Master server.

Use the following command to get the additional options for starting master server

```
spark-class org.apache.spark.deploy.master.Master --help
```



Check that the master node is running correctly by accessing the Spark web UI at http://localhost:8080/



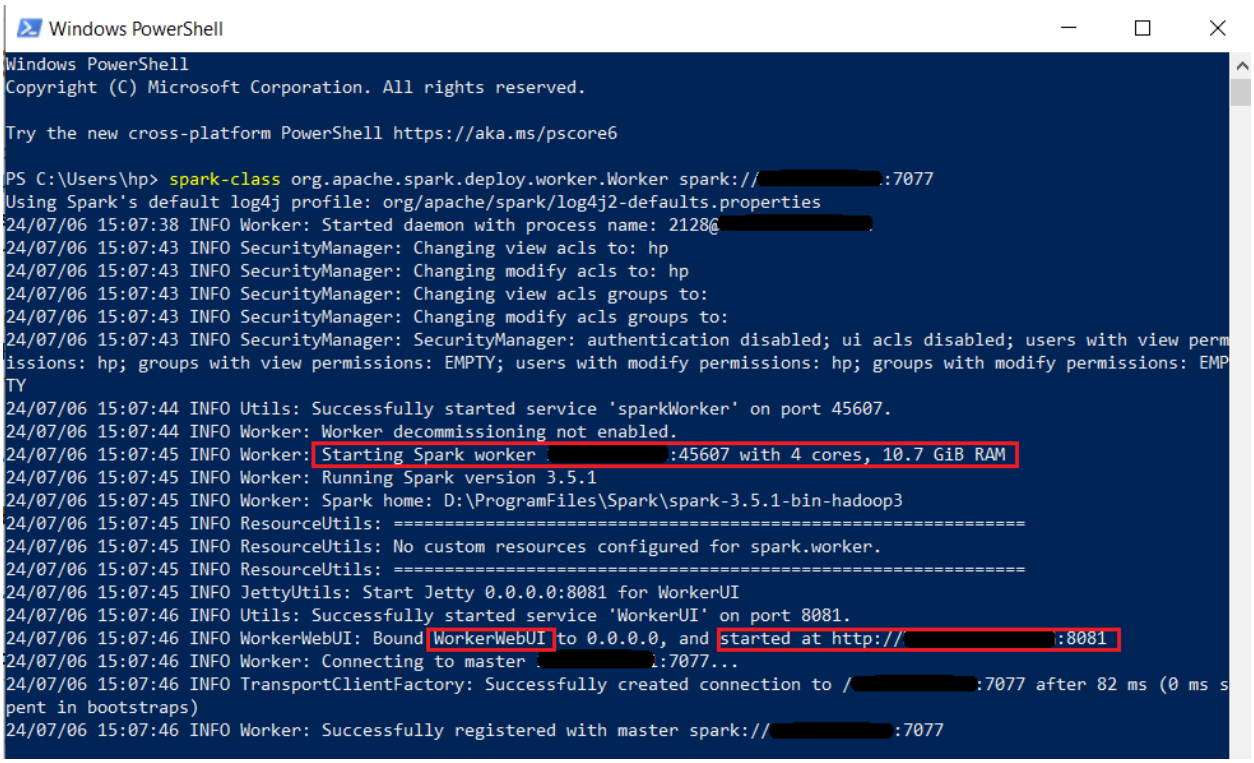
Here, we can see the Spark Master service is running at URL `spark://<ipaddress>:7077` and worker nodes as 0 since we have not started Spark worker service yet.

3.4.2. Start Spark Worker:

The worker nodes are responsible for executing the tasks assigned by the Spark master. We need to start the worker node by specifying the Spark master URL.

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the Spark Worker service *(Make sure that you replace `<ipaddress>` with IP address mentioned in your Spark Master URL)*

```
spark-class org.apache.spark.deploy.worker.Worker
spark://<ipaddress>:7077
```

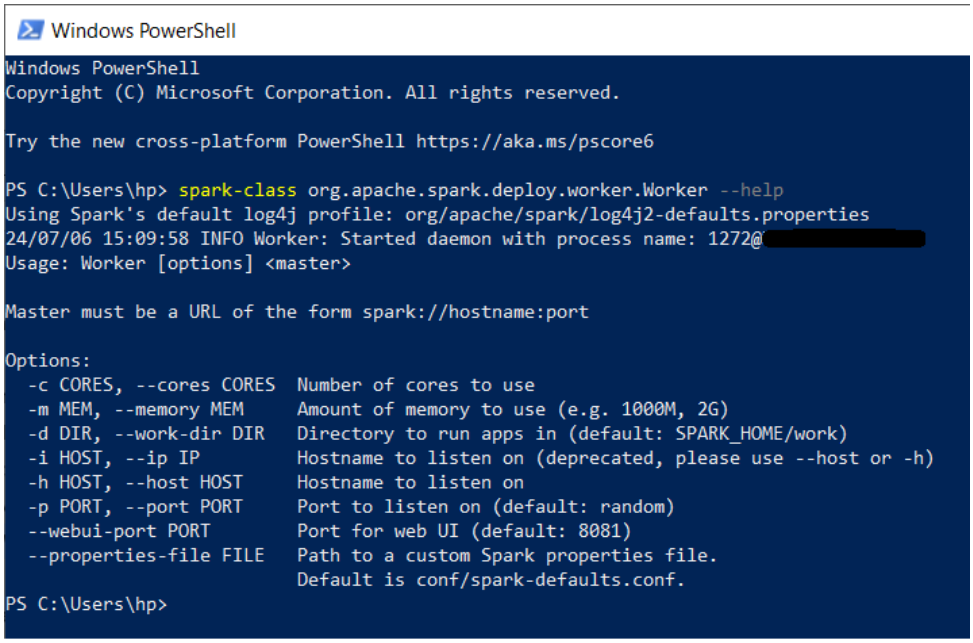


Here, we can see that Spark worker is started at `<ipaddress>:45607` (on a random port) with available number of cores and RAM running on the machine. **WorkerWebUI** is started at http://<ipaddress>:8081 by default.

Note that we can specify the number of cores to use and memory to use for Worker node, also change the Worker node to listen, port to listen and web UI port by specifying additional options while starting Worker server.

Use the following command to get the additional options for starting worker server

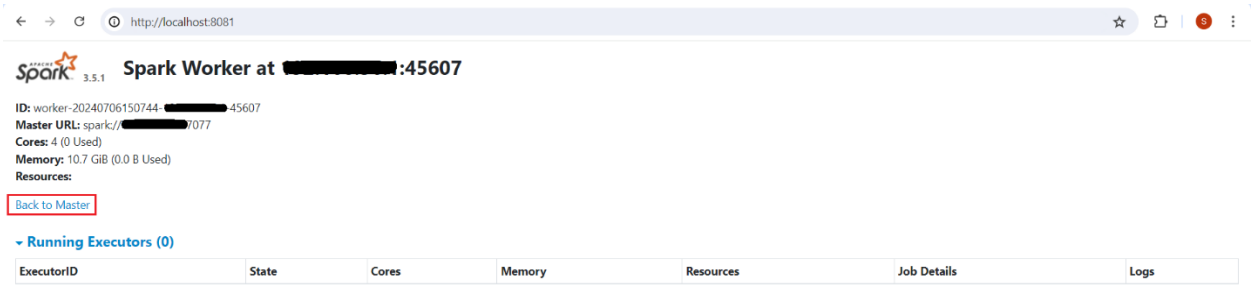
```
spark-class org.apache.spark.deploy.worker.Worker --help
```



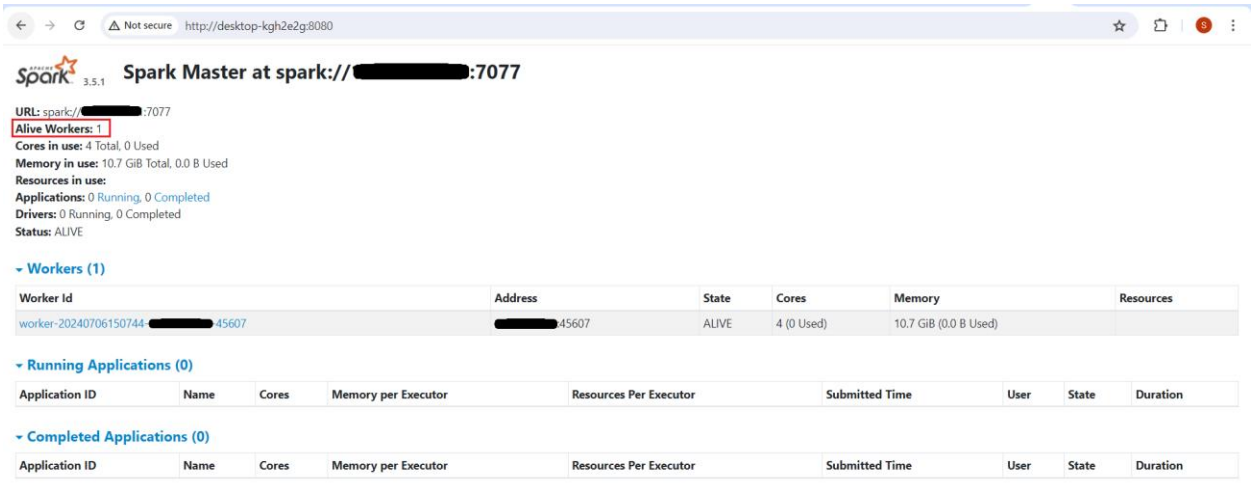
For example, to start the Worker node with 2 cores and 8GB RAM, use this command:

```
spark-class org.apache.spark.deploy.worker.Worker -c 2 -m 8G
spark://<ipaddress>:7077
```

Check that the worker node is connected to the Spark master correctly by accessing the Spark Worker web UI at http://localhost:8081/



Click on **Back to Master** link to open the Master UI where we can see one Alive Worker is available.



3.5. Spark Examples:

Spark provides some in-built example programs such as `pi, wordcount, sort,` etc. available under `SPARK_HOME\examples` directory that can be execute on Spark cluster.

3.5.1. Run Java SparkPI Program:

Let us execute the example Java program `SparkPi` packaged under `spark-examples*.jar` file that is located in `SPARK_HOME\examples\jars` directory.

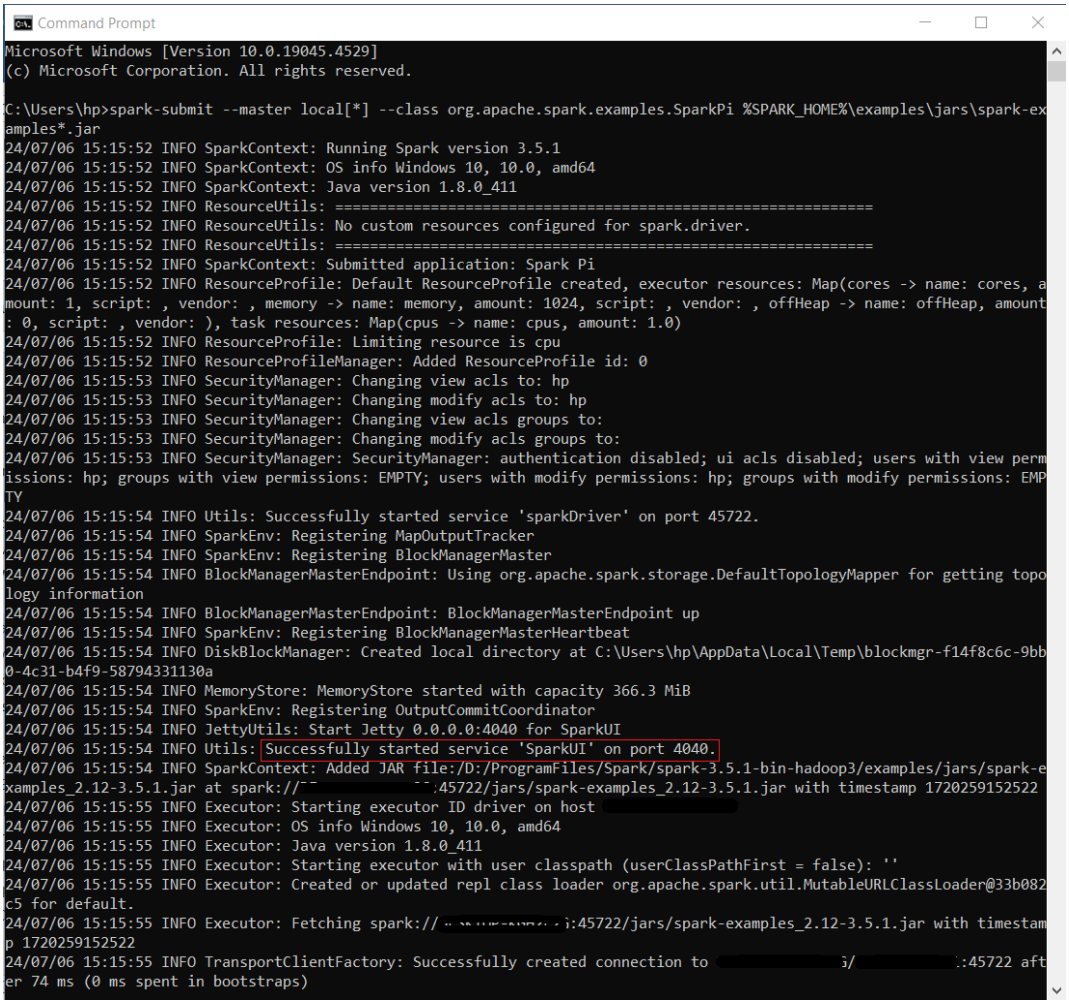
In Local Mode:

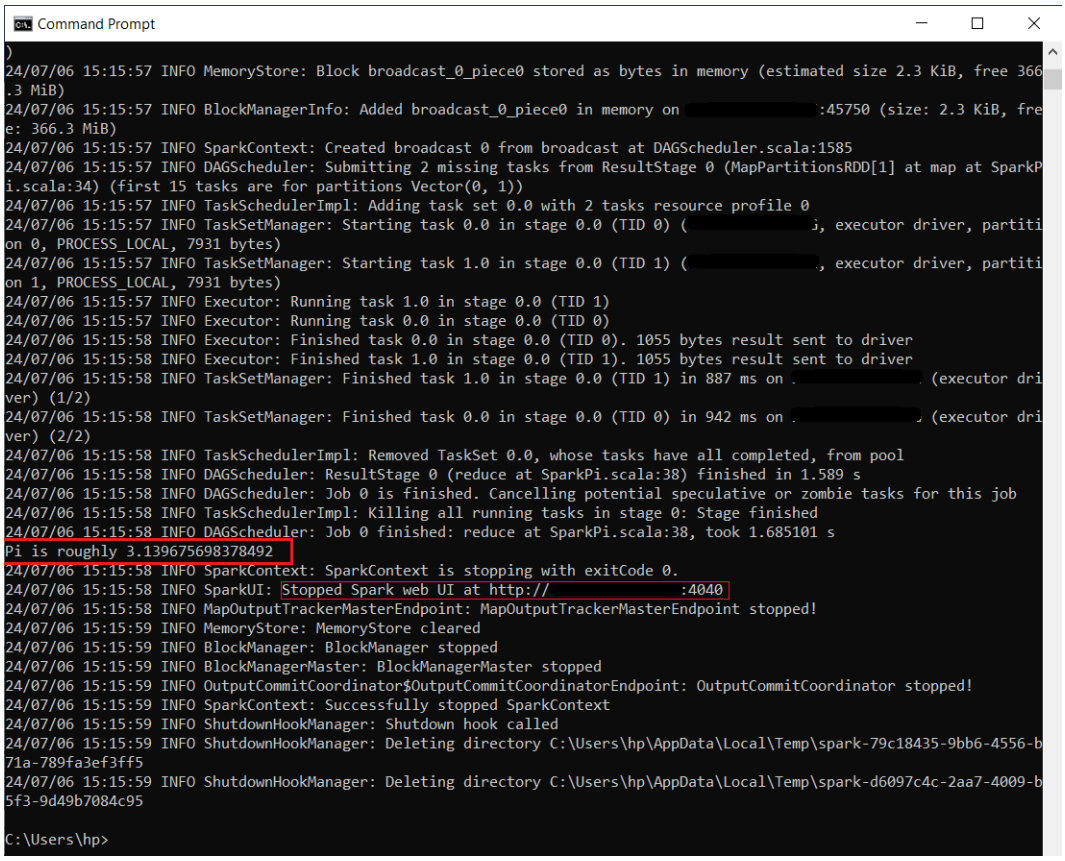
Open a new Command Prompt and run the below command which `SparkPi` application in local mode with all available cores:

```
spark-submit --class org.apache.spark.examples.SparkPi
%SPARK_HOME%\examples\jars\spark-examples*.jar
```

or

```
spark-submit --master local[*] --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```


Command Prompt
Microsoft Windows [Version 10.0.19045.4529]
(c) Microsoft Corporation. All rights reserved.
C:\Users\hp>spark-submit --master local[*] --class org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-examples*.jar
24/07/06 15:15:52 INFO SparkContext: Running Spark version 3.5.1
24/07/06 15:15:52 INFO SparkContext: OS info Windows 10, 10.0, amd64
24/07/06 15:15:52 INFO SparkContext: Java version 1.8.0_411
24/07/06 15:15:52 INFO ResourceUtils: ==
24/07/06 15:15:52 INFO ResourceUtils: No custom resources configured for spark.driver.
24/07/06 15:15:52 INFO ResourceUtils: ==
24/07/06 15:15:52 INFO SparkContext: Submitted application: Spark Pi
24/07/06 15:15:52 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amount: 1024, script: , vendor: , offHeap -> name: offHeap, amount: 0, script: , vendor:), task resources: Map(cpus -> name: cpus, amount: 1.0)
24/07/06 15:15:52 INFO ResourceProfile: Limiting resource is cpu
24/07/06 15:15:52 INFO ResourceProfileManager: Added ResourceProfile id: 0
24/07/06 15:15:53 INFO SecurityManager: Changing view acls to: hp
24/07/06 15:15:53 INFO SecurityManager: Changing modify acls to: hp
24/07/06 15:15:53 INFO SecurityManager: Changing view acls groups to:
24/07/06 15:15:53 INFO SecurityManager: Changing modify acls groups to:
24/07/06 15:15:53 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: hp; groups with view permissions: EMPTY; users with modify permissions: hp; groups with modify permissions: EMPTY
24/07/06 15:15:54 INFO Utils: Successfully started service 'sparkDriver' on port 45722.
24/07/06 15:15:54 INFO SparkEnv: Registering MapOutputTracker
24/07/06 15:15:54 INFO SparkEnv: Registering BlockManagerMaster
24/07/06 15:15:54 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for getting topology information
24/07/06 15:15:54 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
24/07/06 15:15:54 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
24/07/06 15:15:54 INFO DiskBlockManager: Created local directory at C:\Users\hp\AppData\Local\Temp\blockmgr-f14f8c6c-9bb0-4c31-b4f9-58794331130a
24/07/06 15:15:54 INFO MemoryStore: MemoryStore started with capacity 366.3 MiB
24/07/06 15:15:54 INFO SparkEnv: Registering OutputCommitCoordinator
24/07/06 15:15:54 INFO JettyUtils: Start Jetty 0.0.0.0:4040 for SparkUI
24/07/06 15:15:54 INFO Utils: Successfully started service 'SparkUI' on port 4040.
24/07/06 15:15:54 INFO SparkContext: Added JAR file:/D:/ProgramFiles/Spark/spark-3.5.1-bin-hadoop3/examples/jars/spark-examples_2.12-3.5.1.jar at spark://[redacted]:45722/jars/spark-examples_2.12-3.5.1.jar with timestamp 1720259152522
24/07/06 15:15:55 INFO Executor: Starting executor ID driver on host [redacted]
24/07/06 15:15:55 INFO Executor: OS info Windows 10, 10.0, amd64
24/07/06 15:15:55 INFO Executor: Java version 1.8.0_411
24/07/06 15:15:55 INFO Executor: Starting executor with user classpath (userClassPathFirst = false): ''
24/07/06 15:15:55 INFO Executor: Created or updated repl class loader org.apache.spark.util.MutableURLClassLoader@33b082c5 for default.
24/07/06 15:15:55 INFO Executor: Fetching spark://[redacted]:45722/jars/spark-examples_2.12-3.5.1.jar with timestamp 1720259152522
24/07/06 15:15:55 INFO TransportClientFactory: Successfully created connection to [redacted]:45722 after 74 ms (0 ms spent in bootstraps)



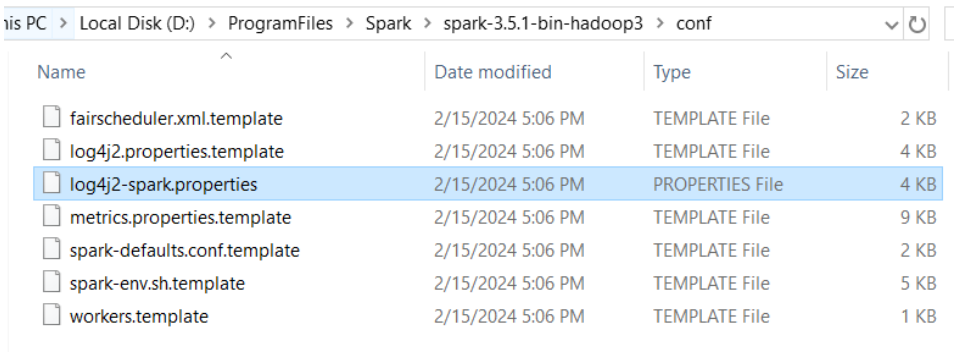
It displayed the output of approximate Pi value which is **3.1396**. In the console log, we can also see that Spark web UI was started and stopped at http://localhost:4040

Note:

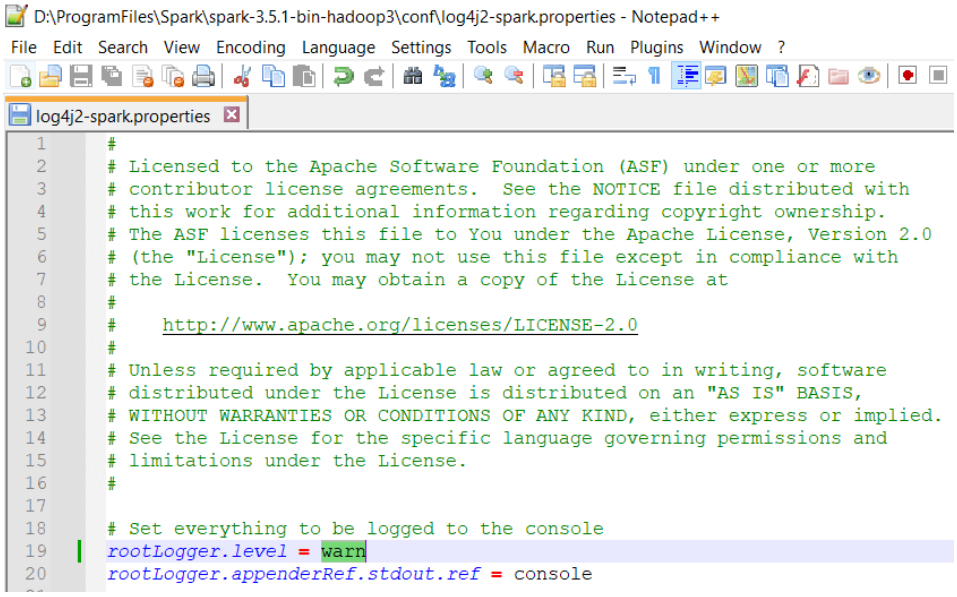
By default, Spark displays **INFO** logging on the console.

Follow the below steps to turn off the unnecessary INFO logging:

- Go to `SPARK_HOME\conf` folder, take the copy of `log4j2.properties.template` file and rename as `log4j2-spark.properties`.

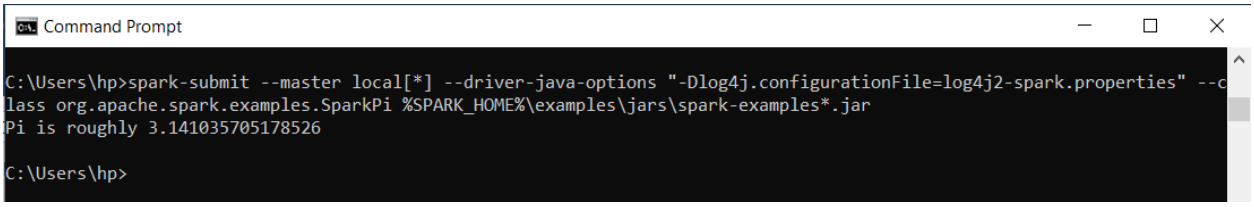


- Open `log4j2-spark.properties` and change the `rootLogger.level` value from `info` to `warn` as shown below:



Now, execute the below `spark-submit` command and we will not see any **INFO** level logging on the console.

```
spark-submit --master local[*] --driver-java-options "-
Dlog4j.configurationFile=log4j2-spark.properties" --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```



In Client Mode:

To run the `SparkPi` application in client mode, use the below command *(Make sure that you replace `<ipaddress>` with the IP address mentioned in your Spark Master URL)*

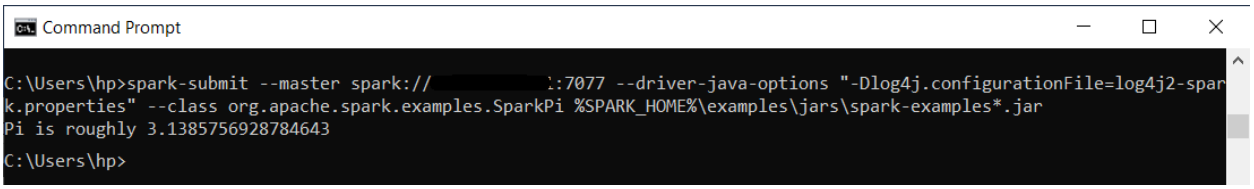
```
spark-submit --master spark://<ipaddress>:7077 --deploy-mode client -
-class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```

or

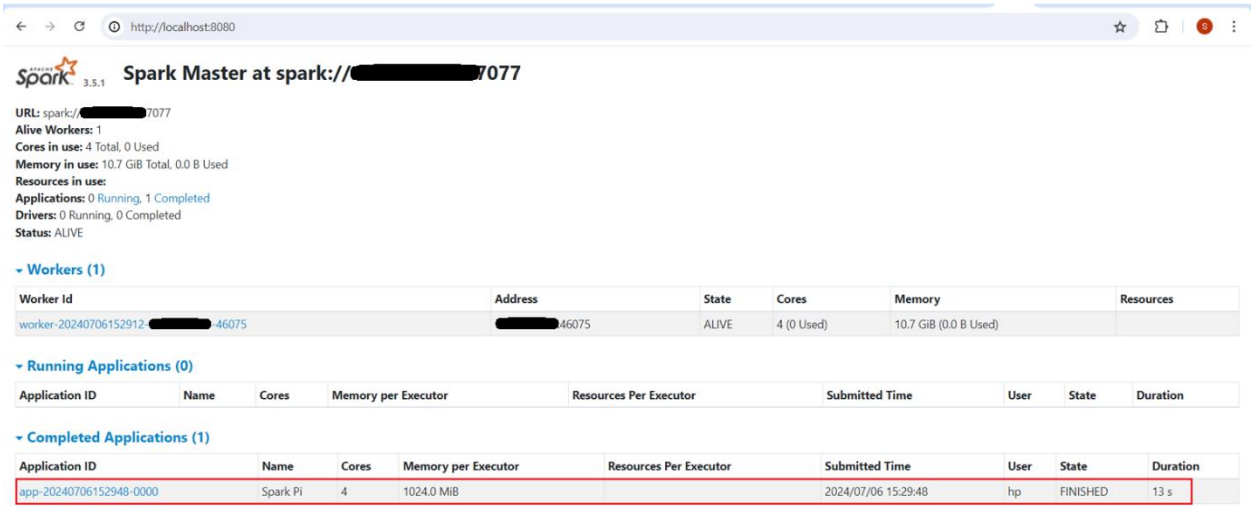
```
spark-submit --master spark://<ipaddress>:7077 --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```

If you want to skip the INFO logging on console, use the extra property `--driver-java-options "-Dlog4j.configurationFile=log4j2-spark.properties"` in the `spark-submit` command as below:

```
spark-submit --master spark://<ipaddress>:7077 --driver-java-options
"-Dlog4j.configurationFile=log4j2-spark.properties" --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```



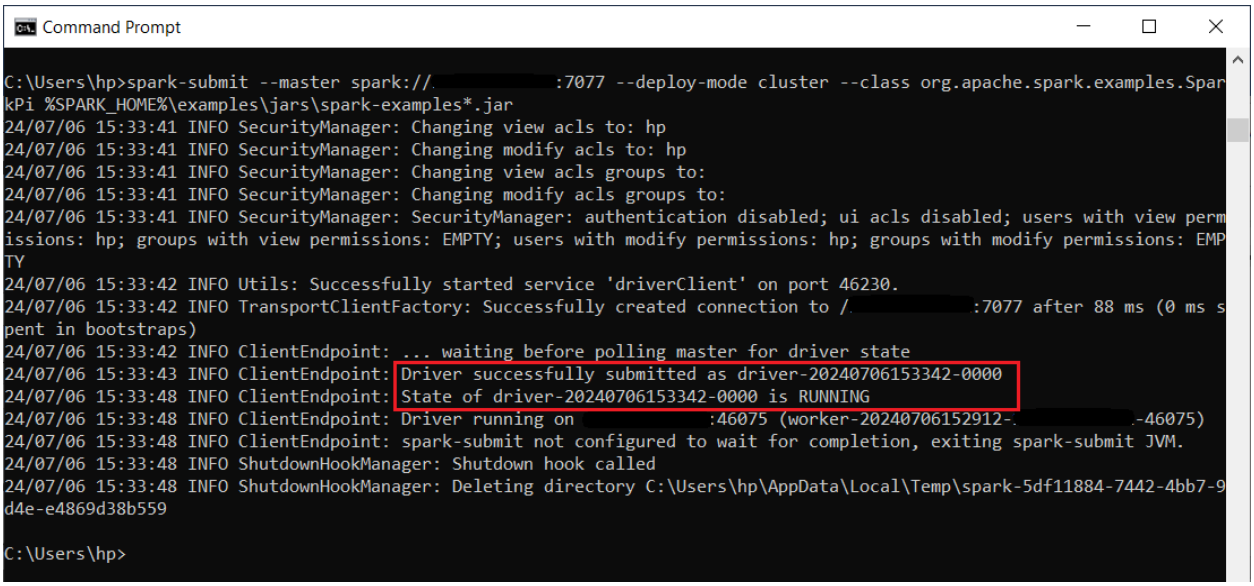
On the Master Web UI (http://localhost:8080), we can see that **Spark Pi** application has been submitted and completed.



<u>In Cluster Mode:</u>

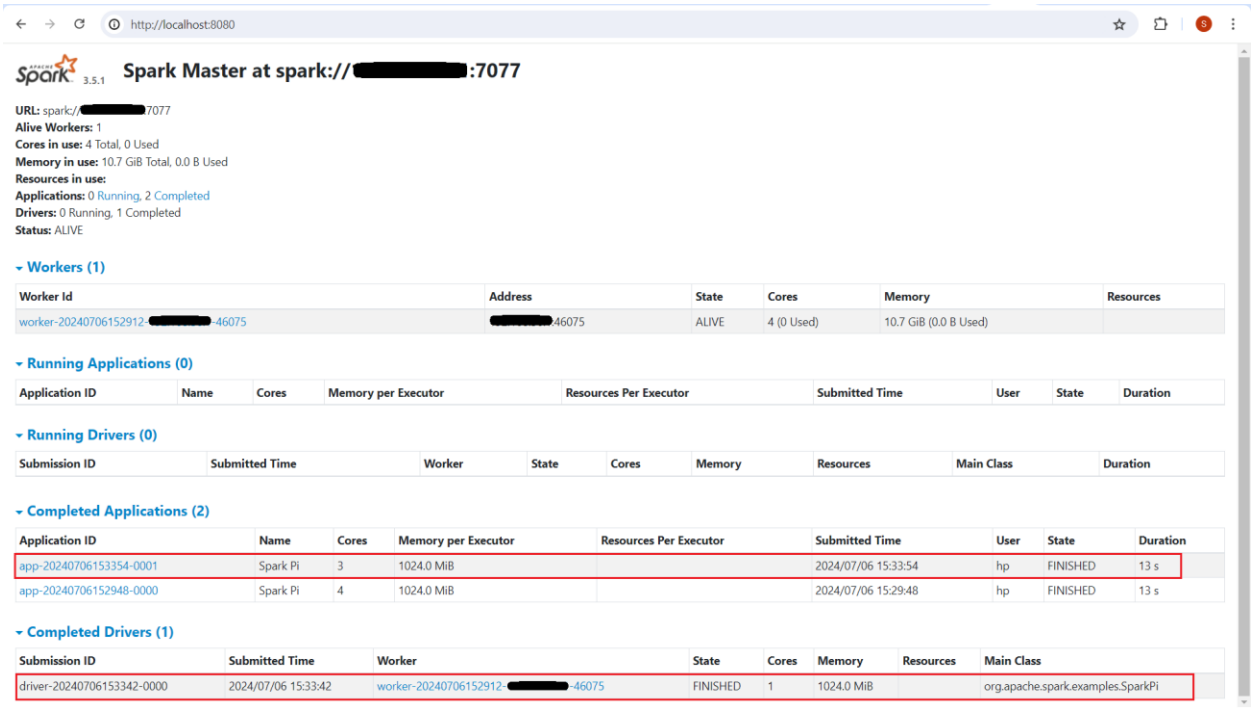
To run the `SparkPi` application in cluster mode, use the below command *(Make sure that you replace `<ipaddress>` with the IP address mentioned in your Spark Master URL)*

```
spark-submit --master spark://<ipaddress>:7077 --deploy-mode cluster
--class org.apache.spark.examples.SparkPi
%SPARK_HOME%\examples\jars\spark-examples*.jar
```



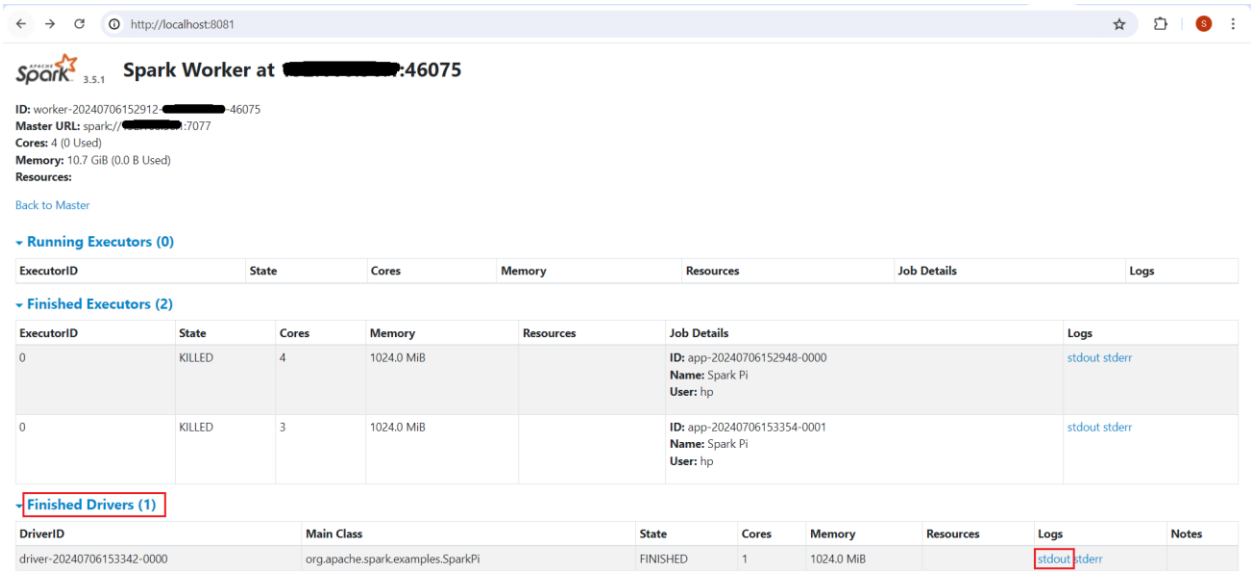
Since we used “cluster” deploy mode, we can see that the Spark application has been submitted as Driver and once the Driver started running, it exited the `spark-submit` JVM.

We can track the further application status on the Master Web UI at http://localhost:8080



On the Master Web UI, we can see that a new application ID has been submitted and finished. We can also see the application submission details under **Completed Drivers** section.

To see the application output, go to the Worker Web UI http://localhost:8081/ and click on **stdout** link under **Finished Drivers** section.

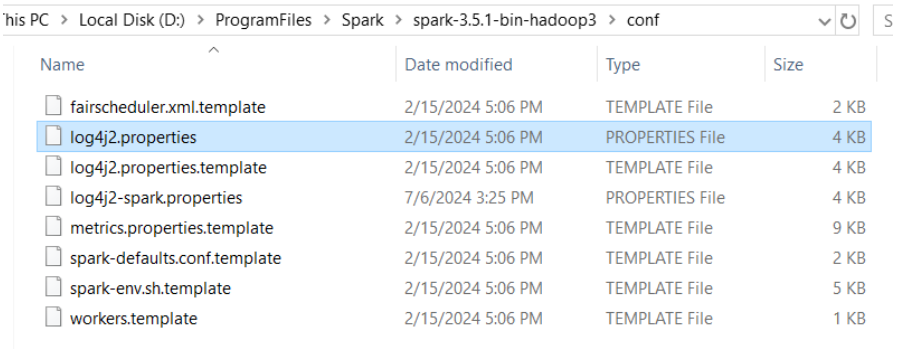


We can see the standard output *"Pi is roughly 3.135"* on the log page for the respective Driver ID.

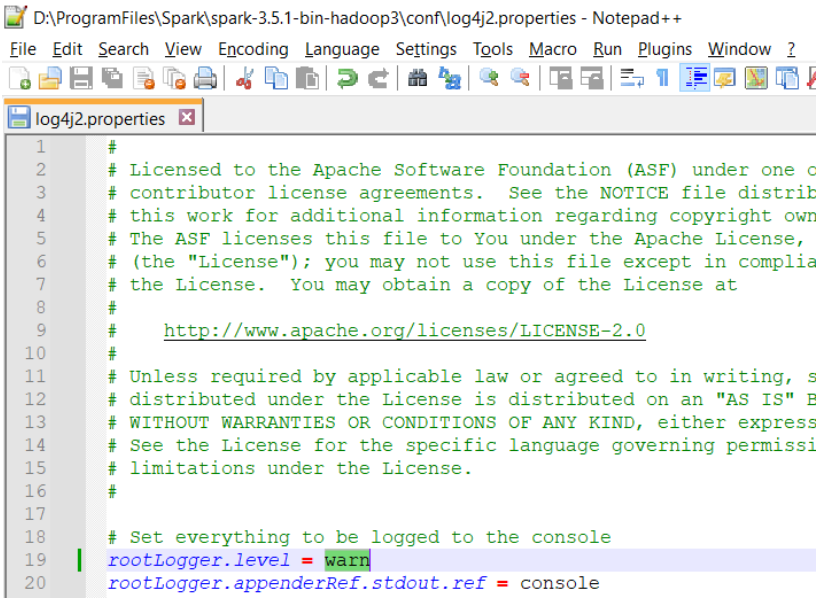
Note:

Follow the below steps to skip **INFO** logging on console while executing Spark in cluster mode:

- Go to `SPARK_HOME\conf` folder, take the copy of `log4j2.properties.template` file and rename as `log4j2.properties`.



- Open `log4j2.properties` and change the `rootLogger.level` value from `info` to `warn` as shown below:



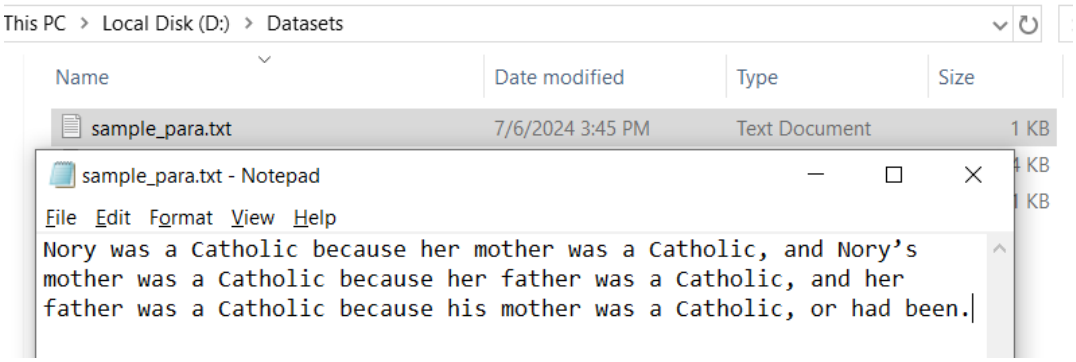
Now, execute the `spark-submit` command with cluster deploy mode and we will not see any INFO messages on the console.

```
spark-submit --master spark://<ipaddress>:7077 --deploy-mode cluster
--class org.apache.spark.examples.SparkPi
%SPARK_HOME%\examples\jars\spark-examples*.jar
```

3.5.2. Run Python wordcount Program:

Let us execute the Python program `wordcount` which counts the occurrences of each word in the given input file. This program is located in `SPARK_HOME\examples\src\main\python` directory.

First, create an input file named `sample_para.txt` with some random text and place it in `D:\Datasets` folder.



Next, we will run the `wordcount.py` program in local mode, client mode and cluster mode.

In Local Mode:

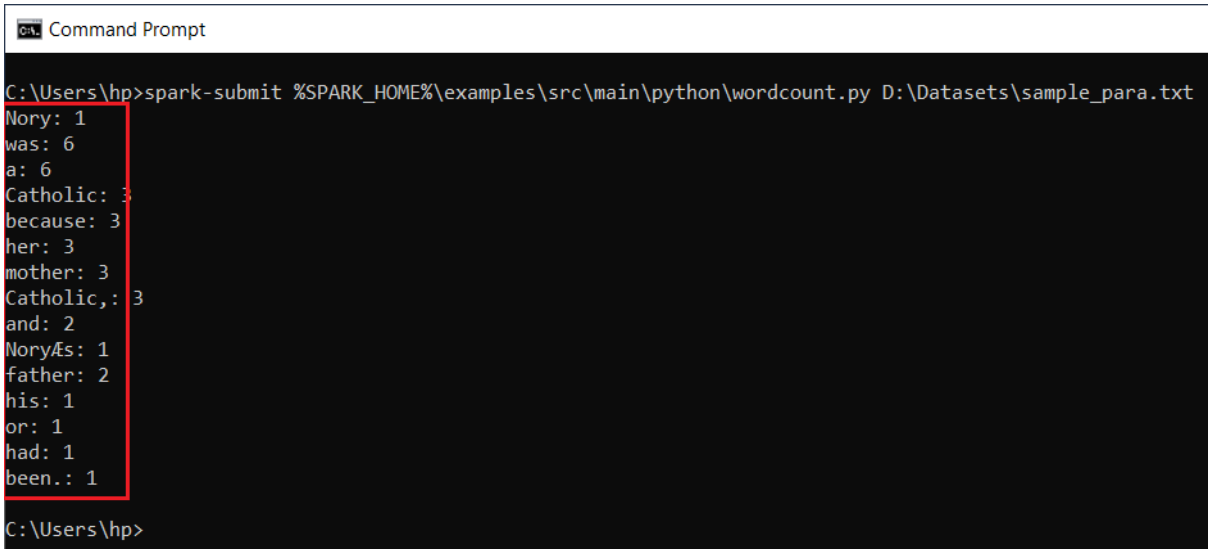
Use this command to execute the **PythonWordCount** application in local mode with all available cores:

```
spark-submit --master local[*]
%SPARK_HOME%\examples\src\main\python\wordcount.py
D:\Datasets\sample_para.txt
```

or

```
spark-submit %SPARK_HOME%\examples\src\main\python\wordcount.py
D:\Datasets\sample_para.txt
```

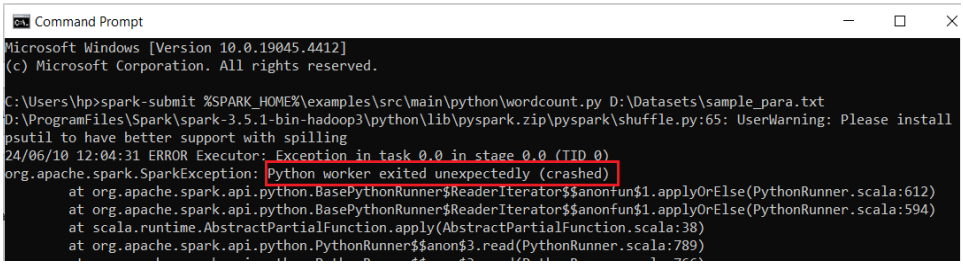
After the above command is executed, it displays the count of each word available in the input file on the console.



Here, we are not seeing any **INFO** messages on the console here because we have set `rootLogger.level` to `warn` in `SPARK_HOME\conf\log4j2.properties` file.

Note:

While executing Python programs in Spark, you may receive error *SparkException: Python worker exited unexpectedly (crashed)* as below:



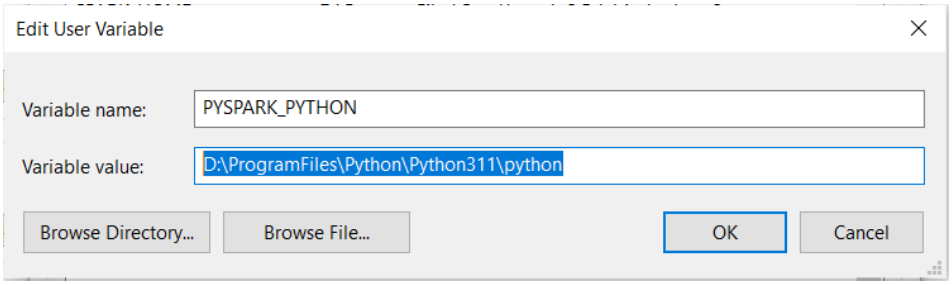
This error mostly occurs due to the latest Python version 3.12.

Follow the below steps to resolve the above error:

- Install the lower version of Python (say 3.11.9) from the official python downloads website. I installed Python 3.11.9 version in `D:\ProgramFiles\Python\Python311` folder.

- Open **Environment Variables** dialog, select `PYSPARK_PYTHON` variable under **User variables** tab and press **Edit** button

- Update the value to the new location where the older version of `python.exe` file is available *(in my case, it is `D:\ProgramFiles\Python\Python311\python`)*. Press **OK** and then **OK** to apply changes



- Open a new command prompt and start executing PySpark programs.

In Client Mode:

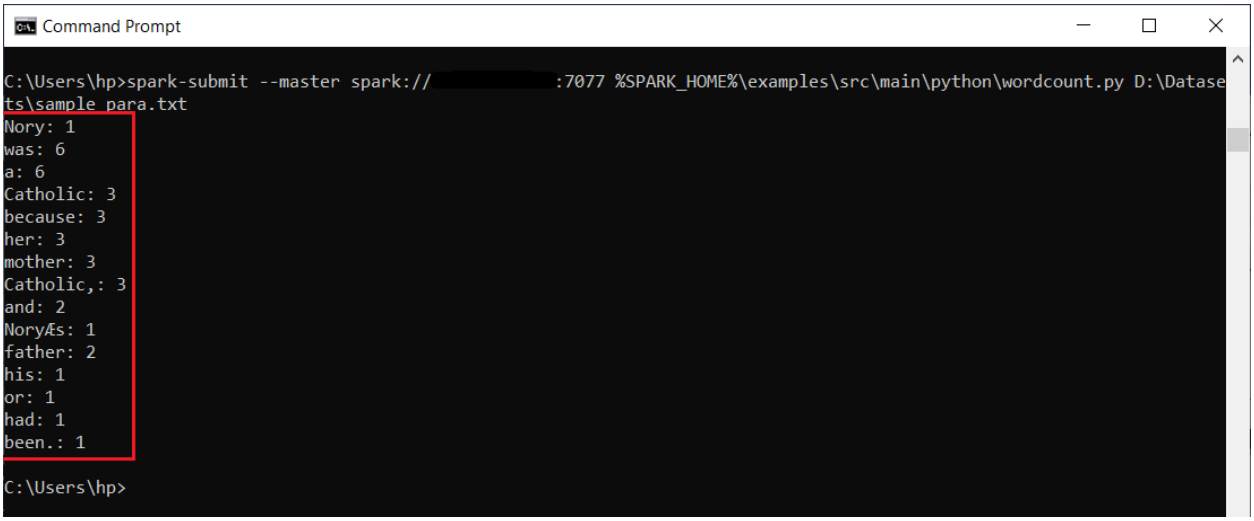
Use this command to execute the **PythonWordCount** application in client mode *(Make sure that you replace `<ipaddress>` with the IP address mentioned in your Spark Master URL)*

```
spark-submit --master spark://<ipaddress>:7077 --deploy-mode client
%SPARK_HOME%\examples\src\main\python\wordcount.py
D:\Datasets\sample_para.txt
```

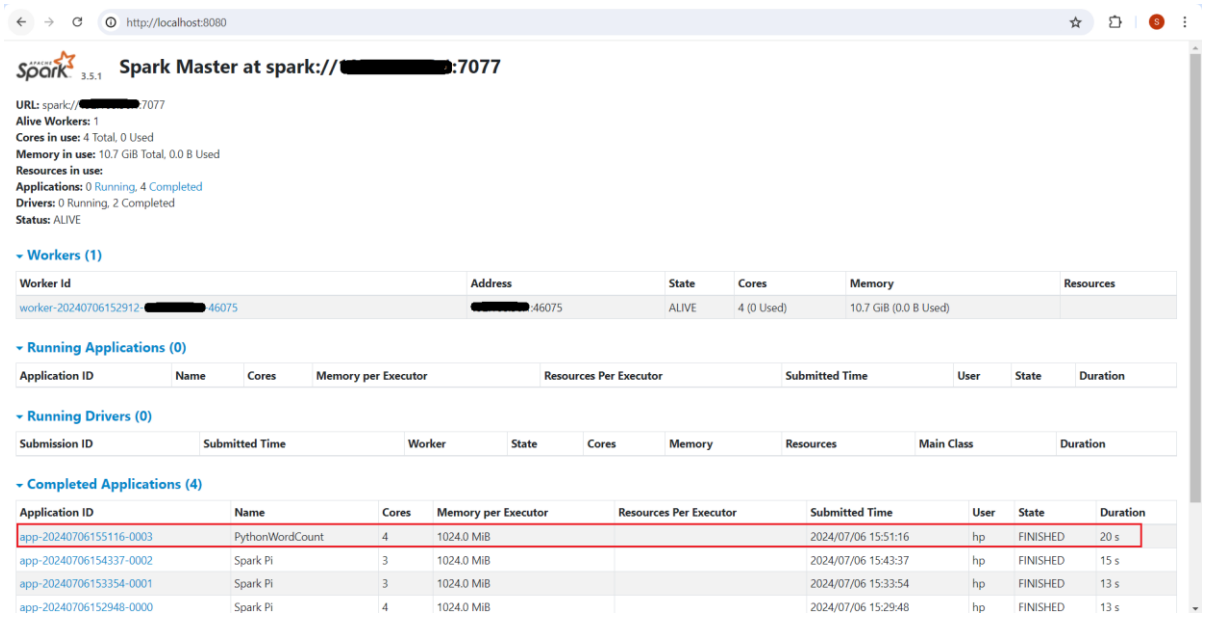
or

```
spark-submit --master spark://<ipaddress>:7077
%SPARK_HOME%\examples\src\main\python\wordcount.py
D:\Datasets\sample_para.txt
```

After the above command is executed, it displays the count of each word available in the input file on the console.



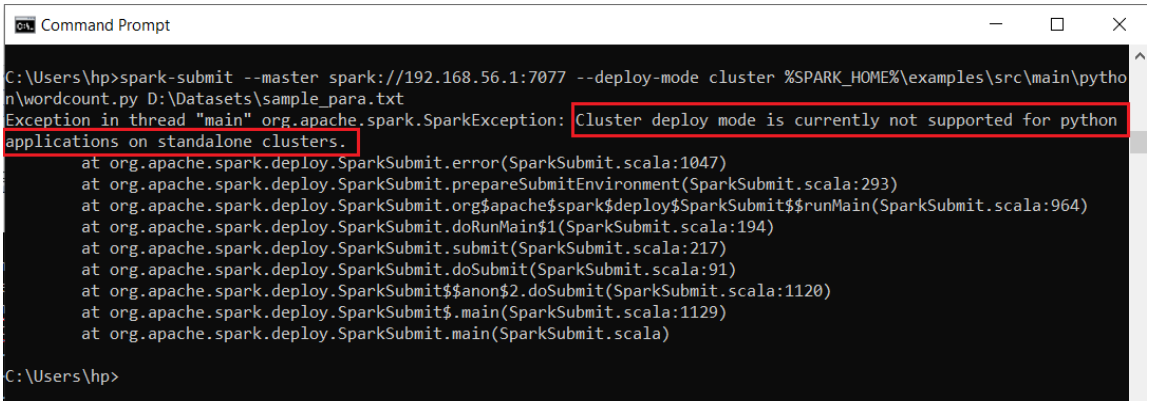
On the Master Web UI (http://localhost:8080), we can see that **PythonWorkdCount** application has been submitted and completed.



In Cluster Mode:

Use this command to execute the **PythonWordCount** application in cluster mode *(Make sure that you replace `<ipaddress>` with the IP address mentioned in your Spark Master URL)*

```
spark-submit --master spark://<ipaddress>:7077 –-deploy-mode cluster
%SPARK_HOME%\examples\src\main\python\wordcount.py
D:\Datasets\sample_para.txt
```



However, it throws **SparkException** because Spark does not support Cluster deploy mode of Python applications on Standalone clusters.

4. Install Spark on Hadoop:

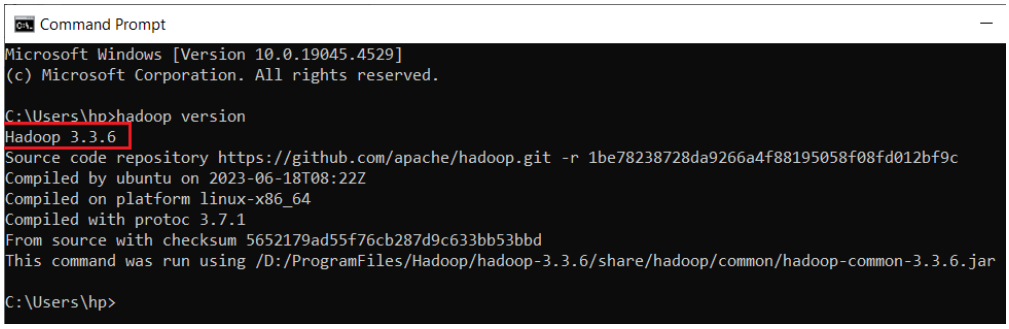
Now, we will see how to install Spark on top of existing Hadoop cluster.

4.1. Verify Hadoop Installation:

If you do not have Hadoop running, refer here to install Hadoop on Windows operating system.

Open **Command Prompt** or **Windows PowerShell** and run the following command:

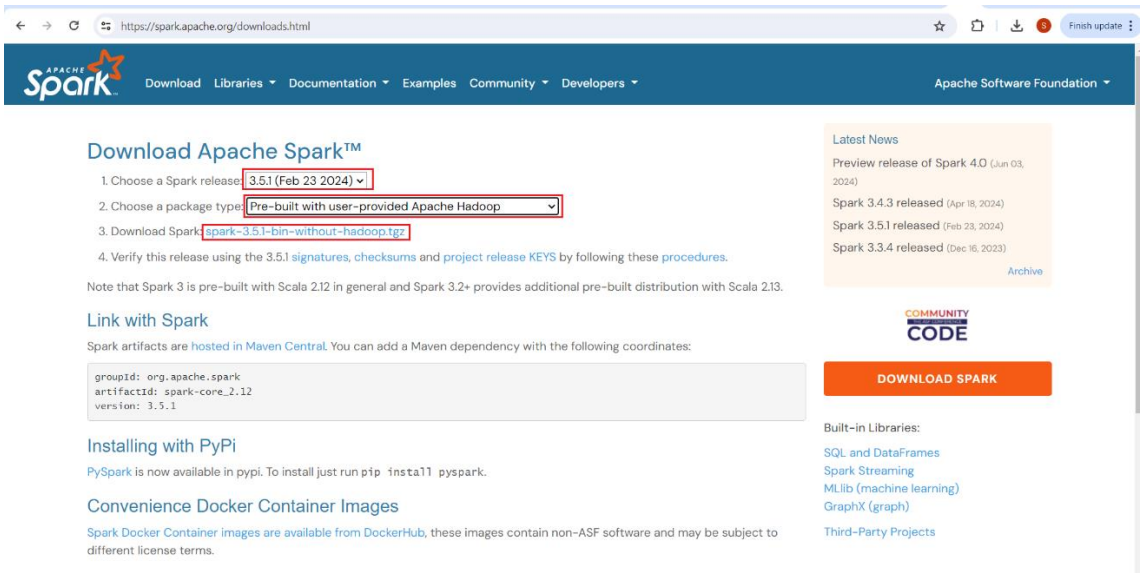
```
hadoop version
```



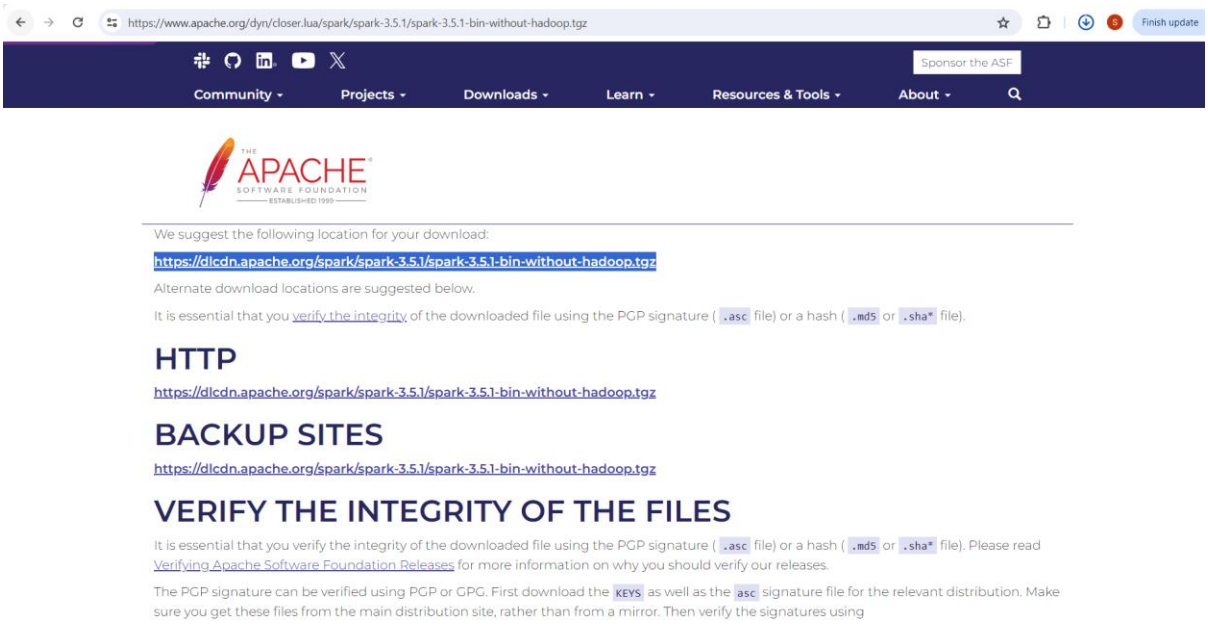
It shows that **Hadoop 3.3.6** version has been installed.

4.2. Download Spark Binaries:

Download the latest **Spark 3.5.1** version from the Apache Spark Downloads page. In this page, choose the **package type** as **Pre-built with user-provided Apache Hadoop** and click on `spark-3.5.1-bin-without-hadoop.tgz` binary file.

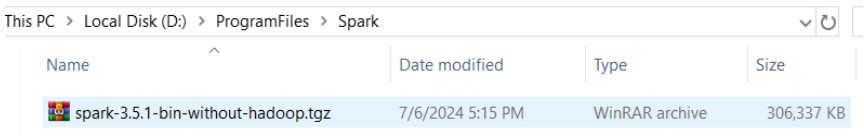


You will be navigated to spark-3.5.1 mirror website where click on the suggested location for spark-3.5.1-bin-wihtout-hadoop.tgz file which gets downloaded to your **Downloads** folder.

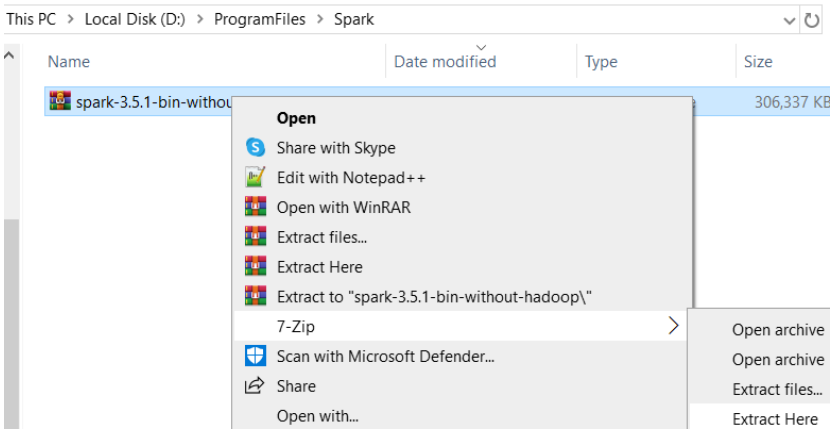


After the binary file is downloaded, unpack it using any file archiver (**7zip** or **WinRAR**) utility as below:

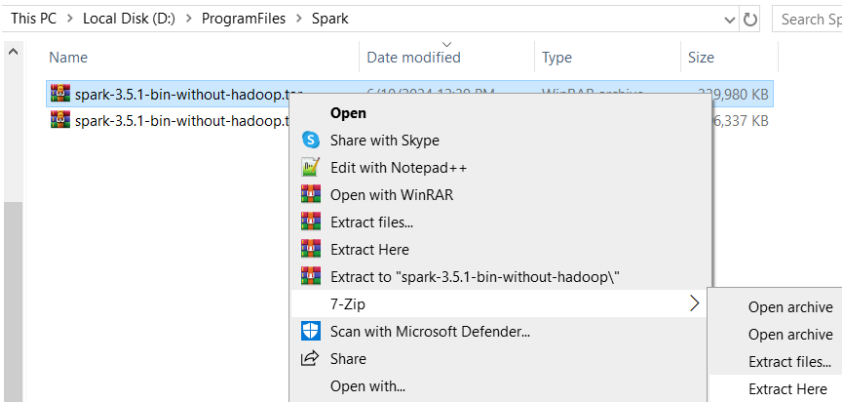
- Choose the installation directory in your machine and copy `spark-3.5.1-bin-without-hadoop.tgz` file to that directory. Here, we are choosing Spark installation directory as `D:\ProgramFiles\Spark`.



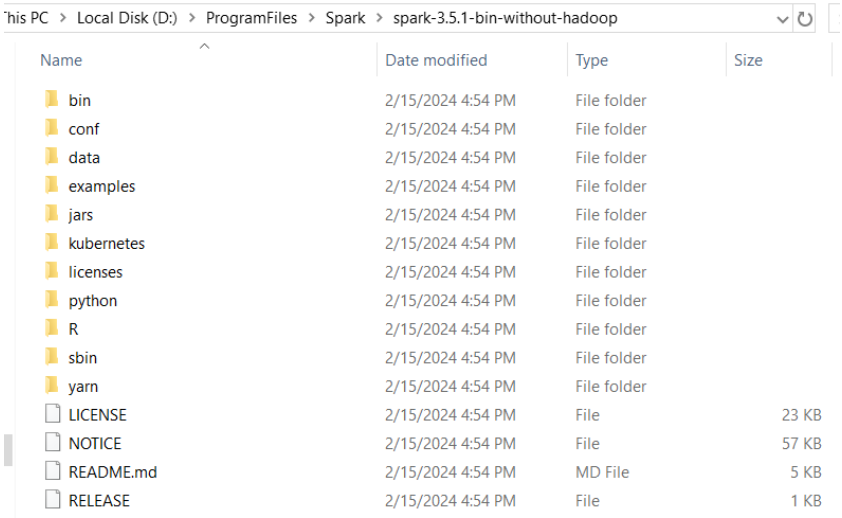
- Right click on `spark-3.5.1-bin-without-hadoop.tgz` and choose **7-Zip -> Extract Here** option which extracts a new packed file `spark-3.5.1-bin-without-hadoop.tar`.



- Next, unpack `spark-3.5.1-without-hadoop.tar` file using 7zip utility.



- The tar file extraction may take few minutes to finish. After finishing, you see a folder named `spark-3.5.1-bin-without-hadoop` which consists of Spark binaries and libraries.



4.3. Set up Environment Variables:

After installing Spark binaries, we should configure the below environment variables defining the Spark default paths.

- **SPARK_HOME**: This is the Spark installation directory path in the machine *(in our case, it is `D:\ProgramFiles\Spark\spark-3.5.1-bin-without-hadoop`)*

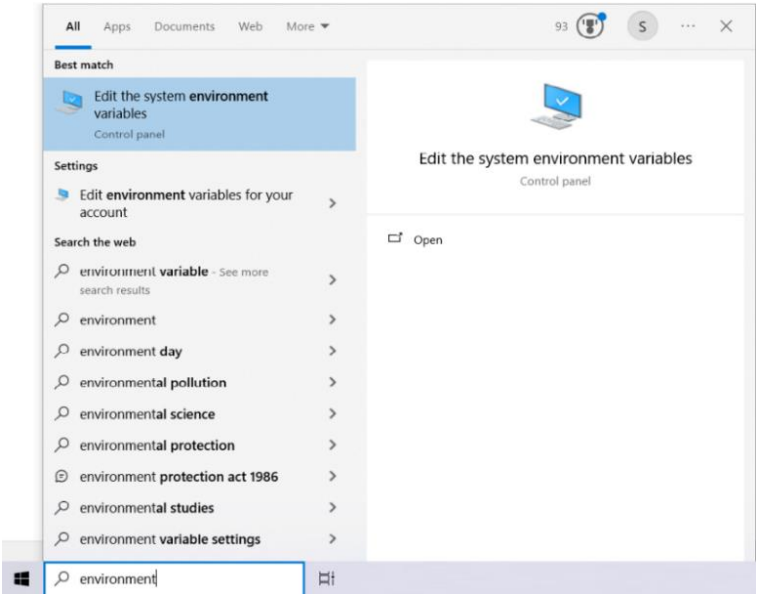
- **PYTHONPATH**: This is the Spark python directory path which should be. *`%SPARK_HOME%\python`*

- **PYSPARK_PYTHON**: Set this variable to the location where *`python.exe`* file is available *(in my machine, it is `D:\ProgramFiles\Anaconda\anaconda3\python`)* which is needed for Spark to execute the Python code.

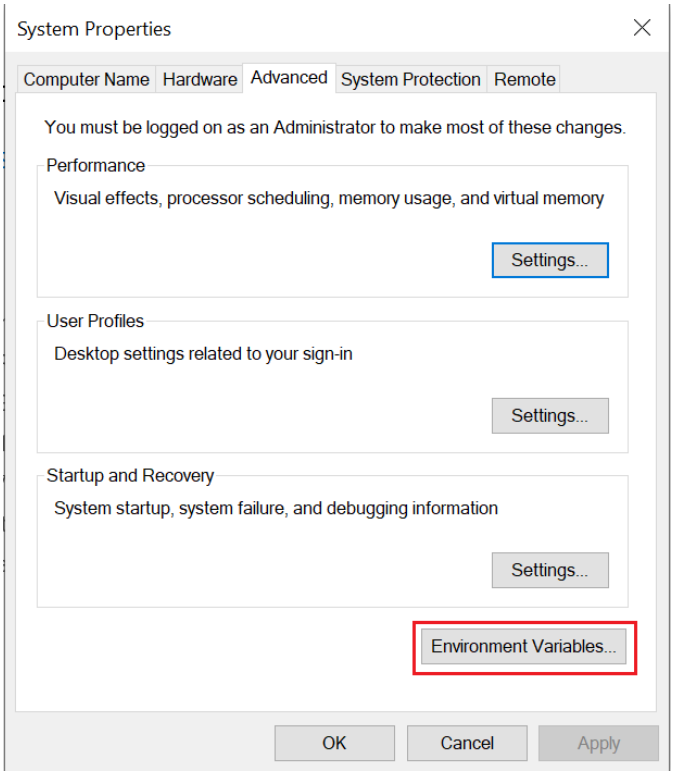
These variables need to be added to either **User environment variables** or **System environment variables** depending on Spark configuration needed **for a single user** or **for multiple users**.

In this tutorial, we will add User environment variables since we are configuring Spark for a single user. If you would like to configure Spark for multiple users, then define System environment variables.

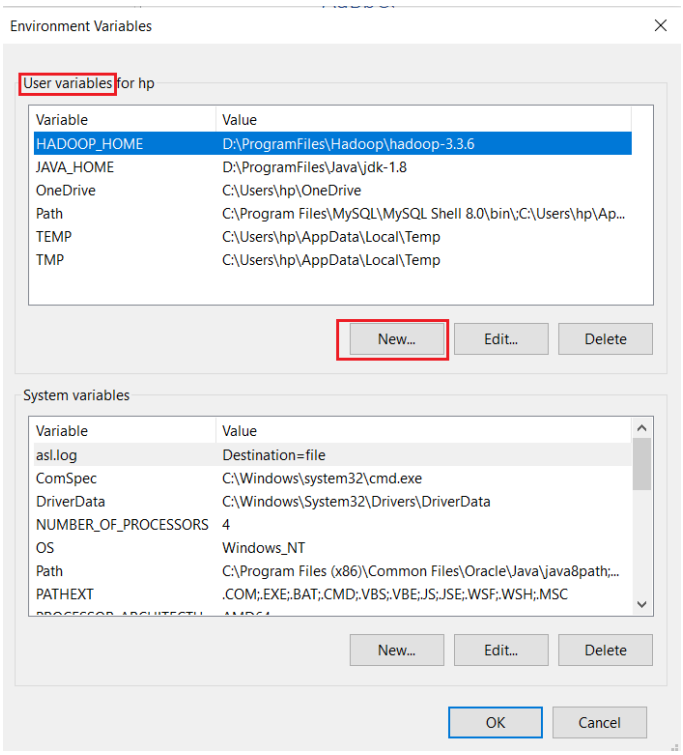
In the Windows search bar, start typing “environment variables” and select the first match which opens up **System Properties** dialog.



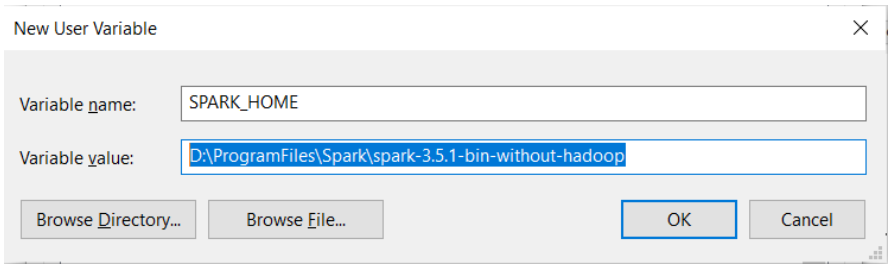
On the **System Properties** window, press **Environment Variables** button.



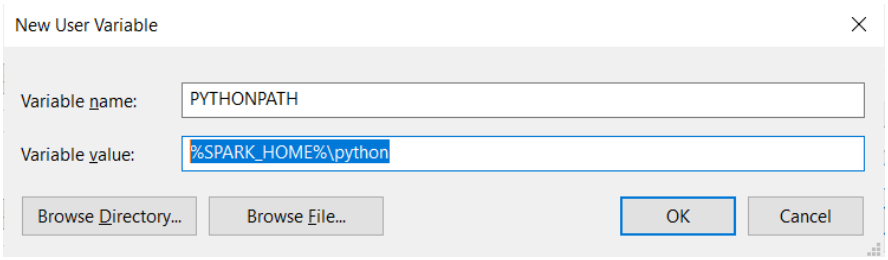
In the **Environment Variables** dialog, click on **New** under **User variables** section.



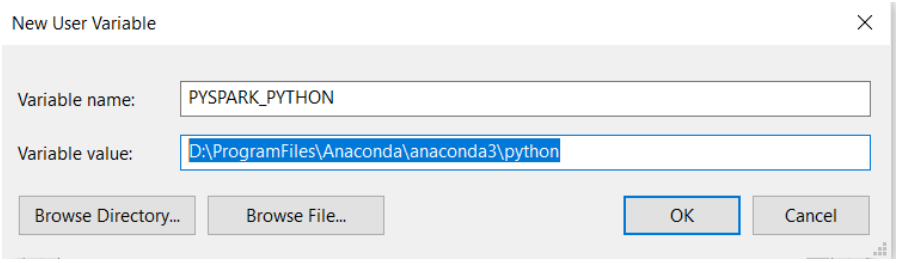
Add `SPARK_HOME` variable and press OK.



Click on **New** again and add `PYTHONPATH` variable and press OK.

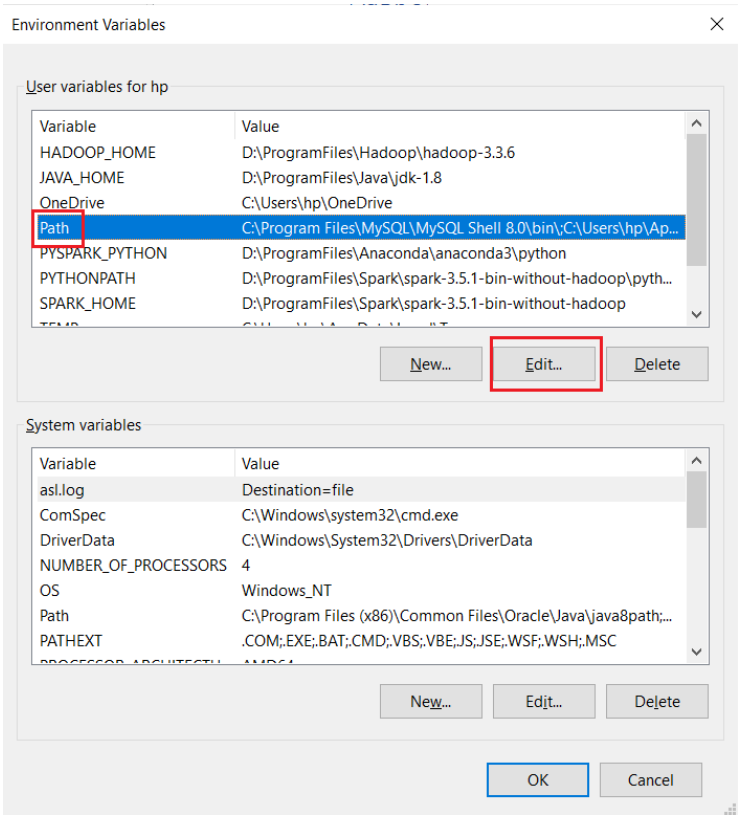


Click on **New** again and add `PYSPARK_PYTHON` variable and press OK.

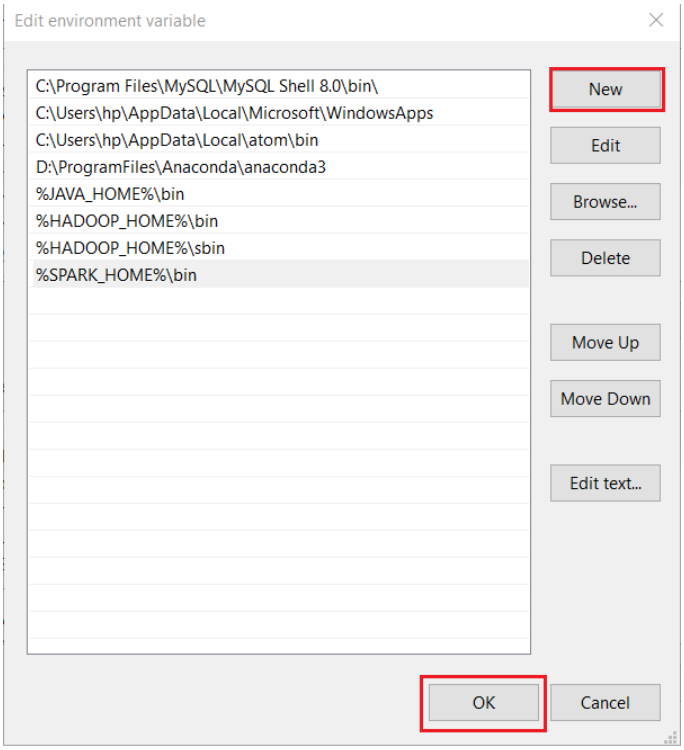


Now, we will update `PATH` variable to add Spark binary paths.

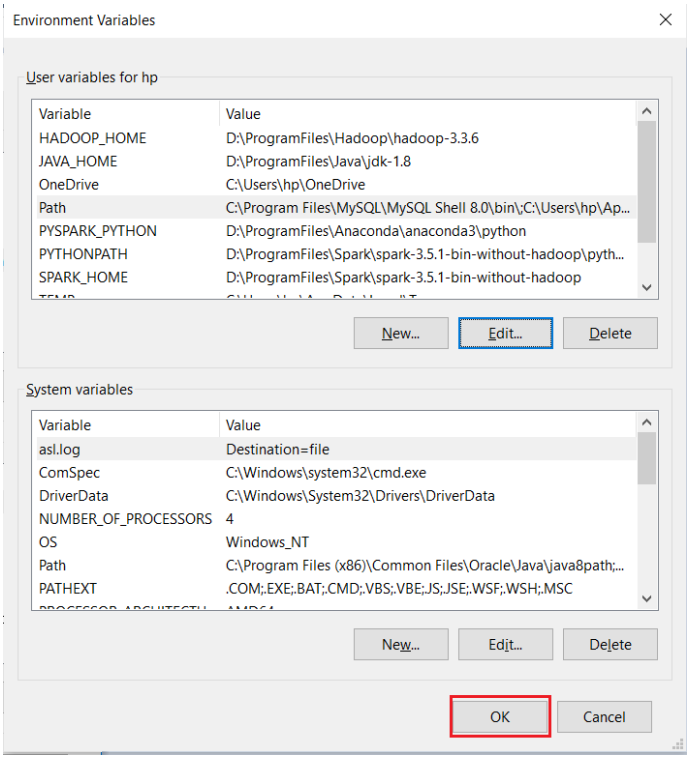
Select `PATH` variable under **User Variables** and press **Edit** button.



Press **New** and add `%SPARK_HOME%\bin` value and press OK.



Press OK again to apply environment variable changes and close window.



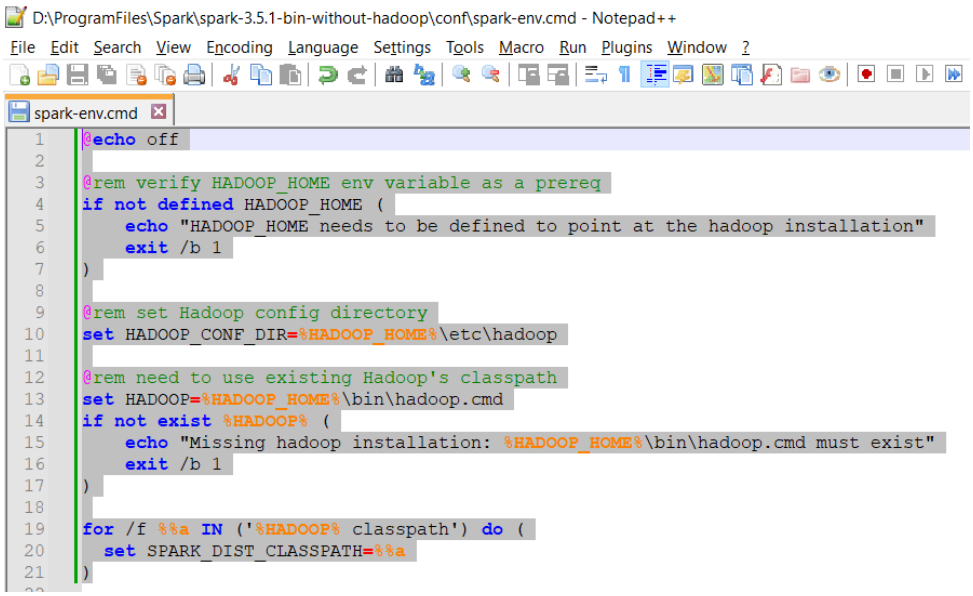
4.4. Configure Spark Environment:

To use Spark's "Hadoop Free" build i.e to use Spark on top of existing Hadoop cluster, we need to set `SPARK_DIST_CLASSPATH` environment variable in `SPARK_HOME\conf\spark-env.cmd` file.

Go to `SPARK_HOME\conf` location, create `spark-env.cmd` file and add the following lines

```
@echo off

@rem verify HADOOP_HOME env variable as a prereq
if not defined HADOOP_HOME (
        echo "HADOOP_HOME needs to be defined to point at the hadoop
installation"
        exit /b 1
)

@rem set Hadoop config directory
set HADOOP_CONF_DIR=%HADOOP_HOME%\etc\hadoop

@rem need to use existing Hadoop's classpath
set HADOOP=%HADOOP_HOME%\bin\hadoop.cmd
if not exist %HADOOP% (
```

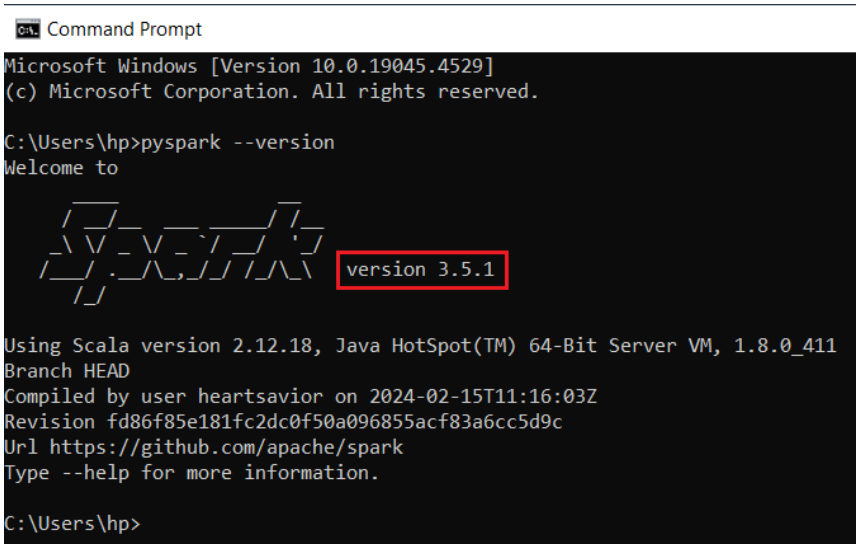
```
        echo "Missing hadoop installation: %HADOOP_HOME%\bin\hadoop.cmd must
exist"
        exit /b 1
)

for /f %%a IN ('%HADOOP% classpath') do (
  set SPARK_DIST_CLASSPATH=%%a
)
```



4.5. Verify Spark Installation:

Open Windows Command prompt or Windows PowerShell prompt and run the following command:

```
pyspark --version
```



After the above command is executed, you can see the installed **Spark version 3.5.1**.

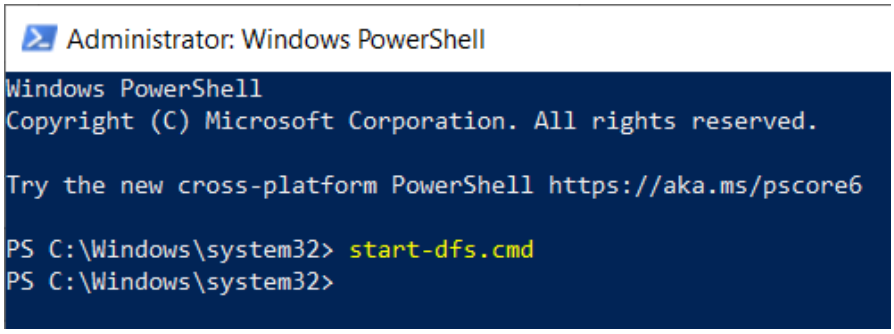
4.6. Start Hadoop Services:

Open **Windows PowerShell** or **Command Prompt** as **Administrator** and start services.

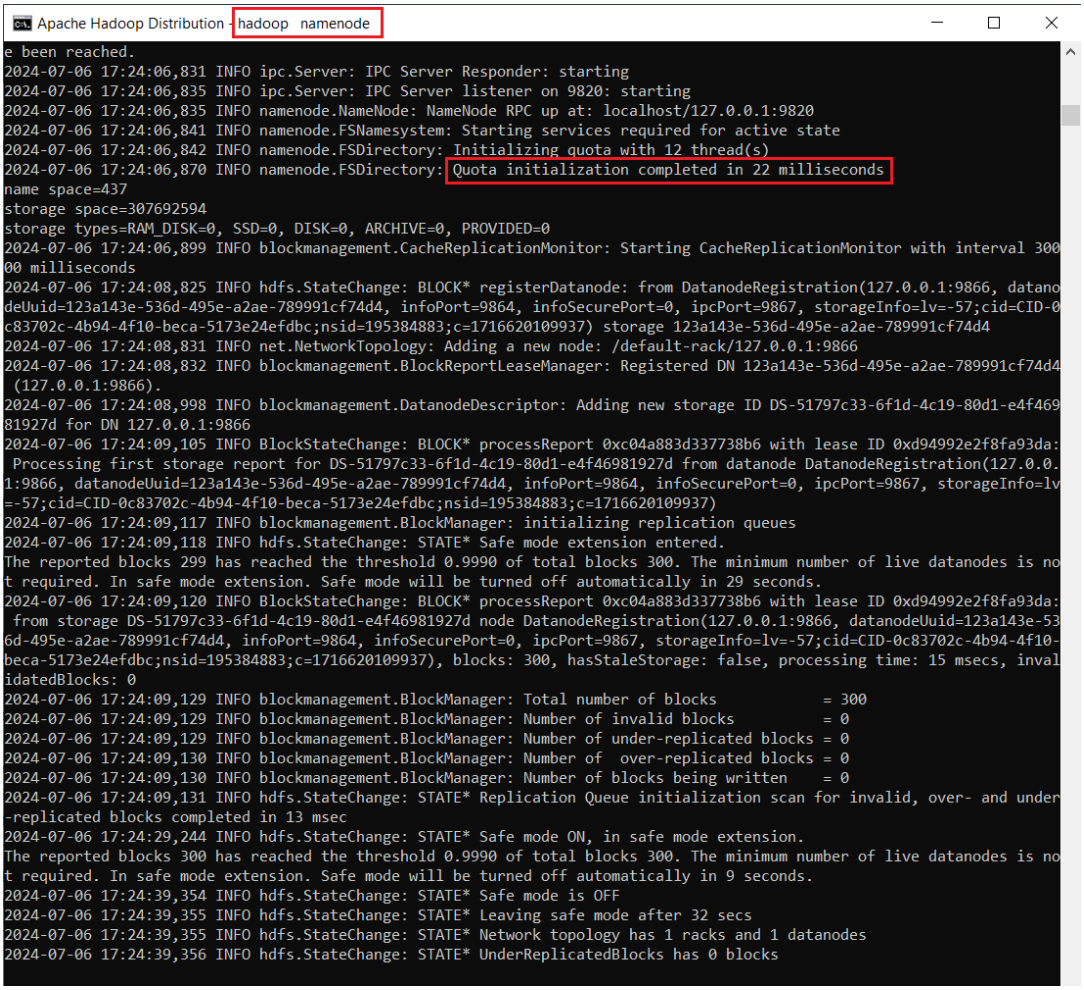
4.6.1. Start Hadoop Nodes:

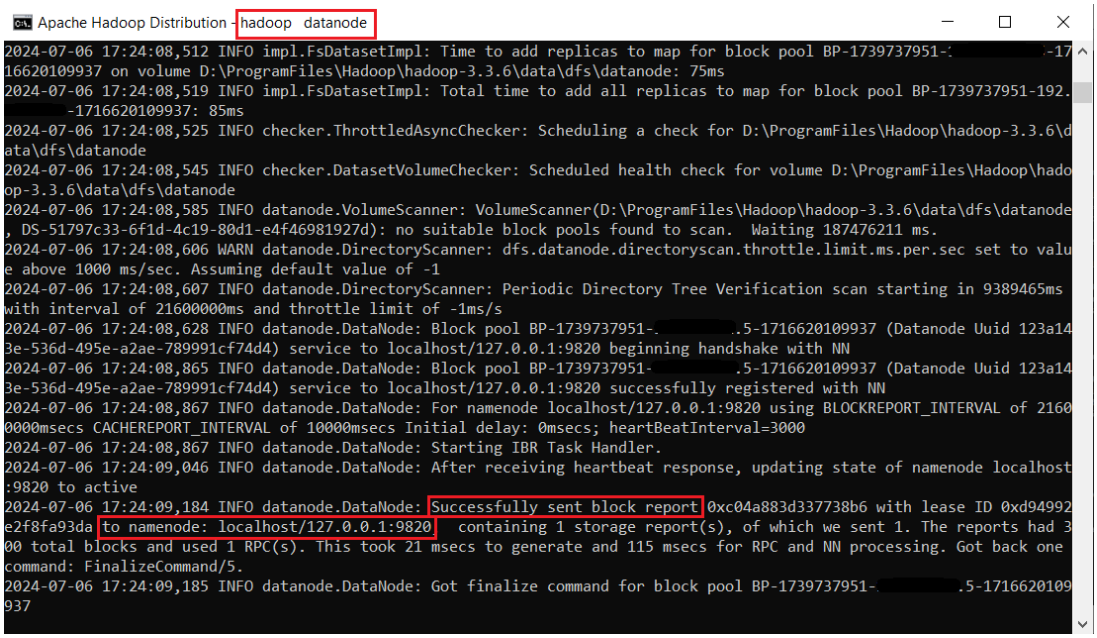
Run the following command to start the Hadoop nodes.

```
start-dfs.cmd
```



After executing the above command, it opens up two command prompt windows - one for the **namenode** and other for the **datanode** as below. Wait until `namenode` service says *"Quota initialization completed"* and `datanode` service says *"Successfully sent block report to namenode: localhost/127.0.0.1:9820"*.

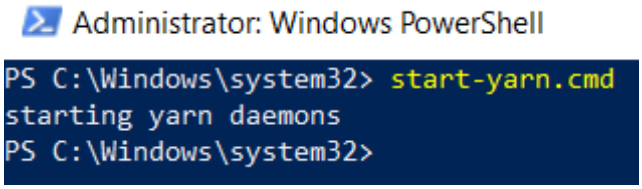

Apache Hadoop Distribution - hadoop namenode
e been reached.
2024-07-06 17:24:06,831 INFO ipc.Server: IPC Server Responder: starting
2024-07-06 17:24:06,835 INFO ipc.Server: IPC Server listener on 9820: starting
2024-07-06 17:24:06,835 INFO namenode.NameNode: NameNode RPC up at: localhost/127.0.0.1:9820
2024-07-06 17:24:06,841 INFO namenode.FSNamesystem: Starting services required for active state
2024-07-06 17:24:06,842 INFO namenode.FSDirectory: Initializing quota with 12 thread(s)
2024-07-06 17:24:06,870 INFO namenode.FSDirectory: Quota initialization completed in 22 milliseconds
name space=437
storage space=307692594
storage types=RAM_DISK=0, SSD=0, DISK=0, ARCHIVE=0, PROVIDED=0
2024-07-06 17:24:06,899 INFO blockmanagement.CacheReplicationMonitor: Starting CacheReplicationMonitor with interval 300
00 milliseconds
2024-07-06 17:24:08,825 INFO hdfs.StateChange: BLOCK* registerDatanode: from DatanodeRegistration(127.0.0.1:9866, datano
deUuid=123a143e-536d-495e-a2ae-789991cf74d4, infoPort=9864, infoSecurePort=0, ipcPort=9867, storageInfo=lv=-57;cid=CID-0
c83702c-4b94-4f10-beca-5173e24efdbc;nsid=195384883;c=1716620109937) storage 123a143e-536d-495e-a2ae-789991cf74d4
2024-07-06 17:24:08,831 INFO net.NetworkTopology: Adding a new node: /default-rack/127.0.0.1:9866
2024-07-06 17:24:08,832 INFO blockmanagement.BlockReportLeaseManager: Registered DN 123a143e-536d-495e-a2ae-789991cf74d4
(127.0.0.1:9866).
2024-07-06 17:24:08,998 INFO blockmanagement.DatanodeDescriptor: Adding new storage ID DS-51797c33-6f1d-4c19-80d1-e4f469
81927d for DN 127.0.0.1:9866
2024-07-06 17:24:09,105 INFO BlockStateChange: BLOCK* processReport 0xc04a883d337738b6 with lease ID 0xd94992e2f8fa93da:
Processing first storage report for DS-51797c33-6f1d-4c19-80d1-e4f46981927d from datanode DatanodeRegistration(127.0.0.
1:9866, datanodeUuid=123a143e-536d-495e-a2ae-789991cf74d4, infoPort=9864, infoSecurePort=0, ipcPort=9867, storageInfo=lv
=-57;cid=CID-0c83702c-4b94-4f10-beca-5173e24efdbc;nsid=195384883;c=1716620109937)
2024-07-06 17:24:09,117 INFO blockmanagement.BlockManager: initializing replication queues
2024-07-06 17:24:09,118 INFO hdfs.StateChange: STATE* Safe mode extension entered.
The reported blocks 299 has reached the threshold 0.9990 of total blocks 300. The minimum number of live datanodes is no
t required. In safe mode extension. Safe mode will be turned off automatically in 29 seconds.
2024-07-06 17:24:09,120 INFO BlockStateChange: BLOCK* processReport 0xc04a883d337738b6 with lease ID 0xd94992e2f8fa93da:
from storage DS-51797c33-6f1d-4c19-80d1-e4f46981927d node DatanodeRegistration(127.0.0.1:9866, datanodeUuid=123a143e-53
6d-495e-a2ae-789991cf74d4, infoPort=9864, infoSecurePort=0, ipcPort=9867, storageInfo=lv=-57;cid=CID-0c83702c-4b94-4f10-
beca-5173e24efdbc;nsid=195384883;c=1716620109937), blocks: 300, hasStaleStorage: false, processing time: 15 msecs, inval
idatedBlocks: 0
2024-07-06 17:24:09,129 INFO blockmanagement.BlockManager: Total number of blocks = 300
2024-07-06 17:24:09,129 INFO blockmanagement.BlockManager: Number of invalid blocks = 0
2024-07-06 17:24:09,129 INFO blockmanagement.BlockManager: Number of under-replicated blocks = 0
2024-07-06 17:24:09,130 INFO blockmanagement.BlockManager: Number of over-replicated blocks = 0
2024-07-06 17:24:09,130 INFO blockmanagement.BlockManager: Number of blocks being written = 0
2024-07-06 17:24:09,131 INFO hdfs.StateChange: STATE* Replication Queue initialization scan for invalid, over- and under
-replicated blocks completed in 13 msec
2024-07-06 17:24:29,244 INFO hdfs.StateChange: STATE* Safe mode ON, in safe mode extension.
The reported blocks 300 has reached the threshold 0.9990 of total blocks 300. The minimum number of live datanodes is no
t required. In safe mode extension. Safe mode will be turned off automatically in 9 seconds.
2024-07-06 17:24:39,354 INFO hdfs.StateChange: STATE* Safe mode is OFF
2024-07-06 17:24:39,355 INFO hdfs.StateChange: STATE* Leaving safe mode after 32 secs
2024-07-06 17:24:39,355 INFO hdfs.StateChange: STATE* Network topology has 1 racks and 1 datanodes
2024-07-06 17:24:39,356 INFO hdfs.StateChange: STATE* UnderReplicatedBlocks has 0 blocks


Apache Hadoop Distribution - hadoop datanode
2024-07-06 17:24:08,512 INFO impl.FsDatasetImpl: Time to add replicas to map for block pool BP-1739737951-
16620109937 on volume D:\ProgramFiles\Hadoop\hadoop-3.3.6\data\dfs\datanode: 75ms
2024-07-06 17:24:08,519 INFO impl.FsDatasetImpl: Total time to add all replicas to map for block pool BP-1739737951-192.
-1716620109937: 85ms
2024-07-06 17:24:08,525 INFO checker.ThrottledAsyncChecker: Scheduling a check for D:\ProgramFiles\Hadoop\hadoop-3.3.6\d
ata\dfs\datanode
2024-07-06 17:24:08,545 INFO checker.DatasetVolumeChecker: Scheduled health check for volume D:\ProgramFiles\Hadoop\hado
op-3.3.6\data\dfs\datanode
2024-07-06 17:24:08,585 INFO datanode.VolumeScanner: VolumeScanner(D:\ProgramFiles\Hadoop\hadoop-3.3.6\data\dfs\datanode
, DS-51797c33-6f1d-4c19-80d1-e4f46981927d): no suitable block pools found to scan. Waiting 187476211 ms.
2024-07-06 17:24:08,606 WARN datanode.DirectoryScanner: dfs.datanode.directoryscan.throttle.limit.ms.per.sec set to valu
e above 1000 ms/sec. Assuming default value of -1
2024-07-06 17:24:08,607 INFO datanode.DirectoryScanner: Periodic Directory Tree Verification scan starting in 9389465ms
with interval of 21600000ms and throttle limit of -1ms/s
2024-07-06 17:24:08,628 INFO datanode.DataNode: Block pool BP-1739737951-.5-1716620109937 (Datanode Uuid 123a14
3e-536d-495e-a2ae-789991cf74d4) service to localhost/127.0.0.1:9820 beginning handshake with NN
2024-07-06 17:24:08,865 INFO datanode.DataNode: Block pool BP-1739737951-.5-1716620109937 (Datanode Uuid 123a14
3e-536d-495e-a2ae-789991cf74d4) service to localhost/127.0.0.1:9820 successfully registered with NN
2024-07-06 17:24:08,867 INFO datanode.DataNode: For namenode localhost/127.0.0.1:9820 using BLOCKREPORT_INTERVAL of 2160
0000msecs CACHEREPORT_INTERVAL of 10000msecs Initial delay: 0msecs; heartBeatInterval=3000
2024-07-06 17:24:08,867 INFO datanode.DataNode: Starting IBR Task Handler.
2024-07-06 17:24:09,046 INFO datanode.DataNode: After receiving heartbeat response, updating state of namenode localhost
:9820 to active
2024-07-06 17:24:09,184 INFO datanode.DataNode: Successfully sent block report 0xc04a883d337738b6 with lease ID 0xd94992
e2f8fa93da to namenode: localhost/127.0.0.1:9820 containing 1 storage report(s), of which we sent 1. The reports had 3
00 total blocks and used 1 RPC(s). This took 21 msecs to generate and 115 msecs for RPC and NN processing. Got back one
command: FinalizeCommand/5.
2024-07-06 17:24:09,185 INFO datanode.DataNode: Got finalize command for block pool BP-1739737951-.5-1716620109
937

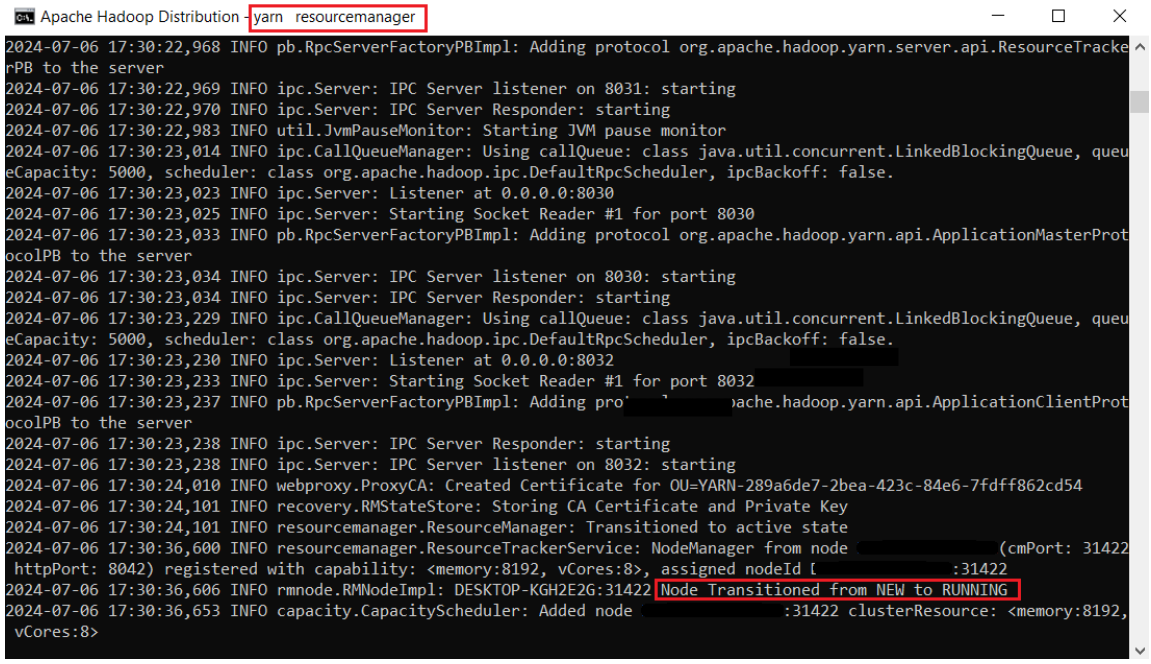
4.6.2. Start Hadoop YARN:

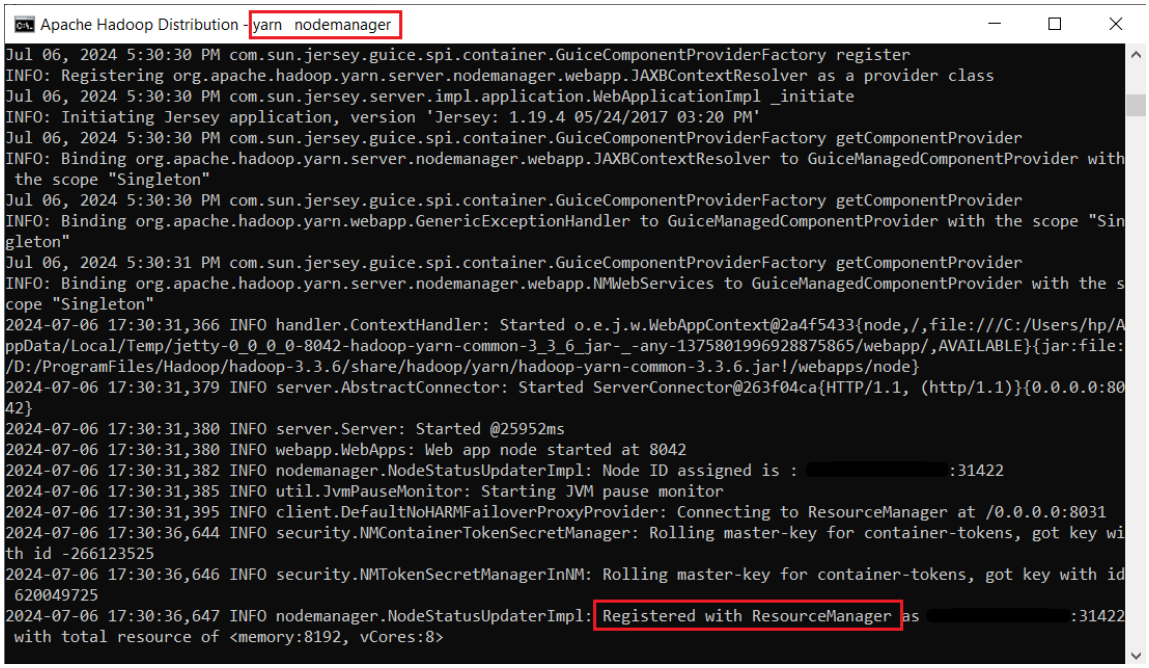
Next, start the Hadoop YARN services using the following command

```
start-yarn.cmd
```



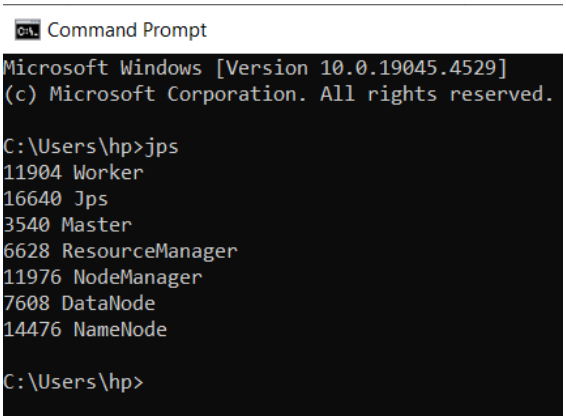
After executing the above command, it opens up two command prompt windows - one for the **resourcemanager** and other for the **nodemanager** as below. Wait until `resourcemanager` service says "*Transitioned to active state*" and `nodemanager` service says "*Registered with ResourceManager*"





After Hadoop services are started, verify if `NameNode`, `DataNode`, `ResourceManager` and `NodeManager` services are running by executing the following command

```
jps
```

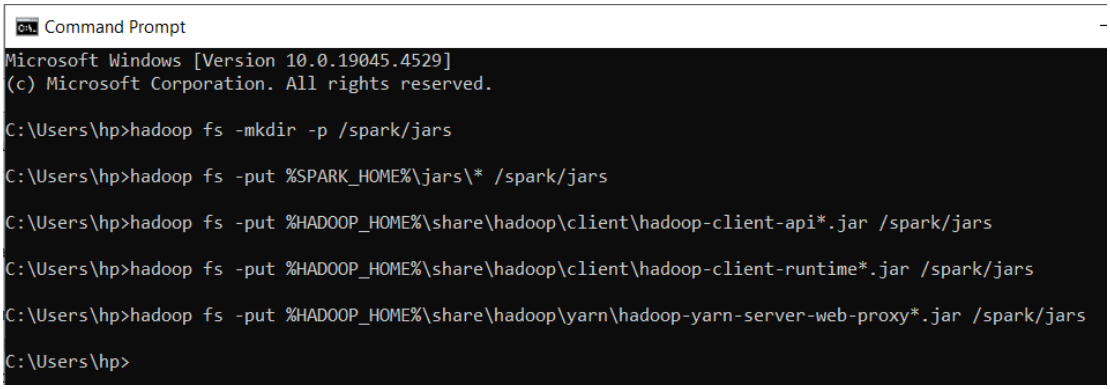


4.7. Set Spark YARN Directory:

To run Spark applications on YARN cluster in Windows, we need to copy Spark jar files into HDFS and set `spark.yarn.jars` property in `spark-defaults.conf` file.

First, run the following commands to create a HDFS directory `/spark/jars` and copy Spark jar files and the necessary Hadoop jar files into HDFS directory.

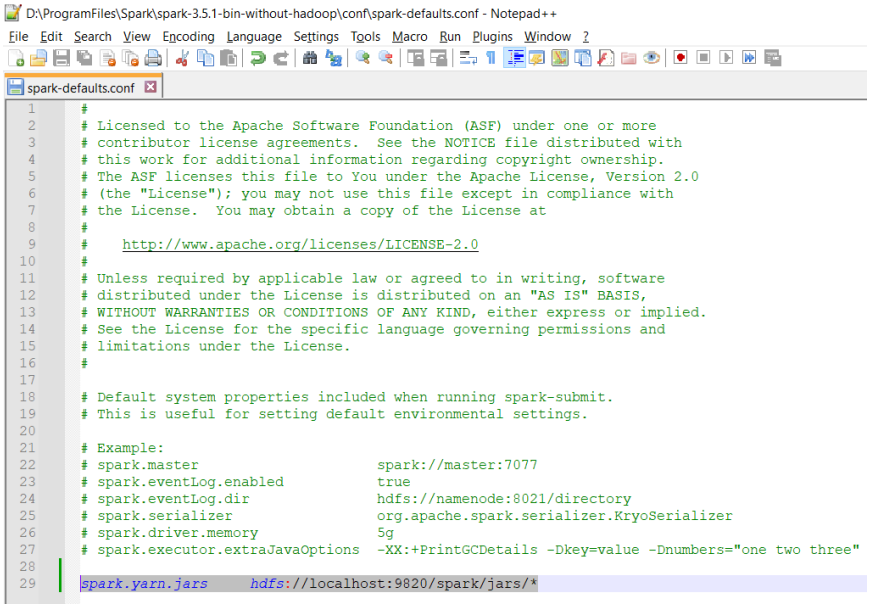
```
hadoop fs -mkdir -p /spark/jars
hadoop fs -put %SPARK_HOME%\jars\* /spark/jars
hadoop fs -put %HADOOP_HOME%\share\hadoop\client\hadoop-client-
api*.jar /spark/jars
hadoop fs -put %HADOOP_HOME%\share\hadoop\client\hadoop-client-
runtime*.jar /spark/jars
hadoop fs -put %HADOOP_HOME%\share\hadoop\yarn\hadoop-yarn-server-
web-proxy*.jar /spark/jars
```



Go to `SPARK_HOME\conf` directory, take the copy of `spark-defaults.conf.template` and rename it as `spark-defaults.conf`.

Add the following line in `spark-defaults.conf` file:

```
spark.yarn.jars         hdfs://localhost:9820/spark/jars/*
```



4.8. Spark Examples:

Spark provides some in-built example programs such as `pi`, `wordcount`, `sort`, etc. available under `SPARK_HOME\examples` directory that can be execute on Spark cluster.

4.8.1. Run Java SparkPI Program:

Let us execute the example Java program `SparkPi` packaged under `spark-examples*.jar` file that is located in `SPARK_HOME\examples\jars` directory.

In Local Mode:

Open a new **Command Prompt** and run the below command which executes `SparkPi` application in local mode with all available cores:

```
spark-submit --class org.apache.spark.examples.SparkPi
%SPARK_HOME%\examples\jars\spark-examples*.jar
```

or

```
spark-submit --master local[*] --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```

It displayed the output of approximate Pi value which is **3.140**.

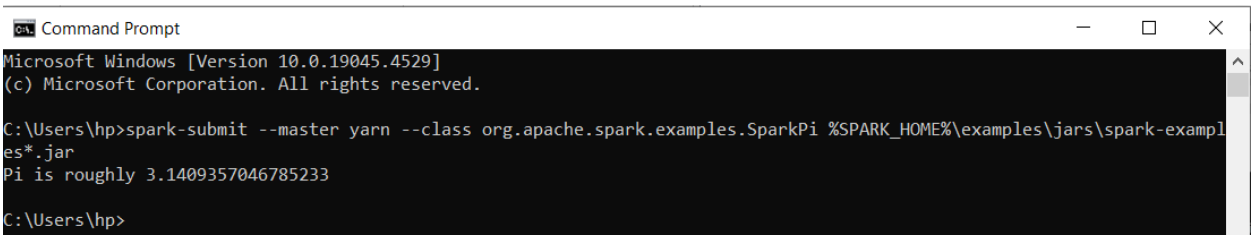
In YARN Client Mode:

Use the following command to run the `SparkPi` application in client mode on YARN cluster:

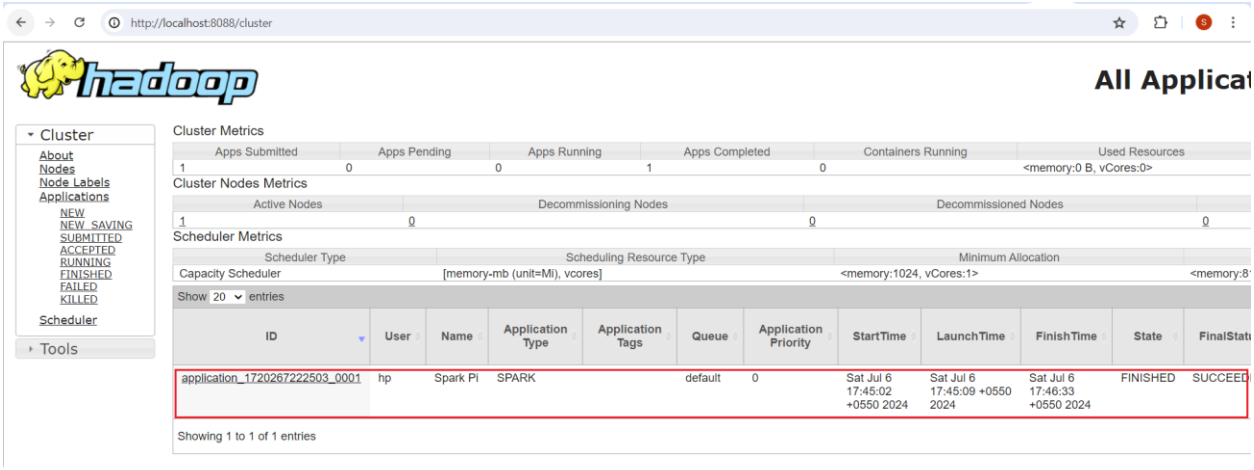
```
spark-submit --master yarn --deploy-mode client --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```

or

```
spark-submit --master yarn --class org.apache.spark.examples.SparkPi
%SPARK_HOME%\examples\jars\spark-examples*.jar
```



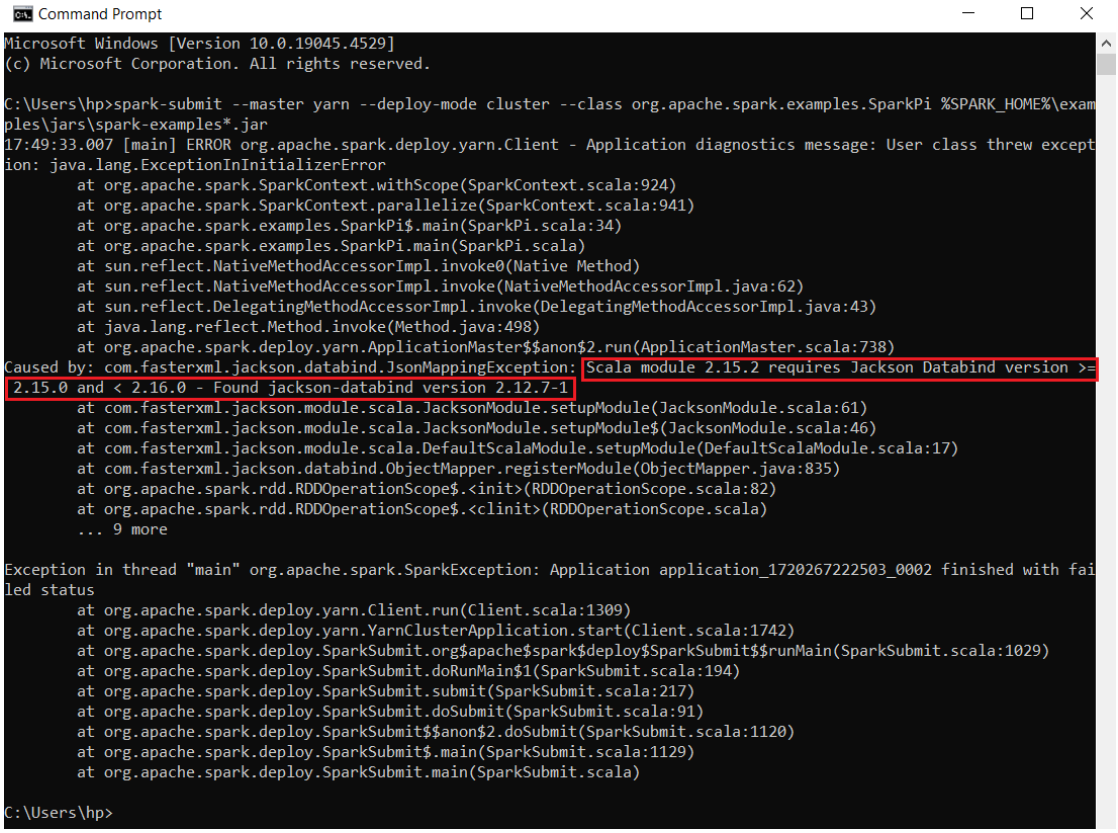
On the YARN UI Web UI (http://localhost:8088/cluster), we can see that **Spark Pi** application has been submitted and completed.



In YARN Cluster Mode:

To run the `SparkPi` application in cluster mode on YARN, use the below command:

```
spark-submit --master yarn --deploy-mode cluster --class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```

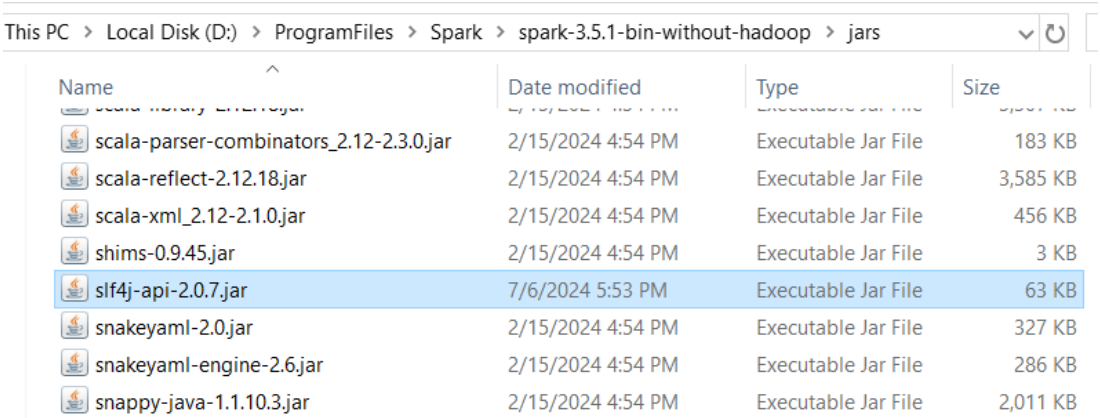


Here, we can see an error *"Scala module 2.15.2 requires Jackson Databind version >= 2.15.0 and < 2.16.0 - Found jackson-databind version 2.12.7-1"* due to the conflict between Hadoop JARs and Spark JARs. Note that this error occurs only in Windows operating system.

Follow the below steps to resolve the above error:

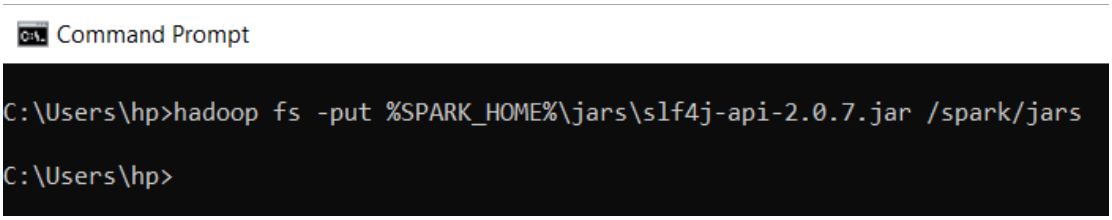
- Download the latest `slf4j-api-2.0.7.jar` file from https://mvnrepository.com/artifact/org.slf4j/slf4j-api/2.0.7

- Place `slf4j-api-2.0.7.jar` file in `%SPARK_HOME%\jars` directory.



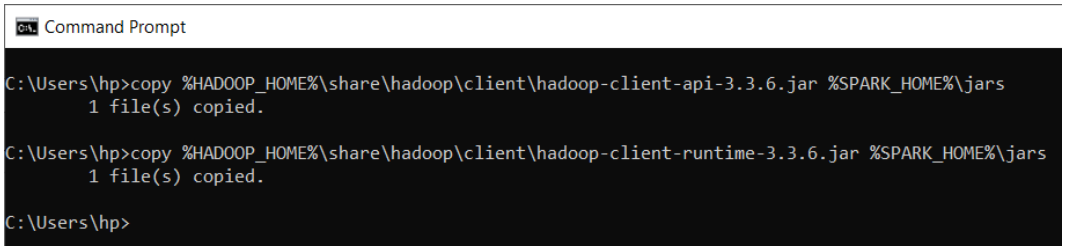
- Copy `slf4j-api-2.0.7.jar` file from `%SPARK_HOME%\jars` to HDFS location where Spark jars are available using the following command:

```
hadoop fs -put %SPARK_HOME%\jars\slf4j-api-2.0.7.jar /spark/jars
```



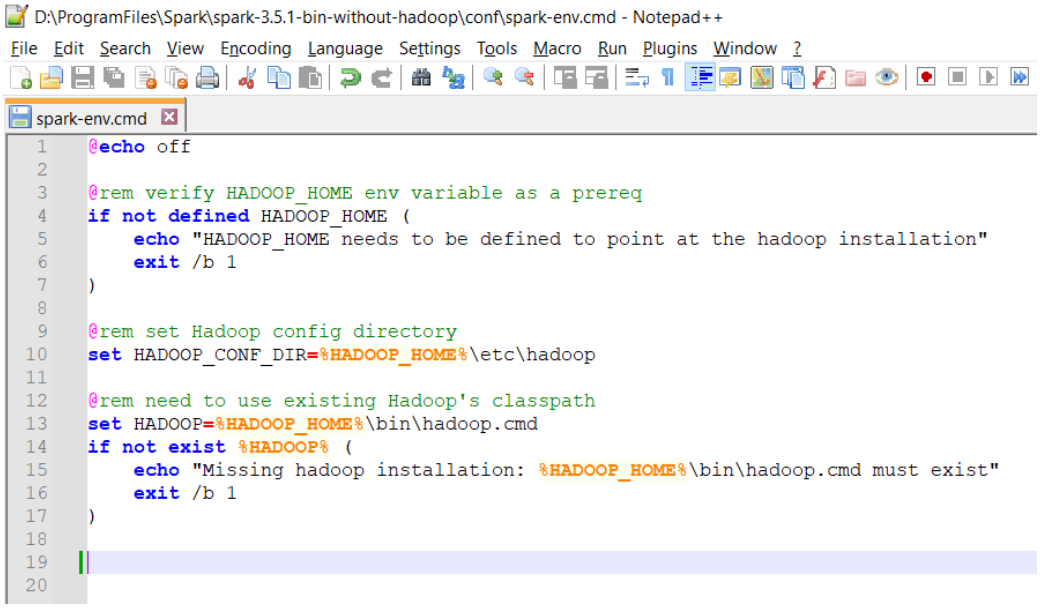
- Copy `hadoop-client-api-*.jar` and `hadoop-client-runtime-*.jar` files from `%HADOOP_HOME%\share\hadoop\client` directory to `%SPARK_HOME%\jars` directory in your local system:

```
copy %HADOOP_HOME%\share\hadoop\client\hadoop-client-api-*.jar
%SPARK_HOME%\jars
copy %HADOOP_HOME%\share\hadoop\client\hadoop-client-runtime-
*.jar %SPARK_HOME%\jars
```



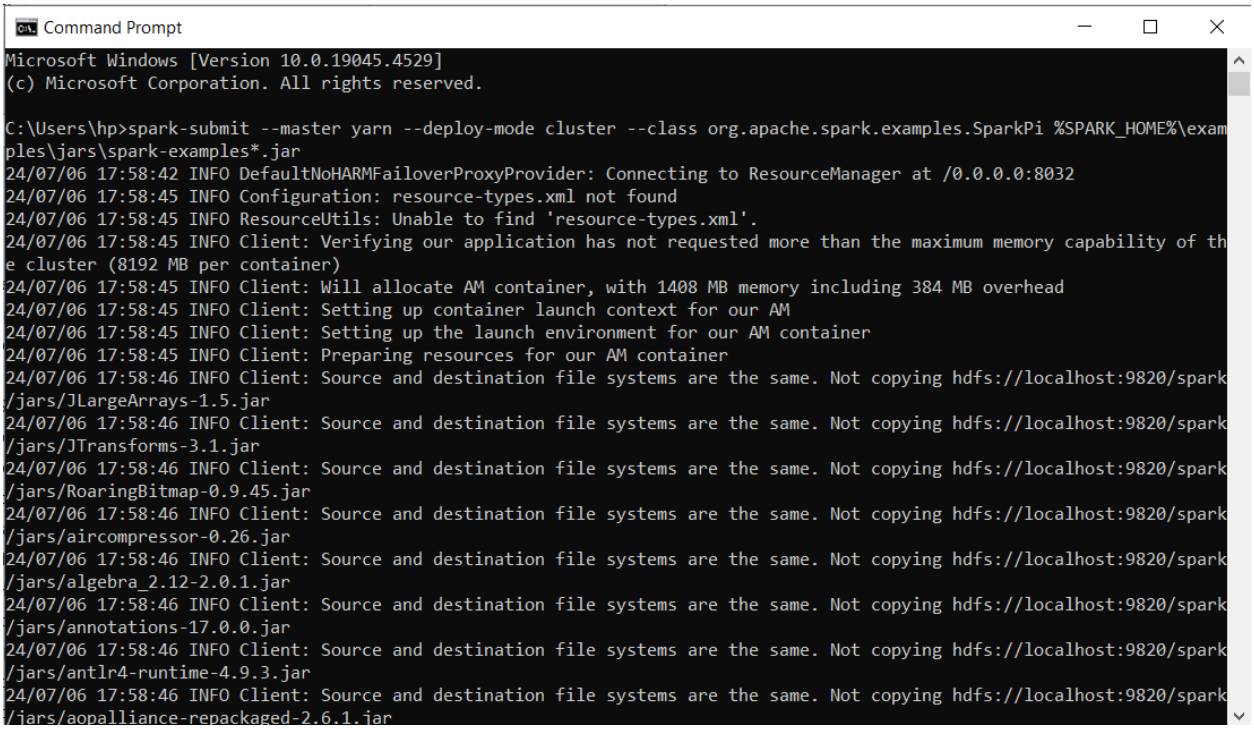
- Remove the following lines in `spark-env.cmd` file.

```
for /f %%a IN ('%HADOOP% classpath') do (
  set SPARK_DIST_CLASSPATH=%%a
)
```



Now, run the `SparkPi` application in cluster mode on YARN using the below command:

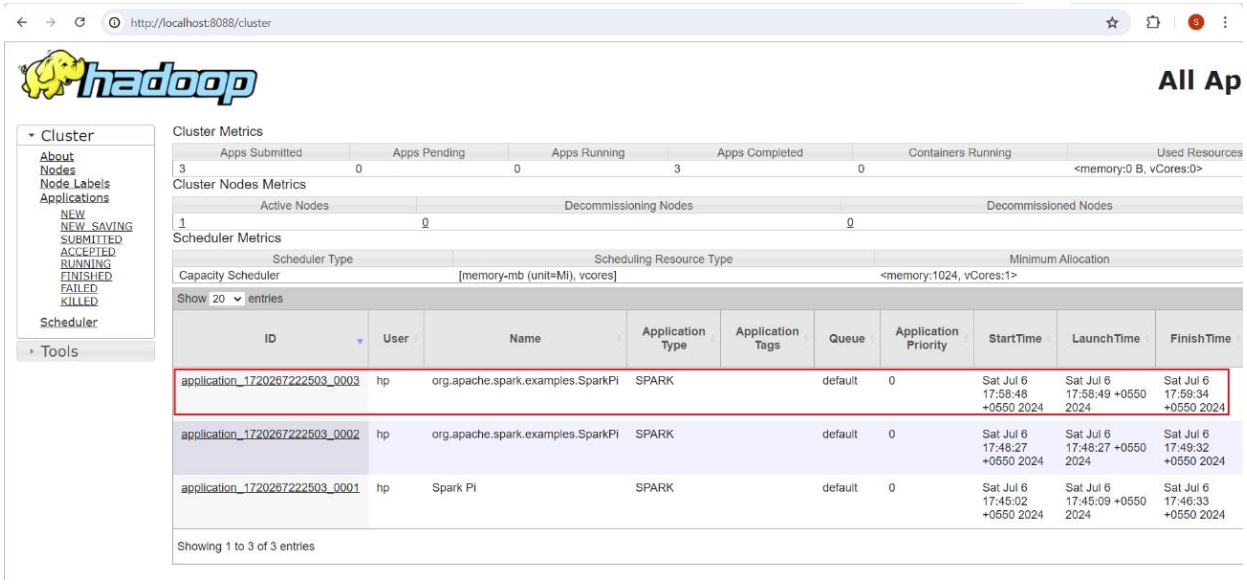
```
spark-submit --master yarn –-deploy-mode cluster –-class
org.apache.spark.examples.SparkPi %SPARK_HOME%\examples\jars\spark-
examples*.jar
```



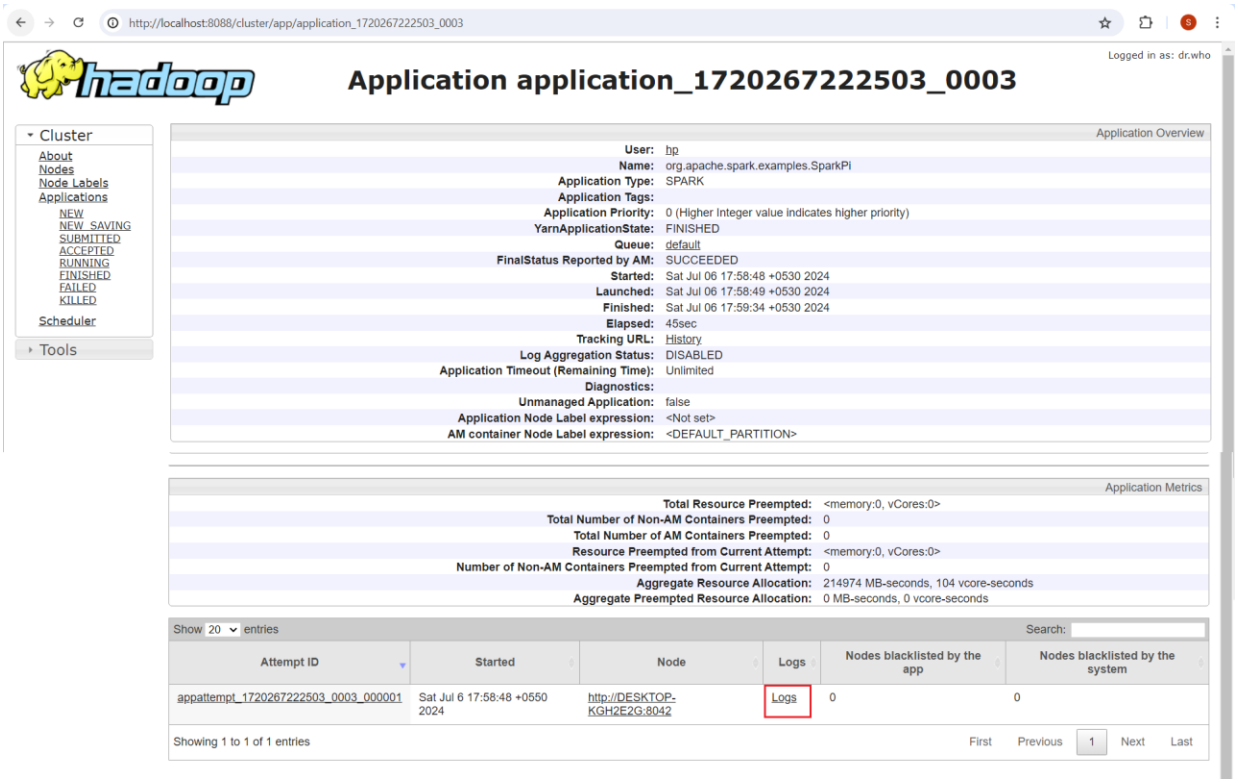
```
24/07/06 17:59:03 INFO Client: Application report for application_1720267222503_0003 (state: RUNNING)
24/07/06 17:59:03 INFO Client:
        client token: N/A
        diagnostics: N/A
        ApplicationMaster host: DESKTOP-KGH2E2G
        ApplicationMaster RPC port: 26347
        queue: default
        start time: 1720268928686
        final status: UNDEFINED
        tracking URL: http://DESKTOP-KGH2E2G:8088/proxy/application_1720267222503_0003/
        user: hp
24/07/06 17:59:33 INFO Client: Application report for application_1720267222503_0003 (state: RUNNING)
24/07/06 17:59:34 INFO Client: Application report for application_1720267222503_0003 (state: FINISHED)
24/07/06 17:59:34 INFO Client:
        client token: N/A
        diagnostics: N/A
        ApplicationMaster host: DESKTOP-KGH2E2G
        ApplicationMaster RPC port: 26347
        queue: default
        start time: 1720268928686
        final status: SUCCEEDED
        tracking URL: http://DESKTOP-KGH2E2G:8088/proxy/application_1720267222503_0003/
        user: hp
24/07/06 17:59:34 INFO ShutdownHookManager: Shutdown hook called
24/07/06 17:59:34 INFO ShutdownHookManager: Deleting directory C:\Users\hp\AppData\Local\Temp\spark-6ed7031b-4749-4ee7-b
8f2-a1278d7923ce
24/07/06 17:59:34 INFO ShutdownHookManager: Deleting directory C:\Users\hp\AppData\Local\Temp\spark-536cd9ef-ad37-4409-a
589-c949acdbccbb

C:\Users\hp>
```

Here, we can see that application has been submitted to YARN and FINISHED successfully. Since we executed the Spark application in cluster mode on YARN, the output of application can be seen in YARN application itself.

Open YARN UI: http://localhost:8088/cluster and click on the latest **Application ID** value.



On the respective application details, click on **Logs** link which opens container logs.

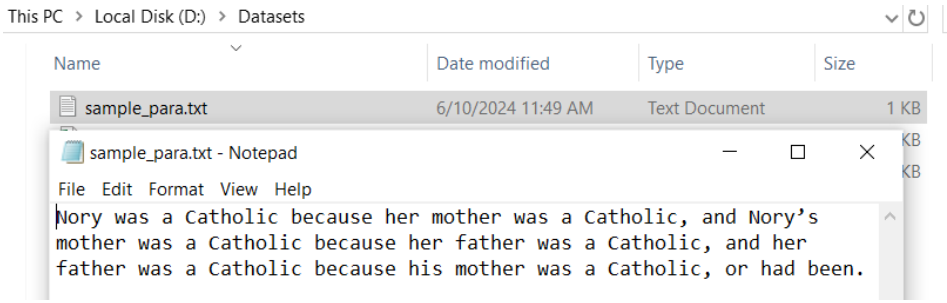


On the container logs, click on **stdout** link to see the output

4.8.2. Run Python wordcount Program:

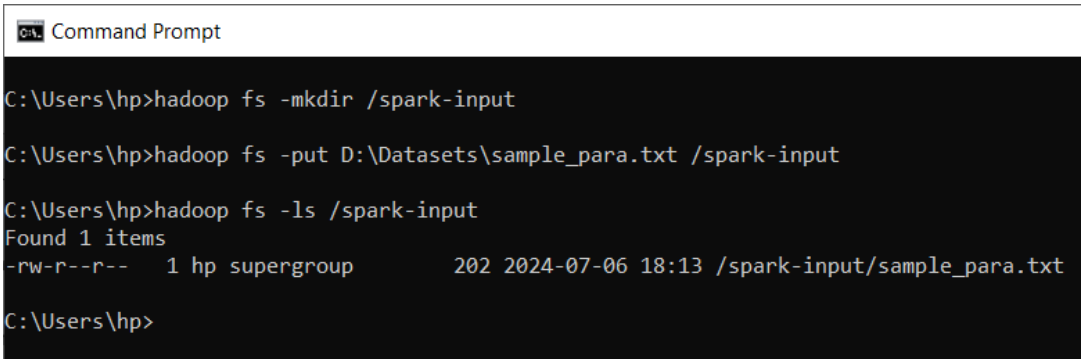
Let us execute the Python program `wordcount` which counts the occurrences of each word in the given input file. This program is located in `SPARK_HOME\examples\src\main\python` directory.

First, create an input file named `sample_para.txt` with some random text and place it in `D:\Datasets` folder.

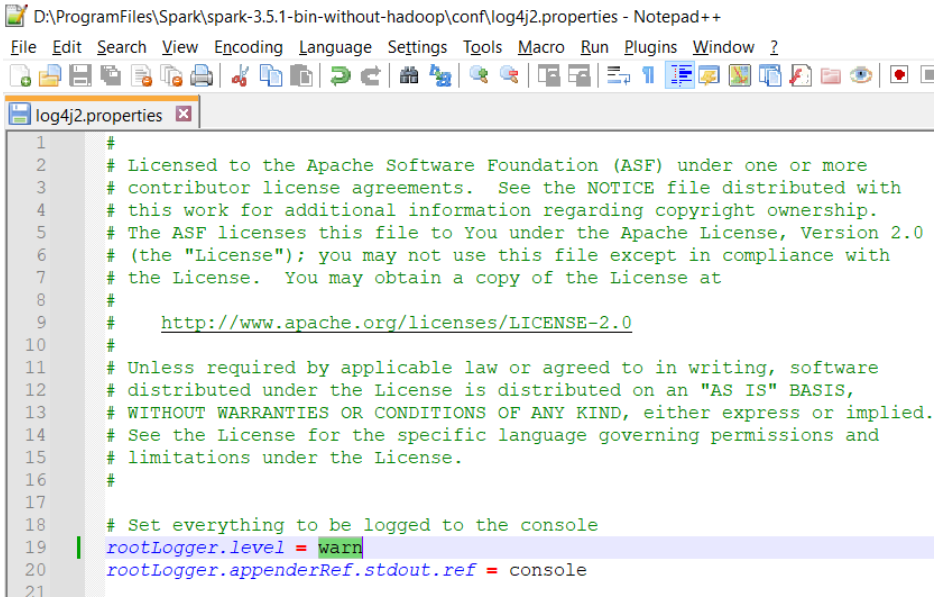


Since our Spark is running on top of Hadoop cluster, we need to move the above file into HDFS using the following commands:

```
hadoop fs -mkdir /spark-input
hadoop fs -put D:\Datasets\sample_para.txt /spark-input
hadoop fs -ls /spark-input
```



Note: To avoid INFO logging on the console while executing the `spark-submit` command, go to `%SPARK_HOME%\conf` location, copy `log4j2.properties.template` file as `log4j2.properties` and change `rootLogger.level` value from `info` to `warn` in `log4j2.properties` file as shown below:



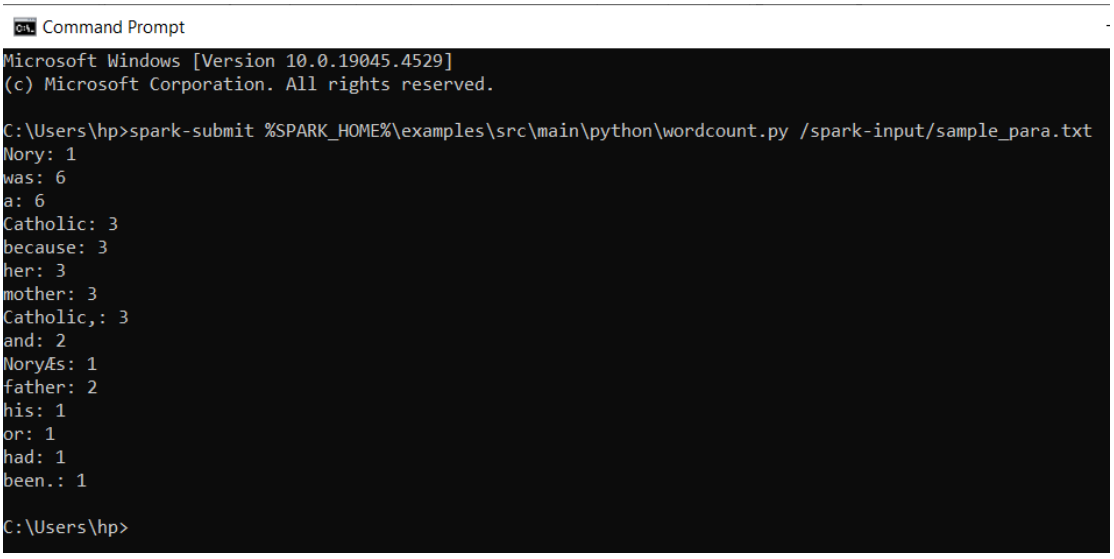
In Local Mode:

Open a new **Command Prompt** and run the below command which executes `wordcount.py` application in local mode with all available cores:

```
spark-submit --master local[*]
%SPARK_HOME%\examples\src\main\python\wordcount.py /spark-
input/sample_para.txt
```

or

```
spark-submit %SPARK_HOME%\examples\src\main\python\wordcount.py
/spark-input/sample_para.txt
```

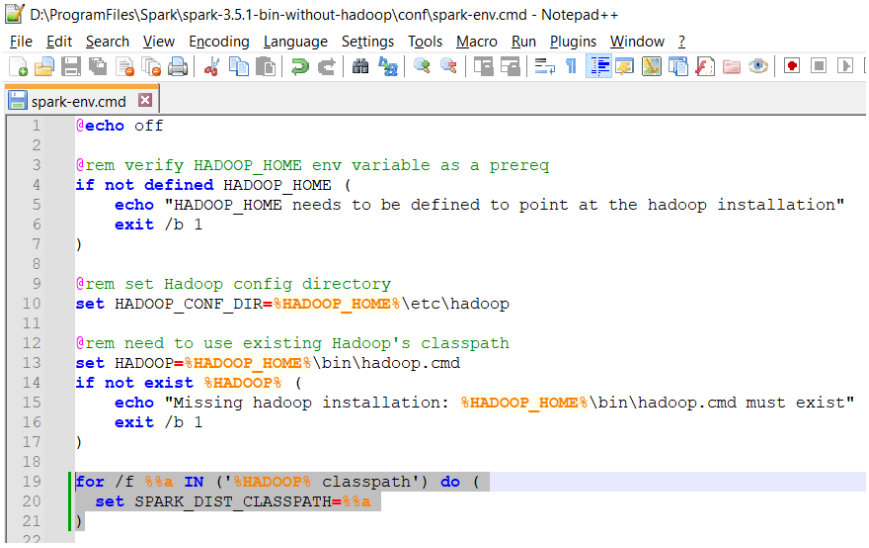


It displayed the count of each word in `sample_para.txt` file.

In YARN Client Mode:

Before executing in "YARN client" mode, make sure to add the following lines in `spark-env.cmd` file *(since we removed earlier to run in "YARN cluster" mode)*

```
for /f %%a IN ('%HADOOP% classpath') do (
  set SPARK_DIST_CLASSPATH=%%a
)
```

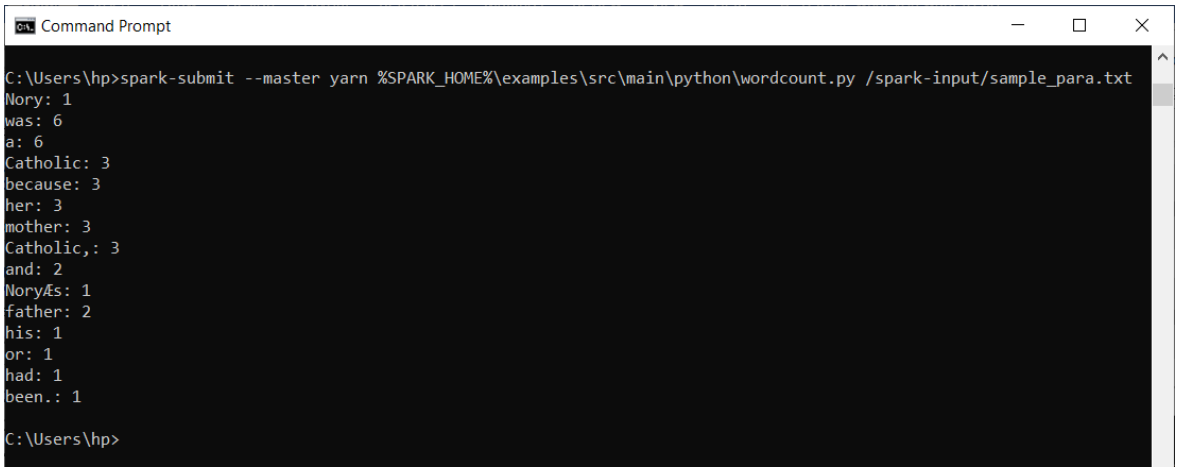


Now run the following command to submit the `wordcount.py` application in client mode on YARN cluster:

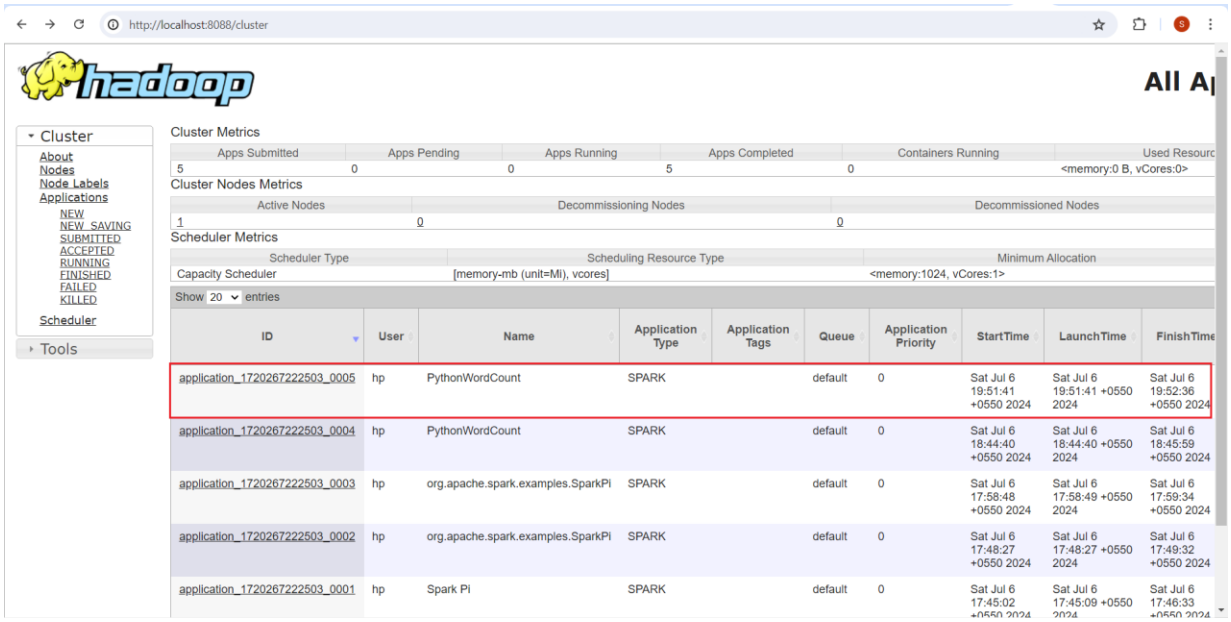
```
spark-submit --master yarn --deploy-mode client
%SPARK_HOME%\examples\src\main\python\wordcount.py /spark-
input/sample_para.txt
```

or

```
spark-submit --master yarn
%SPARK_HOME%\examples\src\main\python\wordcount.py /spark-
input/sample_para.txt
```



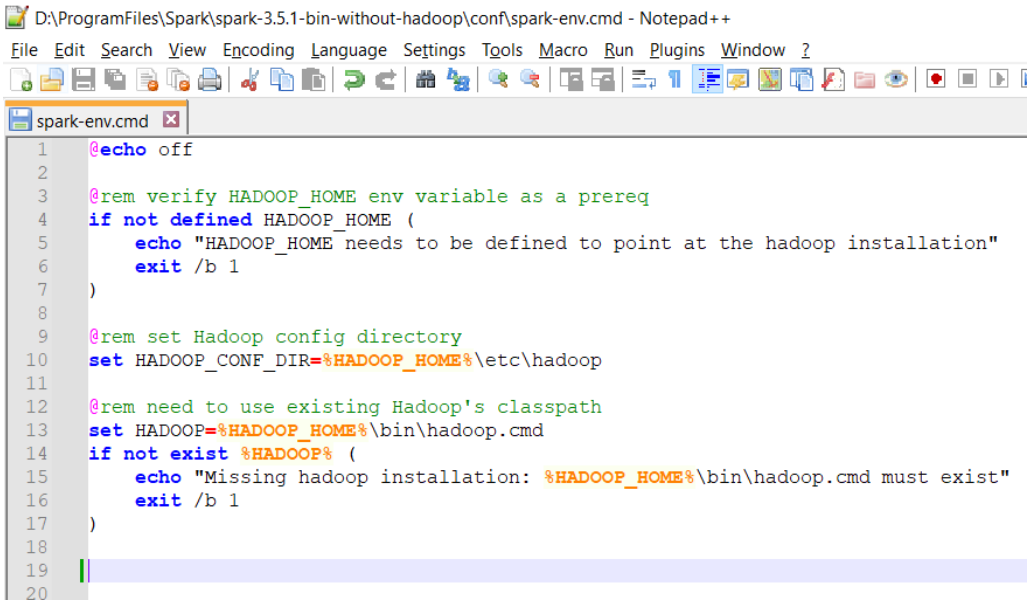
On the YARN UI Web UI (http://localhost:8088/cluster), we can see that **PythonWordCount** application has been submitted and completed.



In YARN Cluster Mode:

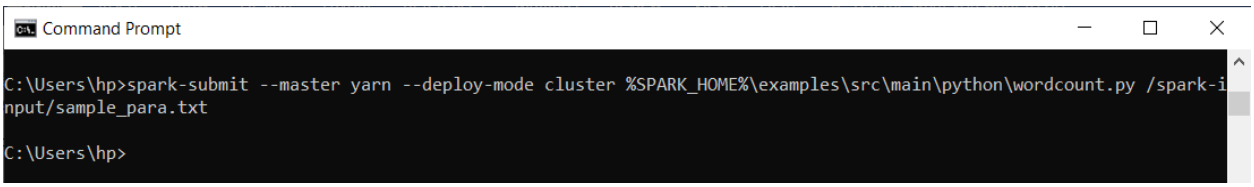
Before executing in "YARN cluster" mode, make sure to remove the following lines in `spark-env.cmd` file.

```
for /f %%a IN ('%HADOOP% classpath') do (
  set SPARK_DIST_CLASSPATH=%%a
)
```



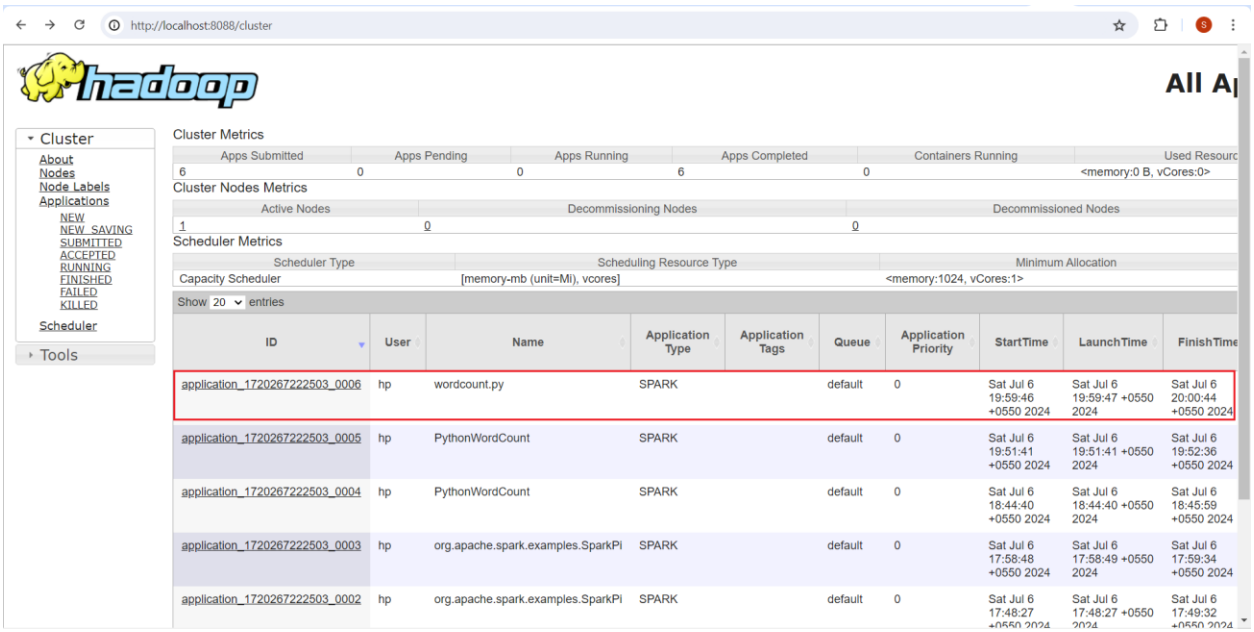
Now, run the below command which executes `wordcount.py` application in cluster mode on YARN:

```
spark-submit --master yarn --deploy-mode cluster
%SPARK_HOME%\examples\src\main\python\wordcount.py /spark-
input/sample_para.txt
```

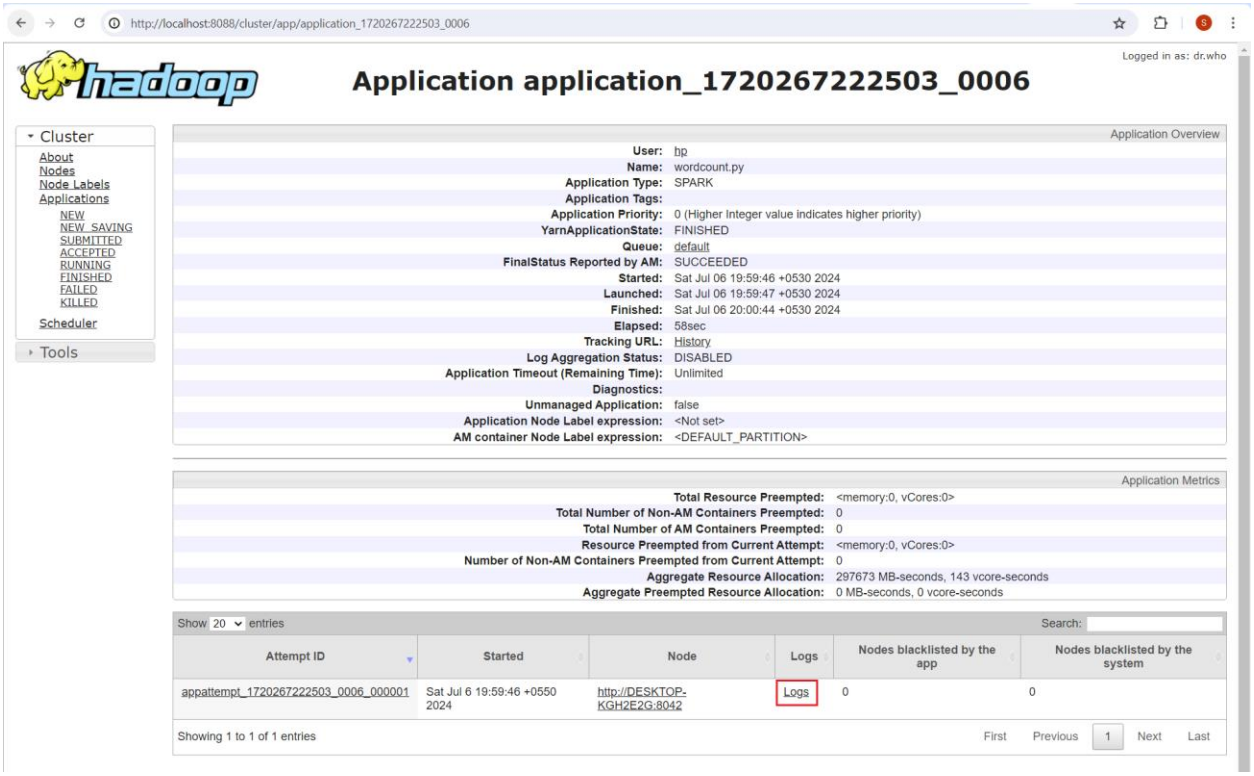


Since we executed the Spark application in cluster mode on YARN, the output of application can be seen in YARN application itself.

Open YARN UI: http://localhost:8088/cluster and click on the latest **Application ID** value.



On the respective application details, click on **Logs** link which opens container logs.



On the container logs, click on **stdout** link to see the output

We can see that it displayed the count of each word available in `sample_para.txt` file.

5. Spark CLIs:

Apache spark provides various command line interfaces such as PySpark Shell, Spark Shell, Spark SQL, Spark R, etc. to execute the spark code in various programming languages.

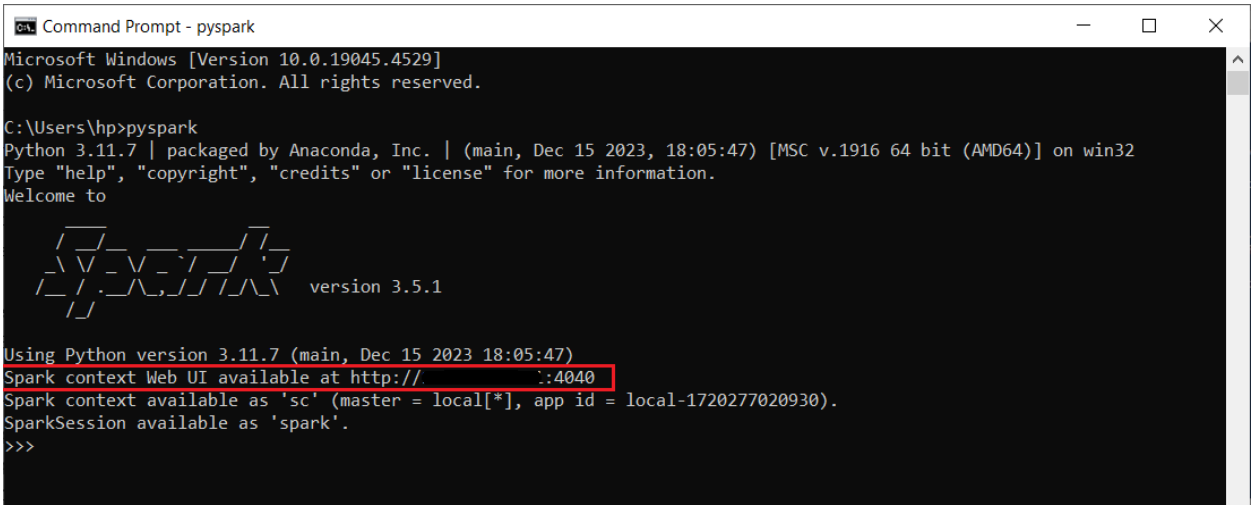
When any spark client such as `pyspark`, `spark-shell`, `spark-sql`, `spark-r`, etc is launched or `spark-submit` command is executed, it creates a **Spark Context Web UI** which runs on **4040** port by default. If the default port 4040 is occupied by other service, Spark tries to create the web interface on the next available port incremented by 1 (i.e. 4041 port). Even if 4041 port is also occupied by any other service, then Spark tries to open Web UI on the next available port incremented by 1.

Note: Spark Web UI is opened only when a spark client is active and will be closed when exited from the spark client.

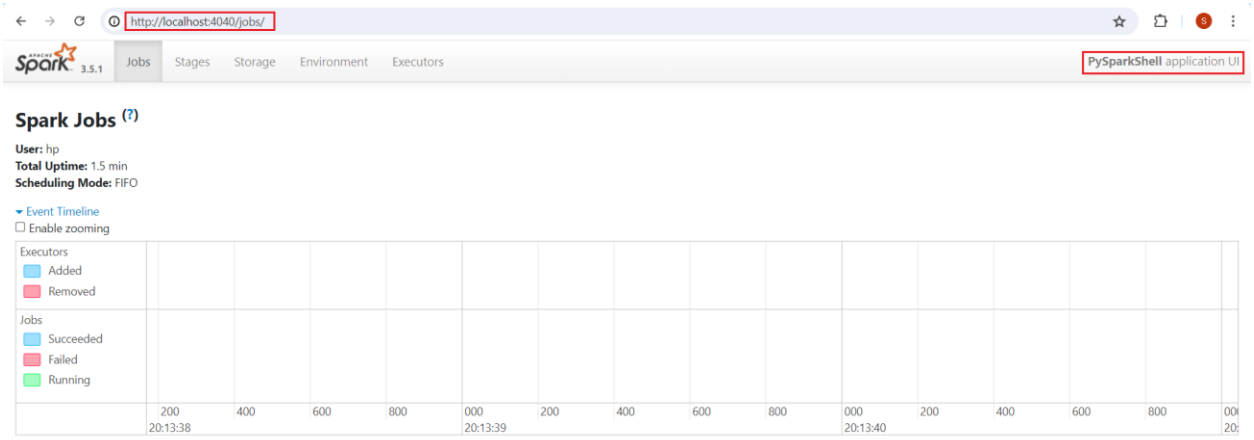
5.1. Pyspark:

Open either **Windows PowerShell** or **Command Prompt** and simply run the following command to start PySpark shell.

```
pyspark
```



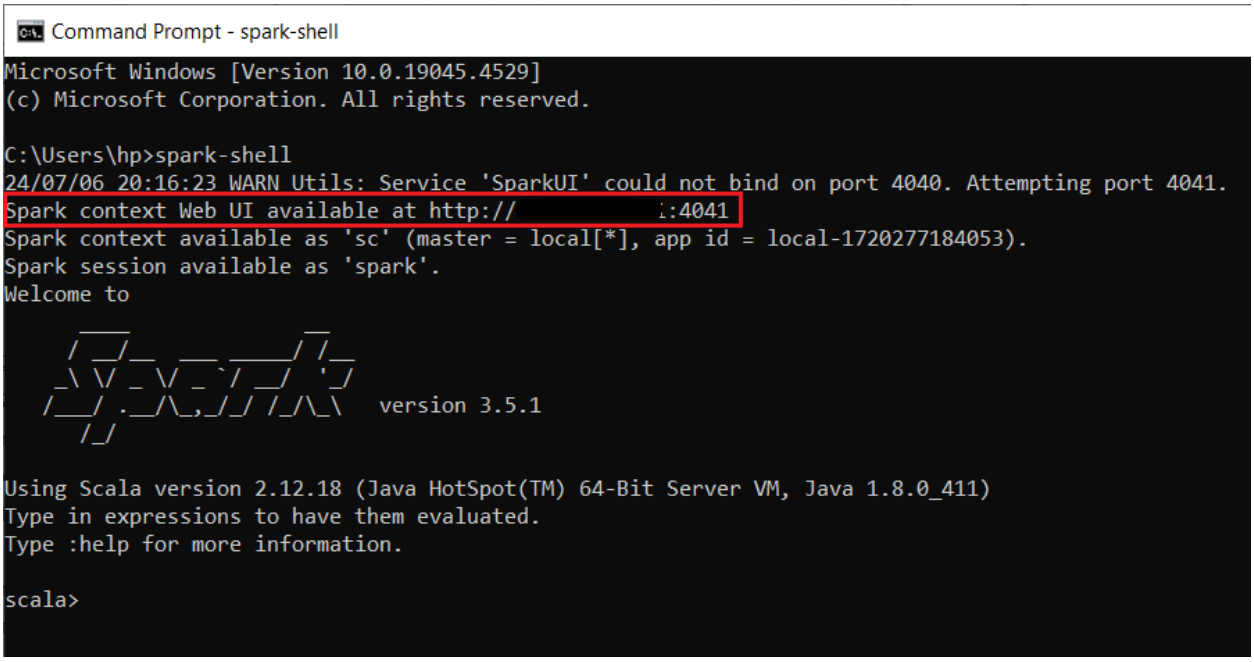
PySparkShell application UI available at http://localhost:4040/ *(I opened* `pyspark` *shell first and is active)*



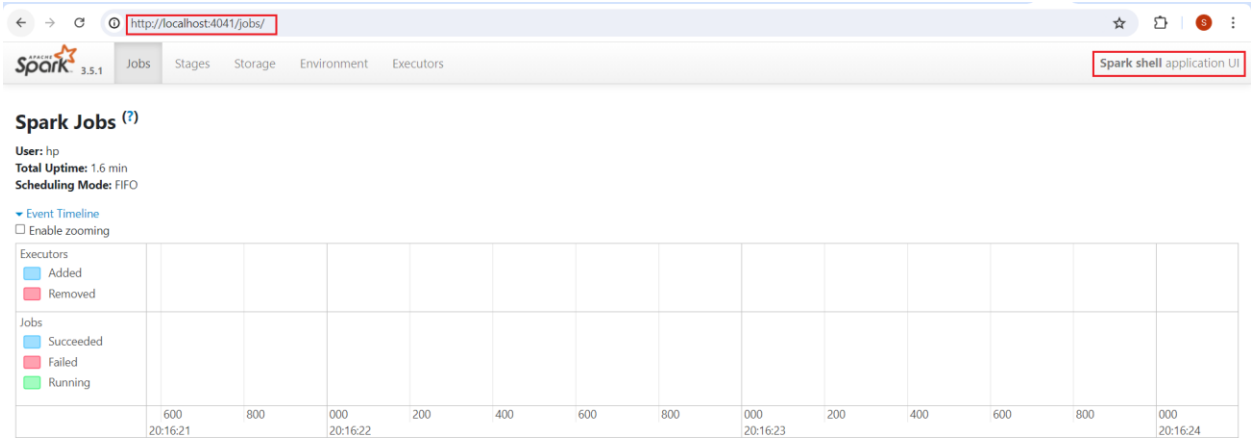
5.2. Spark Shell:

Open another **Windows PowerShell** or **Command Prompt** and run the following command to start Spark shell which opens `scala>` terminal by default.

```
spark-shell
```



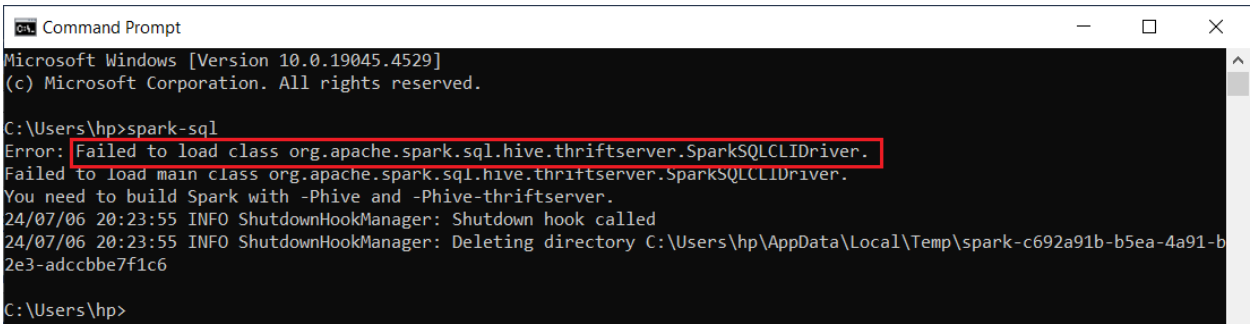
Spark Shell application UI available at http://localhost:4041/ *(I opened `spark-shell` while `pyspark` session is still active)*



5.3. Spark SQL:

Open another **Windows PowerShell** or **Command Prompt** and run the following command to start Spark SQL.

```
spark-sql
```

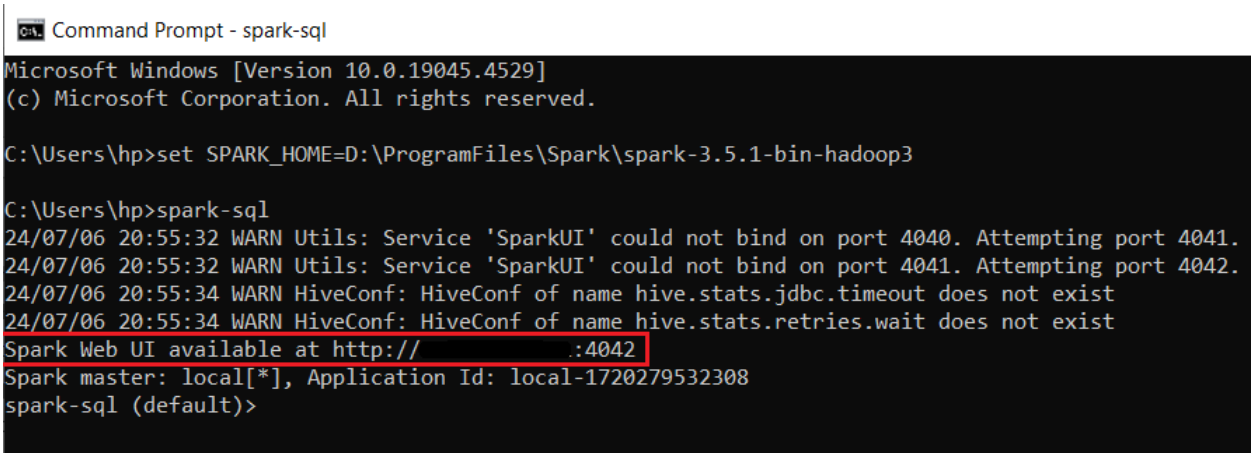


Here, we are encountering *"Error: Failed to load class org.apache.spark.sql.hive.thriftserver.SparkSQLCLIDriver"* which occurs when we are running Spark version without built-in Hadoop libraries which do not include **Spark Hive** and **Spark Hive Thrift Server** and other dependent packages.

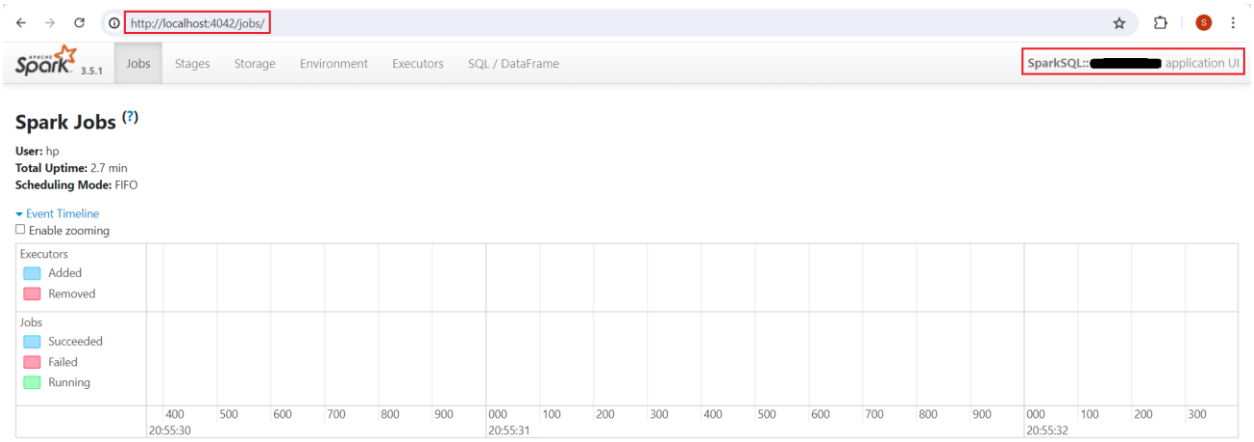
To resolve this issue, we need to either copy all files from `jars` folder in `spark-3.5.1-bin-hadoop3` location to the current `%SPARK_HOME%\jars` location or set the `SPARK_HOME` environment variable to the location where `spark-3.5.1-bin-hadoop3` version was installed.

Here, I am setting `SPARK_HOME` environment variable to the location where my `spark-3.5.1-bin-hadoop3` version was installed and then launching `spark-sql` using below commands:

```
set SPARK_HOME=D:\ProgramFiles\Spark\spark-3.5.1-bin-hadoop3

spark-sql
```



Spark SQL application UI available at http://localhost:4042/ *(I opened `spark-sql` while `pyspark` and `spark-shell` sessions are still active)*



6. Spark Web UI:

Apache Spark provides the following UIs:

1. Master web UI
2. Worker web UI
3. Application web UI

6.1. Master Web UI:

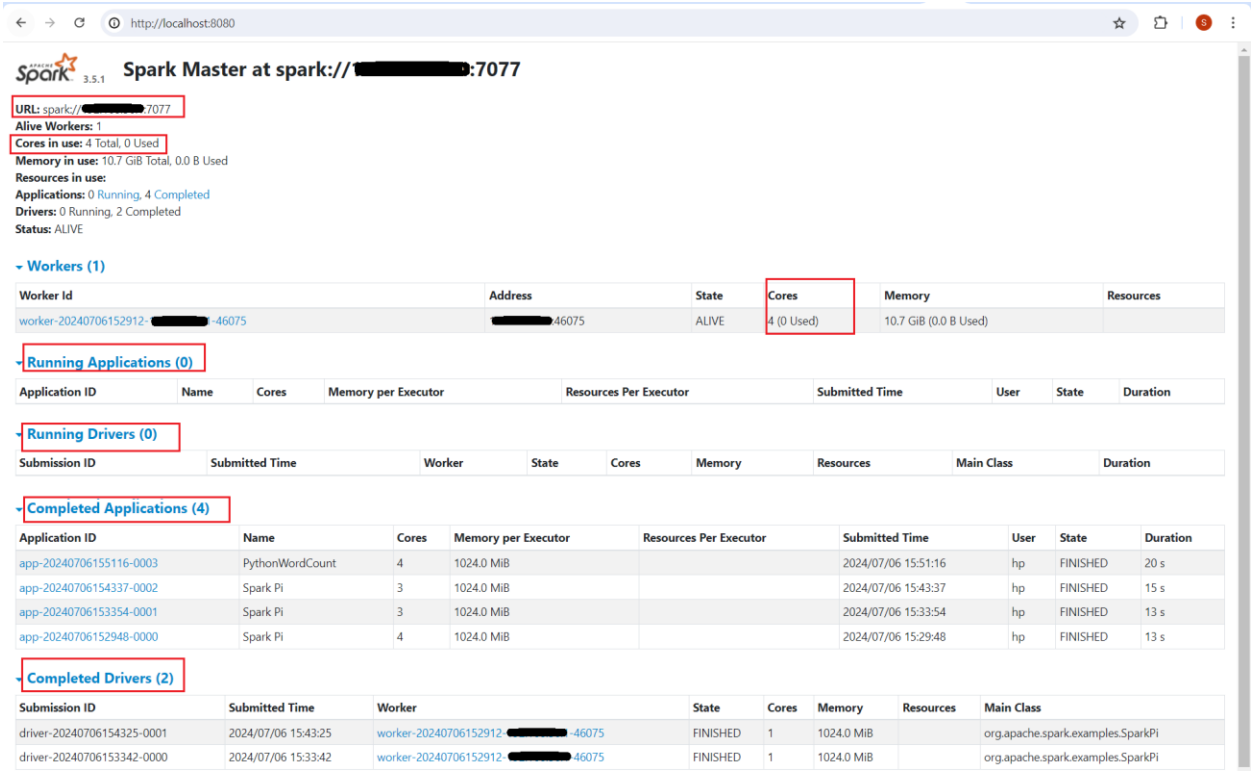
The master web UI is accessible at http://localhost:8080/ by default and will be available when Spark Master is running.

Start the Master service using the below command if not already running.

```
spark-class org.apache.spark.deploy.master.Master
```

The master web UI provides an overview of the Spark cluster and displays the following information:

- Master URL and REST URL.
- CPUs and memory allocated to the Spark cluster and to each application.
- Worker status and its allotted resources
- Information about the active and completed applications, such as their status, allotted resources, and duration.
- Information about the active and completed drivers, such as their status and allotted resources.



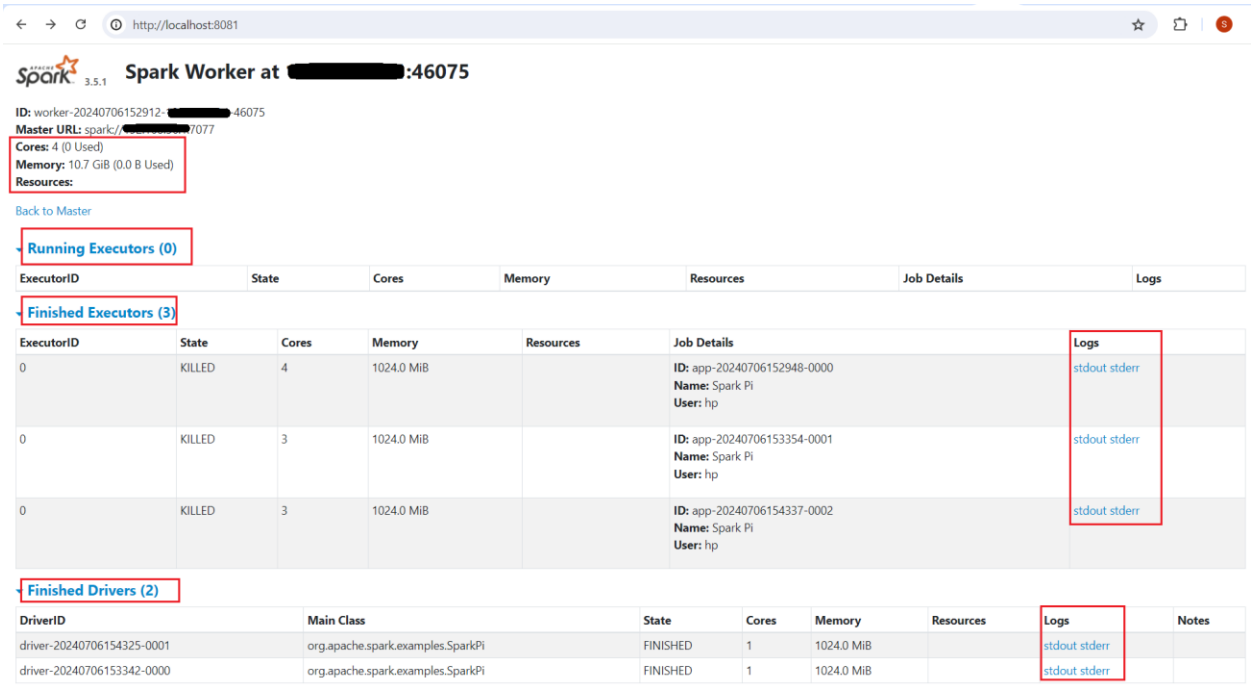
6.2. Worker Web UI:

The worker web UI is accessible at http://localhost:8081/ by default and will be available when Spark Worker is running.

Start the Worker service using the below command if not already running *(Make sure that you replace `<ipaddress>` with IP address mentioned in your Spark Master URL)*

```
spark-class org.apache.spark.deploy.worker.Worker
spark://<ipaddress>:7077
```

The worker web UI provides an overview of the executors and drivers that are spawned by the worker process. It displays information about the allotted resources, status, and provides links to log files for each of these child processes. We can use this web UI to see how many executors are currently running and the amount of resources allotted to each.



6.3. Application Web UI:

Each Spark application launches its own instance of the web UI. The application web UI is accessible at http://localhost:4040/jobs/ by default and will be available only when a Spark application is running.

The application web UI provides a wealth of information about the Spark application and can be a useful tool to debug the application. This web UI has the following tabs:

- **Jobs**: It displays a summary page of all jobs in the Spark application and a detailed page for each job. The summary page shows high-level information, such as the status, duration, and progress of all jobs and the overall event timeline. When you click on a job on the summary page, you see the detailed page for that job. The detailed page further shows the event timeline, DAG visualization, and all stages of the job.

- **Stages**: It displays a summary page that shows the current state of all stages of all jobs in the Spark application, and, when you click on a stage, a detailed page for that stage. The details page shows the event timeline, DAG visualization, and all tasks for the stage.

- **Storage:** It displays information about the RDDs and DataFrames, if any, in the application. The summary page shows the storage levels, sizes and partitions of all RDDs, and the detailed page shows the sizes and using executors for all partitions in an RDD.

- **Environment:** It displays the values for the different environment and configuration variables, including JVM, Spark, and system properties.

- **Executors:** It displays summary information about the executors that were created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

- **SQL:** If the application executes Spark SQL queries, the SQL tab displays information, such as the duration, jobs, and physical and logical plans for the queries.

- **Structured Streaming:** When running Structured Streaming jobs in micro-batch mode, a Structured Streaming tab will be available on the Web UI. It displays scheduling delay and processing time for each micro-batch in the data stream, which can be useful for troubleshooting the streaming application.

- **JDBC/ODBC Server:** This tab is visible when Spark is running as a distributed SQL engine. It shows information about sessions and submitted SQL operations.

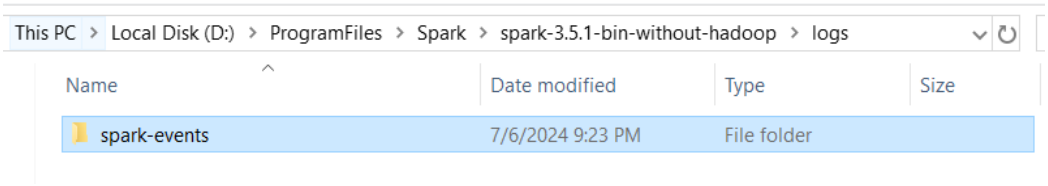
7. Spark History Server:

The Spark History Server is a web UI that displays the logging data from Spark jobs that were run with event logging enabled. It works only when files were not flushed before the Spark Master attempted to build a history user interface.

Before starting the Spark History server, we need to define Spark Event Log directory and Spark History Log directory in `spark-defaults.conf` file.

To store Spark logs on local file system:

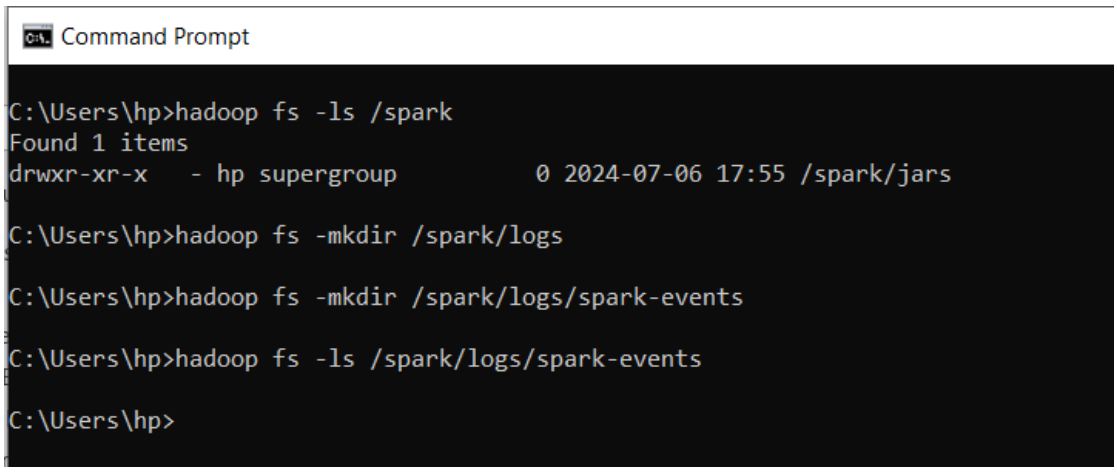
Go to `SPARK_HOME` location and create `logs` folder and create another folder named `spark-events` inside `logs` folder.



To store Spark logs on HDFS:

Create `/spark/logs` folder and create another folder named `spark-events` inside `/spark/logs` folder using these commands:

```
hadoop fs -ls /spark
hadoop fs -mkdir /spark/logs
hadoop fs -mkdir /spark/logs/spark-events
hadoop fs -ls /spark/logs/spark-events
```



Now, go to `SPARK_HOME/conf` location and edit the `spark-defaults.conf` file *(if it is not available, rename `spark-defaults.conf.template` to `spark-defaults.conf`)* to enable event logging.

Depending on where you want to enable event logging, add the following lines in the `spark-defaults.conf` file and ensure that the paths mentioned in `spark.history.fs.logDirectory` and `spark.eventLog.dir` are valid and accessible. Otherwise, Spark History server would fail to start.

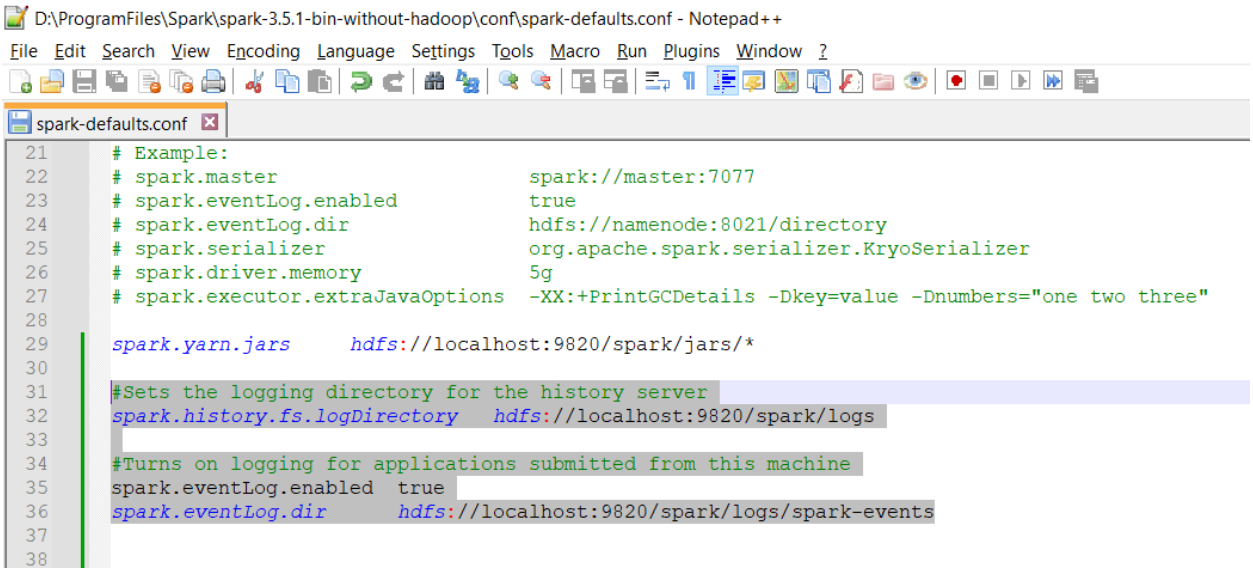
To enable event logging on local file system:

```
#Sets the logging directory for the history server
spark.history.fs.logDirectory file:///D:/ProgramFiles/Spark/spark-3.5.1-bin-
without-hadoop/logs

#Turns on logging for applications submitted from this machine
spark.eventLog.enabled  true
spark.eventLog.dir              file:///D:/ProgramFiles/Spark/spark-3.5.1-bin-
without-hadoop/logs/spark-events
```

To enable event logging on HDFS:

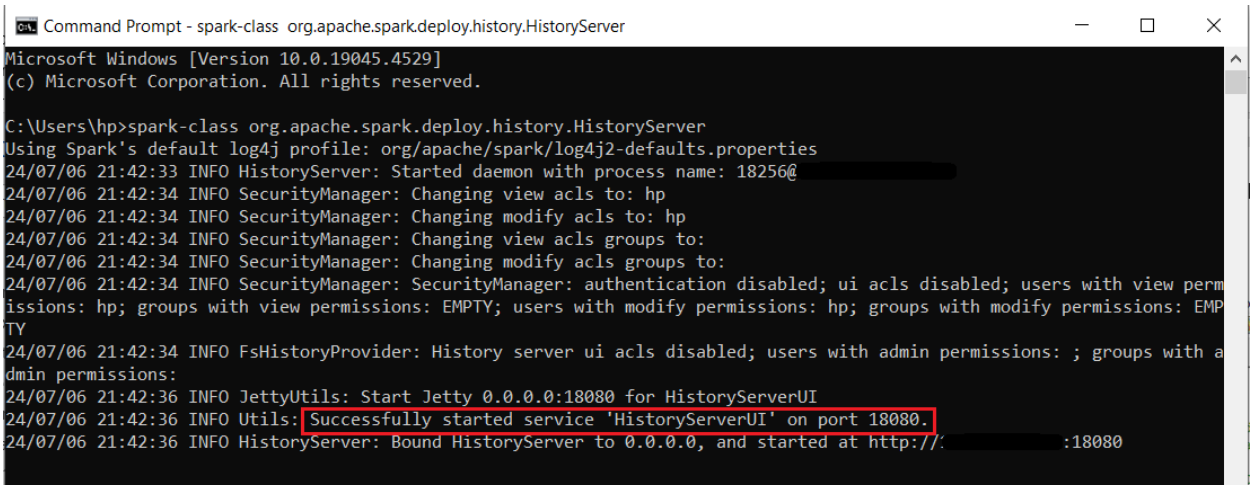
```
#Sets the logging directory for the history server
spark.history.fs.logDirectory   hdfs://localhost:9820/spark/logs

#Turns on logging for applications submitted from this machine
spark.eventLog.enabled  true
spark.eventLog.dir              hdfs://localhost:9820/spark/logs/spark-events
```

Since I want to enable event logging on HDFS, I added the above lines of code in `spark-defaults.conf` file as shown below:

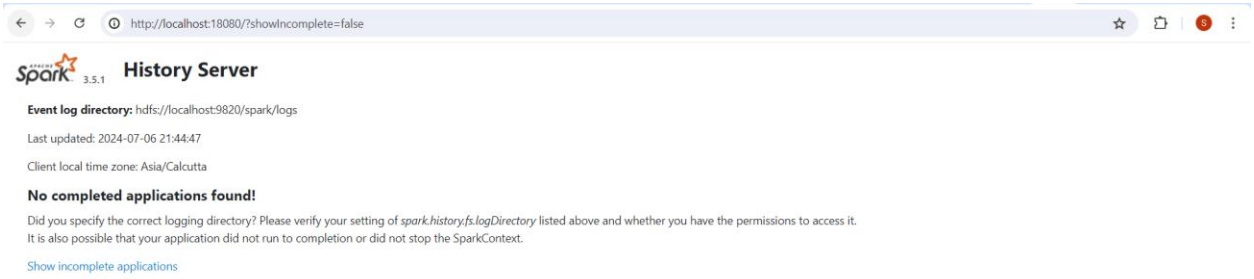


Now, start the Spark history server in Windows using the following command

```
spark-class org.apache.spark.deploy.history.HistoryServer
```



Once the history server is started, its UI is accessible under default port 18080 at http://localhost:18080/



This History server is useful to track the running and completed Spark applications even if the spark application web UI is not available.

Note:

When event logging is enabled, the default behavior is for all logs to be saved, which causes the storage to grow over time.

To enable automated cleanup, edit `spark-defaults.conf` file and add the following lines to perform cleanup daily and delete logs older than 7 days.

```
spark.history.fs.cleaner.enabled true
spark.history.fs.cleaner.interval 1d
spark.history.fs.cleaner.maxAge 7d
```

8. Write Sample PySpark Code:

Let us write some sample Pyspark code to read a CSV file and write the data into JSON format.

We will write the code to read/write CSV file in local file system as well as in HDFS.

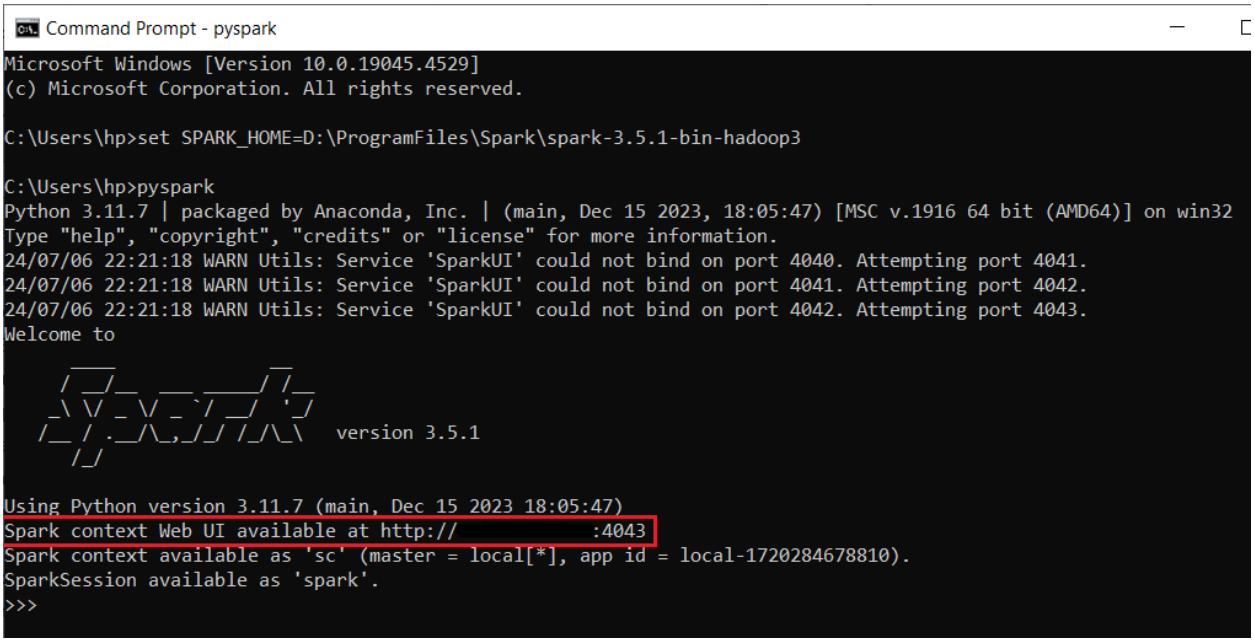
8.1. Read/Write Locally:

Since our `SPARK_HOME` is currently set to `spark-3.5.1-bin-without-hadoop` version, all files are expected to be read or written to HDFS.

To read or write files in local file system, we would need additional jar files that doesnot come with `spark-3.5.1-bin-without-hadoop` version. So, let us temporarily set the `SPARK_HOME` environment variable to the location where `spark-3.5.1-bin-hadoop3` version was installed and then launch `pyspark` CLI.

Open a new command prompt and run the following commands *(here, my `spark-3.5.1-bin-hadoop3` version is installed under `D:\ProgramFiles\Spark` location)*

```
set SPARK_HOME=D:\ProgramFiles\Spark\spark-3.5.1-bin-hadoop3

pyspark
```



Note that my `pyspark` interface has opened the Spark contextweb UI at http://localhost:4043

At the `pyspark` prompt, write the below lines of code to read a file into dataframe and displays output.

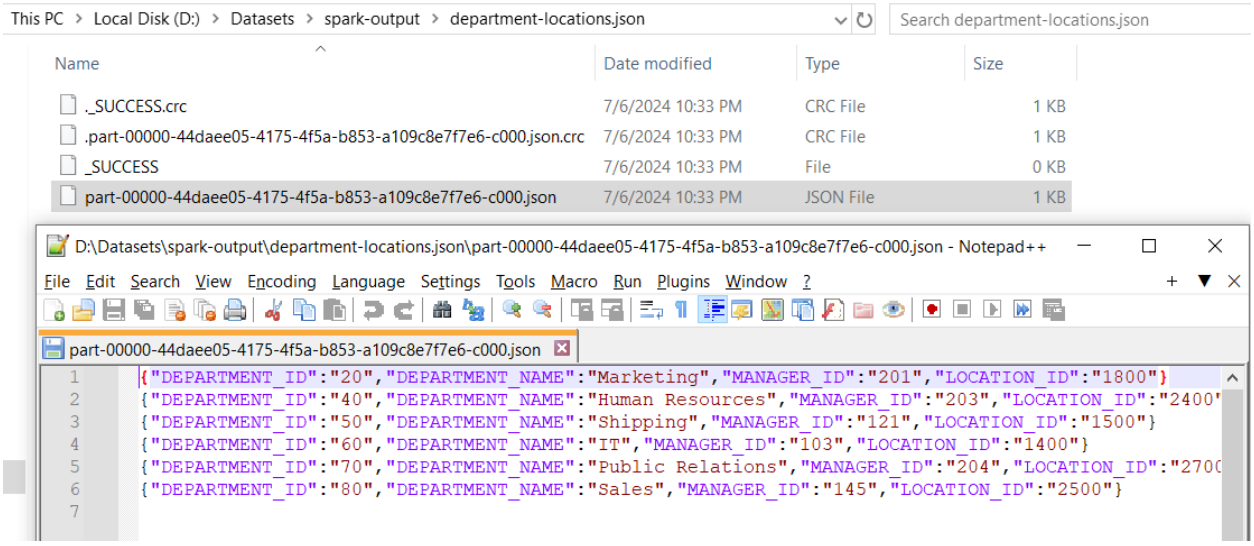
```
df=spark.read.option("header",True).csv("D:\Datasets\departments.csv")
df.printSchema()
df.show(5)
df.count()
df.createOrReplaceTempView("DEPARTMENTS")
df2=spark.sql("select DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID from
DEPARTMENTS where LOCATION_ID <> 1700")
df2.show()
df2.write.format("json").save("D:\Datasets\spark-output\department-locations.json")
```

Here, I am using the input file `departments.csv` available under `D:\Datasets` folder and writing into output location `D:\Datasets\spark-output\department-locations.json`.

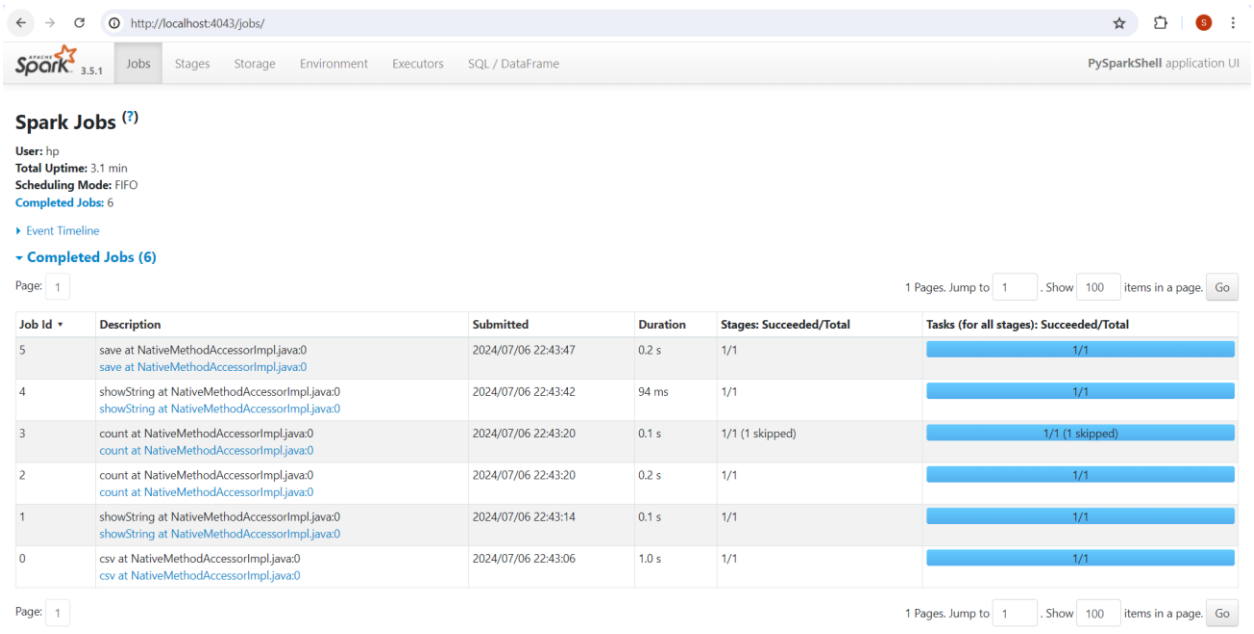
Note: You can copy `departments.csv` file from here and place it under `D:\Datasets` folder.

```
>>> df=spark.read.option("header",True).csv("D:\Datasets\departments.csv")
>>> df.printSchema()
root
 |-- DEPARTMENT_ID: string (nullable = true)
 |-- DEPARTMENT_NAME: string (nullable = true)
 |-- MANAGER_ID: string (nullable = true)
 |-- LOCATION_ID: string (nullable = true)

>>> df.show(5)
+-------------+---------------+----------+-----------+
|DEPARTMENT_ID|DEPARTMENT_NAME|MANAGER_ID|LOCATION_ID|
+-------------+---------------+----------+-----------+
|           10| Administration|       200|       1700|
|           20|      Marketing|       201|       1800|
|           30|     Purchasing|       114|       1700|
|           40|Human Resources|       203|       2400|
|           50|       Shipping|       121|       1500|
+-------------+---------------+----------+-----------+
only showing top 5 rows

>>> df.count()
27
>>> df2 = df.select("*").filter(df.LOCATION_ID != 1700)
>>> df2.show()
+-------------+---------------+----------+-----------+
|DEPARTMENT_ID| DEPARTMENT_NAME|MANAGER_ID|LOCATION_ID|
+-------------+---------------+----------+-----------+
|           20|      Marketing|       201|       1800|
|           40| Human Resources|       203|       2400|
|           50|       Shipping|       121|       1500|
|           60|             IT|       103|       1400|
|           70|Public Relations|       204|       2700|
|           80|          Sales|       145|       2500|
+-------------+---------------+----------+-----------+

>>> df2.write.format("json").save("D:\Datasets\spark-output\department-locations.json")
>>>
```

After the above code is executed, we can see that the output has been generated in `D:\Datasets\spark-output` location.



Open Spark Web UI at http://localhost:4043 where we can notice some jobs submitted for the code that was executed from the Pyspark shell.



8.2. Read/Write HDFS:

First, let us copy the input file `departments.csv` into HDFS

```
hadoop fs -put "D:\Datasets\departments.csv" /spark-input
```

```
hadoop fs -ls /spark-input
```

```
C:\Users\hp>hadoop fs -mkdir /spark-input

C:\Users\hp>hadoop fs -put "D:\Datasets\departments.csv" /spark-input

C:\Users\hp>hadoop fs -ls /spark-input
Found 1 items
-rw-r--r--   1 hp supergroup        709 2024-07-07 00:34 /spark-input/departments.csv

C:\Users\hp>
```

Next, open pyspark shell using below command

```
pyspark
```

```
C:\Users\hp>pyspark
Python 3.11.7 | packaged by Anaconda, Inc. | (main, Dec 15 2023, 18:05:47) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
24/07/07 00:35:48 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
24/07/07 00:35:48 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
24/07/07 00:35:48 WARN Utils: Service 'SparkUI' could not bind on port 4042. Attempting port 4043.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.1
      /_/

Using Python version 3.11.7 (main, Dec 15 2023 18:05:47)
Spark context Web UI available at http://192.168.56.1:4043
Spark context available as 'sc' (master = local[*], app id = local-1720292748177).
SparkSession available as 'spark'.
>>>
```

On the pyspark shell, write the below lines of code to read the above input file into dataframe and displays output

```
df=spark.read.options(header=True, inferSchema=True).csv("/spark-
input/departments.csv")
df.printSchema()
df.show(5, truncate=False)
df.count()
df.createOrReplaceTempView("DEPARTMENTS")
df2=spark.sql("select DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID from
DEPARTMENTS where LOCATION_ID <> 1700")
df2.show()
df2.write.format("json").save("/spark-output/department-locations.json")
```

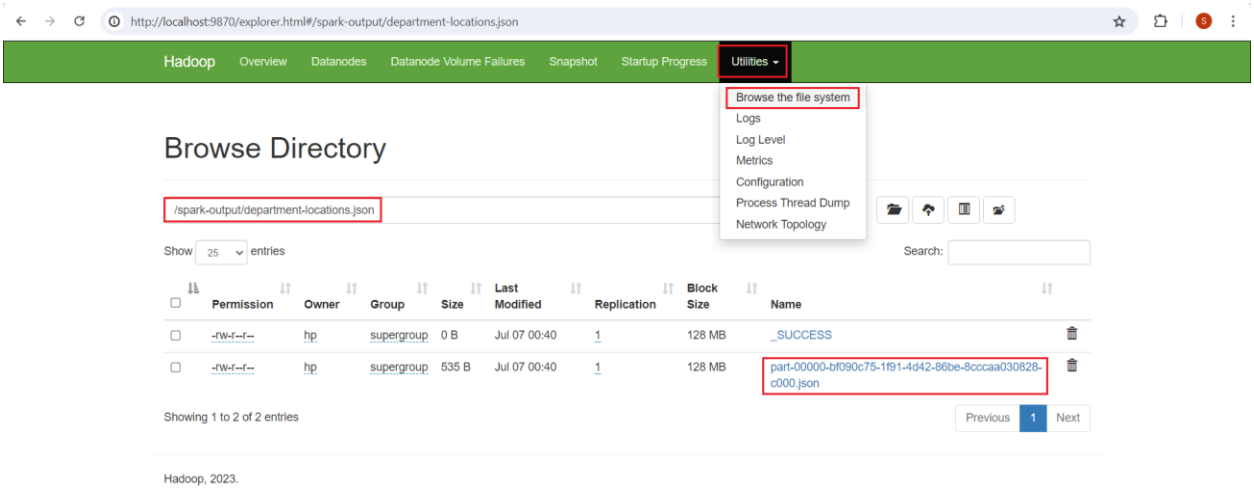
```
>>> df=spark.read.options(header=True, inferSchema=True).csv("/spark-input/departments.csv")
>>> df.printSchema()
root
 |-- DEPARTMENT_ID: integer (nullable = true)
 |-- DEPARTMENT_NAME: string (nullable = true)
 |-- MANAGER_ID: string (nullable = true)
 |-- LOCATION_ID: integer (nullable = true)

>>> df.show(5, truncate=False)
+-------------+---------------+----------+-----------+
|DEPARTMENT_ID|DEPARTMENT_NAME|MANAGER_ID|LOCATION_ID|
+-------------+---------------+----------+-----------+
|10           |Administration |200       |1700       |
|20           |Marketing      |201       |1800       |
|30           |Purchasing     |114       |1700       |
|40           |Human Resources|203       |2400       |
|50           |Shipping       |121       |1500       |
+-------------+---------------+----------+-----------+
only showing top 5 rows

>>> df.count()
27
>>> df.createOrReplaceTempView("DEPARTMENTS")
>>> df2=spark.sql("select DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID from DEPARTMENTS where LOCATION_ID <>
1700")
>>> df2.show()
+-------------+---------------+----------+-----------+
|DEPARTMENT_ID| DEPARTMENT_NAME|MANAGER_ID|LOCATION_ID|
+-------------+---------------+----------+-----------+
|           20|      Marketing|       201|       1800|
|           40|Human Resources|       203|       2400|
|           50|       Shipping|       121|       1500|
|           60|             IT|       103|       1400|
|           70|Public Relations|       204|       2700|
|           80|          Sales|       145|       2500|
+-------------+---------------+----------+-----------+

>>> df2.write.format("json").save("/spark-output/department-locations.json")
>>>
```

Open another command prompt and run the following commands to see the output file created in HDFS.

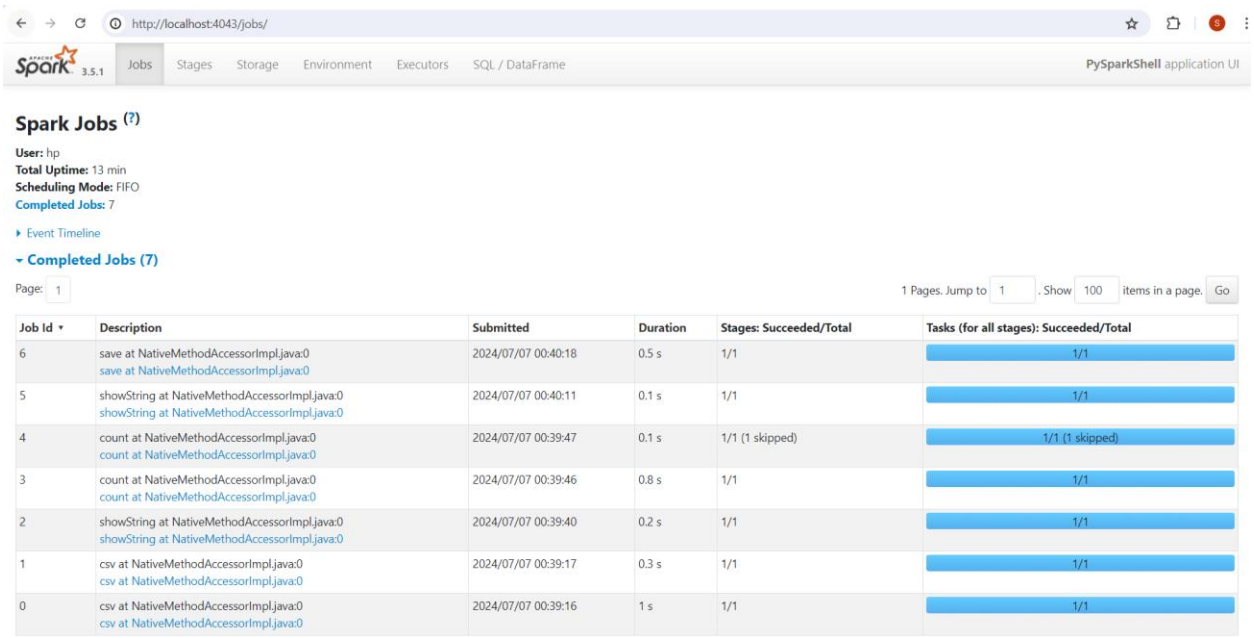
```
hadoop fs -ls /spark-output/department-locations.json
hadoop fs -cat /spark-output/department-locations.json/part-00000-*.json
```

```
C:\Users\hp>hadoop fs -ls /spark-output/department-locations.json
Found 2 items
-rw-r--r--   1 hp supergroup          0 2024-07-07 00:40 /spark-output/department-locations.json/_SUCCESS
-rw-r--r--   1 hp supergroup        535 2024-07-07 00:40 /spark-output/department-locations.json/part-00000-bf090c75-1f9
1-4d42-86be-8cccaa030828-c000.json

C:\Users\hp>hadoop fs -cat /spark-output/department-locations.json/part-00000-*.json
{"DEPARTMENT_ID":20,"DEPARTMENT_NAME":"Marketing","MANAGER_ID":"201","LOCATION_ID":1800}
{"DEPARTMENT_ID":40,"DEPARTMENT_NAME":"Human Resources","MANAGER_ID":"203","LOCATION_ID":2400}
{"DEPARTMENT_ID":50,"DEPARTMENT_NAME":"Shipping","MANAGER_ID":"121","LOCATION_ID":1500}
{"DEPARTMENT_ID":60,"DEPARTMENT_NAME":"IT","MANAGER_ID":"103","LOCATION_ID":1400}
{"DEPARTMENT_ID":70,"DEPARTMENT_NAME":"Public Relations","MANAGER_ID":"204","LOCATION_ID":2700}
{"DEPARTMENT_ID":80,"DEPARTMENT_NAME":"Sales","MANAGER_ID":"145","LOCATION_ID":2500}

C:\Users\hp>
```

The same is visible in NameNode UI: http://localhost:9870/dfshealth.html
In NameNode UI, go to **Utilities** tab and select **Browse the file system** option. Search for `/spark-output/department-locations.json` directory where we can see `part-00000-*.json` file is available.



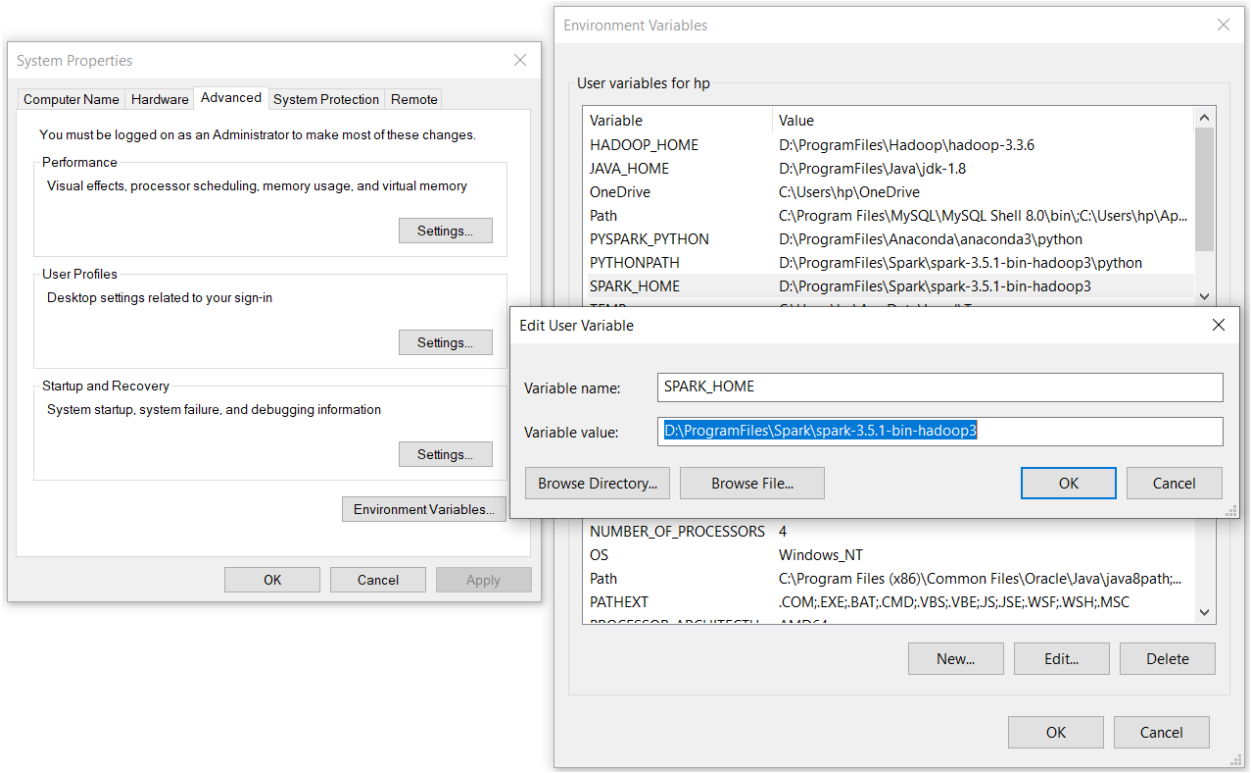
Open Spark Web UI at http://localhost:4043 where we can notice some jobs submitted for the code that we executed from the Pyspark shell.



9. Spark SQL:

The Spark SQL CLI is a conventional tool to run the Hive metastore service in local mode and execute queries from command line. It creates an embedded `metastore_db` directory *(a relational database to manage the metadata of relational entities such as databases, tables, columns partitions)* in the location from where ever `spark-sql` is started.

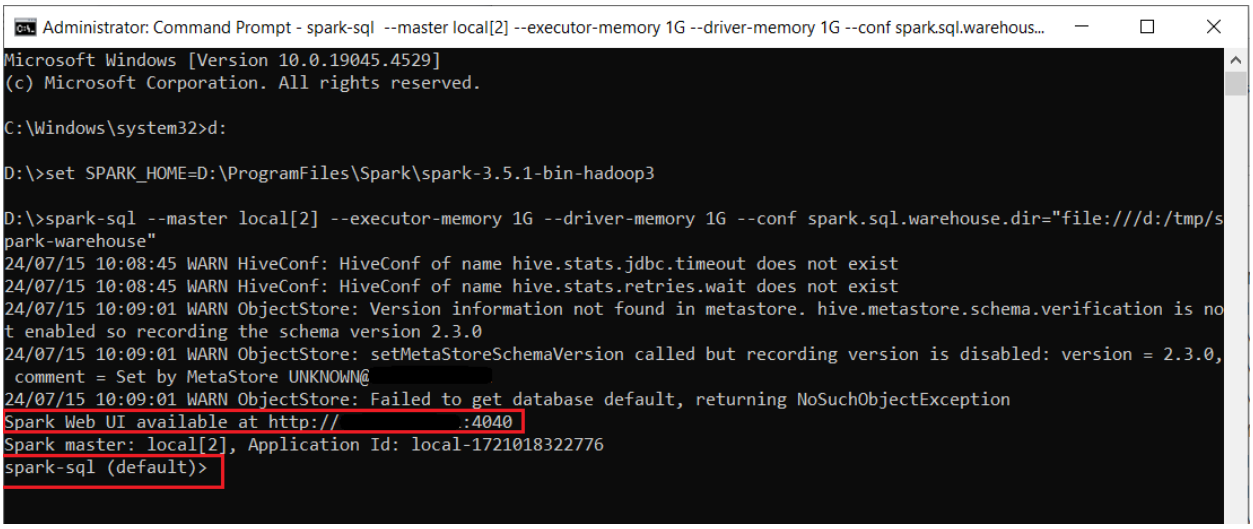
`spark-sql` command line interface requires hive libraries that comes with `spark-3.5.1-bin-hadoop3` version, so set your `SPARK_HOME` environment varaible to the location where `spark-3.5.1-bin-hadoop3` version was installed.



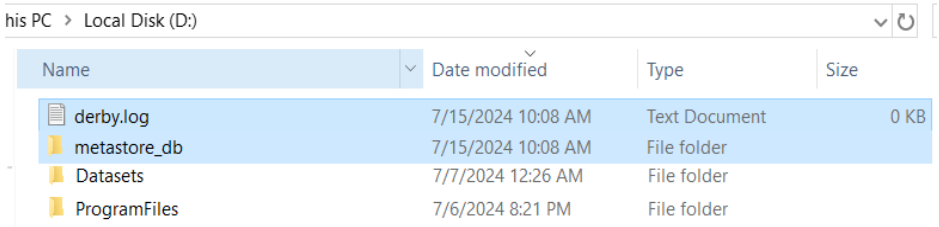
Open a new **Command Prompt** in **Administrator** mode and run the following command:

```
spark-sql --master local[2] --executor-memory 1G --driver-memory 1G --conf spark.sql.warehouse.dir="file:///d:/tmp/spark-warehouse"
```

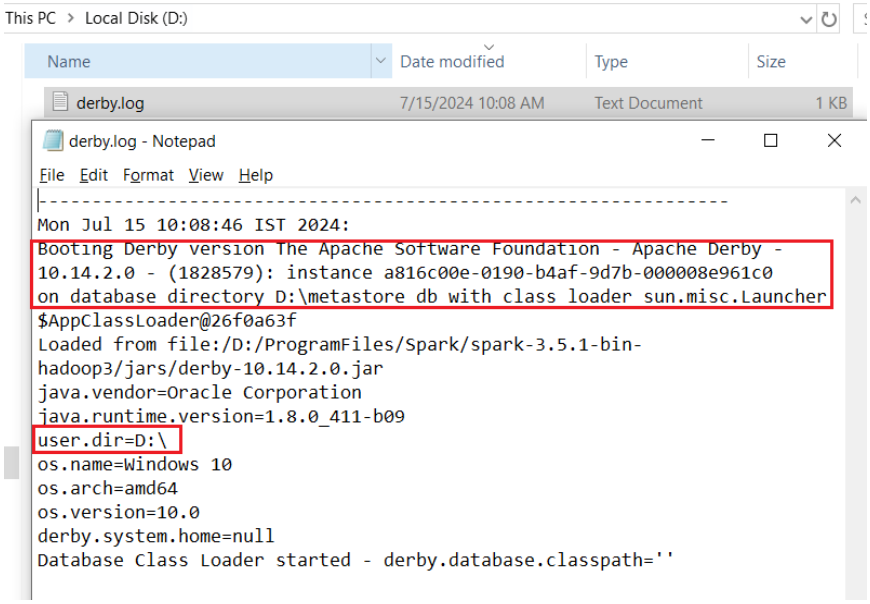
Note that we have set `spark.sql.warehouse.dir` property to `D:\tmp\spark-warehouse` where Spark SQL stores all relational entities such as databases, tables, columns, etc.



As soon as `spark-sql` is launched, we can see that `metastore_db` directory and `derby.log` file is created in the location from where it was launched.



`derby.log` file tells us it has booted the Apache Derby database on the specific database directory.



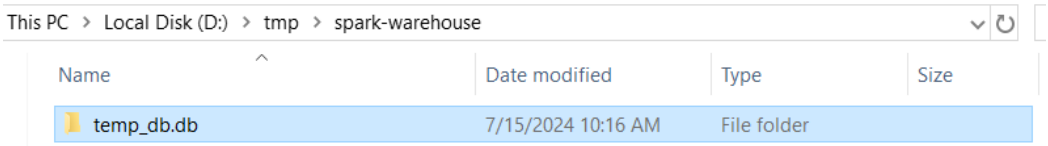
On the `spark-sql>` prompt, let us run some Hive queries to create a database, table and load data into it.

- Create a database named `temp_db`

```
create database temp_db;
show databases;
```

```
spark-sql (default)> show databases;
default
Time taken: 4.516 seconds, Fetched 1 row(s)
spark-sql (default)> create database temp_db;
24/07/15 10:16:35 WARN ObjectStore: Failed to get database temp_db, returning NoSuchObjectException
24/07/15 10:16:35 WARN ObjectStore: Failed to get database temp_db, returning NoSuchObjectException
24/07/15 10:16:35 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
24/07/15 10:16:35 WARN ObjectStore: Failed to get database temp_db, returning NoSuchObjectException
Time taken: 1.371 seconds
spark-sql (default)> show databases;
default
temp_db
Time taken: 0.053 seconds, Fetched 2 row(s)
spark-sql (default)>
```

As soon as the above queries are executed, we can see that `spark-warehouse` directory is created at `D:\tmp` location and `temp_db.db` folder is created under `spark-warehouse` directory.



- Create a table named `employees` in `temp_db` database. This table is created based on the columns data in the CSV file that we are going to load.

```
use temp_db;

create table employees(employee_id int, first_name string, last_name string,
email string, phone_number string, hire_date string, job_id string, salary
int, commission_pct int, manager_id int, department_id int) row format
delimited fields terminated by ',' tblproperties
('skip.header.line.count'='1');
```

```
spark-sql (default)> use temp_db;
Time taken: 0.043 seconds
spark-sql (temp_db)> create table employees(employee_id int, first_name string, last_name string, email string, phone_nu
mber string, hire_date string, job_id string, salary int, commission_pct int, manager_id int, department_id int) row for
mat delimited fields terminated by ',' tblproperties ('skip.header.line.count'='1');
24/07/15 10:42:29 WARN SessionState: METASTORE_FILTER_HOOK will be ignored, since hive.security.authorization.manager is
 set to instance of HiveAuthorizerFactory.
24/07/15 10:42:29 WARN HiveConf: HiveConf of name hive.internal.ss.authz.settings.applied.marker does not exist
24/07/15 10:42:29 WARN HiveConf: HiveConf of name hive.stats.jdbc.timeout does not exist
24/07/15 10:42:29 WARN HiveConf: HiveConf of name hive.stats.retries.wait does not exist
24/07/15 10:42:29 WARN HiveMetaStore: Location: file:/d:/tmp/spark-warehouse/temp_db.db/employees specified for non-exte
rnal table:employees
Time taken: 2.068 seconds
spark-sql (temp_db)>
```

On the local system, `employees` folder is created in `D:\tmp\temp_db.db` location.

- Verify the format of the `employees` table created.

```
describe formatted employees;
```

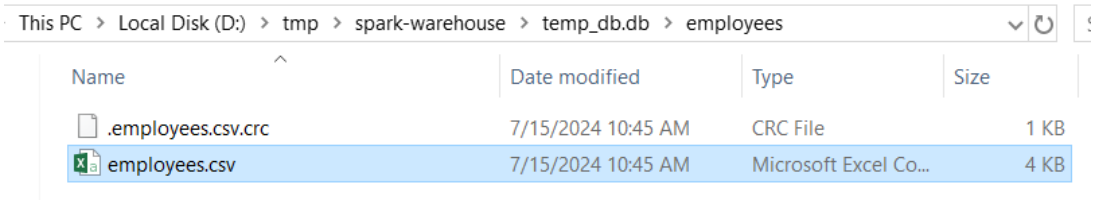
```
spark-sql (temp_db)> describe formatted employees;
employee_id             int
first_name              string
last_name               string
email                   string
phone_number            string
hire_date               string
job_id                  string
salary                  int
commission_pct          int
manager_id              int
department_id           int

# Detailed Table Information
Catalog                 spark_catalog
Database                temp_db
Table                   employees
Owner                   hp
Created Time            Mon Jul 15 10:42:29 IST 2024
Last Access             UNKNOWN
Created By              Spark 3.5.1
Type                    MANAGED
Provider                hive
Table Properties        [skip.header.line.count=1, transient_lastDdlTime=1721020805]
Location                file:/d:/tmp/spark-warehouse/temp_db.db/employees
Serde Library           org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat             org.apache.hadoop.mapred.TextInputFormat
OutputFormat            org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Storage Properties      [serialization.format=,, field.delim=,]
Partition Provider      Catalog
Time taken: 0.1 seconds, Fetched 29 row(s)
spark-sql (temp_db)>
```

- Load data from `employees.csv` file located in the local directory `D:\Datasets` into the hive table `employees` *(You can download `employees.csv` file from [this location] and copy to `D:\Datasets` folder in your machine).*

```
load data local inpath 'file:///D:/Datasets/employees.csv' into table
employees;
```

```
spark-sql (temp_db)> load data local inpath 'file:///D:\Datasets\employees.csv' into table employees;
[LOAD_DATA_PATH_NOT_EXISTS] LOAD DATA input path does not exist: file:///D:Datasetsemployees.csv.
spark-sql (temp_db)> load data local inpath 'file:///D:/Datasets/employees.csv' into table employees;
Time taken: 0.913 seconds
spark-sql (temp_db)>
```

In local system, there is a `employees.csv` file created in `D:\tmp\temp_db.db\employees` location.



- Select top 5 records in `employees` table from Hive CLI.

```
set spark.hadoop.hive.cli.print.header=true;
select * from employees limit 5;
```

```
spark-sql (temp_db)> set spark.hadoop.hive.cli.print.header=true;
spark.hadoop.hive.cli.print.header      true
Time taken: 0.02 seconds, Fetched 1 row(s)
spark-sql (temp_db)> select * from employees limit 5;
NULL    FIRST_NAME      LAST_NAME       EMAIL   PHONE_NUMBER    HIRE_DATE       JOB_ID  NULL    NULL    NULL    NULL
198     Donald  OConnell        DOCONNEL        650.507.9833    21-JUN-07       SH_CLERK        2600    NULL    124
        50
199     Douglas Grant   DGRANT  650.507.9844    13-JAN-08       SH_CLERK        2600    NULL    124     50
200     Jennifer        Whalen  JWHALEN 515.123.4444    17-SEP-03       AD_ASST 4400    NULL    101     10
201     Michael Hartstein       MHARTSTE        515.123.5555    17-FEB-04       MK_MAN  13000   NULL    100     20
Time taken: 0.177 seconds, Fetched 5 row(s)
spark-sql (temp_db)>
```

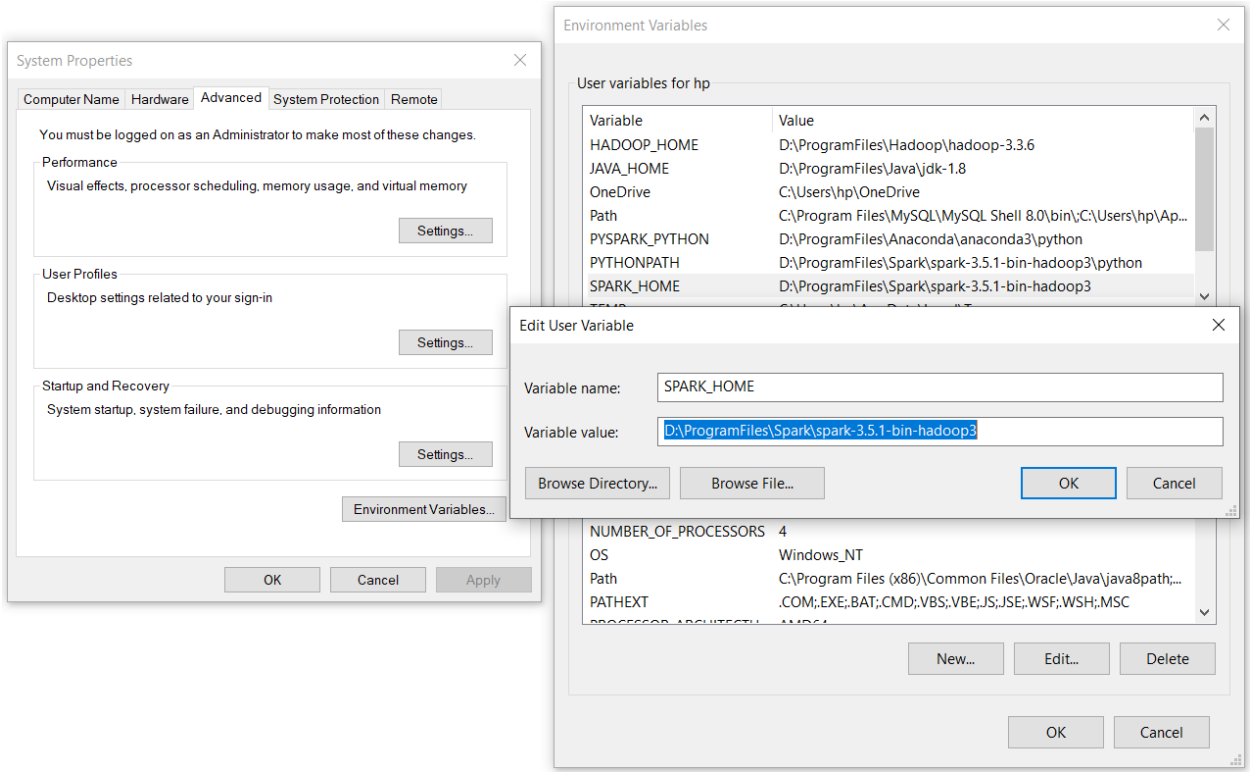
Note: By default, Spark SQL will not skip headers although we set the table property `tblproperties ('skip.header.line.count'='1')`.

Similarly, we can run any queries from `spark-sql` CLI that connects to Hive metastore database running locally.

10. Spark Beeline:

Spark provides `beeline` CLI that runs on Thrift JDBC/ODBC server *(corresponds to hiveserver2 service in built-in Hive)*. Spark SQL can also act as a distributed query engine using its JDBC/ODBC server.

Since `beeline` command line interface requires hive libraries that comes with `spark-3.5.1-bin-hadoop3` version, make sure that your `SPARK_HOME` environment varaible is set to the location where `spark-3.5.1-bin-hadoop3` version was installed.

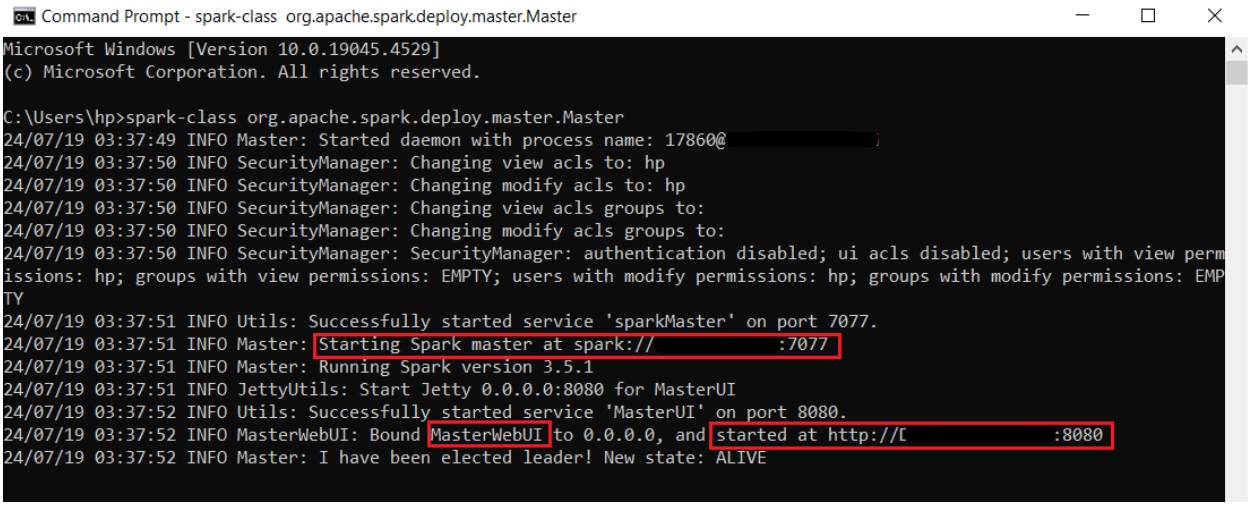


To start using Spark provided `beeline`, we need to first start the Spark Master, Worker and Thrift JDBC/ODBC services.

10.1.1. Start Spark Master:

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the Spark Master service:

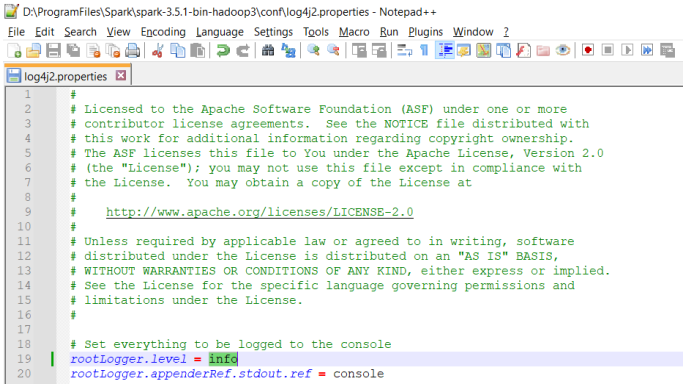
```
spark-class org.apache.spark.deploy.master.Master
```



Here, we can see that Spark master is started at spark://<ipaddress>:7077 i.e listening on default port `7077` and **MasterWebUI** is started at http://<ipaddress>:8080 by default.

Note:

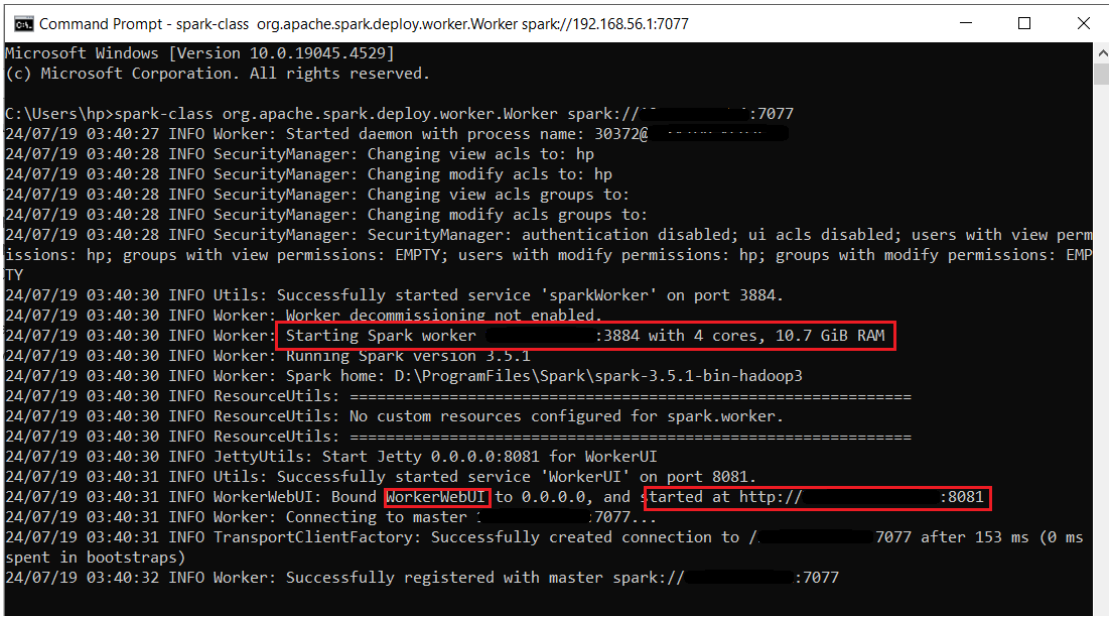
If you are unable to see any logging on the console, then open `log4j2.properties` file in `SPARK_HOME\conf` location and change the `rootLogger.level` property to `info` value.



10.1.2. Start Spark Worker:

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the Spark Worker service *(Make sure that you replace `<ipaddress>` with IP address mentioned in your Spark Master URL)*

```
spark-class org.apache.spark.deploy.worker.Worker
spark://<ipaddress>:7077
```



Here, we can see that Spark worker is started at `<ipaddress>:3884` (on a random port) with available number of cores and RAM running on the machine. **WorkerWebUI** is started at http://<ipaddress>:8081 by default.

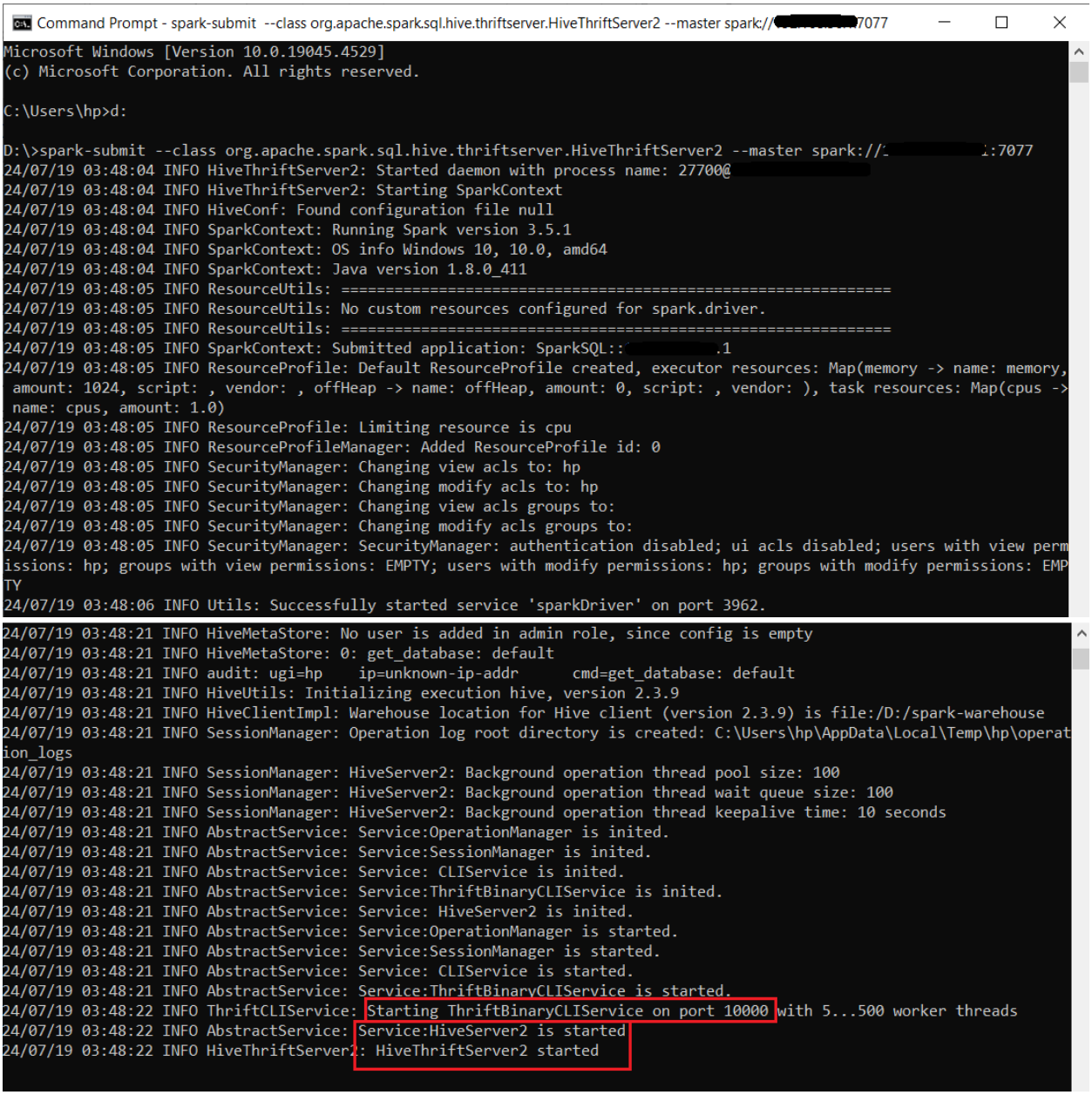
10.1.3. Start Spark Thrift Server:

Spark Thrift services tries to access (or create if not available) an embedded `metastore_db` directory *(a relational database to manage the metadata of relational entities such as databases, tables, columns partitions)* in the location from where ever it is started.

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the Spark Thrift service *(Make sure that you replace `<ipaddress>` with IP address mentioned in your Spark Master URL)*

Note: To connect to the existing `metastore_db` *(that was already created by Spark SQL),* make sure to navigate to the location where `metastore_db` is available and execute this command:

```
spark-submit --class
org.apache.spark.sql.hive.thriftserver.HiveThriftServer2 --master
spark://<ipaddress>:7077
```

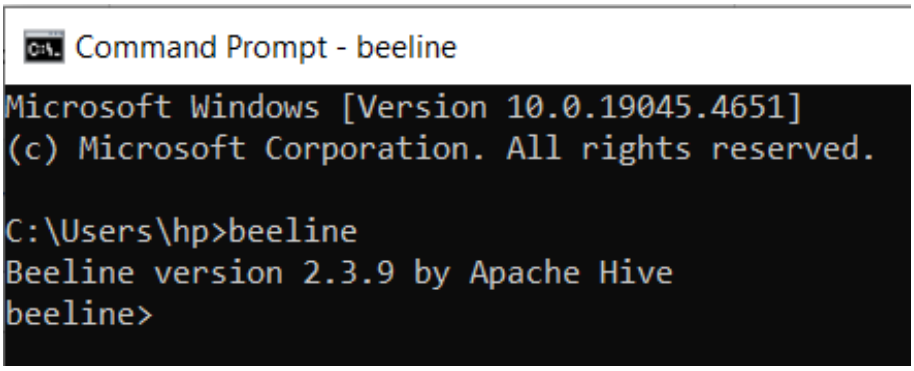


Here, we can see that Spark **ThriftBinaryCLIService** is started on port **10000** and **HiveServer2** and **HiveThriftServer2** services are started.

10.1.4. Start Beeline:

Open a new **Command Prompt** or **Windows PowerShell** and run the following command to start the beeline CLI:

```
beeline
```



Here, we can see the Beeline version **2.3.9.**

Now, connect to the JDBC/ODBC thrift server in beeline using the following command. When Beeline asks for a username and password, simply enter the username on your machine and a blank password.

```
!connect jdbc:hive2://localhost:10000
```

```
beeline> !connect jdbc:hive2://localhost:10000
Connecting to jdbc:hive2://localhost:10000
Enter username for jdbc:hive2://localhost:10000: hp
Enter password for jdbc:hive2://localhost:10000:
24/07/24 22:50:51 INFO Utils: Supplied authorities: localhost:10000
24/07/24 22:50:51 INFO Utils: Resolved authority: localhost:10000
Connected to: Spark SQL (version 3.5.1)
Driver: Hive JDBC (version 2.3.9)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000>
```

Our Beeline has been successfully connected to Thrift server running on 10000 port.

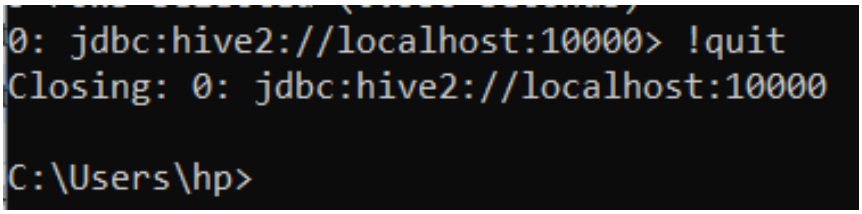
Now, we can run queries similar to Spark SQL

```
show databases;
use temp_db;
show tables;
select * from employees limit 5;
```

```
0: jdbc:hive2://localhost:10000> show databases;
+------------+
| namespace  |
+------------+
| default    |
| temp_db    |
+------------+
2 rows selected (0.262 seconds)
0: jdbc:hive2://localhost:10000> use temp_db;
+---------+
| Result  |
+---------+
+---------+
No rows selected (0.083 seconds)
0: jdbc:hive2://localhost:10000> show tables;
+------------+------------+--------------+
| namespace  | tableName  | isTemporary  |
+------------+------------+--------------+
| temp_db    | employees  | false        |
+------------+------------+--------------+
1 row selected (0.149 seconds)
0: jdbc:hive2://localhost:10000> select * from employees limit 5;
+--------------+-------------+------------+-----------+---------------+------------+-----------+---------+--------------
---+-------------+----------------+
| employee_id  | first_name  | last_name  |   email   | phone_number  | hire_date  |  job_id   | salary  | commission_pc
t  | manager_id  | department_id  |
+--------------+-------------+------------+-----------+---------------+------------+-----------+---------+--------------
---+-------------+----------------+
| NULL         | FIRST_NAME  | LAST_NAME  | EMAIL     | PHONE_NUMBER  | HIRE_DATE  | JOB_ID    | NULL    | NULL
   | NULL        | NULL           |
| 198          | Donald      | OConnell   | DOCONNEL  | 650.507.9833  | 21-JUN-07  | SH_CLERK  | 2600    | NULL
   | 124         | 50             |
| 199          | Douglas     | Grant      | DGRANT    | 650.507.9844  | 13-JAN-08  | SH_CLERK  | 2600    | NULL
   | 124         | 50             |
| 200          | Jennifer    | Whalen     | JWHALEN   | 515.123.4444  | 17-SEP-03  | AD_ASST   | 4400    | NULL
   | 101         | 10             |
| 201          | Michael     | Hartstein  | MHARTSTE  | 515.123.5555  | 17-FEB-04  | MK_MAN    | 13000   | NULL
   | 100         | 20             |
+--------------+-------------+------------+-----------+---------------+------------+-----------+---------+--------------
---+-------------+----------------+
5 rows selected (0.536 seconds)
0: jdbc:hive2://localhost:10000>
```

Use this command to exit out of beeline shell:

```
!quit
```



11. Spark with Jupiter Notebook:

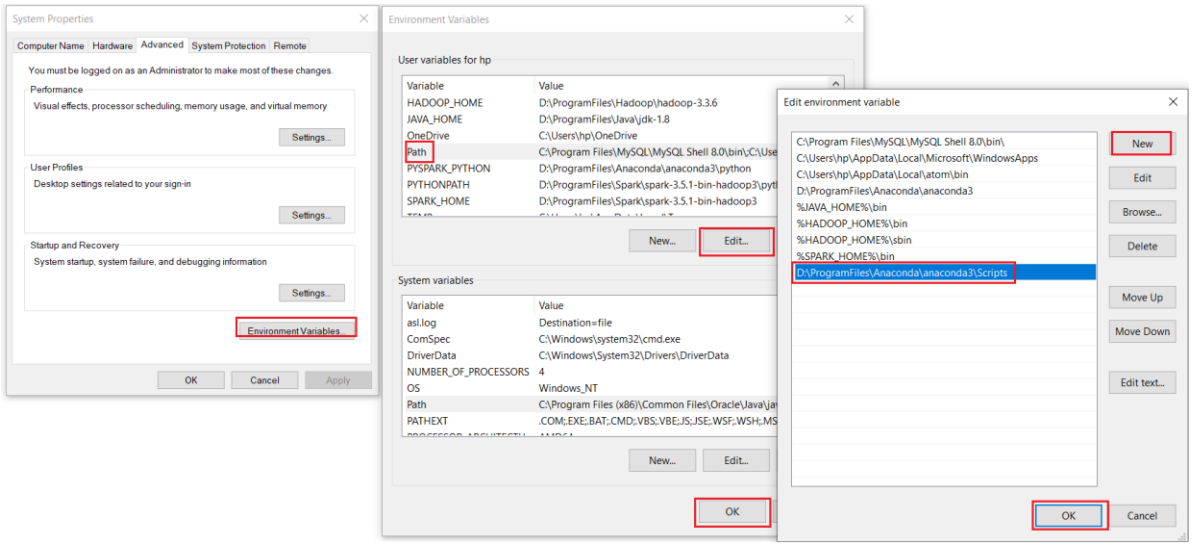
Now, we will see how to integrate **Pyspark** with **Jupiter Notebook** available in Anaconda installation so that we can execute Spark code on Jupiter Notebook directly.

First, make sure that Anaconda scripts location is added to `PATH` environment variable.

In the Windows search bar, start typing "environment variables" and select the first match which opens up **System Properties** dialog where press **Environment Variables** button.

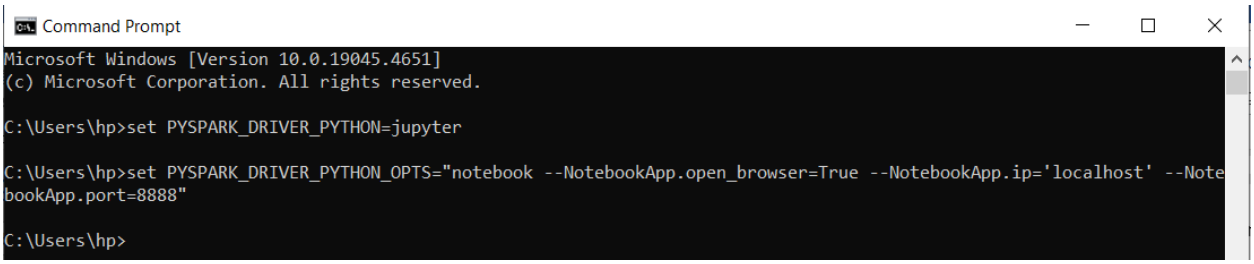
In the **Environment Variables** dialog, select `PATH` variable under **User Variables** and press **Edit** button. Then press **New** and add the value `D:\ProgramFiles\Anaconda\anaconda3\Scripts` *(the path where Anaconda distribution was installed)* and press OK. Then press OK on **Environment Variables** dialog

Note: My Anaconda is installed under `D:\ProgramFiles`. If you have installed in a different location, update the above path appropriately.



Now, open a new command prompt and set the following environment variables for pyspark to execute Jupyter.

```
set PYSPARK_DRIVER_PYTHON=jupyter
set PYSPARK_DRIVER_PYTHON_OPTS="notebook --
NotebookApp.open_browser=True --NotebookApp.ip='localhost' --
NotebookApp.port=8888"
```



Note: If you are installing Spark on a Virtual Machine and would like to access Jupyter from your host browser, you should set the `NotebookApp.ip` flag to `--NotebookApp.ip='0.0.0.0'` in the above command so that your VM's Jupyter server will accept external connections. You can then access Jupyter notebook from the host machine on port 8888.

Execute pyspark with the following command:

```
pyspark --master local[*] --executor-memory 1G --driver-memory 1G --conf spark.sql.warehouse.dir="file:///D:/tmp/spark-warehouse" --packages com.amazonaws:aws-java-sdk-pom:1.10.34 --packages org.apache.hadoop:hadoop-aws:2.8.0
```

The above `pyspark` command performs the following duties:

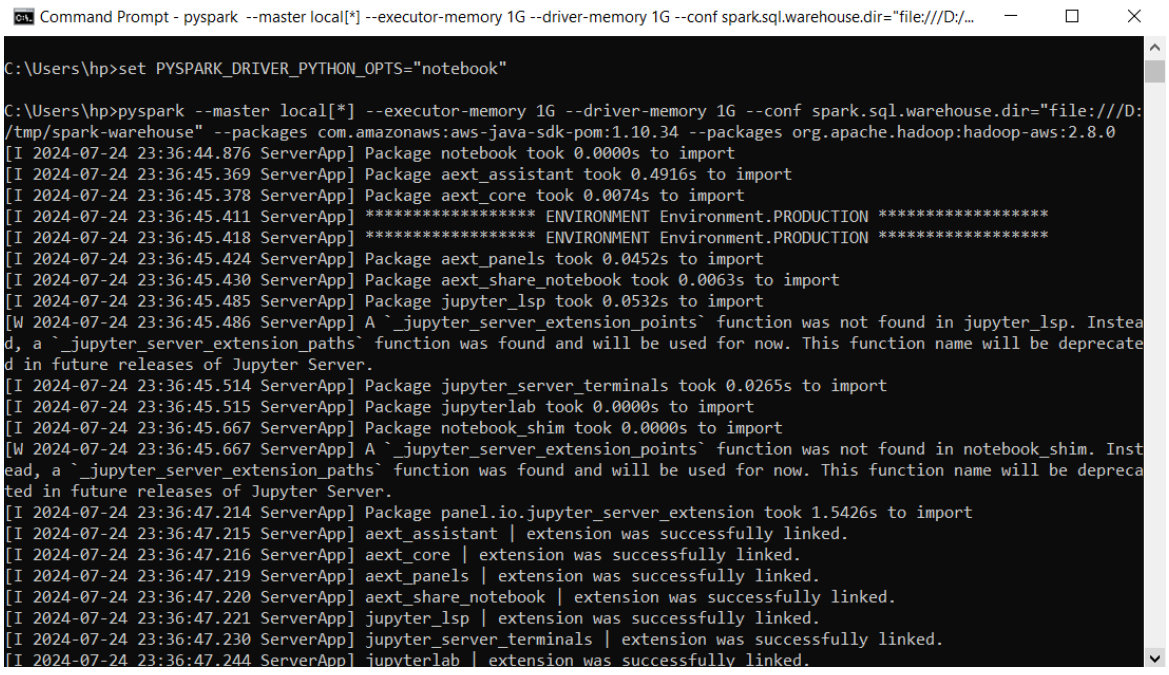
- Execute `pyspark` locally (local master) with all cores of the computer
- Set the Spark Executor memory to 1GB
- Set the Spark Driver memory to 1GB
- Set the Spark SQL Warehouse directory to `D:\tmp\spark-warehouse` in local system to store Spark SQL data
- Set specific packages such as `amazonaws:aws-java-sdk-pom` and `org.apache.hadoop:hadoop-aws` for pyspark to load *(These packages are necessary to access AWS S3 repositories from Spark)*

Note: After executing the above command, you may encounter an error *"Jupyter command `jupyter-notebook --NotebookApp.open_browser=True --NotebookApp.ip='localhost' --NotebookApp.port=8888` not found"* as below.

```
C:\Users\hp>pyspark --master local[*] --executor-memory 1G --driver-memory 1G --conf spark.sql.warehouse.dir="file:///D:
/tmp/spark-warehouse" --packages com.amazonaws:aws-java-sdk-pom:1.10.34 --packages org.apache.hadoop:hadoop-aws:2.8.0
usage: jupyter [-h] [--version] [--config-dir] [--data-dir] [--runtime-dir] [--paths] [--json] [--debug] [subcommand]

Jupyter: Interactive Computing

positional arguments:
  subcommand     the subcommand to launch

options:
  -h, --help     show this help message and exit
  --version      show the versions of core jupyter packages and exit
  --config-dir   show Jupyter config dir
  --data-dir     show Jupyter data dir
  --runtime-dir  show Jupyter runtime dir
  --paths        show all Jupyter paths. Add --json for machine-readable format.
  --json         output paths as machine-readable json
  --debug        output debug information about paths

Available subcommands: console dejavu events execute kernel kernelspec lab labextension labhub migrate nbconvert
notebook qtconsole run script server troubleshoot trust

Jupyter command `jupyter-notebook --NotebookApp.open_browser=True --NotebookApp.ip='localhost' --NotebookApp.port=8888`
not found.

C:\Users\hp>
```

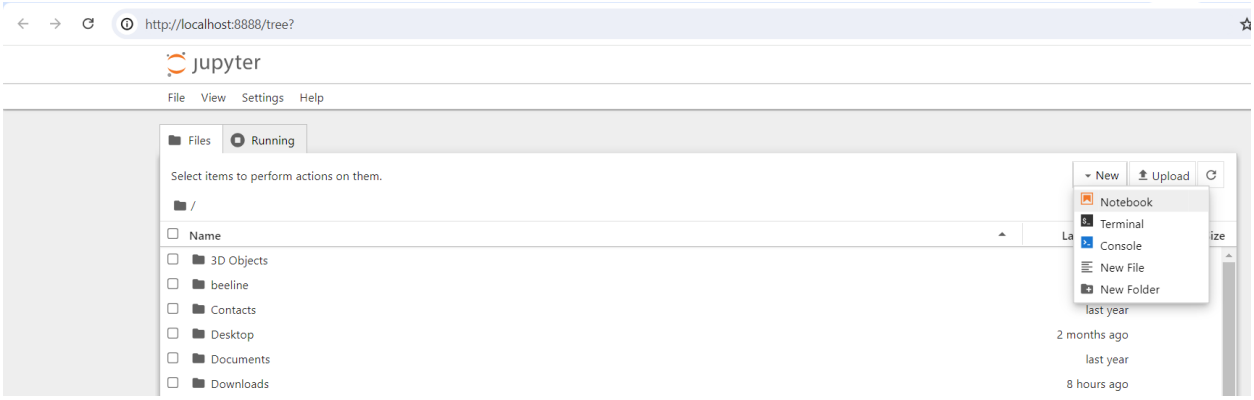
To resolve the above error, set `PYSPARK_DRIVER_PYTHON_OPTS` to "`notebook`" and launch pyspark with these commands:

```
set PYSPARK_DRIVER_PYTHON_OPTS="notebook"

pyspark --master local[*] --executor-memory 1G --driver-memory 1G --
conf spark.sql.warehouse.dir="file:///D:/tmp/spark-warehouse" --
packages com.amazonaws:aws-java-sdk-pom:1.10.34 --packages
org.apache.hadoop:hadoop-aws:2.8.0
```

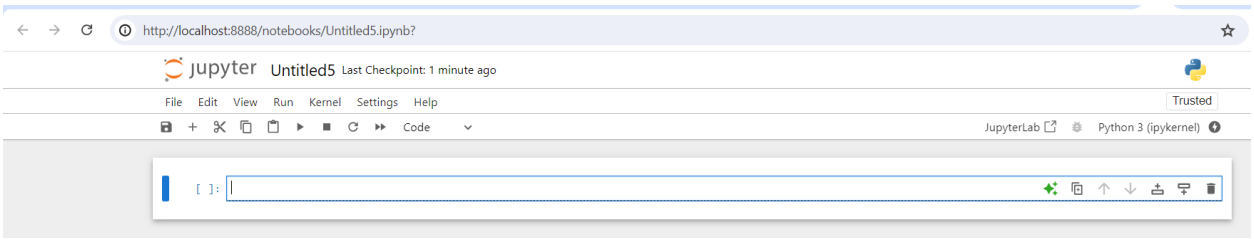


```
[I 2024-07-24 23:36:47.244 ServerApp] jupyterlab | extension was successfully linked.
[I 2024-07-24 23:36:47.256 ServerApp] notebook | extension was successfully linked.
[I 2024-07-24 23:36:47.913 ServerApp] notebook_shim | extension was successfully linked.
[I 2024-07-24 23:36:47.914 ServerApp] panel.io.jupyter_server_extension | extension was successfully linked.
[I 2024-07-24 23:36:47.968 ServerApp] notebook_shim | extension was successfully loaded.
[I 2024-07-24 23:36:47.970 ServerApp] Registered aext_assistant server extension
[I 2024-07-24 23:36:47.970 ServerApp] aext_assistant | extension was successfully loaded.
[I 2024-07-24 23:36:47.973 ServerApp] Registered aext_core server extension
[I 2024-07-24 23:36:47.974 ServerApp] aext_core | extension was successfully loaded.
[I 2024-07-24 23:36:47.978 ServerApp] Registered aext_panels server extension
[I 2024-07-24 23:36:47.978 ServerApp] aext_panels | extension was successfully loaded.
[I 2024-07-24 23:36:47.980 ServerApp] Registered aext_share_notebook_server server extension
[I 2024-07-24 23:36:47.980 ServerApp] aext_share_notebook | extension was successfully loaded.
[I 2024-07-24 23:36:47.986 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2024-07-24 23:36:47.987 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2024-07-24 23:36:47.997 LabApp] JupyterLab extension loaded from D:\ProgramFiles\Anaconda\anaconda3\Lib\site-packages
\jupyterlab
[I 2024-07-24 23:36:47.997 LabApp] JupyterLab application directory is D:\ProgramFiles\Anaconda\anaconda3\share\jupyter\
lab
[I 2024-07-24 23:36:47.999 LabApp] Extension Manager is 'pypi'.
[I 2024-07-24 23:36:48.003 ServerApp] jupyterlab | extension was successfully loaded.
[I 2024-07-24 23:36:48.015 ServerApp] notebook | extension was successfully loaded.
[I 2024-07-24 23:36:48.016 ServerApp] panel.io.jupyter_server_extension | extension was successfully loaded.
[I 2024-07-24 23:36:48.018 ServerApp] Serving notebooks from local directory: C:\Users\hp
[I 2024-07-24 23:36:48.018 ServerApp] Jupyter Server 2.10.0 is running at:
[I 2024-07-24 23:36:48.019 ServerApp] http://localhost:8888/tree?token=1c42b1e14b12732ad71f7b4fe3e034a3252c6b7983fcb3ca
[I 2024-07-24 23:36:48.019 ServerApp]     http://127.0.0.1:8888/tree?token=1c42b1e14b12732ad71f7b4fe3e034a3252c6b7983fcb
3ca
[I 2024-07-24 23:36:48.022 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirm
ation).
```

It will open Jupyter application URL http://localhost:8888/tree? where press **New** button and select **Notebook** option.

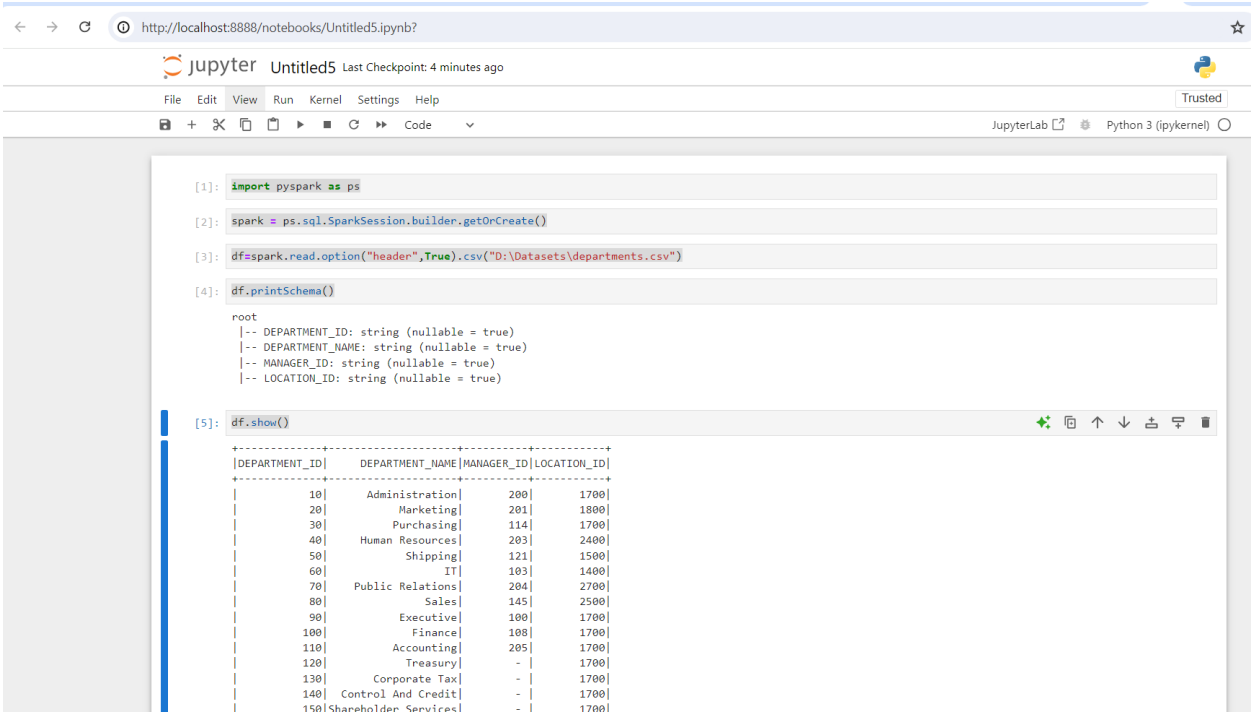


It opens an `Untitled.ipynb` application where we can execute the python code that runs on Spark engine.



Execute the following lines of code on the Jupyter notebook to create a Spark Session, read a CSV file `D:\Datasets\departments.csv` into a DataFrame and display the schema of DataFrame and its data.

```
import pyspark as ps
spark = ps.sql.SparkSession.builder.getOrCreate()
df=spark.read.option("header",True).csv("D:\Datasets\departments.csv")
df.printSchema()
df.show()
```



Congratulations!! You have installed Spark 3.5.1 version as a Standalone as well as on Hadoop cluster with Python API implementation successfully and executed Python Spark code and queries on Spark SQL and Beeline. You also got an exposure on how to integrate the Spark engine with Jupiter Notebook provided by Anaconda distribution.