Top Spark Interview Questions:

Q1) What is Apache Spark?

Apache Spark is an **Analytics engine for processing data at large-scale**. It provides high-level APIs (Application Programming Interface) in multiple programming languages like Java, Scala, Python and R. It provides an optimized engine that supports general execution of graphs. It also supports an upscale set of higher-level tools including Spark SQL for SQL and structured processing of data, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing.

Q2) What is an RDD in Apache Spark?

RDD Stands for **Resilient Distributed Dataset**. From a top-level perspective, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. **RDD is an abstract term provided by Spark, which means a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel so they automatically recover from node failures making them fault-tolerant**.

RDD's can be created in two ways:

1. **Parallelizing** an **existing collection** in your driver program.
2. **Referencing** a dataset from an **external storage system**, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop Input Format.

RDD's support two types of operations:

1. **Transformations**: which create a new dataset from an existing one, e.g.: MAP.
2. **Actions**: which return a value to the driver program after running a computation on the dataset. e.g.: REDUCE.

All transformations in Spark are lazy, meaning, they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.

One of the most important capabilities in Spark is **persisting** (or **caching**) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or

datasets derived from it). This allows future actions to be much faster (often by more than 10x).

Q3) Why use Spark on top of Hadoop?

While Apache Hadoop is a framework which allows us to **store and process big data** in a distributed environment, Apache Spark is only a **data processing engine** developed to provide **faster and easy-to-use analytics** than Hadoop MapReduce. So, we store data in the Hadoop File System and use YARN for resource allocation on top of which we use Spark for processing data fast. Hadoop Map Reduce can't process data fast and Spark doesn't have its own Data Storage so they both compensate for each other's drawbacks and come strong together.

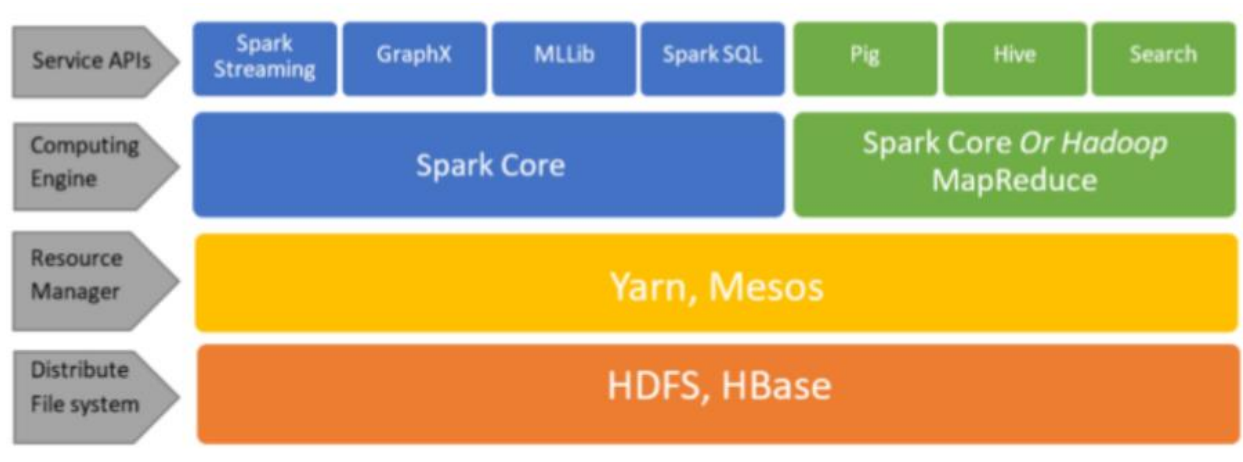Note: We can use Spark Core or Hadoop Map Reduce as a Computing Engine.

Image reference: Towards Data Science: Jeroen Schmidt.

Q4) How to install Spark on windows?

Prerequisites:

1. A system running Windows 10
2. A user account with administrator privileges (required to install software, modify file permissions, and modify system PATH)
3. Command Prompt or Powershell
4. A tool to extract .tar files, such as 7-Zip
5. Already installed Java
6. Already installed Python

Install Apache Spark on Windows

Step 1: Download Apache Spark

1. Open a browser and navigate to https://spark.apache.org/downloads.html.

2. Under the *Download Apache Spark* heading, there are two drop-down menus. Use the current non-preview version.

- In our case, in **Choose a Spark release** drop-down menu select **2.4.5 (Feb 05 2020)**.
- In the second drop-down **Choose a package type,** leave the selection **Pre-built for Apache Hadoop 2.7**.

3. Click the ***spark-2.4.5-bin-hadoop2.7.tgz*** link.
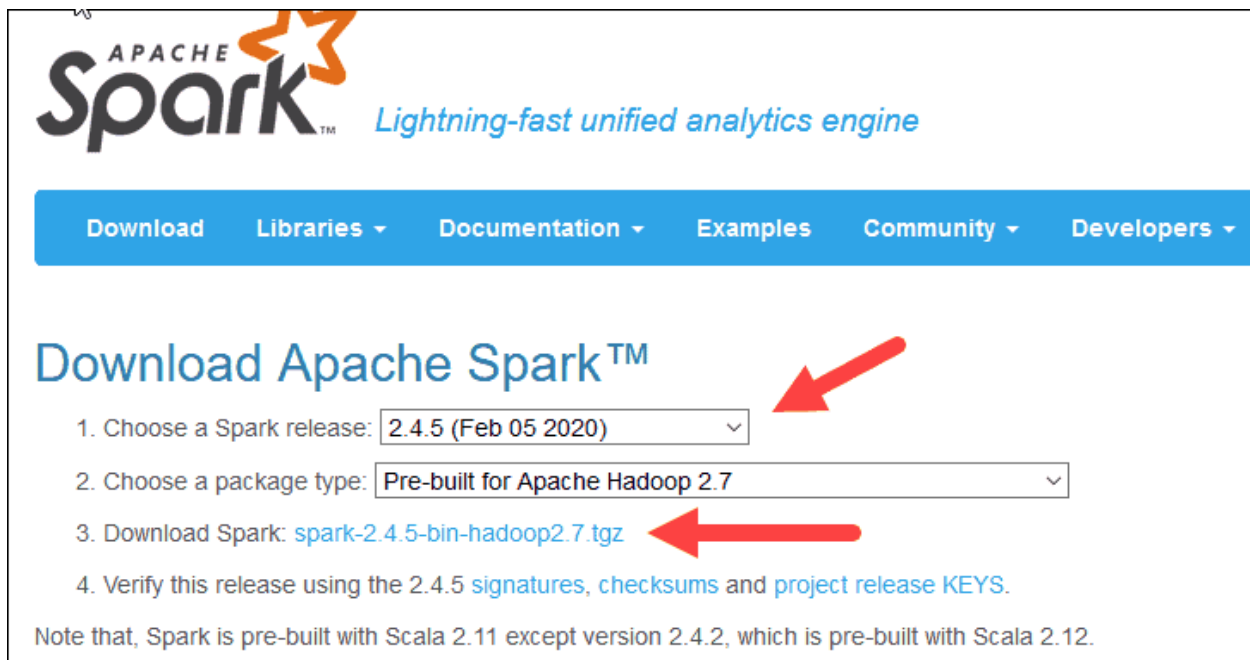
4. A page with a list of mirrors loads where you can see different servers to download from. Pick any from the list and save the file to your Downloads folder.
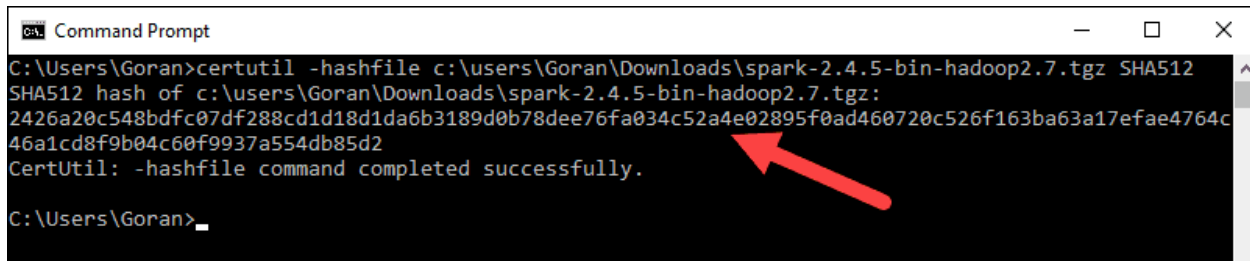
Step 2: Verify Spark Software File

1. Verify the integrity of your download by checking the **checksum** of the file. This ensures you are working with unaltered, uncorrupted software.

2. Navigate back to the *Spark Download* page and open the **Checksum** link, preferably in a new tab.

3. Next, open a command line and enter the following command:

certutil -hashfile c:\users\username\Downloads\spark-2.4.5-bin-hadoop2.7.tgz SHA512

4. Change the username to your username. The system displays a long alphanumeric code, along with the message **Certutil: -hashfile completed successfully**.

5. Compare the code to the one you opened in a new browser tab. If they match, your download file is uncorrupted.

Step 3: Install Apache Spark

Installing Apache Spark involves **extracting the downloaded file** to the desired location.

1. Create a new folder named *Spark* in the root of your C: drive. From a command line, enter the following:

cd \

mkdir Spark

2. In Explorer, locate the Spark file you downloaded.
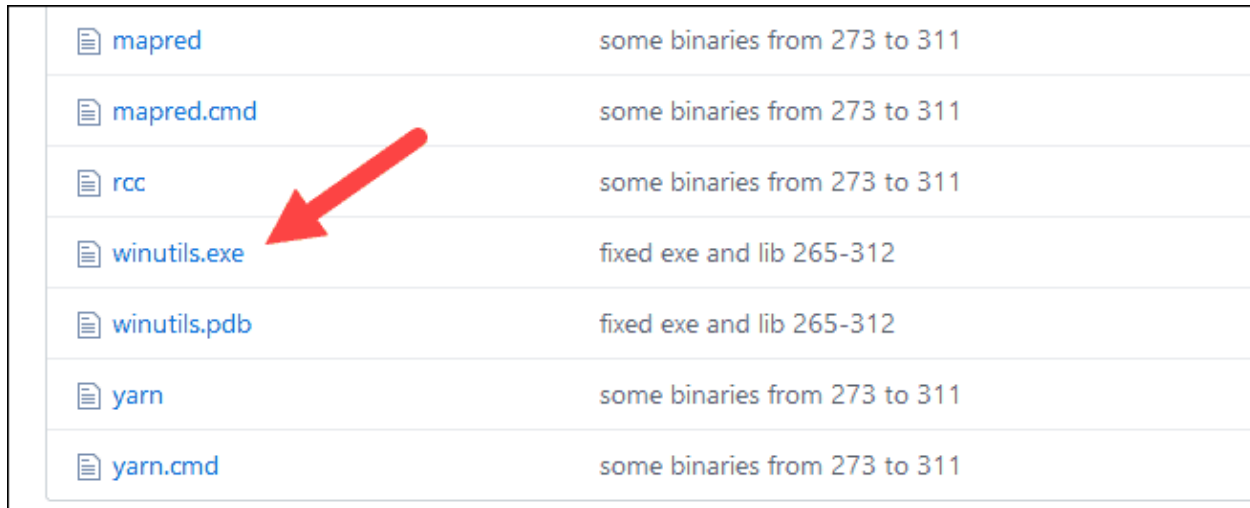
3. Right-click the file and extract it to *C:\Spark* using the tool you have on your system.

4. Now, your *C:\Spark* folder has a new folder *spark-2.4.5-bin-hadoop2.7* with the necessary files inside.

Step 4: Add winutils.exe File

Download the **winutils.exe** file for the underlying Hadoop version for the Spark installation you downloaded.

1. Navigate to this URL https://github.com/cdarlint/winutils and inside the **bin** folder, locate **winutils.exe**, and click it.

📄 mapred	some binaries from 273 to 311
📄 mapred.cmd	some binaries from 273 to 311
📄 rcc	some binaries from 273 to 311
📄 winutils.exe	fixed exe and lib 265-312
📄 winutils.pdb	fixed exe and lib 265-312
📄 yarn	some binaries from 273 to 311
📄 yarn.cmd	some binaries from 273 to 311

2. Find the **Download** button on the right side to download the file.

3. Now, create new folders *Hadoop* and **bin** on C: using Windows Explorer or the Command Prompt.

4. Copy the winutils.exe file from the Downloads folder to **C:\hadoop\bin**.

Step 5: Configure Environment Variables

This step adds the Spark and Hadoop locations to your system PATH. It allows you to run the Spark shell directly from a command prompt window.

1. Click **Start** and type *environment*.

2. Select the result labeled ***Edit the system environment variables***.

3. A System Properties dialog box appears. In the lower-right corner, click **Environment Variables** and then click **New** in the next window.
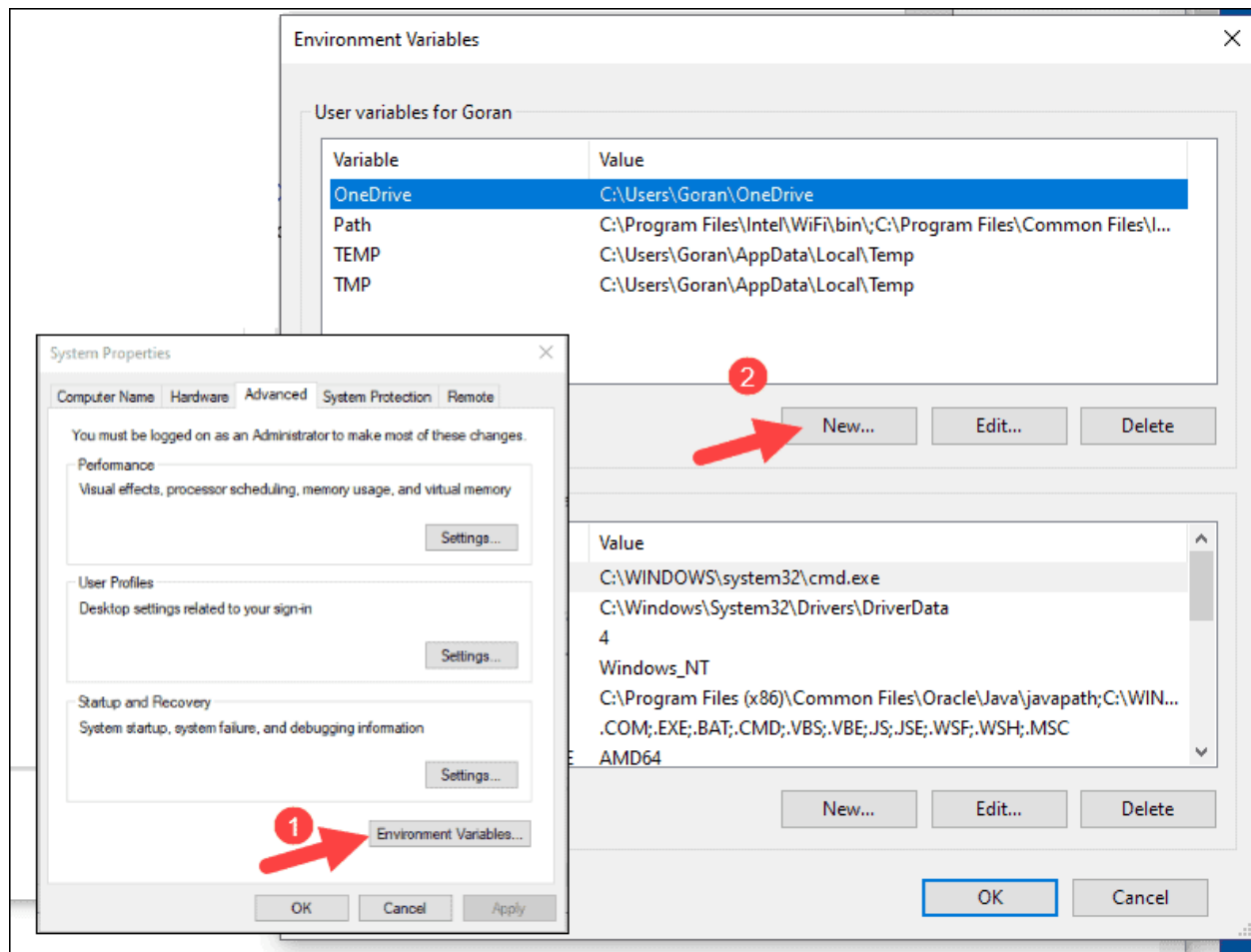
4. For *Variable Name* type **SPARK_HOME**.

5. For *Variable Value* type **C:\Spark\spark-2.4.5-bin-hadoop2.7** and click OK. If you changed the folder path, use that one instead.
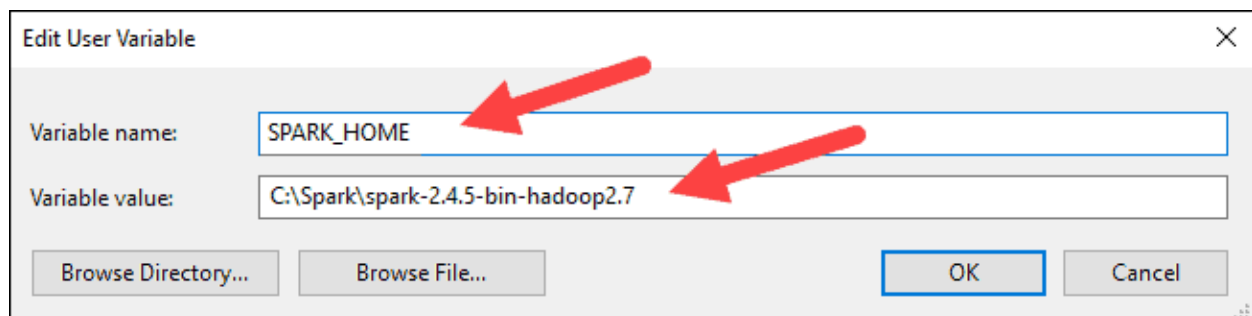
6. In the top box, click the **Path** entry, then click **Edit**. Be careful with editing the system path. Avoid deleting any entries already on the list.
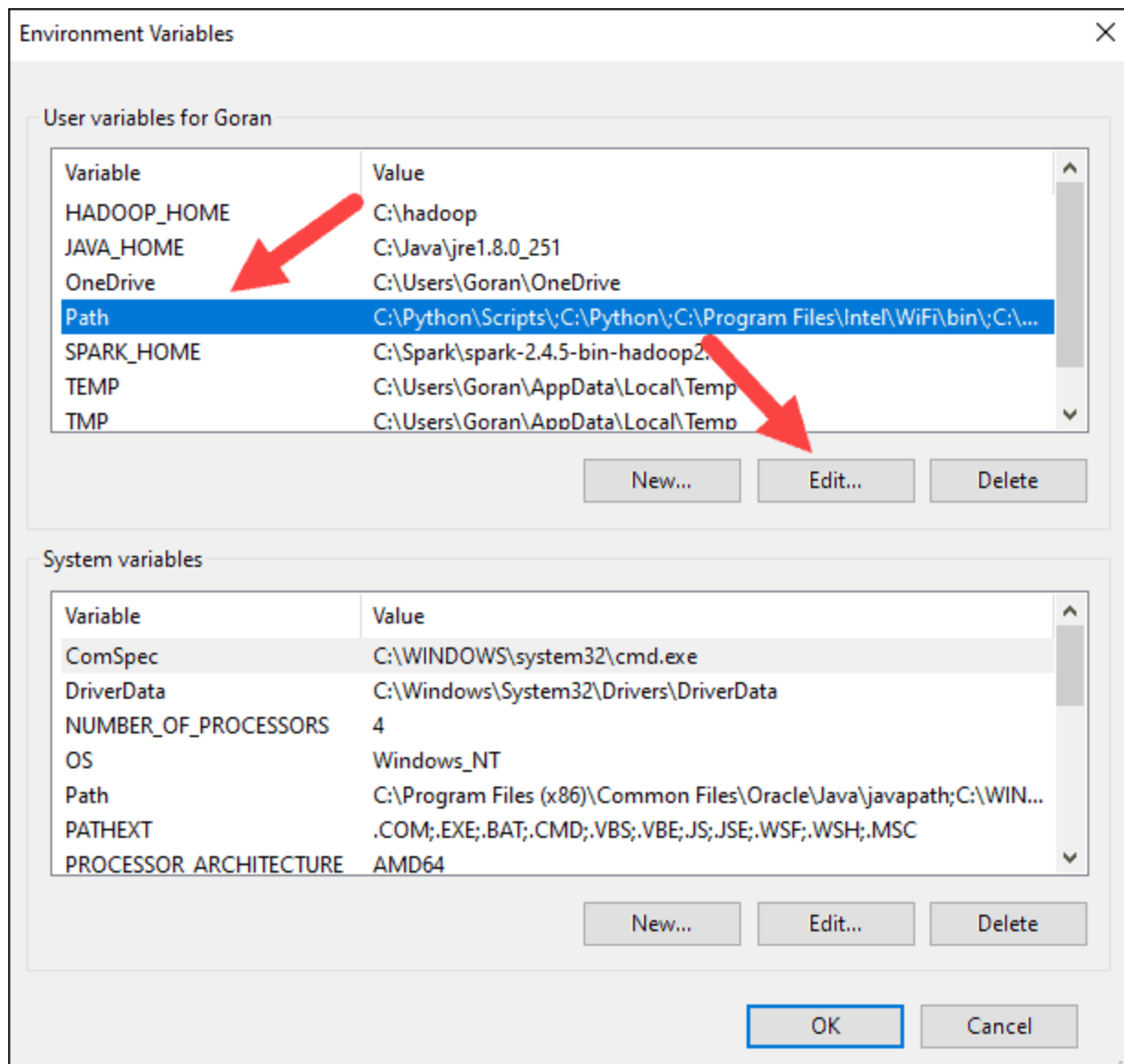
7. You should see a box with entries on the left. On the right, click **New**.

8. The system highlights a new line. Enter the path to the Spark folder **C:\Spark\spark-2.4.5-bin-hadoop2.7\bin**. We recommend using **%SPARK_HOME%\bin** to avoid possible issues with the path.
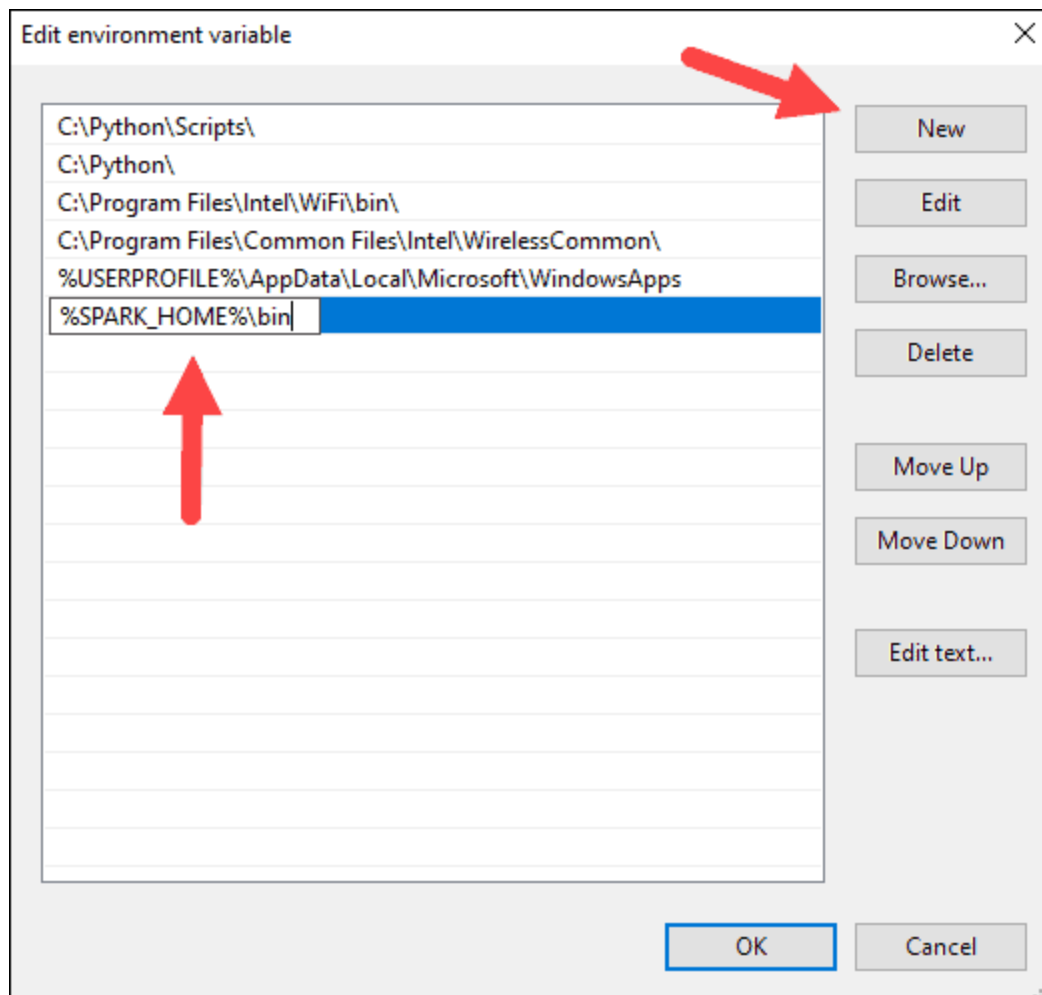
9. Repeat this process for Hadoop and Java.

- For Hadoop, the variable name is **HADOOP_HOME** and for the value use the path of the folder you created earlier: **C:\hadoop.** Add **C:\hadoop\bin** to the **Path variable** field, but we recommend using **%HADOOP_HOME%\bin**.
- For Java, the variable name is **JAVA_HOME** and for the value use the path to your Java JDK directory (in our case it's **C:\Program Files\Java\jdk1.8.0_251**).

10. Click **OK** to close all open windows.

Step 6: Launch Spark

1. Open a new command-prompt window using the right-click and **Run as administrator**:

2. To start Spark, enter:

C:\Spark\spark-2.4.5-bin-hadoop2.7\bin\spark-shell

If you set the **environment path** correctly, you can type **spark-shell** to launch Spark.

3. The system should display several lines indicating the status of the application. You may get a Java pop-up. Select **Allow access** to continue.
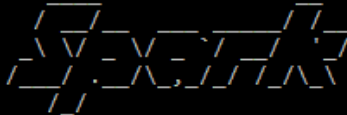
4. Finally, the Spark logo appears, and the prompt displays the **Scala shell**.

4. Open a web browser and navigate to **http://localhost:4040/**.

5. You can replace **localhost** with the name of your system.

6. You should see an Apache Spark shell Web UI. The example below shows the *Executors* page.

Executors

▸ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	To
Active(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0
Total(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0

Executors

Show 20 ▼ entries   Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Com
driver	DESKTOP-SFBGHOU:61547	Active	0	0.0 B / 434 MB	0.0 B	4	0	0	0

Showing 1 to 1 of 1 entries   Previous   1   Next

7. To exit Spark and close the Scala shell, press **ctrl-d** in the command-prompt window.

Step 7: Test Spark

In this example, we will launch the Spark shell and use Scala to read the contents of a file. You can use an existing file, such as the *README* file in the Spark directory, or you can create your own. We created *pnaptest* with some text.

1. Open a command-prompt window and navigate to the folder with the file you want to use and launch the Spark shell.

2. First, state a variable to use in the Spark context with the name of the file. Remember to add the file extension if there is any.

val x =sc.textFile("pnaptest")

3. The output shows an RDD is created. Then, we can view the file contents by using this command to call an action:

x.take(11).foreach(println)

This command instructs Spark to print 11 lines from the file you specified. To perform an action on this file (**value x**), add another value **y**, and do a map transformation.

4. For example, you can print the characters in reverse with this command:

val y = x.map(_.reverse)

5. The system creates a child RDD in relation to the first one. Then, specify how many lines you want to print from the value **y**:

y.take(11).foreach(println)

```
scala> y.take(11).foreach(println)
01 swodniW rof selbairaV tnemnorivnE krapS ehcapA

 EMOH_KRAPS :emaN elbairaV
7.2poodah-nib-5.4.2-kraps\krapS\:C :eulaV elbairaV
nib\%EMOH_KRAPS% :htaP

 EMOH_POODAH :emaN elbairaV
poodah\:C :eulaV elbairaV
nib\%EMOH_POODAH% :htaP

moc.panxineohp
```

The output prints 11 lines of the *pnaptest* file in the reverse order.

When done, exit the shell using **ctrl-d**.

Conclusion:

You should now have a working installation of Apache Spark on Windows 10 with all dependencies installed. Get started running an instance of Spark in your Windows environment.

Q5) What is a dag in spark?

(Directed Acyclic Graph) DAG in Apache Spark is a set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD.
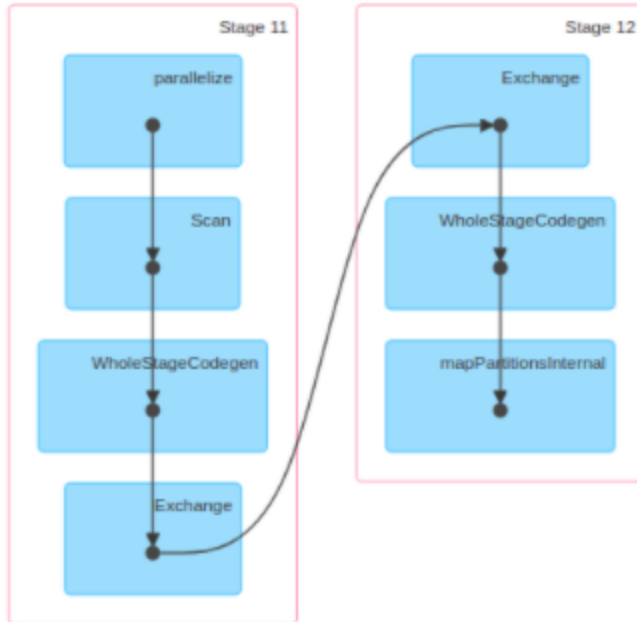
In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task.

Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage. In the stage view, the details of all RDDs belonging to that stage are expanded. The Scheduler splits the Spark RDD into stages based on various transformation applied. Each stage is comprised of tasks, based on the partitions of the RDD, which will perform same computation in parallel. The graph here refers to navigation, and directed and acyclic refers to how it is done.

The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications.

RDD Objects   DAGScheduler   TaskScheduler   Worker

Cluster manager

Threads

Block manager

DAG   TaskSet   Task

```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

split graph into *stages* of tasks

submit each stage as ready

launch tasks via cluster manager

retry failed or straggling tasks

execute tasks

store and serve blocks

agnostic to operators!

stage failed

doesn't know about stages

Following points will explain its working:

1. As you enter your code in spark console (creating RDD's and applying operators), Spark creates a operator graph.

2. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages/tasks.

3. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage. This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages.

4.The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies among stages.

5.The Worker executes the tasks. A new JVM is started per job. The worker knows only about the code that is passed to it.

Q6) What is a dataframe in spark?

A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame is simply a type alias of Dataset[Row]. While, in Java API, users need to use Dataset<Row> to represent a DataFrame.

Some of the key features of DataFrame in Spark are:

i. DataFrame is a distributed collection of data organized in named column. It is equivalent to the table in RDBMS.

ii. It can deal with both structured and unstructured data formats. For e.g. Avro, CSV, elastic search, and Cassandra. It also deals with storage systems HDFS, HIVE tables, MySQL, etc.

iv. The DataFrame API's are available in various programming languages. For e.g. Java, Scala, Python, and R.

v. It provides Hive compatibility. We can run unmodified Hive queries on existing Hive warehouse.

vi. It can scale from kilobytes of data on the single laptop to petabytes of data on a large cluster.

Q7) What is an action in spark?

Actions, which return a value to the driver program after running a computation on the dataset.

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. E.g. Count, Collect.

Q8) In which city did the first spark summit take place in 2013?

The first spark summit took place in Downtown San Francisco on Dec 2nd 2013.

Q9) Which tells spark how and where to access a cluster?

The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster. To create a SparkContext you first need to build a SparkConf object that contains information about your application.

Only one SparkContext may be active per JVM. You must stop() the active SparkContext before creating a new one.

val conf = new SparkConf().setAppName(appName).setMaster(master)

new SparkContext(conf)

The appName parameter is a name for your application to show on the cluster UI. master is a Spark, Mesos or YARN cluster URL, or a special "local" string to run in local mode. In practice, when running on a cluster, you will not want to hardcode master in the program, but rather launch the application with spark-submit and receive it there. However, for local testing and unit tests, you can pass "local" to run Spark in-process.

Q10) What is an accumulator in spark?

There are two main abstractions in spark:

1. RDD's: collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.
2. Shared Variables: shared variables that can be used in parallel operations

By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared **variables**: *broadcast variables*, which can be used to cache a value in memory on all nodes, and **accumulators**, which are variables that are only "added" to, such as counters and sums.

Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.

Accumulators are variables that are only "added" to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can

be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

As a user, you can create named or unnamed accumulators. As seen in the image below, a named accumulator (in this instance counter) will display in the web UI for the stage that modifies that accumulator. Spark displays the value for each accumulator modified by a task in the "Tasks" table.

Accumulators in the Spark UI

Tracking accumulators in the UI can be useful for understanding the progress of running stages (NOTE: this is not yet supported in Python).

A numeric accumulator can be created by calling SparkContext.longAccumulator() or SparkContext.doubleAccumulator() to accumulate values of type Long or Double, respectively. Tasks running on a cluster can then add to it using the add method. However, they cannot read its value. Only the driver program can read the accumulator's value, using its value method.

The code below shows an accumulator being used to add up the elements of an array:

scala> val accum = sc.longAccumulator("My Accumulator")

accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))

scala> accum.value

res2: Long = 10

While this code used the built-in support for accumulators of type Long, programmers can also create their own types by subclassing AccumulatorV2. The AccumulatorV2 abstract class has several methods which one has to override: reset for resetting the accumulator to zero, add for adding another value into the accumulator, merge for merging another same-type accumulator into this one. Other methods that must be

overridden are contained in the API documentation. For example, supposing we had a MyVector class representing mathematical vectors, we could write:

```scala
class VectorAccumulatorV2 extends AccumulatorV2[MyVector, MyVector] {

  private val myVector: MyVector = MyVector.createZeroVector

  def reset(): Unit = {
    myVector.reset()
  }

  def add(v: MyVector): Unit = {
    myVector.add(v)
  }
  ...
}

// Then, create an Accumulator of this type:
val myVectorAcc = new VectorAccumulatorV2
// Then, register it into spark context:
sc.register(myVectorAcc, "MyVectorAcc1")
```

Note that, when programmers define their own type of AccumulatorV2, the resulting type can be different than that of the elements added.

For accumulator updates performed inside actions only, Spark guarantees that each task's update to the accumulator will only be applied once, i.e. restarted tasks will not

update the value. In transformations, users should be aware of that each task's update may be applied more than once if tasks or job stages are re-executed.

Accumulators do not change the lazy evaluation model of Spark. If they are being updated within an operation on an RDD, their value is only updated once that RDD is computed as part of an action. Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map().

Important FAQs

How do I prepare for a Spark Interview?

You can start by going through various blogs available online that provide you with important questions. These questions will help you gain confidence in some of the commonly asked questions. You can also read blogs on Apache Spark provided by sites like Great Learning that will help you brush up your knowledge.

What is Spark good for?

Spark is good for training machine learning algorithms, stream processing, data integration, and interactive analytics.

Is Spark hard to learn?

Learning Spark is not difficult if you have a brief understanding of python and other programming languages. The APIs are provided in Python, Java, and Scala. You can take up the Apache Spark course on Great Learning and start learning.

Why Spark is faster than MapReduce?

Spark uses RDDs, i.e., Resilient Distributed Datasets which support multiple map operations in the memory. MapReduce has to write down interim results to the disk. Hence, Spark is faster.

What is a Spark core?

Spark Core is the fundamental unit of the entire Spark project which provides all sorts of functionalities like scheduling, task-dispatching, and I/O operations, etc.

When should you not use Spark?

You should not use Spark if you want query per/sec to get data to put on the website to end-users. As for each request, Spark will load the file data to search for the one single record in it.

What is Spark SQL?

Spark SQL streamlines the process of querying data stored both in external sources and RDDs (Spark's distributed datasets). Spark SQL effectively blurs the lines between relational tables and RDDs.

What is RDD in spark?

RDD in Spark is Resilient Distributed Datasets; it enables Spark to have multiple map operations in the memory, making it very fast.