This is the first article of the "Big Data Processing with Apache Spark" series. Please see also: Part 2: Spark SQL, Part 3: Spark Streaming, Part 4: Spark Machine Learning, Part 5 Spark ML Data Pipelines, Part 6 Graph Data Analytics with Spark GraphX.

What is Spark

Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.

Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm.

First of all, Spark gives us a comprehensive, unified framework to manage big data processing requirements with a variety of data sets that are diverse in nature (text data, graph data etc) as well as the source of data (batch v. real-time streaming data).

Spark enables applications in Hadoop clusters to run up to 100 times faster in memory and 10 times faster even when running on disk.

Spark lets you quickly write applications in Java, Scala, or Python. It comes with a built-in set of over 80 high-level operators. And you can use it interactively to query data within the shell.

In addition to Map and Reduce operations, it supports SQL queries, streaming data, machine learning and graph data processing. Developers can use these capabilities stand-alone or combine them to run in a single data pipeline use case.

In this first installment of Apache Spark article series, we'll look at what Spark is, how it compares with a typical MapReduce solution and how it provides a complete suite of tools for big data processing.

Hadoop and Spark

Hadoop as a big data processing technology has been around for 10 years and has proven to be the solution of choice for processing large data sets. MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multi-pass computations and algorithms. Each step in the data processing workflow has one Map phase and one Reduce phase and you'll need to convert any use case into MapReduce pattern to leverage this solution.

The Job output data between each step has to be stored in the distributed file system before the next step can begin. Hence, this approach tends to be slow due to replication & disk storage. Also, Hadoop solutions typically include clusters that are hard to set up and manage. It also requires the

integration of several tools for different big data use cases (like Mahout for Machine Learning and Storm for streaming data processing).

If you wanted to do something complicated, you would have to string together a series of MapReduce jobs and execute them in sequence. Each of those jobs was high-latency, and none could start until the previous job had finished completely.

Spark allows programmers to develop complex, multi-step data pipelines using directed acyclic graph (DAG) pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data.

Spark runs on top of existing Hadoop Distributed File System (HDFS) infrastructure to provide enhanced and additional functionality. It provides support for deploying Spark applications in an existing Hadoop v1 cluster (with SIMR – Spark-Inside-MapReduce) or Hadoop v2 YARN cluster or even Apache Mesos.

We should look at Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop. It's not intended to replace Hadoop but to provide a comprehensive and unified solution to manage different big data use cases and requirements.

Spark Features

Spark takes MapReduce to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, the performance can be several times faster than other big data technologies.

Spark also supports lazy evaluation of big data queries, which helps with optimization of the steps in data processing workflows. It provides a higher level API to improve developer productivity and a consistent architect model for big data solutions.

Spark holds intermediate results in memory rather than writing them to disk which is very useful especially when you need to work on the same dataset multiple times. It's designed to be an execution engine that works both in-memory and on-disk. Spark operators perform external operations when data does not fit in memory. Spark can be used for processing datasets that larger than the aggregate memory in a cluster.

Spark will attempt to store as much as data in memory and then will spill to disk. It can store part of a data set in memory and the remaining data on the disk. You have to look at your data and use cases to assess the memory requirements. With this in-memory data storage, Spark comes with performance advantage.

Other Spark features include:

- Supports more than just Map and Reduce functions.
- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
- Provides concise and consistent APIs in Scala, Java and Python.
- Offers interactive shell for Scala and Python. This is not available in Java yet.

Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment. It currently supports the following languages for developing applications using Spark:

- Scala
- Java

- Python
- Clojure
- R

Spark Ecosystem

Other than Spark Core API, there are additional libraries that are part of the Spark ecosystem and provide additional capabilities in Big Data analytics and Machine Learning areas.

These libraries include:

- **Spark Streaming:**
  - Spark Streaming can be used for processing the real-time streaming data. This is based on micro batch style of computing and processing. It uses the DStream which is basically a series of RDDs, to process the real-time data.
- **Spark SQL:**
  - Spark SQL provides the capability to expose the Spark datasets over JDBC API and allow running the SQL like queries on Spark data using traditional BI and visualization tools. Spark SQL allows the users to ETL their data from different formats it's currently in (like JSON, Parquet, a Database), transform it, and expose it for ad-hoc querying.
- **Spark MLlib:**
  - MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.
- **Spark GraphX:**
  - GraphX is the new (alpha) Spark API for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multi-graph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

Outside of these libraries, there are others like BlinkDB and Tachyon.

BlinkDB is an approximate query engine and can be used for running interactive SQL queries on large volumes of data. It allows users to trade-off query accuracy for response time. It works on large data sets by running queries on data samples and presenting results annotated with meaningful error bars.

Tachyon is a memory-centric distributed file system enabling reliable file sharing at memory-speed across cluster frameworks, such as Spark and MapReduce. It caches working set files in memory, thereby avoiding going to disk to load datasets that are frequently read. This enables different jobs/queries and frameworks to access cached files at memory speed.

And there are also integration adapters with other products like Cassandra (Spark Cassandra Connector) and R (SparkR). With Cassandra Connector, you can use Spark to access data stored in a Cassandra database and perform data analytics on that data.

Following diagram (Figure 1) shows how these different libraries in Spark ecosystem are related to each other.

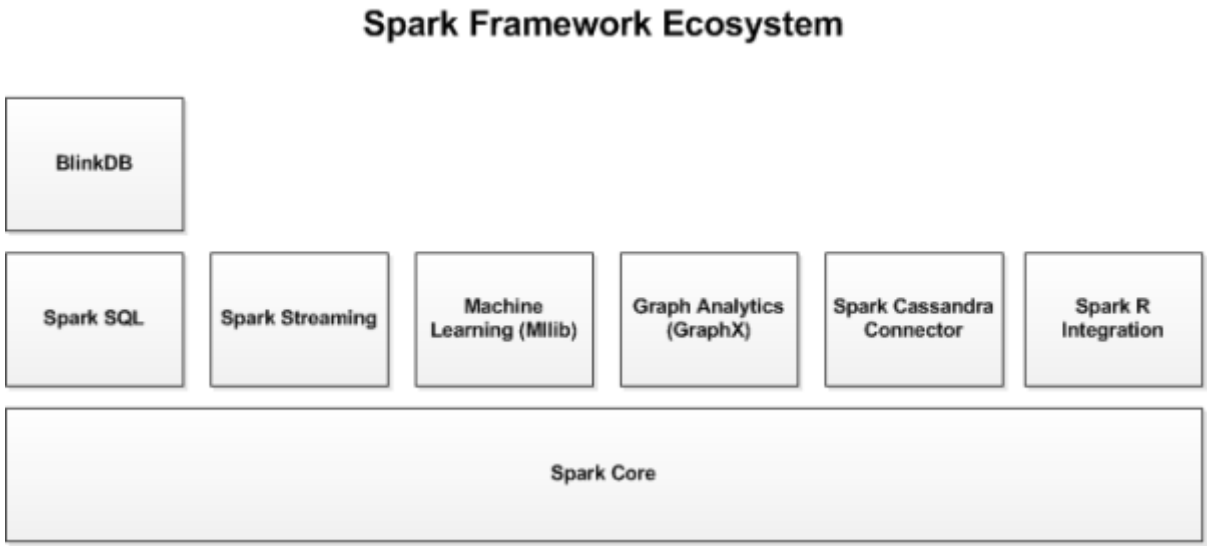

Figure 1. Spark Framework Libraries

We'll explore these libraries in future articles in this series.

Spark Architecture

Spark Architecture includes following three main components:

- Data Storage
- API
- Management Framework

Let’s look at each of these components in more detail.

Data Storage:

Spark uses HDFS file system for data storage purposes. It works with any Hadoop compatible data source including HDFS, HBase, Cassandra, etc.

API:

The API provides the application developers to create Spark based applications using a standard API interface. Spark provides API for Scala, Java, and Python programming languages.

Following are the website links for the Spark API for each of these languages.

- **Scala API**
- **Java**
- **Python**

Resource Management:

Spark can be deployed as a Stand-alone server or it can be on a distributed computing framework like Mesos or YARN.

Figure 2 below shows these components of Spark architecture model.

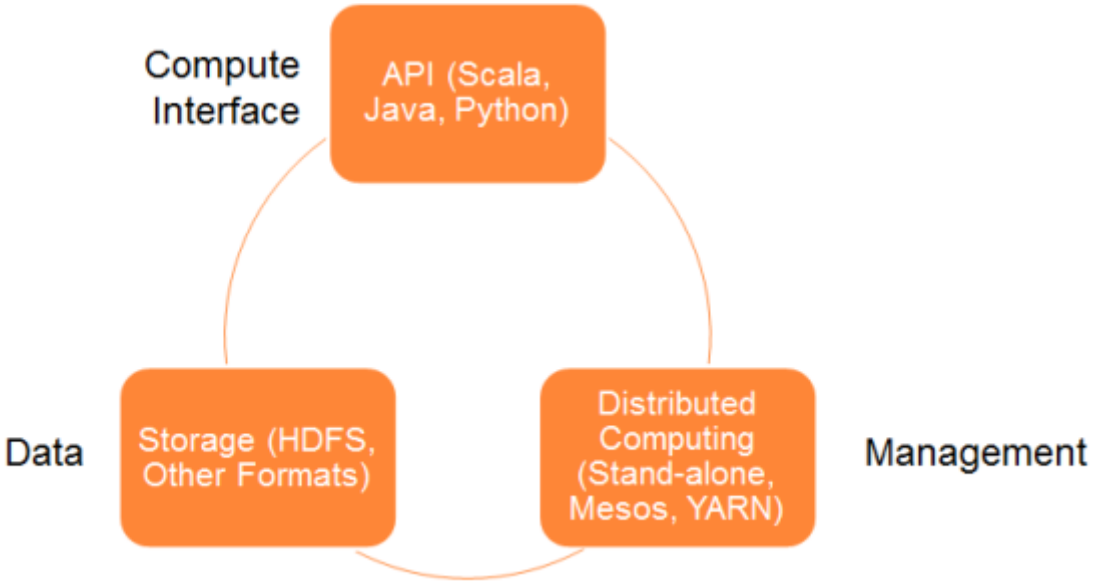

Figure 2. Spark Architecture

Resilient Distributed Datasets

Resilient Distributed Dataset (based on Matei's research paper) or RDD is the core concept in Spark framework. Think about RDD as a table in a database. It can hold any type of data. Spark stores data in RDD on different partitions.

They help with rearranging the computations and optimizing the data processing.

They are also fault tolerance because an RDD know how to recreate and recompute the datasets.

RDDs are immutable. You can modify an RDD with a transformation but the transformation returns you a new RDD whereas the original RDD remains the same.

RDD supports two types of operations:

- Transformation
- Action

Transformation: Transformations don't return a single value, they return a new RDD. Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.

Some of the Transformation functions are `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, `aggregateByKey`, `pipe`, and `coalesce`.

Action: Action operation evaluates and returns a new value. When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.

Some of the Action operations are `reduce`, `collect`, `count`, `first`, `take`, `countByKey`, and `foreach`.

How to Install Spark

There are few different to install and use Spark. You can install it on your machine as a stand-alone framework or use one of Spark Virtual Machine (VM) images available from vendors like Cloudera, HortonWorks, or MapR. Or you can also use Spark installed and configured in the cloud (like Databricks Cloud).

In this article, we'll install Spark as a stand-alone framework and launch it locally. Spark 1.2.0 version was released recently. We'll use this version for sample application code demonstration.

How to Run Spark

When you install Spark on the local machine or use a Cloud based installation, there are few different modes you can connect to Spark engine.

The following table shows the Master URL parameter for the different modes of running Spark.

Master URL	Description
Local	Run Spark locally with one worker thread (i.e. no parallelism at all).
local[K]	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
spark://HOST:PORT	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
mesos://HOST:PORT	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use mesos://zk://....
yarn-client	Connect to a YARN cluster in client mode. The cluster location will be found based on the HADOOP_CONF_DIR variable.
yarn-cluster	Connect to a YARN cluster in cluster mode. The cluster location will be found based on HADOOP_CONF_DIR.

How to Interact with Spark

Once Spark is up and running, you can connect to it using the Spark shell for interactive data analysis. Spark Shell is available in both Scala and Python languages. Java doesn't support an interactive shell yet, so this feature is currently not available in Java.

You use the commands `spark-shell.cmd` and `pyspark.cmd` to run Spark Shell using Scala and Python respectively.

Spark Web Console

When Spark is running in any mode, you can view the Spark job results and other statistics by accessing Spark Web Console via the following URL:

`http://localhost:4040`

Spark Console is shown in Figure 3 below with tabs for Stages, Storage, Environment, and Executors.

(Click on the image to enlarge it)

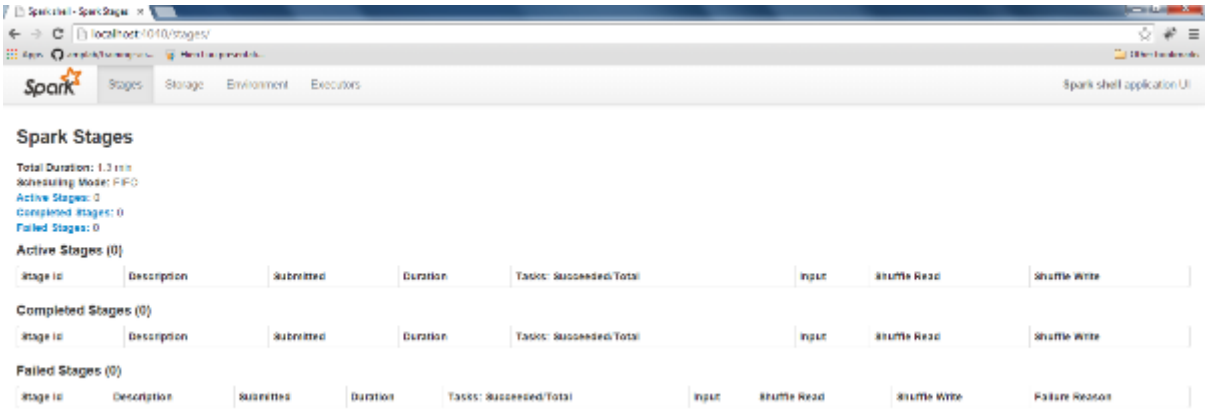

Figure 3. Spark Web Console

Shared Variables

Spark provides two types of shared variables to make it efficient to run the Spark programs in a cluster. These are Broadcast Variables and Accumulators.

Broadcast Variables: Broadcast variables allow to keep read-only variable cached on each machine instead of sending a copy of it with tasks. They can be used to give the nodes in the cluster copies of large input datasets more efficiently.

Following code snippet shows how to use the broadcast variables.

```
//
// Broadcast Variables
//
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Accumulators: Accumulators are only added using an associative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Tasks running on the cluster can add to an accumulator variable using the add method. However, they cannot read its value. Only the driver program can read the accumulator's value.

The code snippet below shows how to use Accumulator shared variable:

```
//
// Accumulators
//

val accum = sc.accumulator(0, "My Accumulator")

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

accum.value
```

Sample Spark Application

The sample application I cover in this article is a simple Word Count application. This is the same example one would cover when they are learning Big Data processing with Hadoop. We'll perform some data analytics queries on a text file. The text file and the data set in this example are small, but same Spark queries can be used for large size data sets, without any modifications in the code.

To keep the discussion simple, we'll use the Spark Scala Shell.

First, let's look at how to install Spark on your local machine.

Pre-Requisites:

- You will need Java Development Kit (JDK) installed for Spark to work locally. This is covered in Step 1 below.
- You will also need to install Spark software on your laptop. The instructions on how to do this are covered in the Step 2 below.

Note: These instructions are for Windows environment. If you are using a different operating system environment, you'll need to modify the system variables and directory paths to match your environment.

I. INSTALL JDK:

1) Download JDK from Oracle website. JDK version 1.7 is recommended.

Install JDK in a directory name without spaces. For Windows users, install JDK in a folder like `c:\dev`, not in "`c:\Program Files`". "`Program Files`" directory has a space in the name and this causes problems when software is installed in this folder.

NOTE: DO NOT INSTALL JDK or Spark Software (described in Step 2) in "`c:\Program Files`" directory.

2) After installing JDK, verify it was installed correctly by navigating to "bin" folder under JDK 1.7 directory and typing the following command:

```
java -version
```

If JDK is installed correctly, the above command would display the Java version.

II. INSTALL SPARK SOFTWARE:

Download the latest Spark version from Spark website. Latest version at the time of publication of this article is Spark 1.2. You can choose a specific Spark installation depending on the Hadoop version. I downloaded Spark for Hadoop 2.4 or later, and the file name is `spark-1.2.0-bin-hadoop2.4.tgz`.

Unzip the installation file to a local directory (For example, c:\dev).

To verify Spark installation, navigate to spark directory and launch Spark Shell using the following commands. This is for Windows. If you are using Linux or Mac OS, please edit the commands to work on your OS.

```
c:
cd c:\dev\spark-1.2.0-bin-hadoop2.4
```

```
bin\spark-shell
```

If Spark was installed correctly, you should the see the following messages in the output on the console.

```
….
15/01/17 23:17:46 INFO HttpServer: Starting HTTP Server
15/01/17 23:17:46 INFO Utils: Successfully started service 'HTTP class server'
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.0
      /_/

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.
Type :help for more information.
….
15/01/17 23:17:53 INFO BlockManagerMaster: Registered BlockManager
15/01/17 23:17:53 INFO SparkILoop: Created spark context..
Spark context available as sc.
```

You can type the following commands to check if Spark Shell is working correctly.

```
sc.version
```

(or)

```
sc.appName
```

After this step, you can exit the Spark Shell window by typing the following command:

```
:quit
```

To launch Spark Python Shell, you need to have Python installed on your machine. You can download and install Anaconda which is a free Python distribution and includes several popular Python packages for science, math, engineering, and data analysis.

Then you can run the following commands:

```
c:
cd c:\dev\spark-1.2.0-bin-hadoop2.4
bin\pyspark
```

Word Count Application

Once you have Spark installed and have it up and running, you can run the data analytics queries using Spark API.

These are simple commands to read the data from a text file and process it. We'll look at advanced use cases of using Spark framework in the future articles in this series.

First, let's use Spark API to run the popular Word Count example. Open a new Spark Scala Shell if you don't already have it running. Here are the commands for this example.

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

val txtFile = "README.md"
val txtData = sc.textFile(txtFile)
txtData.cache()
```

We call the cache function to store the RDD created in the above step in the cache, so Spark doesn't have to compute it every time we use it for further data queries. Note that cache() is a lazy operation. Spark doesn't immediately store the data in memory when we call cache. It actually takes place when an action is called on an RDD.

Now, we can call the count function to see how many lines are there in the text file.

```
txtData.count()
```

Now, we can run the following commands to perform the word count. The count shows up next to each word in the text file.

```
val wcData = txtData.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceBy

wcData.collect().foreach(println)
```

If you want to look at more code examples of using Spark Core API, checkout Spark documentation on their website.

What's Next

In the future articles of this series, we'll learn more about other parts of Spark ecosytem starting with Spark SQL. Later, we'll look at Spark Streaming, Spark MLlib, and Spark GraphX. We'll also look at the upcoming frameworks like Tachyon and BlinkDB.

Conclusions

In this article, we looked at how Apache Spark framework helps with big data processing and analytics with its standard API. We also looked at how Spark compares with traditional MapReduce implementation like Apache Hadoop. Spark is based on the same HDFS file storage system as Hadoop, so you can use Spark and MapReduce together if you already have significant investment and infrastructure setup with Hadoop.

You can also combine the Spark processing with Spark SQL, Machine Learning and Spark Streaming as we'll see in a future article.

With several integrations and adapters on Spark, you can combine other technologies with Spark. An example of this is to use Spark, Kafka, and Apache Cassandra together where Kafka can be used for the streaming data coming in, Spark to do the computation, and finally Cassandra NoSQL database to store the computation result data.

But keep in mind, Spark is a less mature ecosystem and needs further improvements in areas like security and integration with BI tools.

References

- Spark Main Website
- Spark Examples
- Spark Summit 2014 Conference Presentation and Videos
- Spark on Databricks website

About the Author

Srini Penchikala currently works as Software Architect at a financial services organization in Austin, Texas. He has over 20 years of experience in software architecture, design and development. Srini is currently authoring a book on NoSQL Database Patterns topic. He is also the co-author of "Spring Roo in Action" book from Manning Publications. He has presented at conferences like JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. Srini also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, OReilly Network (ONJava), DevX Java, java.net and JavaWorld. He is a Lead Editor for NoSQL Databases community at InfoQ.

This is the first article of the "Big Data Processing with Apache Spark" series. Please see also: Part 2: Spark SQL, Part 3: Spark Streaming, Part 4: Spark Machine Learning, Part 5 Spark ML Data Pipelines, Part 6 Graph Data Analytics with Spark GraphX.