BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

Topic: Apache Spark

Introduction

What is Apache Spark?

Apache Spark is an open source parallel processing framework for running large-scale data analytics applications across clustered computers. It can handle both batch and real-time analytics and data processing workloads.

Spark Core, the heart of the project that provides distributed task transmission, scheduling and I/O functionality provides programmers with a potentially faster and more flexible alternative to MapReduce, the software framework to which early versions of Hadoop were tied. Spark's developers say it can run jobs 100 times faster than MapReduce when processed in-memory, and 10 times faster on disk.

How Apache Spark works

Apache Spark can process data from a variety of data repositories, including the Hadoop Distributed File System (HDFS), NoSQL databases and relational data stores, such as Apache Hive. Spark supports in-memory processing to boost the performance of big data analytics applications, but it can also perform conventional disk-based processing when data sets are too large to fit into the available system memory.

The Spark Core engine uses the resilient distributed data set, or RDD, as its basic data type. The RDD is designed in such a way so as to hide much of the computational complexity from users. It aggregates data and partitions it across a server cluster, where it can then be computed and either moved to a different data store or run through an analytic model. The user doesn't have to define where specific files are sent or what computational resources are used to store or retrieve files. In addition, Spark can handle more than the batch processing applications that MapReduce is limited to running.

Spark languages

Spark was written in Scala, which is considered the primary language for interacting with the Spark Core engine. Out of the box, Spark also comes with API connectors for using Java and Python. Java is not considered an optimal language for data engineering or data science, so many users rely on Python, which is simpler and more geared toward data analysis.

There is also an R programming package that users can download and run in Spark. This enables users to run the popular desktop data science language on larger distributed data sets in Spark and to use it to build applications that leverage machine learning algorithms.

Apache Spark vs. Apache Hadoop

Outside of the differences in the design of Spark and Hadoop MapReduce, many organizations have found these big data frameworks to be complimentary, using them together to solve a broader business challenge.

Hadoop is an open source framework that has the Hadoop Distributed File System (HDFS) as storage, YARN as a way of managing computing resources used by different applications, and an implementation of the MapReduce programming model as an execution engine. In a typical Hadoop implementation, different execution engines are also deployed such as Spark, Tez, and Presto.

Spark is an open source framework focused on interactive query, machine learning, and real-time workloads. It does not have its own storage system, but runs analytics on other storage systems like HDFS, or other popular stores like Amazon Redshift, Amazon S3, Couchbase, Cassandra, and others. Spark on Hadoop leverages YARN to share a common cluster and dataset as other Hadoop engines, ensuring consistent levels of service, and response.

Benefits of Apache Spark:

Fast: Through in-memory caching, and optimized query execution, Spark can run fast analytic queries against data of any size.

Developer Friendly: Apache Spark natively supports Java, Scala, R, and Python, giving you a variety of languages for building your applications. These APIs make it easy for your developers, because they hide the complexity of distributed processing behind simple, high-level operators that dramatically lowers the amount of code required.

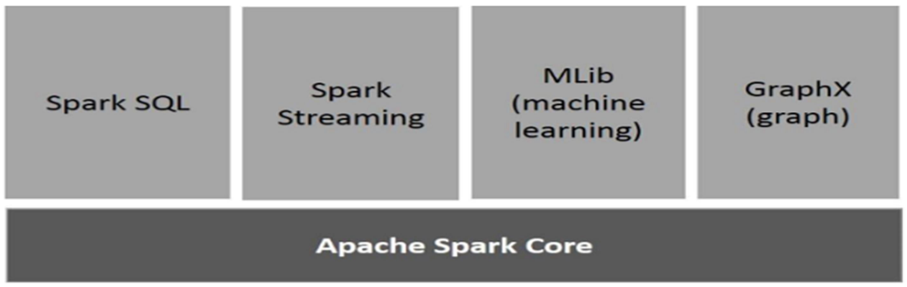
Multiple workloads: Apache Spark comes with the ability to run multiple workloads, including interactive queries, real-time analytics, machine learning, and graph processing. One application can combine multiple workloads seamlessly.

The Spark framework includes:

- Spark Core as the foundation for the platform
- Spark SQL for interactive queries
- Spark Streaming for real-time analytics
- Spark MLlib for machine learning
- Spark GraphX for graph processing

Spark Components:

The following illustration depicts the different components of Spark.



Apache Spark core: Spark Core is the foundation of the platform. It is responsible for memory management, fault recovery, scheduling, distributing & monitoring jobs, and interacting with storage systems. Spark Core is exposed through an application programming interface (APIs) built for Java, Scala, Python and R. These APIs hide the complexity of distributed processing behind simple, high-level operators.

MLlib: Spark includes MLlib, a library of algorithms to do machine learning on data at scale. Machine Learning models can be trained by data scientists with R or Python on any Hadoop data source, saved using MLlib, and imported into a Java or Scala-based pipeline. Spark was designed for fast, interactive computation that runs in memory, enabling machine learning to run quickly. The algorithms include the ability to do classification, regression, clustering, collaborative filtering, and pattern mining.

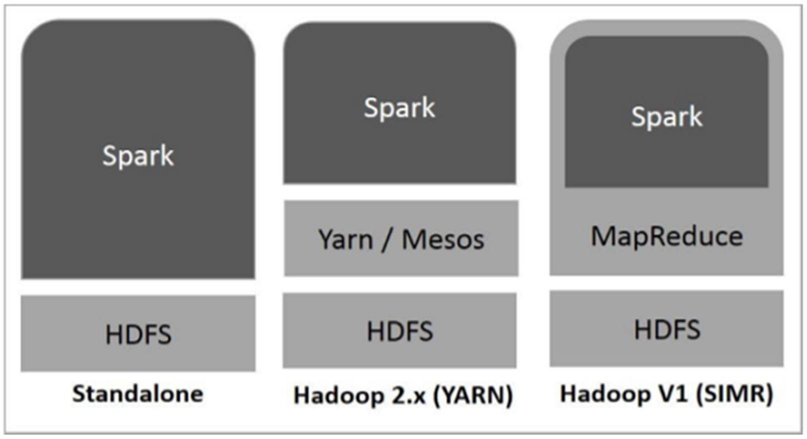
Spark Streaming: Spark streaming is a real-time solution that leverages Spark Core's fast scheduling capability to do streaming analytics. It ingests data in mini-batches, and enables analytics on that data with the same application code written for batch analytics. This improves developer productivity, because they can use the same code for batch processing, and for real-time streaming applications.

Spark SQL: Spark SQL is a distributed query engine that provides low-latency, interactive queries up to 100x faster than MapReduce. It includes a cost-based optimizer, columnar storage, and code generation for fast queries, while scaling to thousands of nodes. Developers can use APIs, available in Scala, Java, Python, and R.

GraphX: Spark GraphX is a distributed graph processing framework built on top of Spark. GraphX provides ETL, exploratory analysis, and iterative graph computation to enable users to interactively build, and transform a graph data structure at scale. It comes with a highly flexible API, and a selection of distributed 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.

Data Sharing is Slow in MapReduce:

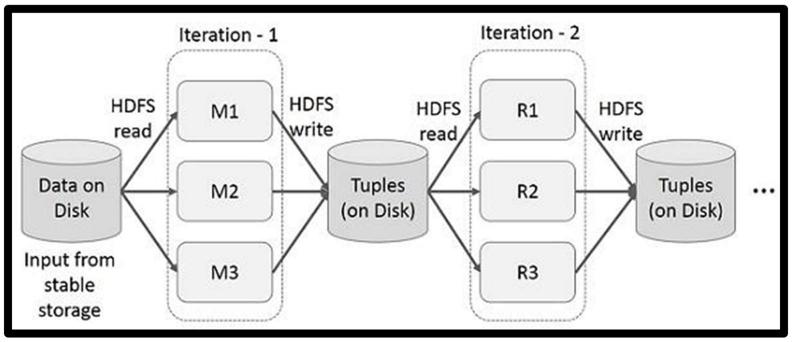
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

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 systems, most of the Hadoop applications spend more than 90% of the time doing HDFS read-write operations.

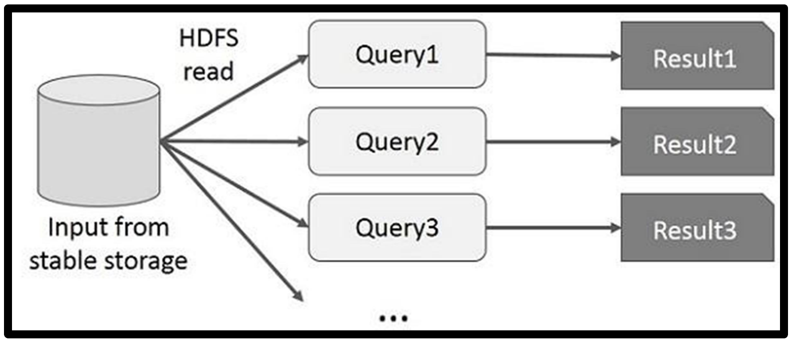
Iterative Operations on MapReduce

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 dominate application execution time.



Apache Spark – Installation

Step 1: Before installing spark, update the system by following command:

$*sudo apt update*

Step 2: check for java version using command

$*java -version*

Step 3: If not present then install java by using following command

```
$sudo apt-get install default-jdk -y
```

Step 4: Create a new directory

```
$mkdir -p spark
```

Step 5: Change your directory to "spark" directory

```
$cd spark
```

Step 6: If wget is not installed

```
$sudo apt-get install wget
```

Step 7: Install spark

```
$sudo wget https://dlcdn.apache.org/spark/spark-3.2.2/spark-3.2.2-bin-hadoop3.2.tgz
```

You can get links for different versions from https://spark.apache.org/downloads.html

Step 8: Now unzip the installed file using following command:

```
$sudo tar -xvf spark-3.2.2-bin-hadoop3.2.tgz
```

Step 9: Now we have to edit the bashrc file. Bashrc can be opened by following command

```
$nano ~/.bashrc
```

Step 10: Append the following two commands in the bottom of the bashrc file.

```
SPARK_HOME=/home/username/spark/spark-3.2.2-bin-hadoop3.2
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Note: *Modify username in above path*

Step 11: Now for applying all the changes, source the bashrc file

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

Note: *Close the current terminal window and open a new one.*

Step 12: Verify whether Spark was installed correctly or not using following command:

```
$Spark-shell
```

Programs Using PySpark

PySpark Installation

Step 1: Install python 3 using below commands

```
$sudo apt install python3
$sudo apt install python3-pip
```

Step 2: Edit bashrc file and add the below line

To open bashrc

```
$gedit ~/.bashrc
```

Add the following line at the end, save it and exit.

```
export PYSPARK_PYTHON=/usr/bin/python3
```

Step 3: To reflect the changes, use the command:

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

Step 4: Close all the terminals. Open a new terminal and check whether PySpark is installed properly. Use the command:

```
$pyspark
```

Spark shell will open.

Executing the programs:

1. Word Count

In new terminal execute the below command to run word count program

```
$spark-submit /home/<username>/WordCount.py /home/<username>/InputFiles/File1/home/<username>/result
```

Note: *Syntax: spark-submit <Argument 0- path of the program file> <Argument 1: Path of Input File> <Argument 3: Path of output directory to be created>*

You can use the python interpreter in Spark shell to observe the output line by line using rdd.collect() method.

2. Inverted Index

Use the below command to run the Inverted Index Program

```
$spark-submit /home/<username>/InvertedIndex.py /home/<username>/InputFiles/ /home/<username>/result2
```

Note: *Syntax: spark-submit <Argument 0- path of the program file> <Argument 1: Path of Input Directory> <Argument 3: Path of output directory to be created>*

You can use the python interpreter in Spark shell to observe the output line by line using rdd.collect() method.

Programs using Scala interpreter in Spark Shell

1) Word Count Program:

Run word count program using Scala Interpreter provided in spark shell.

```
val text = sc.textFile("/home/username/wc_input_file")
val a1 = text.flatMap(line => line.split(" "))
val a2 = a1.map(word => (word,1))
val a3 = a2.reduceByKey(_+_)
a3.collect()
```

Note: *Run line by line and observe Transformations outputs using collect() action.*

2) Inverted Index Program:

Run Inverted Index program using Scala Interpreter provided in spark shell and observe Transformations outputs using collect() action.

```
1) val q1 = sc.wholeTextFiles("/home/username/input_directory/")
```

Note: Create multiple input files in input directory

```
2) val q2 = q1.flatMap {
               case (path, text) => {
                 val words = text.split("""\W+""")
                 val filename = path.split("/").last
                 words.map(word => ((word.toLowerCase(), filename),1))
                }
            }

 3) val q3 = q2.reduceByKey((a, b) => a + b)

 4) val q4 = q3.map{
                case ((word, filename), count) => (word, (filename,count))
            }

 5) val q5 = q4.groupByKey().sortByKey(ascending = true)

 6) val q6 = q5.map { case (word, iterable) =>
                val vec = iterable.toVector.sortBy { case (filename, count) => (-count,
filename) }
                val (locations, counts) = vec.unzip
                val totalCount = counts.reduceLeft( (n1, n2) => n1 + n2)
                (word, locations, totalCount)
            }
```