Apache Spark - Quick Guide

Previous Page Next Page

Apache Spark - Introduction

Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.

As against a common belief, **Spark is not a modified version of Hadoop** and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways – one is **storage** and second is **processing**. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

Apache Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

Features of Apache Spark

Apache Spark has following features.

Speed – Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write

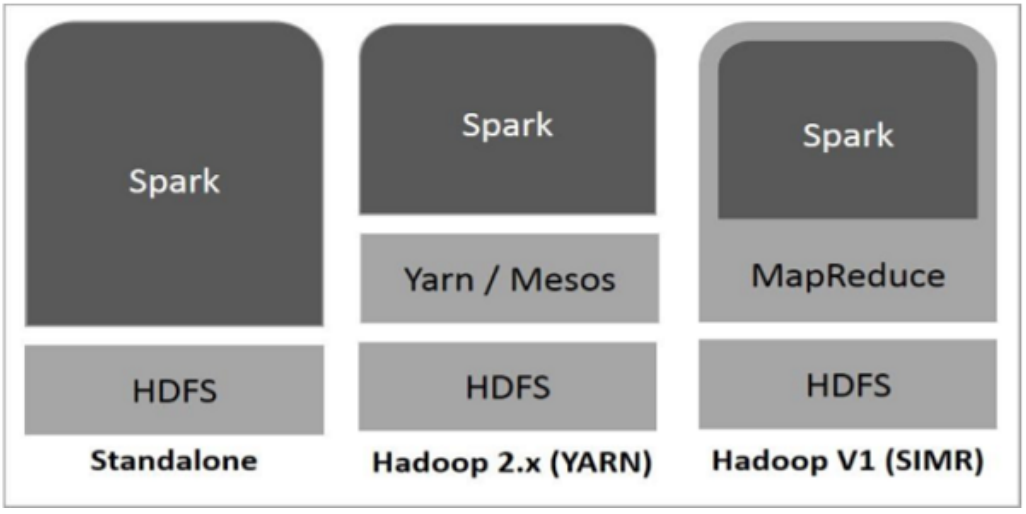
operations to disk. It stores the intermediate processing data in memory.

Supports multiple languages – Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.

Advanced Analytics – Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

Spark Built on Hadoop

The following diagram shows three ways of how Spark can be built with Hadoop components.



There are three ways of Spark deployment as explained below.

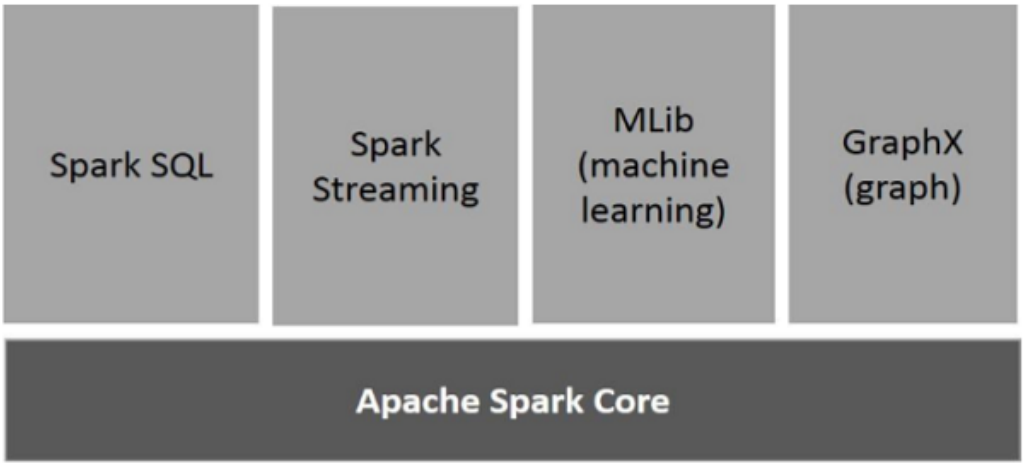
Standalone – Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.

Hadoop Yarn – Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.

Spark in MapReduce (SIMR) – Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

Components of Spark

The following illustration depicts the different components of Spark.



Apache Spark Core

Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

MLlib (Machine Learning Library)

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

GraphX

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

Apache Spark - RDD

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs – **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data Sharing is Slow in MapReduce

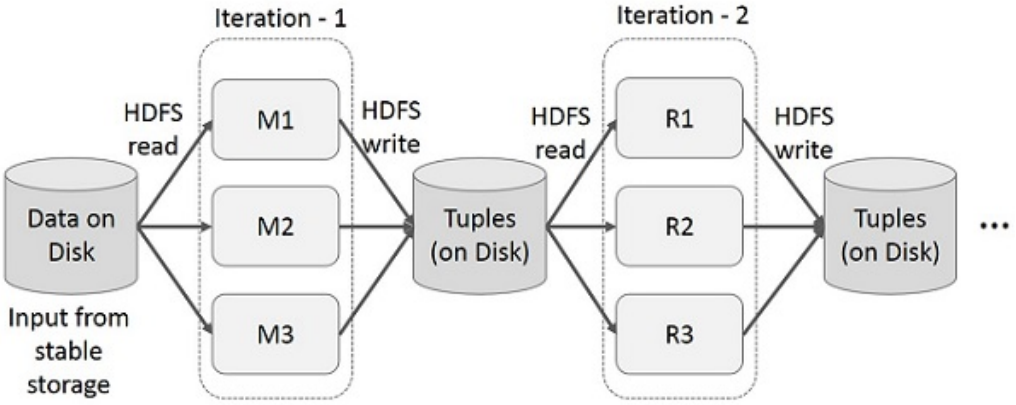
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex – between two MapReduce jobs) is to write it to an external stable storage system (Ex – HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

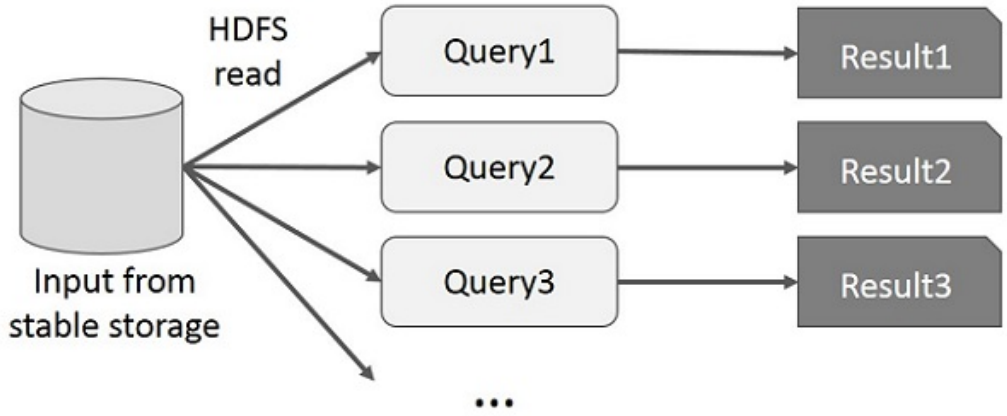
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominates application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

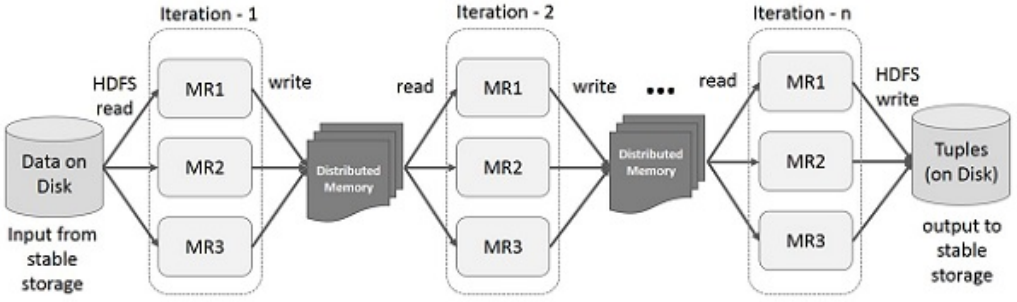
Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

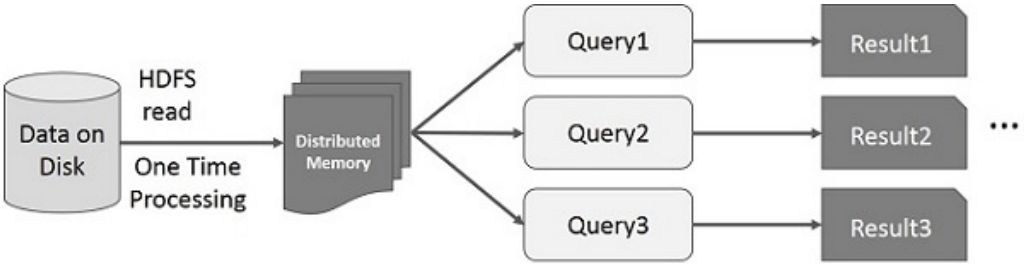
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note – If the Distributed memory (RAM) is sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster

for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

Apache Spark - Installation

Spark is Hadoop's sub-project. Therefore, it is better to install Spark into a Linux based system. The following steps show how to install Apache Spark.

Step 1: Verifying Java Installation

Java installation is one of the mandatory things in installing Spark. Try the following command to verify the JAVA version.

```
$java -version
```

If Java is already, installed on your system, you get to see the following response –

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

In case you do not have Java installed on your system, then Install Java before proceeding to next step.

Step 2: Verifying Scala installation

You should Scala language to implement Spark. So let us verify Scala installation using following command.

```
$scala -version
```

If Scala is already installed on your system, you get to see the following response –

```
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

In case you don't have Scala installed on your system, then proceed to next step for Scala installation.

Step 3: Downloading Scala

Download the latest version of Scala by visit the following link Download Scala . For this tutorial, we are using scala-2.11.6 version. After downloading, you will find the Scala tar file in the download folder.

Step 4: Installing Scala

Follow the below given steps for installing Scala.

Extract the Scala tar file

Type the following command for extracting the Scala tar file.

```
$ tar xvf scala-2.11.6.tgz
```

Move Scala software files

Use the following commands for moving the Scala software files, to respective directory **(/usr/local/scala)**.

```
$ su –
Password:
# cd /home/Hadoop/Downloads/
```

```
# mv scala-2.11.6 /usr/local/scala
# exit
```

Set PATH for Scala

Use the following command for setting PATH for Scala.

```
$ export PATH = $PATH:/usr/local/scala/bin
```

Verifying Scala Installation

After installation, it is better to verify it. Use the following command for verifying Scala installation.

```
$scala -version
```

If Scala is already installed on your system, you get to see the following response –

```
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

Step 5: Downloading Apache Spark

Download the latest version of Spark by visiting the following link Download Spark . For this tutorial, we are using **spark-1.3.1-bin-hadoop2.6** version. After downloading it, you will find the Spark tar file in the download folder.

Step 6: Installing Spark

Follow the steps given below for installing Spark.

Extracting Spark tar

The following command for extracting the spark tar file.

```
$ tar xvf spark-1.3.1-bin-hadoop2.6.tgz
```

Moving Spark software files

The following commands for moving the Spark software files to respective directory **(/usr/local/spark)**.

```
$ su –
Password:

# cd /home/Hadoop/Downloads/
# mv spark-1.3.1-bin-hadoop2.6 /usr/local/spark
# exit
```

Setting up the environment for Spark

Add the following line to **~/.bashrc** file. It means adding the location, where the spark software file are located to the PATH variable.

```
export PATH = $PATH:/usr/local/spark/bin
```

Use the following command for sourcing the ~/.bashrc file.

```
$ source ~/.bashrc
```

Step 7: Verifying the Spark Installation

Write the following command for opening Spark shell.

```
$spark-shell
```

If spark is installed successfully then you will find the following output.

```
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/06/04 15:25:22 INFO SecurityManager: Changing view acls to: hadoop
15/06/04 15:25:22 INFO SecurityManager: Changing modify acls to: hadoop
15/06/04 15:25:22 INFO SecurityManager: SecurityManager: authentication disabled;
   ui acls disabled; users with view permissions: Set(hadoop); users with modify permissions: Set(hadoop)
15/06/04 15:25:22 INFO HttpServer: Starting HTTP Server
15/06/04 15:25:23 INFO Utils: Successfully started service 'HTTP class server' on port 43292.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)
Type in expressions to have them evaluated.
Spark context available as sc
scala>
```

Apache Spark - Core Programming

Spark Core is the base of the whole project. It provides distributed task dispatching, scheduling, and basic I/O functionalities. Spark uses a specialized fundamental data structure known as RDD (Resilient Distributed Datasets) that is a logical collection of data partitioned across machines. RDDs can be created in two ways; one is by referencing datasets in external storage systems and second is by applying transformations (e.g. map, filter, reducer, join) on existing RDDs.

The RDD abstraction is exposed through a language-integrated API. This simplifies programming complexity because the way applications manipulate RDDs is similar to manipulating local collections of data.

Spark Shell

Spark provides an interactive shell – a powerful tool to analyze data interactively. It is available in either Scala or Python language. Spark’s primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop Input Formats (such as HDFS files) or by transforming other RDDs.

Open Spark Shell

The following command is used to open Spark shell.

```
$ spark-shell
```

Create simple RDD

Let us create a simple RDD from the text file. Use the following command to create a simple RDD.

```
scala> val inputfile = sc.textFile(“input.txt”)
```

The output for the above command is

```
inputfile: org.apache.spark.rdd.RDD[String] = input.txt MappedRDD[1] at textFile at <console>:12
```

The Spark RDD API introduces few **Transformations** and few **Actions** to manipulate RDD.

RDD Transformations

RDD transformations returns pointer to new RDD and allows you to create dependencies between RDDs. Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD.

Spark is lazy, so nothing will be executed unless you call some transformation or action that will trigger job creation and execution. Look at the following snippet of the word-count example.

Therefore, RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

> Given below is a list of RDD transformations.

Actions

> The following table gives a list of Actions, which return values.

S.No	Action & Meaning
1	**reduce(func)** 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.
2	**collect()** 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.
3	**count()** Returns the number of elements in the dataset.
4	**first()** Returns the first element of the dataset (similar to take (1)).
5	**take(n)** Returns an array with the first **n** elements of the dataset.
6	**takeSample (withReplacement,num, [seed])** 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.
7	

	takeOrdered(n, [ordering]) Returns the first **n** elements of the RDD using either their natural order or a custom comparator.
8	**saveAsTextFile(path)** Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.
9	**saveAsSequenceFile(path) (Java and Scala)** Writes 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. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
10	**saveAsObjectFile(path) (Java and Scala)** Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
11	**countByKey()** Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
12	**foreach(func)** Runs a function **func** on each element of the dataset. This is usually, done for side effects such as updating an Accumulator or interacting with external storage systems. **Note** – modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Programming with RDD

Let us see the implementations of few RDD transformations and actions in RDD programming with the help of an example.

Example

Consider a word count example – It counts each word appearing in a document. Consider the following text as an input and is saved as an **input.txt** file in a home directory.

input.txt – input file.

```
people are not as beautiful as they look,
as they walk or as they talk.
they are only as beautiful  as they love,
as they care as they share.
```

Follow the procedure given below to execute the given example.

Open Spark-Shell

The following command is used to open spark shell. Generally, spark is built using Scala. Therefore, a Spark program runs on Scala environment.

```
$ spark-shell
```

If Spark shell opens successfully then you will find the following output. Look at the last line of the output "Spark context available as sc" means the Spark container is automatically created spark context object with the name **sc**. Before starting the first step of a program, the SparkContext object should be created.

```
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/06/04 15:25:22 INFO SecurityManager: Changing view acls to: hadoop
15/06/04 15:25:22 INFO SecurityManager: Changing modify acls to: hadoop
15/06/04 15:25:22 INFO SecurityManager: SecurityManager: authentication disabled;
   ui acls disabled; users with view permissions: Set(hadoop); users with modify permissions: Set(hadoop)
15/06/04 15:25:22 INFO HttpServer: Starting HTTP Server
15/06/04 15:25:23 INFO Utils: Successfully started service 'HTTP class server' on port 43292.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)
Type in expressions to have them evaluated.
Spark context available as sc
scala>
```

Create an RDD

First, we have to read the input file using Spark-Scala API and create an RDD.

The following command is used for reading a file from given location. Here, new RDD is created with the name of inputfile. The String which is given as an argument in the textFile("") method is absolute path for the input file name. However, if only the file name is given, then it means that the input file is in the current location.

```
scala> val inputfile = sc.textFile("input.txt")
```

Execute Word count Transformation

Our aim is to count the words in a file. Create a flat map for splitting each line into words (**flatMap(line ⇒ line.split(" ")**).

Next, read each word as a key with a value **'1'** (<key, value> = <word,1>)using map function (**map(word ⇒ (word, 1)**).

Finally, reduce those keys by adding values of similar keys (**reduceByKey(_+_)**).

The following command is used for executing word count logic. After executing this, you will not find any output because this is not an action, this is a transformation; pointing a new RDD or tell spark to what to do with the given data)

```
scala> val counts = inputfile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_+_);
```

Current RDD

While working with the RDD, if you want to know about current RDD, then use the following command. It will show you the description about current RDD and its dependencies for debugging.

```
scala> counts.toDebugString
```

Caching the Transformations

You can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Use the following command to store the intermediate transformations in memory.

```
scala> counts.cache()
```

Applying the Action

Applying an action, like store all the transformations, results into a text file. The String argument for saveAsTextFile(" ") method is the absolute path of output folder. Try the following command to save the output in a text file. In the following example, 'output' folder is in current location.

```
scala> counts.saveAsTextFile("output")
```

Checking the Output

Open another terminal to go to home directory (where spark is executed in the other terminal). Use the following commands for checking output directory.

```
[hadoop@localhost ~]$ cd output/
[hadoop@localhost output]$ ls -1

part-00000
part-00001
_SUCCESS
```

The following command is used to see output from **Part-00000** files.

```
[hadoop@localhost output]$ cat part-00000
```

Output

```
(people,1)
(are,2)
(not,1)
(as,8)
(beautiful,2)
(they, 7)
(look,1)
```

The following command is used to see output from **Part-00001** files.

```
[hadoop@localhost output]$ cat part-00001
```

Output

```
(walk, 1)
(or, 1)
```

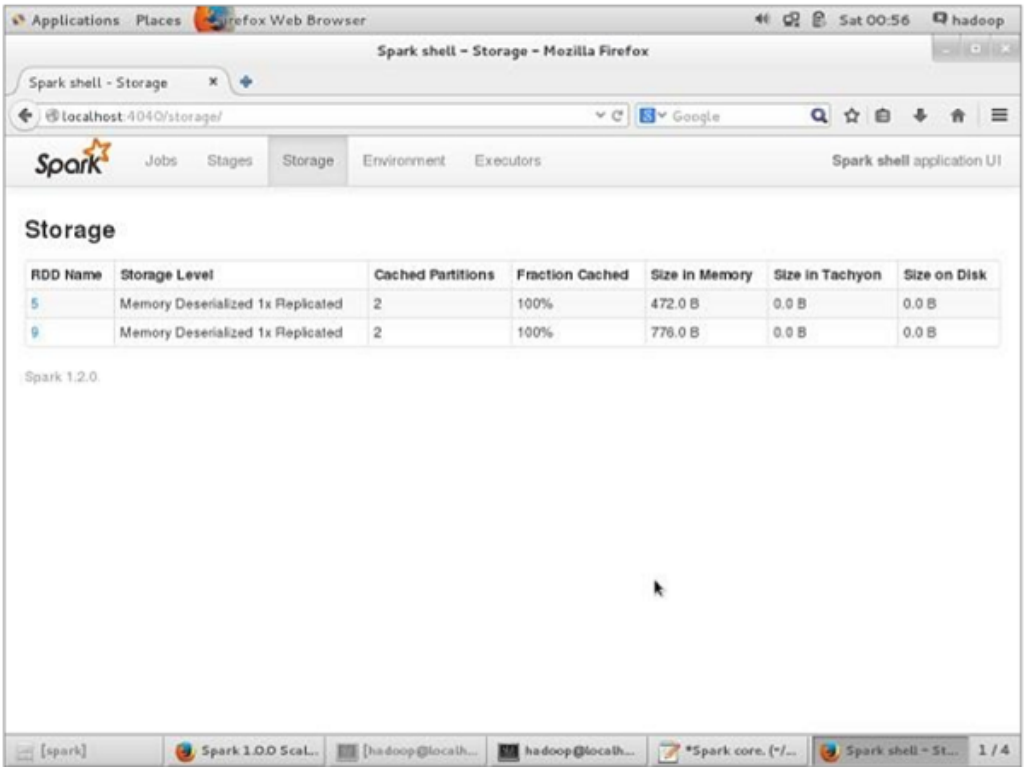
```
(talk, 1)
(only, 1)
(love, 1)
(care, 1)
(share, 1)
```

UN Persist the Storage

Before UN-persisting, if you want to see the storage space that is used for this application, then use the following URL in your browser.

```
http://localhost:4040
```

You will see the following screen, which shows the storage space used for the application, which are running on the Spark shell.



If you want to UN-persist the storage space of particular RDD, then use the following command.

```
Scala> counts.unpersist()
```

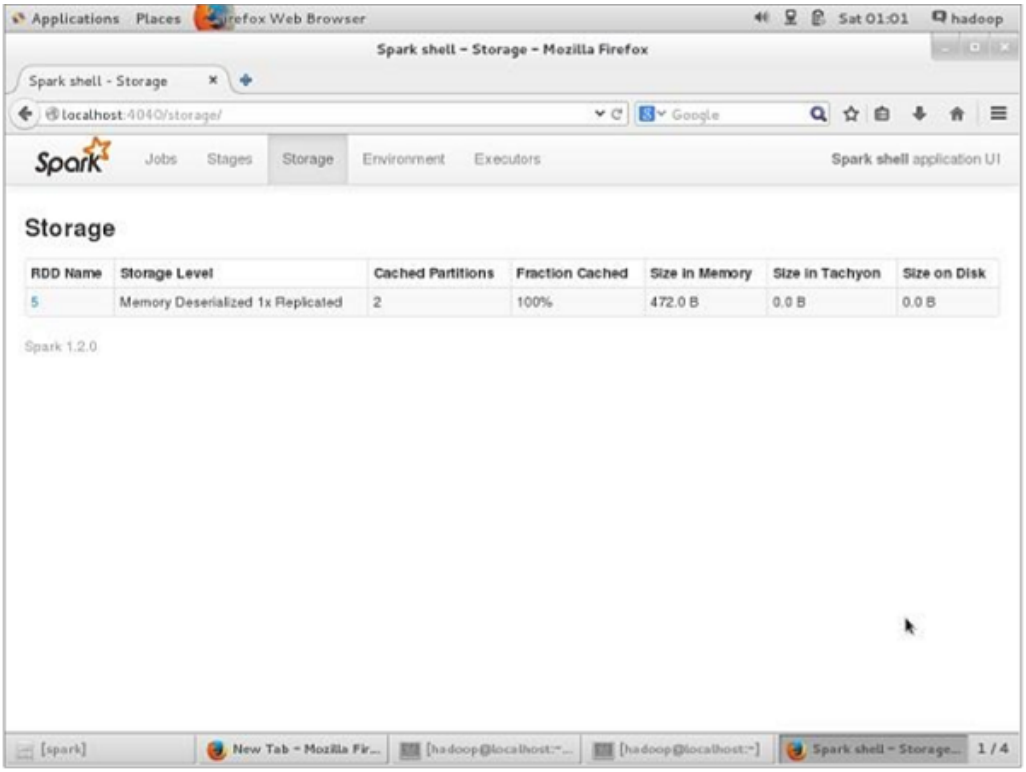
You will see the output as follows –

```
15/06/27 00:57:33 INFO ShuffledRDD: Removing RDD 9 from persistence list
15/06/27 00:57:33 INFO BlockManager: Removing RDD 9
15/06/27 00:57:33 INFO BlockManager: Removing block rdd_9_1
15/06/27 00:57:33 INFO MemoryStore: Block rdd_9_1 of size 480 dropped from memory (free 280061810)
15/06/27 00:57:33 INFO BlockManager: Removing block rdd_9_0
15/06/27 00:57:33 INFO MemoryStore: Block rdd_9_0 of size 296 dropped from memory (free 280062106)
res7: cou.type = ShuffledRDD[9] at reduceByKey at <console>:14
```

For verifying the storage space in the browser, use the following URL.

```
http://localhost:4040/
```

You will see the following screen. It shows the storage space used for the application, which are running on the Spark shell.



Apache Spark - Deployment

Spark application, using spark-submit, is a shell command used to deploy the Spark application on a cluster. It uses all respective cluster managers through a uniform interface. Therefore, you do not have to configure your application for each one.

Example

Let us take the same example of word count, we used before, using shell commands. Here, we consider the same example as a spark application.

Sample Input

The following text is the input data and the file named is **in.txt**.

```
people are not as beautiful as they look,
as they walk or as they talk.
they are only as beautiful  as they love,
as they care as they share.
```

Look at the following program –

SparkWordCount.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark._

object SparkWordCount {
   def main(args: Array[String]) {

      val sc = new SparkContext( "local", "Word Count", "/usr/local/spark", Nil, Map(), Map())

      /* local = master URL; Word Count = application name; */
      /* /usr/local/spark = Spark Home; Nil = jars; Map = environment */
      /* Map = variables to work nodes */
      /*creating an inputRDD to read text file (in.txt) through Spark context*/
      val input = sc.textFile("in.txt")
      /* Transform the inputRDD into countRDD */

      valcount = input.flatMap(line ⇒ line.split(" "))
```

```
      .map(word ⇒ (word, 1))
      .reduceByKey(_ + _)

      /* saveAsTextFile method is an action that effects on the RDD */
      count.saveAsTextFile("outfile")
      System.out.println("OK");
   }
}
```

Save the above program into a file named **SparkWordCount.scala** and place it in a user-defined directory named **spark-application**.

Note – While transforming the inputRDD into countRDD, we are using flatMap() for tokenizing the lines (from text file) into words, map() method for counting the word frequency and reduceByKey() method for counting each word repetition.

Use the following steps to submit this application. Execute all steps in the **spark-application** directory through the terminal.

Step 1: Download Spark Ja

Spark core jar is required for compilation, therefore, download spark-core_2.10-1.3.0.jar from the following link Spark core jar and move the jar file from download directory to **spark-application** directory.

Step 2: Compile program

Compile the above program using the command given below. This command should be executed from the spark-application directory. Here, **/usr/local/spark/lib/spark-assembly-1.4.0-hadoop2.6.0.jar** is a Hadoop support jar taken from Spark library.

```
$ scalac -classpath "spark-core_2.10-1.3.0.jar:/usr/local/spark/lib/spark-assembly-1.4.0-hadoop2.6.0.jar" SparkPi.scala
```

Step 3: Create a JAR

Create a jar file of the spark application using the following command. Here, **wordcount** is the file name for jar file.

```
jar -cvf wordcount.jar SparkWordCount*.class spark-core_2.10-1.3.0.jar/usr/local/spark/lib/spark-assembly-1.4.0-hadoop2.6
```

Step 4: Submit spark application

Submit the spark application using the following command –

```
spark-submit --class SparkWordCount --master local wordcount.jar
```

If it is executed successfully, then you will find the output given below. The **OK** letting in the following output is for user identification and that is the last line of the program. If you carefully read the following output, you will find different things, such as –

successfully started service 'sparkDriver' on port 42954

MemoryStore started with capacity 267.3 MB

Started SparkUI at http://192.168.1.217:4040

Added JAR file:/home/hadoop/piapplication/count.jar

ResultStage 1 (saveAsTextFile at SparkPi.scala:11) finished in 0.566 s

Stopped Spark web UI at http://192.168.1.217:4040

MemoryStore cleared

```
15/07/08 13:56:04 INFO Slf4jLogger: Slf4jLogger started
15/07/08 13:56:04 INFO Utils: Successfully started service 'sparkDriver' on port 42954.
15/07/08 13:56:04 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@192.168.1.217:42954]
15/07/08 13:56:04 INFO MemoryStore: MemoryStore started with capacity 267.3 MB
15/07/08 13:56:05 INFO HttpServer: Starting HTTP Server
15/07/08 13:56:05 INFO Utils: Successfully started service 'HTTP file server' on port 56707.
15/07/08 13:56:06 INFO SparkUI: Started SparkUI at http://192.168.1.217:4040
15/07/08 13:56:07 INFO SparkContext: Added JAR file:/home/hadoop/piapplication/count.jar at http://192.168.1.217:56707/ja
15/07/08 13:56:11 INFO Executor: Adding file:/tmp/spark-45a07b83-42ed-42b3b2c2-823d8d99c5af/userFiles-df4f4c20-a368-4cdd
15/07/08 13:56:11 INFO HadoopRDD: Input split: file:/home/hadoop/piapplication/in.txt:0+54
15/07/08 13:56:12 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 2001 bytes result sent to driver
 (MapPartitionsRDD[5] at saveAsTextFile at SparkPi.scala:11), which is now runnable
15/07/08 13:56:12 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 1 (MapPartitionsRDD[5] at saveAsTextFile
15/07/08 13:56:13 INFO DAGScheduler: ResultStage 1 (saveAsTextFile at SparkPi.scala:11) finished in 0.566 s
15/07/08 13:56:13 INFO DAGScheduler: Job 0 finished: saveAsTextFile at SparkPi.scala:11, took 2.892996 s
OK
15/07/08 13:56:13 INFO SparkContext: Invoking stop() from shutdown hook
15/07/08 13:56:13 INFO SparkUI: Stopped Spark web UI at http://192.168.1.217:4040
15/07/08 13:56:13 INFO DAGScheduler: Stopping DAGScheduler
15/07/08 13:56:14 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
15/07/08 13:56:14 INFO Utils: path = /tmp/spark-45a07b83-42ed-42b3-b2c2823d8d99c5af/blockmgr-ccdda9e3-24f6-491b-b509-3d15
15/07/08 13:56:14 INFO MemoryStore: MemoryStore cleared
15/07/08 13:56:14 INFO BlockManager: BlockManager stopped
15/07/08 13:56:14 INFO BlockManagerMaster: BlockManagerMaster stopped
15/07/08 13:56:14 INFO SparkContext: Successfully stopped SparkContext
15/07/08 13:56:14 INFO Utils: Shutdown hook called
15/07/08 13:56:14 INFO Utils: Deleting directory /tmp/spark-45a07b83-42ed-42b3b2c2-823d8d99c5af
15/07/08 13:56:14 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
```

Step 5: Checking output

After successful execution of the program, you will find the directory named **outfile** in the spark-application directory.

The following commands are used for opening and checking the list of files in the outfile directory.

```
$ cd outfile
$ ls
Part-00000 part-00001 _SUCCESS
```

The commands for checking output in **part-00000** file are –

```
$ cat part-00000
(people,1)
(are,2)
(not,1)
(as,8)
(beautiful,2)
(they, 7)
(look,1)
```

The commands for checking output in part-00001 file are –

```
$ cat part-00001
(walk, 1)
(or, 1)
```

```
(talk, 1)
(only, 1)
(love, 1)
(care, 1)
(share, 1)
```

Go through the following section to know more about the 'spark-submit' command.

Spark-submit Syntax

```
spark-submit [options] <app jar | python file> [app arguments]
```

Options

> The table given below describes a list of **options** –

Advanced Spark Programming

Spark contains two different types of shared variables – one is **broadcast variables** and second is **accumulators**.

Broadcast variables – used to efficiently, distribute large values.

Accumulators – used to aggregate the information of particular collection.

Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node, a copy of a large input dataset, in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage.

The data broadcasted this way is cached in serialized form and is deserialized before running each task. This means that explicitly creating broadcast variables, is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. The broadcast variable is a wrapper around **v**, and its value can be accessed by calling the **value** method. The code given below shows this –

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

Output –

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

After the broadcast variable is created, it should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once. In addition, the object **v** should not be modified after its broadcast, in order to ensure that all nodes get the same value of the broadcast variable.

Accumulators

Accumulators are variables that are only "added" to through an associative 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. If accumulators are created with a name, they will be displayed in **Spark's UI**. This can be useful for understanding the progress of running stages (NOTE – this is not yet supported in Python).

An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**. Tasks running on the cluster can then add to it using the **add** method or the += operator (in Scala and Python). However, they cannot read its value. Only the driver program can read the accumulator's value, using its **value** method.

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

```
scala> val accum = sc.accumulator(0)

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

If you want to see the output of above code then use the following command –

```
scala> accum.value
```

Output

```
res2: Int = 10
```

Numeric RDD Operations

Spark allows you to do different operations on numeric data, using one of the predefined API methods. Spark's numeric operations are implemented with a streaming algorithm that allows building the model, one element at a time.

These operations are computed and returned as a **StatusCounter** object by calling **status()** method.

> The following is a list of numeric methods available in **StatusCounter**.

If you want to use only one of these methods, you can call the corresponding method directly on RDD.

Previous Page Next Page