Apache Spark

UPDATED BY TIM SPANN BIG DATA SOLUTIONS ENGINEER, HORTONWORKS
WRITTEN BY ASHWINI KUNTAMUKKALA SOFTWARE ARCHITECT, SCISPIKE

CONTENTS

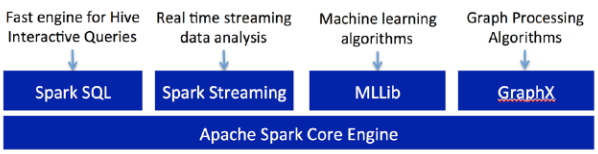
WHY APACHE SPARK?

Apache Spark has become the engine to enhance many of the capabilities of the ever-present Apache Hadoop environment. For Big Data, Apache Spark meets a lot of needs and runs natively on Apache Hadoop's YARN. By running Apache Spark in your Apache Hadoop environment, you gain all the security, governance, and scalability inherent to that platform. Apache Spark is also extremely well integrated with Apache Hive and gains access to all your Apache Hadoop tables utilizing integrated security.

Apache Spark has begun to really shine in the areas of streaming data processing and machine learning. With first-class support of Python as a development language, PySpark allows for data scientists, engineers and developers to develop and scale machine learning with ease. One of the features that has expanded this is the support for Apache Zeppelin notebooks to run Apache Spark jobs for exploration, data cleanup, and machine learning. Apache Spark also integrates with other important streaming tools in the Apache Hadoop space, namely Apache NiFi and Apache Kafka. I like to think of Apache Spark + Apache NiFi + Apache Kafka as the three amigos of Apache Big Data ingest and streaming. The latest version of Apache Spark is 2.2.

ABOUT APACHE SPARK

Apache Spark is an open source, Hadoop-compatible, fast and expressive cluster-computing data processing engine. It was created at AMPLabs in UC Berkeley as part of Berkeley Data Analytics Stack (BDAS). It is a top-level Apache project. The below figure shows the various components of the current Apache Spark stack.



It has six major benefits:

1. Lightning speed of computation because data are loaded in distributed memory (RAM) over a cluster of machines. Data can be quickly transformed iteratively and cached on demand for subsequent usage.
2. Highly accessible through standard APIs built in Java, Scala, Python, R, and SQL (for interactive queries) and has a rich set of machine learning libraries available out of the box.
3. Compatibility with existing Hadoop 2.x (YARN) ecosystems so companies can leverage their existing infrastructure.
4. Convenient download and installation processes. Convenient shell (REPL: Read-Eval-Print-Loop) to interactively learn the APIs.
5. Enhanced productivity due to high-level constructs that keep the focus on content of computation.
6. Multiple user notebook environments supported by Apache Zeppelin.

Also, Spark is implemented in Scala, which means that the code is very succinct and fast and requires JVM to run.

HOW TO INSTALL APACHE SPARK

The following table lists a few important links and prerequisites:

Current Release	2.2.0 @ apache.org/dyn/closer.lua/spark/spark-2.2.0/spark-2.2.0-bin-hadoop2.7.tgz
Downloads Page	spark.apache.org/downloads.html
JDK Version (Required)	1.8 or higher
Scala Version (Required)	2.11 or higher
Python (Optional)	[2.7, 3.5)
Simple Build Tool (Required)	scala-sbt.org
Development Version	github.com/apache/spark

Building Instructions	spark.apache.org/docs/latest/building-spark.html
Maven	3.3.9 or higher
Hadoop + Spark Installation	docs.hortonworks.com/HDPDocuments/Ambari-2.6.0.0/bk_ambari-i0nstallation/content/ch_Getting_Ready.html

Apache Spark can be configured to run standalone or on Hadoop 2 YARN. Apache Spark requires moderate skills in Java, Scala, or Python. Here we will see how to install and run Apache Spark in the standalone configuration.

1. Install JDK 1.8+, Scala 2.11+, Python 3.5+ and Apache Maven.
2. Download Apache Spark 2.2.0 Release.
3. Untar and unzip spark-2.2.0.tgz in a specified directory.
4. Go to the directory and run sbt to build Apache Spark.

```
export MAVEN_OPTS="-Xmx2g -XX:ReservedCodeCacheSize=512m"
mvn -Pyarn -Phadoop-2.7 -Dhadoop.version=2.7.3 -Phive
-Phive thriftserver -DskipTests clean package
```

5. Launch Apache Spark standalone REPL. For Scala, use:

```
./spark-shell
```

For Python, use:

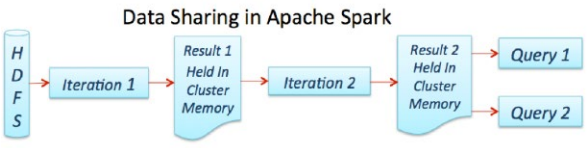
```
./pyspark
```

6. Go to SparkUI @ http://localhost:4040

This is a good quick start, but I recommend utilizing a Sandbox or an available Apache Zeppelin notebook to begin your exploration of Apache Spark.

HOW APACHE SPARK WORKS

The Apache Spark engine provides a way to process data in distributed memory over a cluster of machines. The figure below shows a logical diagram of how a typical Spark job processes information.



RESILIENT DISTRIBUTED DATASET

The core concept in Apache Spark is the resilient distributed dataset (RDD). It is an immutable distributed collection of data, which is partitioned across machines in a cluster. It facilitates two types of operations: transformations and actions. A transformation is an operation such as `filter()`, `map()`, or `union()` on an RDD that yields another RDD. An action is an operation such as `count()`, `first()`, `take(n)`, or `collect()` that triggers a computation, returns a value back to the Driver program, or writes to a stable storage system like Apache Hadoop HDFS. Transformations are lazily evaluated in that they don't run until an action warrants it. The Apache Spark Driver remembers the transformations applied to an RDD, so if a partition is lost (say a worker machine goes down), that partition can easily be reconstructed on some other machine in the cluster. That is why is it called "Resilient."

The following code snippets show how we can do this in Python using the Spark 2 PySpark shell.

```
%spark2.pyspark
guten = spark.read.text('/load/55973-0.txt')
```

COMMONLY USED TRANSFORMATIONS

Transformation & Purpose	Example & Result
`filter(func)` **Purpose:** new RDD by selecting those data elements on which `func` returns true	`shinto = guten.filter( guten.Variable contains("Shinto") )`

`map(func)`
Purpose: return new RDD by applying `func` on each data element
	`val rdd = sc.parallelize(List(1,2,3,4,5))`
	`val times2 = rdd.map(*2) times2. collect()`
	Result:
	`Array[Int] = Array(2, 4, 6, 8, 10)`

`flatMap(func)`
Purpose: Similar to `map` but `func` returns a sequence instead of a value. For example, mapping a sentence into a sequence of words
	`val rdd=sc. parallelize(List("Spark is awesome","It is fun"))`
	`val fm=rdd.flatMap(str=>str. split(" "))`
	`fm.collect()`
	Result:
	`Array[String] = Array(Spark, is, awesome, It, is, fun)`

`reduceByKey(-func,[numTasks])`
Purpose: To aggregate values of a key using a function. "numTasks" is an optional parameter to specify a number of reduce tasks
	`val word1=fm.map(word=>(word,1))`
	`val wrdCnt = word1.`
	`reduceByKey(_+_)wrdCnt.collect()`
	Result:
	`Array[(String, Int)] = Array((is,2), (It,1), (awesome,1), (Spark,1), (fun,1))`

`groupByKey([num-Tasks])`
Purpose: To convert (K,V) to (K,Iterable<V>)
	`val cntWrd = wrdCnt.map{case (word, count) => (count, word)}`
	`cntWrd.groupByKey().collect()`
	Result:
	`Array[(Int, Iterable[String])] = Array((1,ArrayBuffer(It, awesome, Spark, fun)), (2,ArrayBuffer(is)))`

`distinct([num-Tasks])`
Purpose: Eliminate duplicates from RDD
	`fm.distinct().collect()`
	Result:
	`Array[String] = Array(is, It, awesome, Spark, fun)`

COMMONLY USED SET OPERATIONS

Transformation & Purpose	Example & Result

`filter(func)`
Purpose: new RDD by selecting those data elements on which `func` returns true
	`val rdd1=sc.parallelize(List('A','B'))`
	`val rdd2=sc.parallelize(List('B','C'))`
	`rdd1.union(rdd2).collect()`
	Result:
	`Array[Char] = Array(A, B, B, C)`

`cartesian()`
Purpose: new RDD cross product of all elements from source RDD and argument
	`rdd1.cartesian(rdd2).collect()`
	Result:
	`Array[(Char, Char)] = Array((A,B), (A,C), (B,B), (B,C))`

`subtract()`
Purpose: new RDD created by removing data elements in source RDD in common with argument
	`rdd1.subtract(rdd2).collect()`
	Result:
	`Array[Char] = Array(A)`

`join(RDD,[num-Tasks])`
Purpose: When invoked on (K,V) and (K,W), this operation creates a new RDD of (K, (V,W))
	`val personFruit = sc.parallelize(Seq((“Andy”, “Apple”), (“Bob”, “Banana”), (“Charlie”, “Cherry”), (“Andy”,”Apricot”)))`
	`val personSE = sc.parallelize(Seq((“Andy”, “Google”), (“Bob”, “Bing”), (“Charlie”, “Yahoo”), (“Bob”,”AltaVista”)))`
	`personFruit.join(personSE).collect()`
	Result:
	`Array[(String, (String, String))] = Array((Andy,(Apple,Google)), (Andy,(Apricot,Google)), (Charlie,(Cherry,Yahoo)), (Bob,(Banana,Bing)), (Bob,(Banana,AltaVista)))`

cogroup(RDD,[num-Tasks])
Purpose: To convert (K,V) to (K,Iterable<V>)
	`personFruit.cogroup(personSe).collect()`
	Result:
	`Array[(String, (Iterable[String], Iterable[String]))] = Array((Andy,(ArrayBuffer(Apple, Apricot),ArrayBuffer(google))), (Charlie,(ArrayBuffer(Cherry), ArrayBuffer(Yahoo))), (Bob,(ArrayBuffer(Banana), ArrayBuffer(Bing, AltaVista))))`

For a more detailed list of transformations, please refer to spark.apache.org/docs/latest/programming-guide.html#transformations.

COMMONLY USED ACTIONS

Action & Purpose	Example & Result

`count()`
Purpose: get the number of data elements in the RDD
	`val rdd = sc.parallelize (list('A','B','c')) scala> rdd.count()`
	Result:
	`long = 3`

`collect()`
Purpose: get all the data elements in an RDD as an array
	`val rdd = sc.parallelize(list ('A','B','c'))rdd.collect()`
	Result:
	`Array[char] = Array(A, B, c)`

`reduce(func)`
Purpose: Aggregate the data elements in an RDD using this function which takes two arguments and returns one
	`val rdd = sc.parallelize(list(1,2,3,4))`
	`rdd.reduce(_+_)`
	Result:
	`Int = 10`

`take (n)`
Purpose: fetch first n data elements in an RDD, computed by driver program
	`val rdd = sc.parallelize(list(1,2,3,4))`
	`rdd.take(2)`
	Result:
	`Array[Int] = Array(1, 2)`

`foreach(func)`
Purpose: execute function for each data element in RDD, usually used to update an accumulator (discussed later) or interacting with external systems
	`val rdd = sc.parallelize(list(1,2,3,4))`
	`rdd.foreach(x=>println("%s*10=%s". format(x,x*10)))`
	Result:
	`1*10=10 4*10=40 3*10=30 2*10=20`

`first()`
Purpose: retrieves the first data element in RDD. Similar to take(1)
	`val rdd = sc.parallelize(list(1,2,3,4))`
	`rdd.first()`
	Result:
	`Int = 1`

`saveAsTextFile(path)`
Purpose: Writes the content of RDD to a text file or a set of text files to local file system/ HDFS
	`val hamlet = sc.textFile("/users/akuntamukkala/ temp/gutenburg.txt") hamlet.filter(_.contains("Shakespeare")).saveAsTextFile("/users/akuntamukkala/temp/ filtered")`
	Result:
	`akuntamukkala@localhost~/temp/filtered$ ls _SUCCESS part-00000 part-00001`

For a more detailed list of actions, please refer to spark.apache.org/docs/latest/programming-guide.html#actions.

RDD PERSISTENCE

One of the key capabilities in Apache Spark is persisting/caching RDD in cluster memory. This speeds up iterative computation.

The following table shows the various options Spark provides:

Storage Level	Purpose
`MEMORY_ONLY (Default level)`	This option stores RDD in available cluster memory as deserialized Java objects. Some partitions may not be cached if there is not enough cluster memory. Those partitions will be recalculated on the fly as needed.
`MEMORY_AND_DISK`	This option stores RDD as deserialized Java objects. If RDD does not fit in cluster memory, then store those partitions on the disk and read them as needed.
`MEMORY_ONLY_SER`	This options stores RDD as serialized Java objects (One byte array per partition). This is more CPU intensive but saves memory as it is more space efficient. Some partitions may not be cached. Those will be recalculated on the fly as needed.
`MEMORY_ONLY_DISK_SER`	This option is same as above except that disk is used when memory is not sufficient.
`DISK_ONLY`	This option stores the RDD only on the disk
`MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.`	Same as other levels but partitions are replicated on 2 slave nodes
`OFF_HEAP (experimental)`	Works off of JVM heap and must be enabled.

The above storage levels can be accessed through persist() operation on RDD. The `cache()` operation is a convenient way of specifying a `MEMORY_ONLY` option. The **SER** options do not work with Python.

For a more detailed list of persistence options, please refer to http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

Spark uses the Least Recently Used (LRU) algorithm to remove old, unused cached RDDs to reclaim memory. It also provides a convenient `unpersist()` operation to force removal of cached/persisted RDDs.

DATAFRAMES

Action & Purpose	Example & Result

`printSchema()`
Purpose: print out the schema of the DataFrame
	`df.printSchema()`
	Result:
	`root |-- clientIp: string (nullable = true) |-- clientIdentity: string (nullable = true) |-- user: string (nullable = true) |-- dateTime: string (nullable = true) |-- request: string (nullable = true) |-- statusCode: integer (nullable = true) |-- bytesSent: long (nullable = true) |-- referer: string (nullable = true) |-- userAgent: string (nullable = true)`

`collect()`
Purpose: returns all the records as a list of rows
	`df.collect()`
	Result:
	`Row(value=u'The Project Gutenberg EBook of Shinto: The ancient religion of Japan, by ')`

`columns()`
 	
Purpose: returns all the columns as a list
	`df.columns()`
	
	Result:
	`['userAgent', 'referer', 'bytesSent']`

`count()`
 	
Purpose: returns number of rows
	`df.count()`
	
	Result:
	`2`

`createTempView()` **Purpose:** create a local temporary view that can be used in Spark SQL	`df.createTempView("viewName")`

A DataFrame is a distributed collection of data with named columns built on the Dataset interface. You can learn more here: spark.apache.org/docs/latest/sql-programming-guide.html.

SHARED VARIABLES

ACCUMULATORS

Accumulators are variables that can be incremented in distributed tasks.

```
exampleAccumulator = sparkContext.accumulator(1)
exampleAccumulator.add(5)
```

BROADCAST VARIABLES

Using the SparkContext, you can broadcast a read-only value to other tasks. You can set, destroy, and unpersist these values.

```
broadcastVariable = sparkContext.broadcast(500)
broadcastVariable.value
```

SPARK SQL

Spark SQL provides a convenient way to run interactive queries over large data sets using Apache Spark Engine, returning DataFrames. Spark SQL provides two types of contexts, SQLContext and HiveContext, that extend SparkContext functionality.

SQLContext provides access to a simple SQL parser, whereas HiveContext provides access to the HiveQL parser. HiveContext enables enterprises to leverage their existing Hive infrastructure.

Let's see a simple example in Scala:

```
val df = spark.read.csv("customers.txt")
val dfS = spark.sql("select * from customers where gender='M'")
dfs.printSchema()
dfs.show()
```

Here's one in Python for Apache Hive:

```
spark = SparkSession.builder.appName("dzone1").config("spark.sql.
warehouse.dir", "/mydata").enableHiveSupport().getOrCreate()
spark.sql("SELECT * FROM default.myHiveTable")
```

For more practical examples using SQL & HiveQL, please refer to the following link: spark.apache.org/docs/latest/sql-programming-guide.html.

SPARK STREAMING

Spark Streaming provides a scalable, fault tolerant, efficient way of processing streaming data using Spark's simple programming model. It converts streaming data into "micro" batches, which enable Spark's batch programming model to be applied in Streaming use cases. This unified programming model makes it easy to combine batch and interactive data processing with streaming.

The core abstraction in Spark Streaming is Discretized Stream (DStream). DStream is a sequence of RDDs. Each RDD contains data received in a configurable interval of time.

Spark Streaming also provides sophisticated window operators, which help with running efficient computation on a collection of RDDs in a rolling window of time. DStream exposes an API, which contains operators (transformations and output operators) that are applied on constituent RDDs. Let's try and understand this using a simple example:

```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName("appName").
setMaster("masterNode")
val ssc = new StreamingContext(conf, Seconds(1))
val lines = ssc.socketTextStream("localhost", 9999)
```

The above snippet is setting up Spark Streaming Context. Spark Streaming will create an RDD in DStream containing text network streams retrieved every second.

There are many commonly used source data streams for Spark Streaming, including Apache Kafka, Apache HDFS, Twitter, Apache NiFi S2S, Amazon S3, and Amazon Kinesis.

Transformation & Purpose	Example & Result
`map(func)` **Purpose:** Create new DStream by applying this function to tall constituent RDDS in DStream	`lines.map(x=>x.tolnt*10).print()` nc –lk 9999 12 34 Output: 120 340

`flatMap(func)`

Purpose: This is the same as `map` but mapping function can output 0 or more items

```
lines.flatMap(_.split(“ “)).print()
```

nc –lk 9999 Spark is fun	Output: Spark is fun

`count()`

Purpose: create a DStream of RDDs containing a count of the number of data elements

```
lines.flatMap(_.split(“ “)).print()
```

nc –lk 9999 say hello to spark	Output: 4

`reduce(func)`

Purpose: Same as `count` but the value is derived by applying the function

```
lines.map(x=>x.toInt).reduce(_+_).
print()
```

nc –lk 9999 1 3 5 7	Output: 16

`countByValue()`

Purpose: This is same as `map` but mapping function can output 0 or more items

```
lines.map(x=>x.toInt).reduce(_+_).
print()
```

nc –lk 9999 spark spark is fun fun	Output: (is,1) (spark,2) (fun,2)

`reduceByKey (func,[numTasks])`

```
lines.map(x=>x.toInt).reduce(_+_).
print()
```

nc –lk 9999 spark spark is fun fun	Output: (is,1) (spark,2) (fun,2)

`reduceByKey (func,[numTasks])`

```
val words = lines.flatMap(_.split(“ “))
val wordCounts = words.map(x => (x, 1)).
reduceByKey(_+_)
wordCounts.print()
```

nc –lk 9999 spark is fun fun fun	Output: (is,1) (spark,1) (fun,3)

The following example shows how Apache Spark combines Spark batch with Spark Streaming. This is a powerful capability for an all-in-one technology stack. In this example, we read a file containing brand names and filter those streaming data sets that contain any of the brand names in the file.

`transform(func)`

Purpose: Creates a new DStream by applying RDD->RDD transformation to all RDDs in DStream

brandNames.txt
coke
nike
sprite
reebok

```
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc,
Seconds(10))
val lines = ssc.
socketTextStream(“localhost” 9999,
StorageLevel.MEMORY_AND_DISK_SER)
val brands = sc.textFile(“/tmp/names.
txt”)
lines.transform(rdd=> {
rdd.intersection(brands)
}).print()
```

nc –lk 9999 msft apple nike sprite ibm	Output: sprite nike

COMMON WINDOW OPERATIONS

Transformation & Purpose	Example & Result
`window(window Length, slideInterval)`	`val win = lines. window(Seconds(30),Seconds(10)); win. foreachRDD(rdd => { rdd.foreach(x=>println(x+ “ “)) })` nc –lk 9999 10 (0th second) 20 (10 seconds later) 30 (20 seconds later) 40 (30 seconds later) Output: 10 10 20 20 10 30 20 30 40 (drops 10)
`countByWindow (windowLength slideInterval)` **Purpose:** Returns a new sliding window count of elements in a steam	lines.countByWindow(Seconds(30), Seconds (10)).print() nc –lk 9999 10 (0th second) 20 (10 seconds later) 30 (20 seconds later) 40 (30 seconds later) Output: 1 2 3 3

For additional transformation operators, please refer to spark.apache.org/docs/latest/streaming-programming-guide.html#transformations.

Spark Streaming has powerful output operators. We already saw foreachRDD() in the above example. For others, please refer to spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations.

Structured Streaming has been added to Apache Spark and allows for continuous incremental execution of a structured query. There a few input sources supported including files, Apache Kafka, and sockets. Structured Streaming supports windowing and other advanced streaming features. It is recommended when streaming from files that you supply a schema as opposed to letting Apache Spark infer one for you. This is a similar feature of most streaming systems, like Apache NiFi and Hortonworks Streaming Analytics Manager.

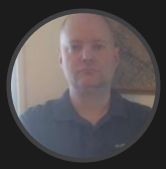
```
val sStream = spark.readStream.json("myJson").load()
sStream.isStreaming

sStream.printSchema
```

For more details on Structured Streaming, please refer to spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

ADDITIONAL RESOURCES

- docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.3/bk_spark-component-guide/content/ch_introduction-spark.html
- hortonworks.com/tutorial/a-lap-around-apache-spark
- hortonworks.com/products/sandbox
- hortonworks.com/tutorial/hands-on-tour-of-apache-spark-in-5-minutes
- hortonworks.com/tutorial/sentiment-analysis-with-apache-spark
- spark.apache.org/downloads.html
- spark.apache.org/docs/latest
- spark.apache.org/docs/latest/quick-start.html
- zeppelin.apache.org
- jaceklaskowski.gitbooks.io/mastering-apache-spark



Written by Tim Spann

is a Big Data Solution Engineer. He helps educate and disseminate performant open source solutions for Big Data initiatives to customers and the community. With over 15 years of experience in various technical leadership, architecture, sales engineering, and development roles, he is well-experienced in all facets of Big Data, cloud, IoT, and microservices. As part of his community efforts, he also runs the Future of Data Meetup in Princeton.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300