Expt. No. 9: Apache Spark Tutorial, What is Spark? Spark Installation, Spark Architecture, Spark Components What is RDD?, RDD Operations ,RDD Persistence RDD Shared Variables.

Prof. Dhanshree S.

Apache Spark

Apache Spark tutorial provides basic and advanced concepts of Spark. Our Spark tutorial is designed for beginners and professionals.

Spark is a unified analytics engine for large-scale data processing including built-in modules for SQL, streaming, machine learning and graph processing.

Our Spark tutorial includes all topics of Apache Spark with Spark introduction, Spark Installation, Spark Architecture, Spark Components, RDD, Spark real time examples and so on.

Prerequisite

Before learning Spark, you must have a basic knowledge of Hadoop.

Audience

Our Spark tutorial is designed to help beginners and professionals.

Problems

We assure you that you will not find any problem with this Spark tutorial. However, if there is any mistake, please post the problem in the contact form.

What is Spark?

Apache Spark is an open-source cluster computing framework. Its primary purpose is to handle the real-time generated data.

Spark was built on the top of the Hadoop MapReduce. It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives. So, Spark process the data much quicker than other alternatives.

History of Apache Spark

The Spark was initiated by Matei Zaharia at UC Berkeley's AMPLab in 2009. It was open sourced in 2010 under a BSD license.

In 2013, the project was acquired by Apache Software Foundation. In 2014, the Spark emerged as a Top-Level Apache Project.

Features of Apache Spark

- **Fast** - It provides high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.
- **Easy to Use** - It facilitates to write the application in Java, Scala, Python, R, and SQL. It also provides more than 80 high-level operators.
- **Generality** - It provides a collection of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming.
- **Lightweight** - It is a light unified analytics engine which is used for large scale data processing.

  Runs Everywhere - It can easily run on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.

Uses of Spark

- **Data integration:** The data generated by systems are not consistent enough to combine for analysis. To fetch consistent data from systems we can use processes like Extract, transform, and load (ETL). Spark is used to reduce the cost and time required for this ETL process.
- **Stream processing:** It is always difficult to handle the real-time generated data such as log files. Spark is capable enough to operate streams of data and refuses potentially fraudulent operations.

- **Machine learning:** Machine learning approaches become more feasible and increasingly accurate due to enhancement in the volume of data. As spark is capable of storing data in memory and can run repeated queries quickly, it makes it easy to work on machine learning algorithms.
- **Interactive analytics:** Spark is able to generate the respond rapidly. So, instead of running pre-defined queries, we can handle the data interactively.

Spark Installation

In this section, we will perform the installation of Spark. So, follow the below steps.

- Download the Apache Spark tar file. Click Here
- Unzip the downloaded tar file.

```
1. sudo tar -xzvf /home/codegyani/spark-2.4.1-bin-hadoop2.7.tgz
```

- Open the bashrc file.

```
1. sudo nano ~/.bashrc
```

- Now, copy the following spark path in the last.

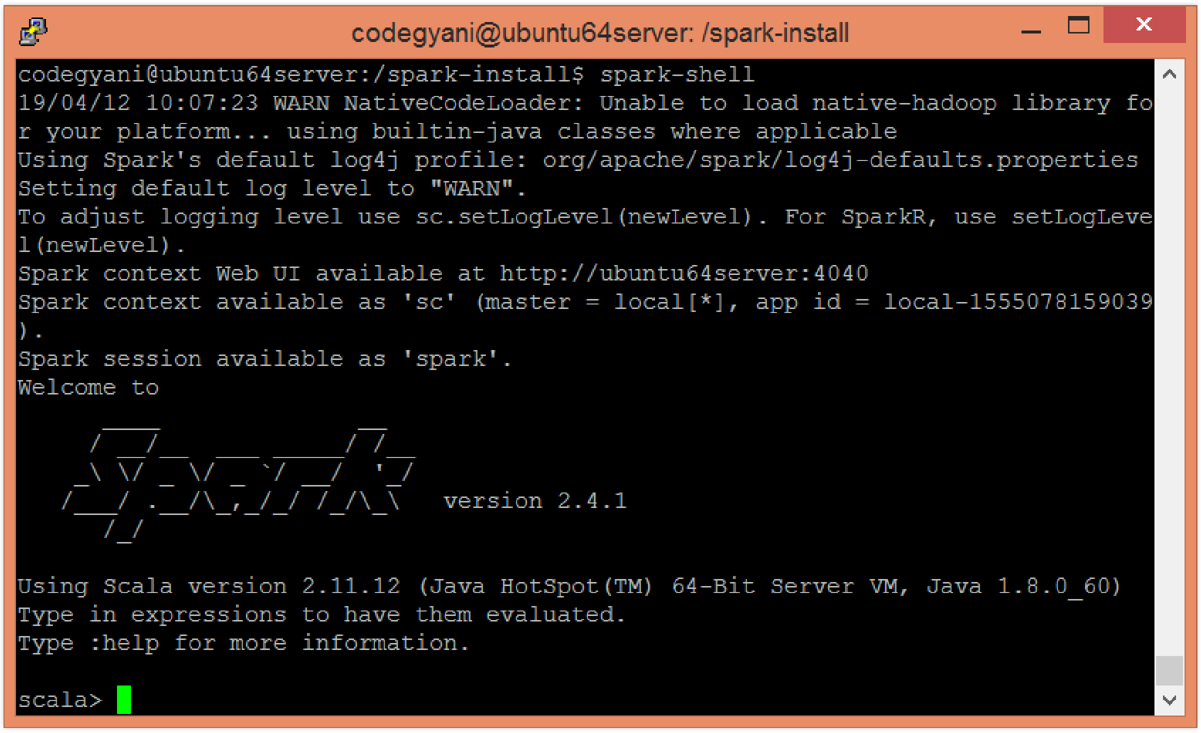
```
1. SPARK_HOME=/ home/codegyani /spark-2.4.1-bin-hadoop2.7
2. export PATH=$SPARK_HOME/bin:$PATH
```

- Update the environment variable

```
1. source ~/.bashrc
```

- Let's test the installation on the command prompt type

```
1. spark-shell
```



```
codegyani@ubuntu64server:/spark-install$ spark-shell
19/04/12 10:07:23 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
Spark context Web UI available at http://ubuntu64server:4040
Spark context available as 'sc' (master = local[*], app id = local-1555078159039
).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.1
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Note - Here, Spark is enabled in Scala mode.

Spark Architecture

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

- Resilient Distributed Dataset (RDD)
- Directed Acyclic Graph (DAG)

Resilient Distributed Datasets (RDD)

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

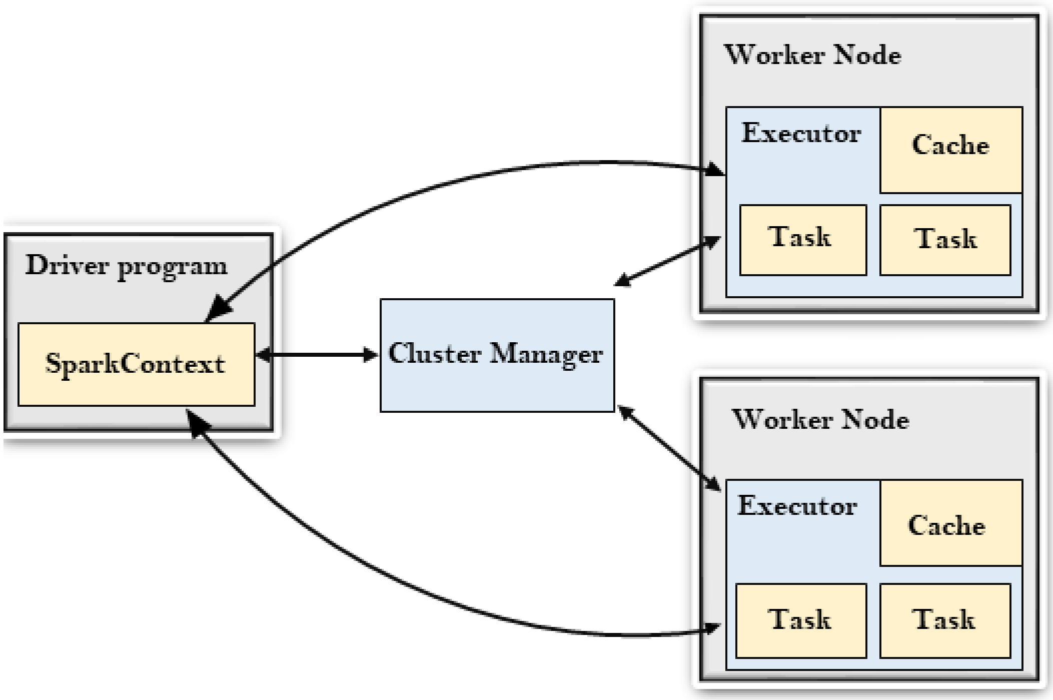
- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

We will learn about RDD later in detail.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.

Let's understand the Spark architecture.



Driver Program

The Driver Program is a process that runs the main() function of the application and creates the **SparkContext** object. The purpose of **SparkContext** is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the **SparkContext** connects to a different type of cluster managers and then perform the following tasks: -

- It acquires executors on nodes in the cluster.
- Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
- At last, the SparkContext sends tasks to the executors to run.

Cluster Manager

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.

- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

- The worker node is a slave node
- Its role is to run the application code in the cluster.

Executor

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.
- Every application contains its executor.

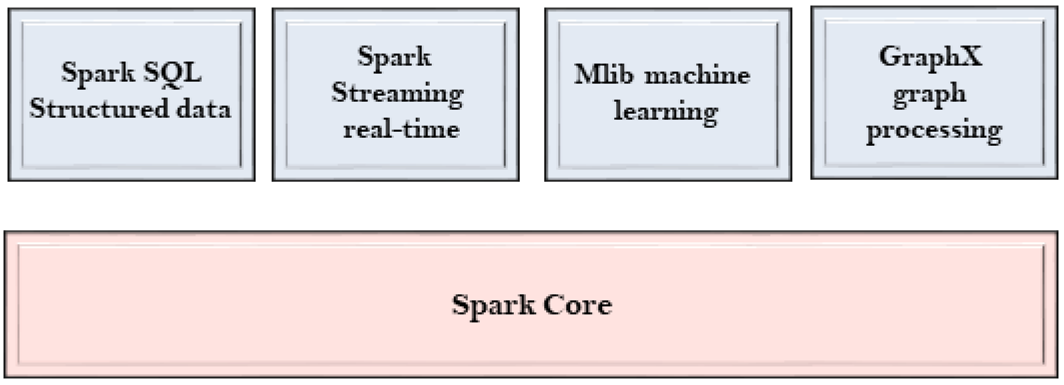
Task

- A unit of work that will be sent to one executor.

Spark Components

The Spark project consists of different types of tightly integrated components. At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.

Let's understand each Spark component in detail.



Spark Core

- The Spark Core is the heart of Spark and performs the core functionality.

- It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.

Spark SQL

- The Spark SQL is built on the top of Spark Core. It provides support for structured data.
- It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language).
- It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.
- It also supports various sources of data like Hive tables, Parquet, and JSON.

Spark Streaming

- Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
- It uses Spark Core's fast scheduling capability to perform streaming analytics.
- It accepts data in mini-batches and performs RDD transformations on that data.
- Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
- The log files generated by web servers can be considered as a real-time example of a data stream.

MLlib

- The MLlib is a Machine Learning library that contains various machine learning algorithms.
- These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
- It is nine times faster than the disk-based implementation used by Apache Mahout.

GraphX

- The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
- It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
- To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

What is RDD?

The RDD (Resilient Distributed Dataset) is the Spark's core abstraction. It is a collection of elements, partitioned across the nodes of the cluster so that we can execute various parallel operations on it.

There are two ways to create RDDs:

- Parallelizing an existing data in the driver program
- Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

Parallelized Collections

To create parallelized collection, call **SparkContext's** parallelize method on an existing collection in the driver program. Each element of collection is copied to form a distributed dataset that can be operated on in parallel.

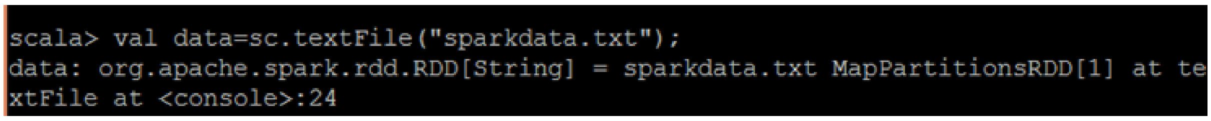
```
val info = Array(1, 2, 3, 4)
val distinfo = sc.parallelize(info)
```

Now, we can operate the distributed dataset (distinfo) parallel such like distinfo.reduce((a, b) => a + b).

External Datasets

In Spark, the distributed datasets can be created from any type of storage sources supported by Hadoop such as HDFS, Cassandra, HBase and even our local file system. Spark provides the support for text files, **SequenceFiles**, and other types of Hadoop **InputFormat**.

SparkContext's textFile method can be used to create RDD's text file. This method takes a URI for the file (either a local path on the machine or a hdfs://) and reads the data of the file.



```
scala> val data=sc.textFile("sparkdata.txt");
data: org.apache.spark.rdd.RDD[String] = sparkdata.txt MapPartitionsRDD[1] at te
xtFile at <console>:24
```

Now, we can operate data on by dataset operations such as we can add up the sizes of all the lines using the map and reduceoperations as follows: data.map(s => s.length).reduce((a, b) => a + b).

Transformation	Description
map(func)	It returns a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	It returns a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Here, each input item can be mapped to zero or more output items, so func should return a sequence rather than a single item.
mapPartitions(func)	It is similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(func)	It is similar to mapPartitions that provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	It samples the fraction fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	It returns a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	It returns a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numPartitions]))	It returns a new dataset that contains the distinct elements of the source dataset.
groupByKey([numPartitions])	It returns a dataset of (K, Iterable) pairs when called on a dataset of (K, V) pairs.
reduceByKey(func, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.
sortByKey([ascending], [numPartitions])	It returns a dataset of key-value pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable, Iterable)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script.
coalesce(numPartitions)	It decreases the number of partitions in the RDD to numPartitions.
repartition(numPartitions)	It reshuffles the data in the RDD randomly to create either more or fewer partitions and balance it across them.
repartitionAndSortWithinPartitions(partitioner)	It repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their

	keys.

RDD Operations

The RDD provides the two types of operations:

- Transformation
- Action

Transformation

In Spark, the role of transformation is to create a new dataset from an existing one. The transformations are considered lazy as they only computed when an action requires a result to be returned to the driver program.

Let's see some of the frequently used RDD Transformations.

Action

In Spark, the role of action is to return a value to the driver program after running a computation on the dataset.

Let's see some of the frequently used RDD Actions.

Action	Description
reduce(func)	It aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	It returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	It returns the number of elements in the dataset.
first()	It returns the first element of the dataset (similar to take(1)).
take(n)	It returns an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	It returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	It returns the first n elements of the RDD using either their natural order or a custom comparator.

Storage Level	Description
MEMORY_ONLY	It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed

saveAsSequenceFile(path) (Java and Scala)	It is used to write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system.
saveAsObjectFile(path) (Java and Scala)	It is used to write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	It is only available on RDDs of type (K, V). Thus, it returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	It runs a function func on each element of the dataset for side effects such as updating an Accumulator or interacting with external storage systems.

RDD Persistence

Spark provides a convenient way to work on the dataset by persisting it in memory across operations. While persisting an RDD, each node stores any partitions of it that it computes in memory. Now, we can also reuse them in other tasks on that dataset.

We can use either persist() or cache() method to mark an RDD to be persisted. Spark?s cache is fault-tolerant. In any case, if the partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

There is an availability of different storage levels which are used to store persisted RDDs. Use these levels by passing a **StorageLevel** object (Scala, Java, Python) to persist(). However, the cache() method is used for the default storage level, which is StorageLevel.MEMORY_ONLY.

The following are the set of storage levels:

	each time they're needed.
MEMORY_AND_DISK	It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	It stores RDD as serialized Java objects (i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.
MEMORY_AND_DISK_SER (Java and Scala)	It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.
DISK_ONLY	It stores the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2 , etc.	It is the same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.

RDD Shared Variables

In Spark, when any function passed to a transformation operation, then it is executed on a remote cluster node. It works on different copies of all the variables used in the function. These variables are copied to each machine, and no updates to the variables on the remote machine are revert to the driver program.

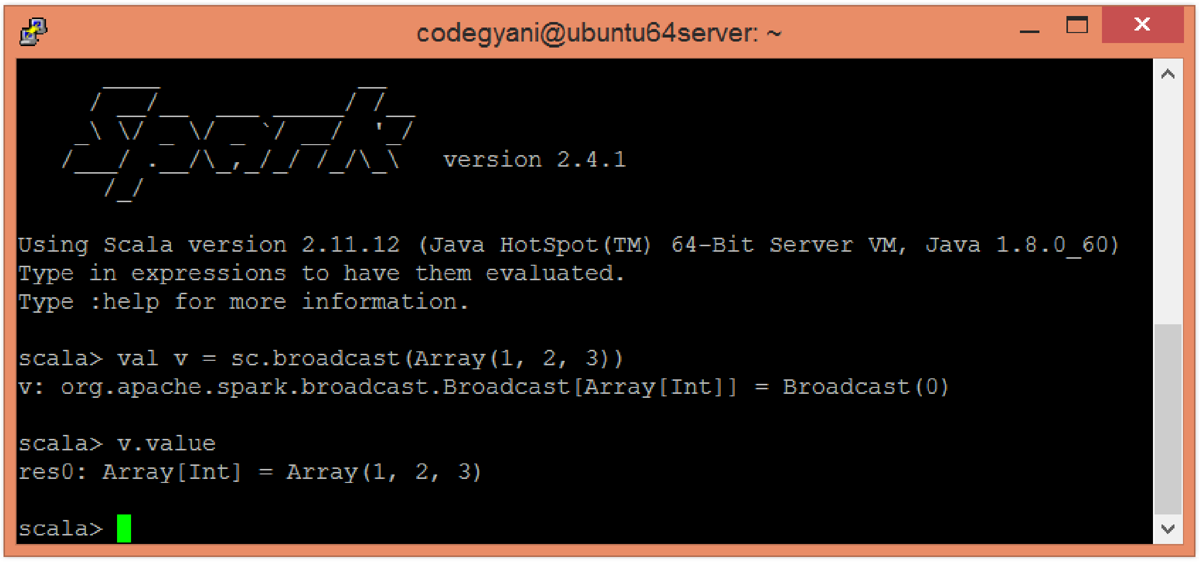
Broadcast variable

The broadcast variables support a read-only variable cached on each machine rather than providing a copy of it with tasks. Spark uses broadcast algorithms to distribute broadcast variables for reducing communication cost.

The execution of spark actions passes through several stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data required by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task.

To create a broadcast variable (let say, v), call SparkContext.broadcast(v). Let's understand with an example.

1. scala> val v = sc.broadcast(Array(1, 2, 3))
2. scala> v.value



codegyani@ubuntu64server: ~

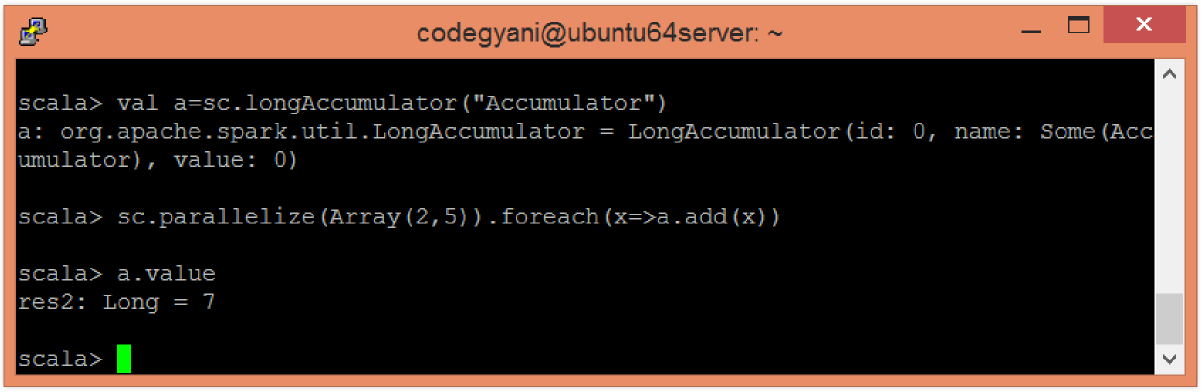
```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.1
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val v = sc.broadcast(Array(1, 2, 3))
v: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> v.value
res0: Array[Int] = Array(1, 2, 3)

scala>
```

Accumulator

The Accumulator are variables that are used to perform associative and commutative operations such as counters or sums. The Spark provides support for accumulators of numeric types. However, we can add support for new types.

To create a numeric accumulator, call SparkContext.longAccumulator() or SparkContext.doubleAccumulator() to accumulate the values of Long or Double type.

1. scala> val a=sc.longAccumulator("Accumulator")
2. scala> sc.parallelize(Array(2,5)).foreach(x=>a.add(x))
3. scala> a.value



codegyani@ubuntu64server: ~

```
scala> val a=sc.longAccumulator("Accumulator")
a: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(Acc
umulator), value: 0)

scala> sc.parallelize(Array(2,5)).foreach(x=>a.add(x))

scala> a.value
res2: Long = 7

scala>
```