spark

new material

Spark Tutorial – Learn Spark Programming

18 Jan, 2018 in Apache Spark Tutorials by Data Flair

1.SPARK- INTRODUCTION

1. Objective

In this Spark Tutorial, we will see an overview of Spark Big Data. We will start with an introduction to Apache Spark Programming. Then we will move to know the Spark History. Moreover, we will learn why Spark is needed. Afterwards, will cover all fundamental of Spark components. Furthermore, we will learn about Spark's core abstraction and Spark RDD. For more detailed insights, we will also cover spark features, Spark limitations and Spark Use cases.

2. Introduction to Spark Programming

What is Spark? Spark Programming is nothing but a general-purpose & lightning fast cluster computing platform. In other words, it is an open source, wide range data processing engine. That reveals development API's, which also qualifies data workers to accomplish streaming, machine learning or SQL workloads which demand repeated access to data sets. However, Spark can perform **batch processing and stream processing**. Batch processing refers, to the processing of the previously collected job in a single batch. Whereas stream processing means to deal with Spark streaming data.

Moreover, it is designed in such a way that it integrates with all the **Big data** tools. Like spark can access any **Hadoop** data source, also can run on Hadoop clusters. Furthermore, Apache Spark extends Hadoop MapReduce to next level. That also includes iterative queries and stream processing.

One more common belief about Spark is that it is an extension of Hadoop. Although that is not true. However, Spark is independent of Hadoop since it has its own **cluster management** system. Basically, it uses Hadoop for storage purpose only.

Although, there is one spark's key feature that it has in-memory cluster computation capability. Also increases the processing speed of an application.

Basically, Apache Spark offers high-level APIs to users, such as **Java**, Scala, **Python**and R. Although, Spark is written in Scala still offers rich APIs in Scala, Java, Python, as well as R. We can say, it is a tool for running spark applications.

Most importantly, on **comparing Spark with Hadoop**, it is 100 times faster than Big Data Hadoop and 10 times faster than accessing data from disk.

3. Spark History

At first, in 2009 Apache Spark was introduced in the UC Berkeley R&D Lab. Which is now known as AMPLab. Afterwards, in 2010 it became open source under BSD license. Further, the spark was donated to Apache Software Foundation, in 2013. Then in 2014, it became top-level Apache project.

4. Why Spark?

Spark Tutorial – Why Spark?

As we know, there was no general purpose computing engine in the industry, since

1. To perform batch processing, we were using **Hadoop MapReduce**.
2. Also, to perform stream processing, we were using Apache Storm / S4.
3. Moreover, for interactive processing, we were using Apache Impala / Apache Tez.
4. To perform graph processing, we were using Neo4j / Apache Giraph.

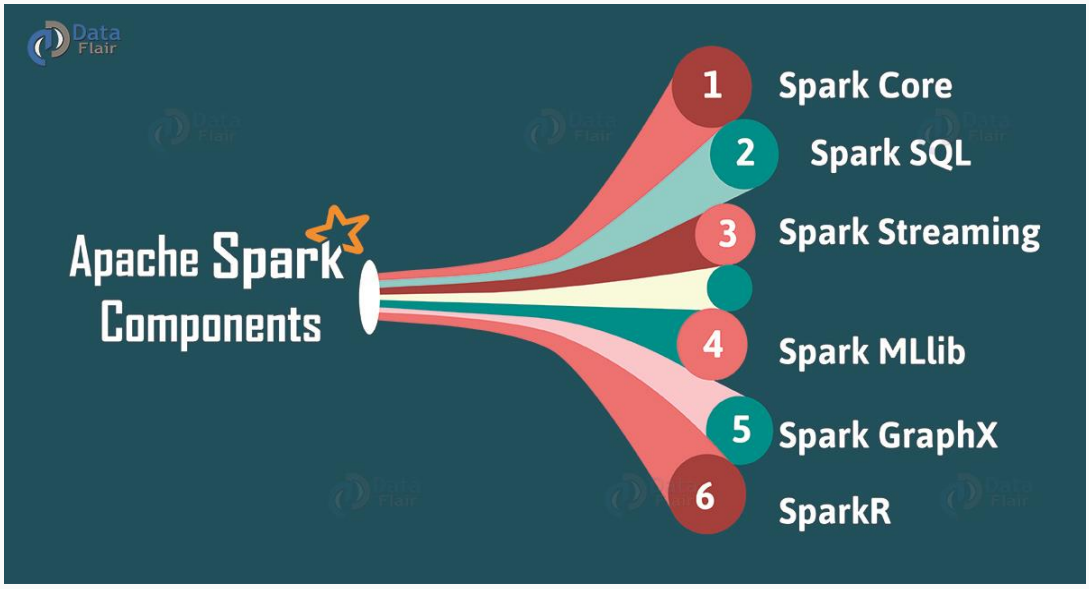
Hence there was no powerful engine in the industry, that can process the data both in real-time and batch mode. Also, there was a requirement that one engine can respond in sub-second and perform in-memory processing.

Therefore, Apache Spark programming enters, it is a powerful open source engine. Since, it offers real-time stream processing, interactive processing, graph processing, in-memory processing as well as batch processing. Even with very fast speed, ease of use and standard interface. Basically, these features create the difference between Hadoop and Spark. Also makes a huge **comparison between Spark vs Storm.**

5. Apache Spark Components

In this Apache Spark Tutorial, we discuss Spark Components. It puts the promise for faster data processing as well as easier development. It is only possible because of its components. All these Spark components resolved the issues that occurred while using Hadoop MapReduce.

Now let's discuss each Spark Ecosystem Component one by one-



Spark Tutorial – Apache Spark Ecosystem Components

a. Spark Core

Spark Core is a central point of Spark. Basically, it provides an execution platform for all the Spark applications. Moreover, to support a wide array of applications, Spark Provides a generalized platform.

b. Spark SQL

On the top of Spark, Spark SQL enables users to run SQL/HQL queries. We can process structured as well as semi-structured data, by using Spark SQL. Moreover, it offers to run unmodified queries up to 100 times faster on existing deployments. To learn **Spark SQL in detail**, follow this link.

c. Spark Streaming

Basically, across live streaming, Spark Streaming enables a powerful interactive and data analytics application. Moreover, the live streams are converted into micro-batches those are executed on top of spark core. Learn **Spark Streaming in detail.**

d. Spark MLlib

Machine learning library delivers both efficiencies as well as the high-quality algorithms. Moreover, it is the hottest choice for a data scientist. Since it is capable of in-memory data processing, that improves the performance of iterative algorithm drastically.

e. Spark GraphX

Basically, Spark GraphX is the graph computation engine built on top of Apache Spark that enables to process graph data at scale.

f. SparkR

Basically, to use Apache Spark from **R**. It is **R package** that gives light-weight frontend. Moreover, it allows data scientists to analyze large datasets. Also allows running jobs interactively on them from the R shell. Although, the main idea behind SparkR was to explore different techniques to integrate the usability of R with the scalability of Spark. Follow the link to learn **SparkR in detail.**

To learn about all the components of Spark in detail, follow link **Apache Spark Ecosystem – Complete Spark Components Guide**

6. Resilient Distributed Dataset – RDD

The key abstraction of Spark is RDD. RDD is an acronym for Resilient Distributed Dataset. It is the fundamental unit of data in Spark. Basically, it is a distributed collection of elements across cluster nodes. Also performs parallel operations. Moreover, Spark RDDs are immutable in nature. Although, it can generate new RDD by transforming existing Spark RDD.Learn about **Spark RDDs in detail.**

a. Ways to create Spark RDD

Basically, there are 3 ways to create Spark RDDs

i. Parallelized collections

By invoking parallelize method in the driver program, we can create parallelized collections.

ii. External datasets

One can create Spark RDDs, by calling a textFile method. Hence, this method takes URL of the file and reads it as a collection of lines.

iii. Existing RDDs

Moreover, we can create new RDD in spark, by applying transformation operation on existing RDDs.

To learn all three **ways to create RDD** in detail, follow the link.

b. Spark RDDs operations

There are two types of operations, which Spark RDDs supports:

i. Transformation Operations

It creates a new Spark RDD from the existing one. Moreover, it passes the dataset to the function and returns new dataset.

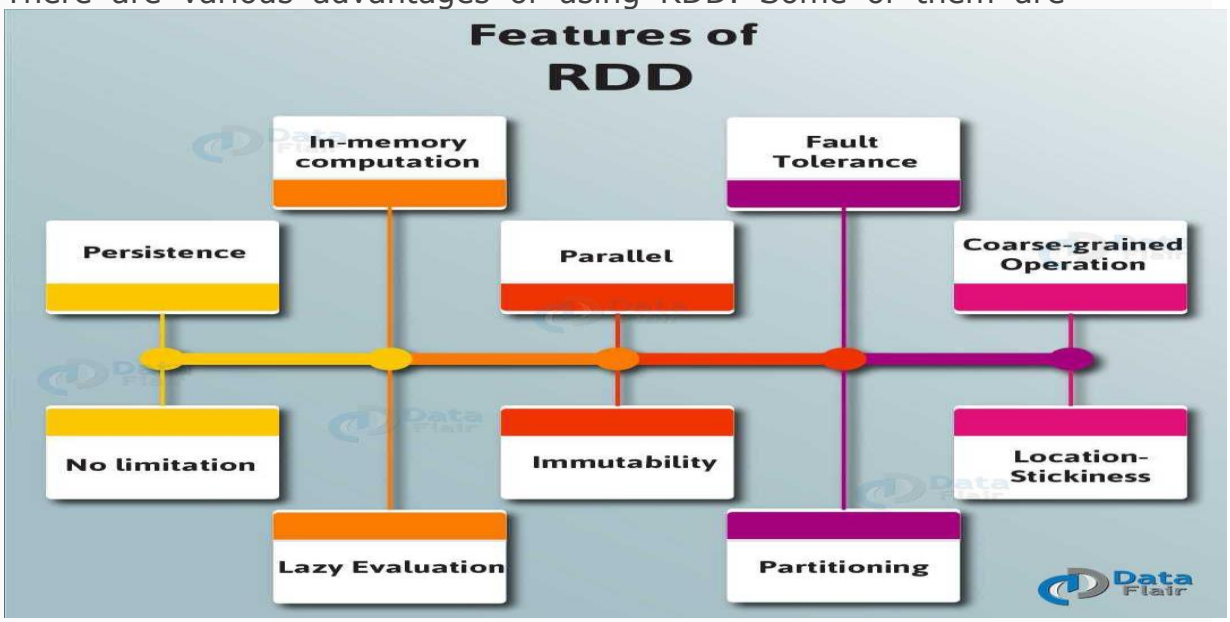
ii. Action Operations

In Apache Spark, Action returns final result to driver program or write it to the external data store.

Learn **RDD Operations in detail.**

c. sparkling Features of Spark RDD

There are various advantages of using RDD. Some of them are



Spark Tutorial – Spark RDD Features

i. In-memory computation

Basically, while storing data in RDD, data is stored in memory for as long as you want to store. It improves the performance by an order of magnitudes by keeping the data in memory.

ii. Lazy Evaluation

Spark Lazy Evaluation means the data inside RDDs are not evaluated on the go. Basically, only after an action triggers all the changes or the computation is performed. Therefore, it limits how much work it has to do. learn **Lazy Evaluation in detail.**

iii. Fault Tolerance

If any worker node fails, by using lineage of operations, we can re-compute the lost partition of RDD from the original one. Hence, it is possible to recover lost data easily. Learn **Fault Tolerance in detail**.

iv. Immutability

Immutability means once we create an RDD, we can not manipulate it. Moreover, we can create a new RDD by performing any transformation. Also, we achieve consistency through immutability.

v. Persistence

In in-memory, we can store the frequently used RDD. Also, we can retrieve them directly from memory without going to disk. It results in the speed of the execution. Moreover, we can perform multiple operations on the same data. It is only possible by storing the data explicitly in memory by calling persist() or cache() function. Learn **Persistence and Caching Mechanism** in detail.

vi. Partitioning

Basically, RDD partition the records logically. Also, distributes the data across various nodes in the cluster. Moreover, the logical divisions are only for processing and internally it has no division. Hence, it provides parallelism.

vii. Parallel

While we talk about parallel processing, RDD processes the data parallelly over the cluster.

viii. Location-Stickiness

To compute partitions, RDDs are capable of defining placement preference. Moreover, placement preference refers to information about the location of RDD. Although, the DAGScheduler places the partitions in such a way that task is close to data as much as possible. Moreover, it speeds up computation.

ix. Coarse-grained Operation

Generally, we apply coarse-grained transformations to Spark RDD. It means the operation applies to the whole dataset not on the single element in the data set of RDD in Spark.

x. Typed

There are several types of Spark RDD. Such as: RDD [int], RDD [long], RDD [string].

xi. No limitation

There are no limitations to use the number of Spark RDD. We can use any no. of RDDs. Basically, the limit depends on the size of disk and memory.

In this Apache Spark tutorial, we cover most Features of Spark RDD to learn more about **RDD Features** follow this link.

7. Spark Tutorial – Spark Streaming

While data is arriving continuously in an unbounded sequence is what we call a data stream. Basically, for further processing, Streaming divides continuous flowing input data into discrete units. Moreover, we can say it is a low latency processing and analyzing of streaming data.

In addition, an extension of the core Spark API Streaming was added to Apache Spark in 2013. That offers scalable, fault-tolerant and high-throughput processing of live data streams. Although, here we can do data ingestion from many sources. Such as Kafka, **Apache Flume**, Amazon Kinesis or TCP sockets. However, we do processing here by using complex algorithms which are expressed with high-level functions such as map, reduce, join and window.

a. Internal working of Spark Streaming

Let's understand its internal working. While live input data streams are received. It further divided into batches by Spark streaming, Afterwards, these batches are processed by the Spark engine to generate the final stream of results in batches.

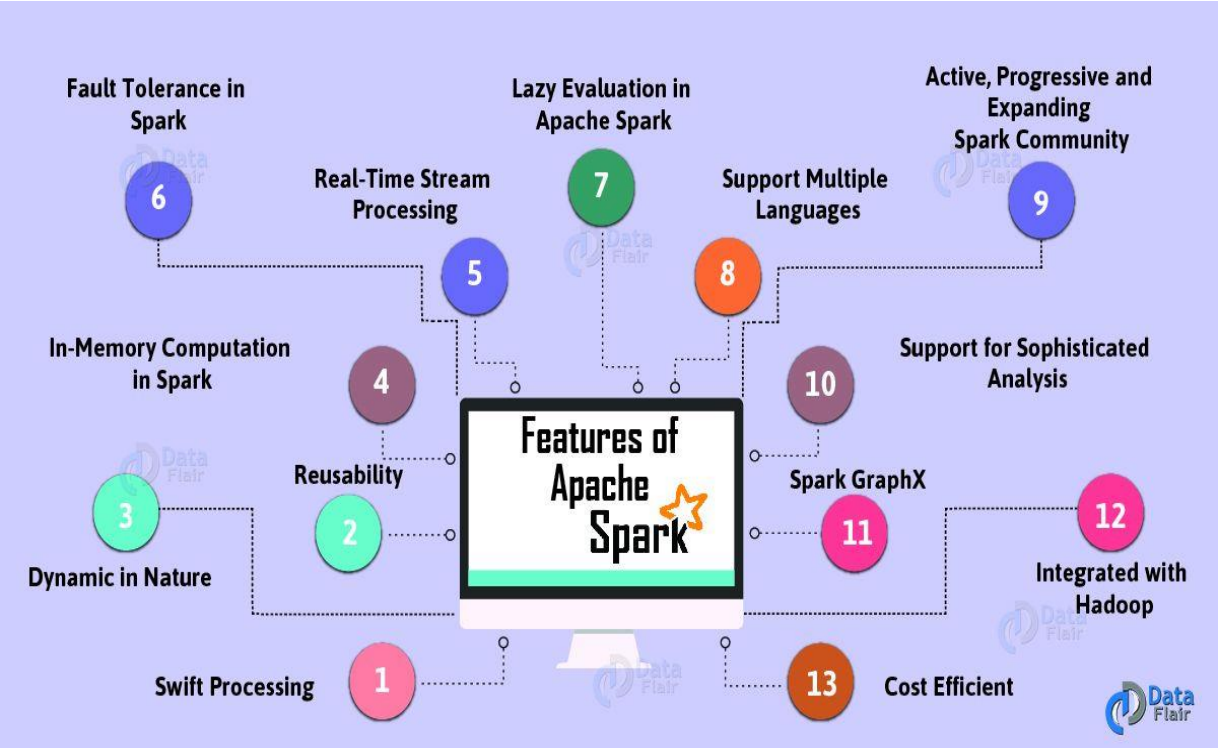
b. Discretized Stream (DStream)

Apache Spark Discretized Stream is the key abstraction of Spark Streaming. That is what we call Spark DStream. Basically, it represents a stream of data divided into small batches. Moreover, DStreams are built on Spark RDDs, Spark's core data abstraction. It also allows Streaming to seamlessly integrate with any other Apache Spark components. Such as Spark MLlib and Spark SQL.

follow this link, to Learn **Concept of Dstream in detail.**

8. Features of Apache Spark

There are several sparkling Apache Spark features:



Apache Spark Tutorial – Features of Apache Spark

a. Swift Processing

Apache Spark offers high data processing speed. That is about 100x faster in memory and 10x faster on the disk. However, it is only possible by reducing the number of read-write to disk.

b. Dynamic in Nature

Basically, it is possible to develop a parallel application in Spark. Since there are 80 high-level operators available in Apache Spark.

c. In-Memory Computation in Spark

The increase in processing speed is possible due to **in-memory processing**. It enhances the processing speed.

d. Reusability

We can easily reuse spark code for batch-processing or join stream against historical data. Also to run ad-hoc queries on stream state.

e. Spark Fault Tolerance

Spark offers fault tolerance. It is possible through Spark's core abstraction-RDD. Basically, to handle the failure of any worker node in the cluster, Spark RDDs are designed. Therefore, the loss of data is reduced to zero.

f. Real-Time Stream Processing

We can do real-time stream processing in Spark. Basically, Hadoop does not support real-time processing. It can only process data which is already present. Hence with Spark Streaming, we can solve this problem.

g. Lazy Evaluation in Spark

All the transformations we make in Spark RDD are Lazy in nature, that is it does not give the result right away rather a new RDD is formed from the existing one. Thus, this increases the efficiency of the system.

h. Support Multiple Languages

Spark supports multiple languages. Such as Java, R, **Scala**, Python. Hence, it shows dynamicity. Moreover, it also overcomes the **limitations of Hadoop** since it can only build applications in **Java.**

i. Support for Sophisticated Analysis

There are dedicated tools in Apache Spark. Such as for streaming data interactive/declarative queries, machine learning which add-on to map and reduce.

j. Integrated with Hadoop

As we know Spark is flexible. It can run independently and also on Hadoop YARN Cluster Manager. Even it can read existing Hadoop data.

k. Spark GraphX

In Spark, a component for graph and graph-parallel computation, we have GraphX. Basically, it simplifies the graph analytics tasks by the collection of graph algorithm and builders.

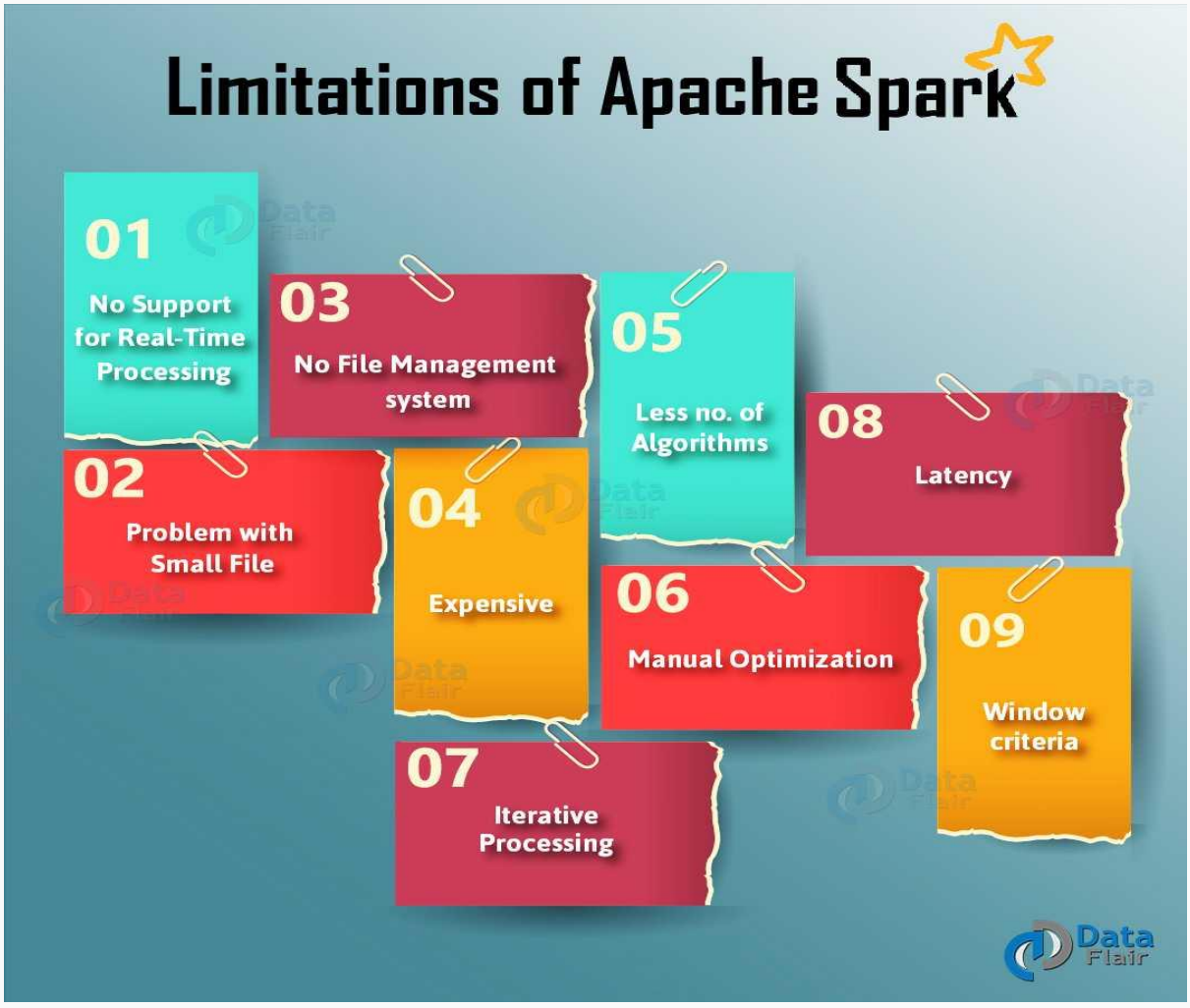
l. Cost Efficient

For Big data problem as in Hadoop, a large amount of storage and the large data center is required during replication. Hence, Spark programming turns out to be a cost-effective solution.

Learn **All features of Apache Spark, in detail.**

9. Limitations of Apache Spark Programming

There are many limitations of Apache Spark. Let’s learn all one by one:



Spark Tutorial – Limitations of Apache Spark Programming

a. No Support for Real-time Processing

Basically, Spark is near real-time processing of live data. In other words, Micro-batch processing takes place in Spark Streaming. Hence we can not say Spark is completely Real-time Processing engine.

b. Problem with Small File

In RDD, each file is a small partition. It means, there is the large amount of tiny partition within an RDD. Hence, if we want efficiency in our processing, the RDDs should be repartitioned into some manageable format. Basically, that demands extensive shuffling over the network.

c. No File Management System

A major issue is Spark does not have its own file management system. Basically, it relies on some other platform like Hadoop or another cloud-based platform.

d. Expensive

While we desire cost-efficient processing of big data, Spark turns out to be very expensive. Since keeping data in memory is quite expensive. However the memory consumption is very high, and it is not handled in a user-friendly manner. Moreover, we require lots of RAM to run in-memory, thus the cost of spark is much higher.

e. Less number of Algorithms

Spark MLlib have very less number of available algorithms. For example, Tanimoto distance.

f. Manual Optimization

It is must that Spark job is manually optimized and is adequate to specific datasets. Moreover, to partition and cache in spark to be correct, it is must to control it manually.

g. Iterative Processing

Basically, here data iterates in batches. Also, each iteration is scheduled and executed separately.

h. Latency

On comparing with **Flink**, Apache Spark has higher latency.

i. Window Criteria

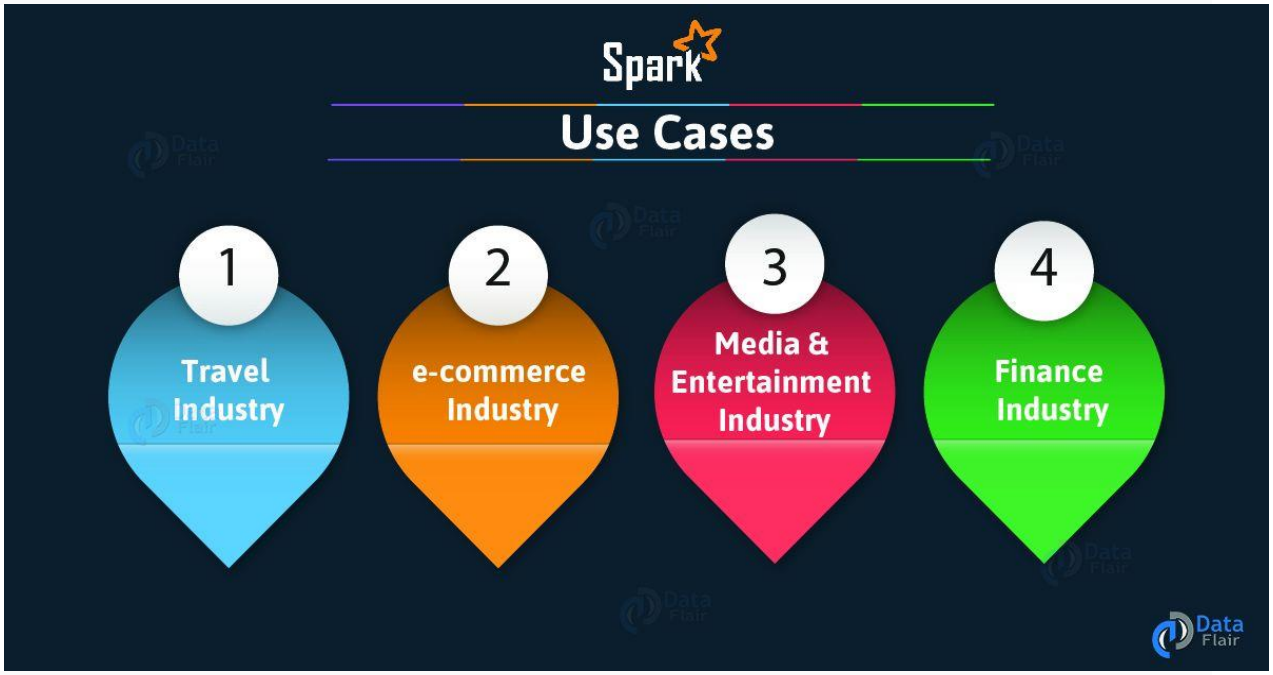
Spark only support time-based window criteria not record based window criteria.

Note: To overcome these limitations of Spark, we can use **Apache Flink – 4G of Big Data**.

Learn All Limitations of Apache Spark, in detail.

10. Apache Spark use cases

There are many industry-specific Apache Spark use cases, let's discuss them one by one:



Spark Tutorial – Apache Spark Use Cases

a. Spark Use Cases in the Finance Industry

There are many banks those are using Spark. Basically, it helps to access and analyze many of the parameters in the bank sector like the emails, social media profiles, call recordings, forum, and many more. Further, it helps to make right decisions for several zones.

b. Apache Spark Use Cases in E-Commerce Industry

Basically, it helps with information about a real-time transaction. Moreover, those are passed to streaming clustering algorithms.

c. Apache Spark Use Cases in Media & Entertainment Industry

We use Spark to identify patterns from the real-time in-game events. Moreover, it helps to respond in order to harvest lucrative business opportunities.

d. Apache Spark Use Cases in Travel Industry

Basically, travel industries are using spark rapidly. Moreover, it helps users to plan a perfect trip by speed up the personalized recommendations. Although, its review process of the hotels in a readable format is done by using Spark.

Apache Spark tutorial cover Spark real-time use Cases, there are many more, follow the link to learn all in detail. **Apache Spark use cases in real time**

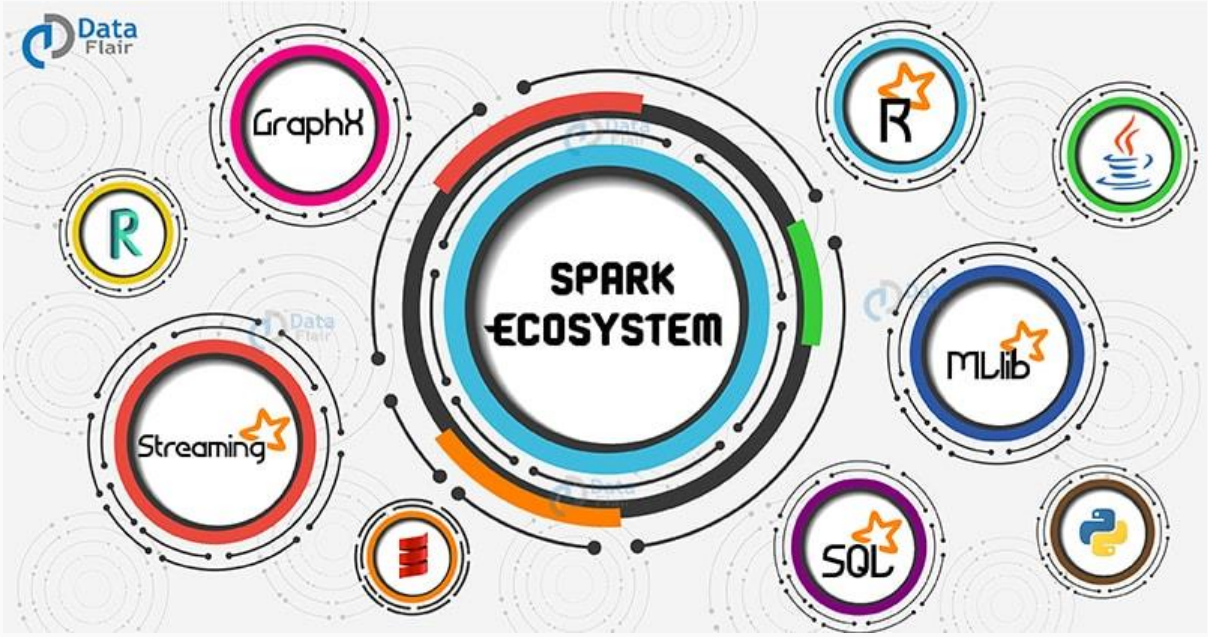
11. Spark Tutorial – Conclusion

As a result, we have seen every aspect of Apache Spark, what is Apache spark programming and spark definition, History of Spark, why Spark is needed, Components of Apache Spark, Spark RDD, Features of Spark RDD, Spark Streaming, Features of Apache Spark, Limitations of Apache Spark, Apache Spark use cases. In this tutorial we were trying to cover all spark notes, hope you get desired information in it if you feel to ask any query, feel free to ask in the comment section.

2.Spark- Ecosystem components

1. Objective

In this tutorial on **Apache Spark ecosystem**, we will learn what is Apache Spark, what is the ecosystem of Apache Spark. It also covers components of Spark ecosystem like **Spark core** component, **Spark SQL**, **Spark Streaming**, **Spark MLlib**, **Spark GraphX** and **SparkR**. We will also learn the features of Apache Spark ecosystem components in this Spark tutorial.

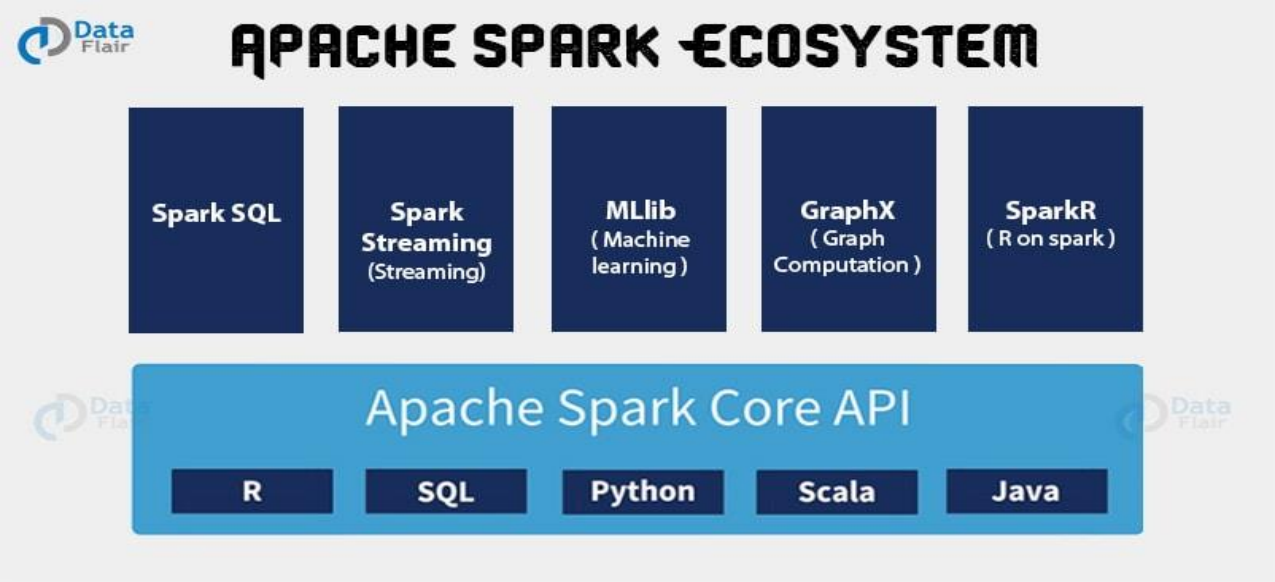


2. Apache Spark Introduction

Apache Spark is general purpose cluster computing system. It provides high-level API in Java, **Scala**, Python, and **R**. Spark provide an optimized engine that supports general execution graph. It also has abundant high-level tools for structured data processing, machine learning, graph processing and streaming. The Spark can either run alone or on an existing cluster manager. Follow this link to Learn more about Apache Spark.

3. Introduction to Apache Spark Ecosystem Components

Following are 6 components in Apache Spark Ecosystem which empower to Apache Spark- Spark Core, Spark SQL, Spark Streaming, Spark MLlib, Spark GraphX, and SparkR.



Let us now learn about these Apache Spark ecosystem components in detail below:

3.1. Apache Spark Core

All the functionalities being provided by Apache Spark are built on the top of **Spark Core**. It delivers speed by providing **in-memory computation** capability. Thus Spark Core is the foundation of parallel and distributed processing of huge dataset.

The key features of Apache Spark Core are:

- It is in charge of essential I/O functionalities.
- Significant in programming and observing the role of the **Spark cluster**.
- Task dispatching.
- Fault recovery.
- It overcomes the snag of **MapReduce** by using in-memory computation.

Spark Core is embedded with a special collection called **RDD** (resilient distributed dataset). RDD is among the abstractions of Spark. *Spark RDD* handles partitioning data across all the nodes in a cluster. It holds them in the memory pool of the cluster as a single unit. There are two operations performed on RDDs: *Transformation* and *Action*-

- **Transformation:** It is a function that produces new RDD from the existing RDDs.
- **Action:** In Transformation, RDDs are created from each other. But when we want to work with the actual dataset, then, at that point we use Action.

Refer these guides to learn more about Spark RDD Transformations & Actions API and Different ways to create RDD in Spark.

3.2. Apache Spark SQL

The **Spark SQL** component is a distributed framework for *structured* data processing. Using Spark SQL, Spark gets more information about the structure of data and the computation. With this information, Spark can perform extra optimization. It uses same execution engine while computing an output. It does not depend on API/ language to express the computation.

Spark SQL works to access structured and semi-structured information. It also enables powerful, interactive, analytical application across both streaming and historical data. Spark SQL is Spark module for structured data processing. Thus, it acts as a distributed SQL query engine.

Features of Spark SQL include:

- Cost based optimizer. Follow Spark SQL Optimization tutorial to learn more.
- Mid query fault-tolerance: This is done by scaling thousands of nodes and multi-hour queries using the Spark engine. Follow this guide to Learn more about Spark fault tolerance.
- Full compatibility with existing **Hive** data.
- **DataFrames** and SQL provide a common way to access a variety of data sources. It includes Hive, Avro, Parquet, ORC, JSON, and JDBC.
- Provision to carry structured data inside Spark programs, using either SQL or a familiar Data Frame API.

3.3. Apache Spark Streaming

It is an add-on to core Spark API which allows scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark can access data from sources like **Kafka**, **Flume**, **Kinesis** or **TCP socket.** It can operate using various algorithms. Finally, the data so received is given to file system, databases and live dashboards. Spark uses *Micro-batching* for real-time streaming.

Micro-batching is a technique that allows a process or task to treat a stream as a sequence of small batches of data. Hence Spark Streaming, groups the live data into small batches. It then delivers it to the batch system for

processing. It also provides fault tolerance characteristics. Learn Spark Streaming in detail from this Apache Spark Streaming Tutorial.

How does Spark Streaming Works?

There are 3 phases of Spark Streaming:

a. GATHERING

The *Spark Streaming* provides two categories of built-in *streaming sources:*

- **Basic sources:** These are the sources which are available in the*StreamingContext* API. Examples: file systems, and socket connections.
- **Advanced sources:** These are the sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. Hence Spark access data from different sources like Kafka, Flume, Kinesis, or TCP sockets.

b. PROCESSING

The gathered data is processed using complex algorithms expressed with a high-level function. For example, map, reduce, join and window. Refer this guide to learn Spark Streaming transformations operations.

c. DATA STORAGE

The Processed data is pushed out to file systems, databases, and live dashboards.

Spark Streaming also provides high-level abstraction. It is known as discretized stream or DStream.

DStream in Spark signifies continuous stream of data. We can form DStream in two ways either from sources such as Kafka, Flume, and Kinesis or by high-level operations on other DStreams. Thus, DStream is internally a sequence of RDDs.

3.4. Apache Spark MLlib (Machine Learning Library)

MLlib in Spark is a scalable Machine learning library that discusses both high-quality algorithm and high speed.

The motive behind MLlib creation is to make machine learning scalable and easy. It contains machine learning libraries that have an implementation of various machine learning algorithms. For example, *clustering, regression, classification and collaborative filtering.* Some lower level machine learning primitives like generic gradient descent optimization algorithm are also present in MLlib.

In Spark Version 2.0 the RDD-based API in *spark.mllib* package entered in maintenance mode. In this release, the DataFrame-based API is the primary Machine Learning API for Spark. So, from now MLlib will not add any new feature to the RDD based API.

The reason MLlib is switching to DataFrame-based API is that it is more user-friendly than RDD. Some of the benefits of using DataFrames are it includes Spark Data sources, SQL DataFrame queries *Tungsten and Catalyst optimizations*, and uniform APIs across languages. MLlib also uses the linear algebra package *Breeze*. Breeze is a collection of libraries for numerical computing and machine learning.

3.5. Apache Spark GraphX

GraphX in Spark is API for graphs and graph parallel execution. It is network graph analytics engine and data store. *Clustering, classification, traversal, searching, and pathfinding* is also possible in graphs. Furthermore, GraphX extends Spark RDD by bringing in light a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

GraphX also optimizes the way in which we can represent vertex and edges when they are primitive data types. To support graph computation it supports fundamental operators (e.g., subgraph, join Vertices, and aggregate Messages) as well as an optimized variant of the *Pregel API*.

3.6. Apache SparkR

SparkR was Apache Spark 1.4 release. The key component of SparkR is SparkR DataFrame. DataFrames are a fundamental data structure for data processing in **R.** The concept of DataFrames extends to other languages with libraries like *Pandas*etc.

R also provides software facilities for data manipulation, calculation, and graphical display. Hence, the main idea behind SparkR was to explore different techniques to integrate the usability of R with the scalability of Spark. It is R package that gives light-weight frontend to use Apache Spark from R.

There are various benefits of SparkR:

- **Data Sources API:** By tying into Spark SQL's data sources API SparkR can read in data from a variety of sources. For example, Hive tables, JSON files, Parquet files etc.
- **Data Frame Optimizations:** SparkR DataFrames also inherit all the optimizations made to the computation engine in terms of code generation, memory management.
- **Scalability to many cores and machines:** Operations that executes on SparkR DataFrames get distributed across all the cores and machines available in the**Spark cluster**. As a result, SparkR DataFrames can run on terabytes of data and clusters with thousands of machines.

4. Conclusion

Apache Spark amplifies the existing **Bigdata** tool for analysis rather than reinventing the wheel. It is Apache Spark Ecosystem Components that make it popular than other Bigdata frameworks. Hence, Apache Spark is a common platform for different types of data processing. For example, real-time data analytics, Structured data processing, graph processing, etc.

Therefore Apache Spark is gaining considerable momentum and is a promising alternative to support ad-hoc queries. It also provide iterative processing logic by replacing MapReduce. It offers interactive code execution using Python and Scala REPL but you can also write and compile your application in Scala and Java.

Got a question about Apache Spark ecosystem component? Notify us by leaving a comment and we will get back to you.

3.Apache Spark Terminologies and Concepts

1. Objective

Here you will learn some of the important Apache Spark Terminologies and Apache Spark Key terms that are being used in the Spark industry. It will help you in learning what is Apache Spark, what does spark engine do, partition in Apache Spark, what does MLlib do, worker node in Spark, Spark SQL and Pair RDD definition, what is lazy evaluation in Spark, Spark Driver and Action in Spark.

2. Introduction to Apache Spark Terminologies

Before we start with testing the knowledge, let us revise our Spark key concepts and understand how Apache Spark is better than Hadoop MapReduce in terms of usage and performance.

4.Apache Spark Installation On Ubuntu- A Beginners Tutorial

1. Objective

This tutorial describes the first step while **learning Apache Spark** i.e. Apache Spark Installation On Ubuntu. This Apache Spark tutorial is a step by step guide for Installation of Spark, the configuration of pre-requisites and launches Spark shell to perform various operations. If you are completely new to Apache Spark, I would recommend you to read these introductory blogs- What is Spark, Spark ecosystem, Spark key abstraction RDD, Spark features, and limitations of Apache Spark.

2. Steps for Apache Spark Installation On Ubuntu

Follow the steps given below for Apache Spark Installation On Ubuntu-

2.1. Deployment Platform

i. Platform Requirements

- **Operating System:** You can use Ubuntu 14.04 or later (other Linux flavors can also be used like CentOS, Redhat, etc.)
- **Spark:** Apache Spark 1.6.1 or later

ii. Setup Platform

If you are using Windows / Mac OS you can create a virtual machine and install Ubuntu using VMWare Player, alternatively, you can create a virtual machine and install Ubuntu using Oracle Virtual Box.

2.2. Prerequisites

i. Install Java 7

a. Install Python Software Properties

```
$sudo apt-get install python-software-properties
```

b. Add Repository

```
$sudo add-apt-repository ppa:webupd8team/java
```

c. Update the source list

```
$sudo apt-get update
```

d. Install Java

```
$sudo apt-get install oracle-java7-installer
```

2.3. Install Apache Spark

i. Download Spark

You can download Apache Spark from the below link. In the package type please select "Pre-built for Hadoop 2.6 and Later"
http://spark.apache.org/downloads.html

Or, you can use direct download link:
http://mirror.fibergrid.in/apache/spark/spark-1.6.1/spark-1.6.1-bin-hadoop2.6.tgz

ii. Untar Spark Setup

```
$tar xzf spark-1.6.1-bin-hadoop2.6.tgz
```

You can find all the scripts and configuration files in newly created directory "spark-1.6.1-bin-hadoop2.6"

iii. Setup Configuration

a. Edit .bashrc

Edit .bashrc file located in user's home directory and add following parameters-

```
1. export JAVA_HOME=<path-to-the-root-of-your-Java-installation> (eg: /usr/lib/jvm/java-7-oracle/)
2. export SPARK_HOME=<path-to-the-root-of-your-spark-installation> (eg: /home/dataflair/spark-1.6.1-bin-hadoop2.6/)
```

2.4. Launch the Spark Shell

Go to Spark home directory (spark-1.6.1-bin-hadoop2.6) and run below command to start Spark Shell

```
$bin/spark-shell.sh
```

Spark shell is launched, now you can play with Spark

i. Spark UI

This is the GUI for Spark Application, in local mode spark shell runs as an application. The GUI provide details about stages, storage (cached RDDs), Environment Variables and executors

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

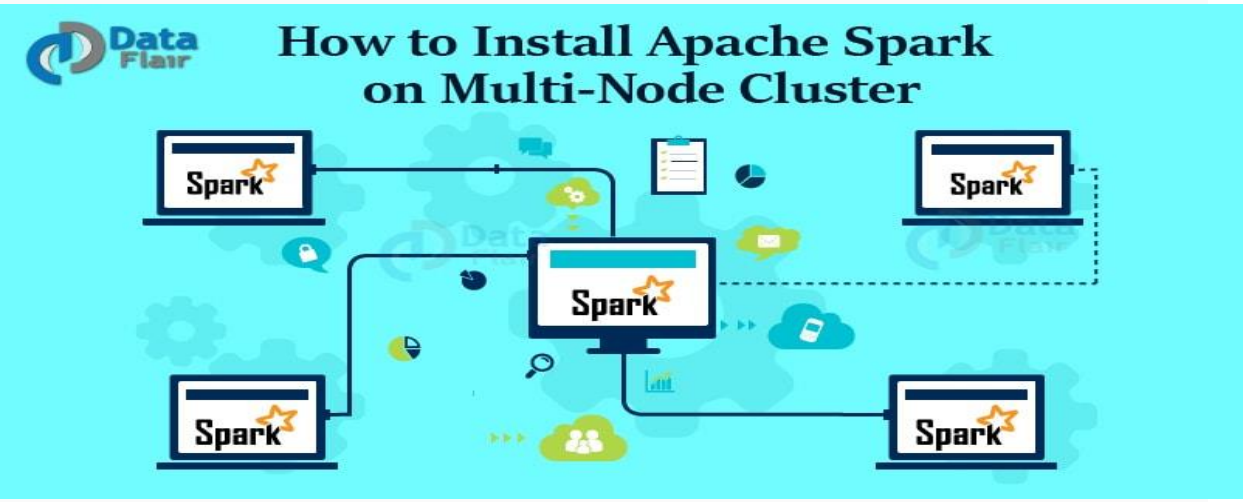
2.5. Spark Commands / Operations

Once you installed Apache Spark, you can play with spark shell to perform the various operation like **transformation and action**, the **creation of RDDs.** Follow this guide for **Shell Commands** to working with Spark.

5.How to Install Apache Spark on Multi Node Cluster

1. Objective

This Spark tutorial explains how to install **Apache Spark** on a multi-node cluster. This guide provides step by step instructions to deploy and configure Apache Spark on the real multi-node cluster. Once the setup and installation are done you can play with Spark and process data.



2. Steps to install Apache Spark on multi-node cluster

Follow the steps given below to easily install Apache Spark on a multi-node cluster.

2.1. Recommended Platform

- **OS –** Linux is supported as a development and deployment platform. You can use Ubuntu 14.04 / 16.04 or later (you can also use other Linux flavors like CentOS, Redhat, etc.). Windows is supported as a dev platform. (If you are new to Linux follow this **Linux commands manual**).
- **Spark –** Apache Spark 2.x

For Apache Spark Installation On Multi-Node Cluster, we will be needing multiple nodes, either you can use Amazon AWS or follow this guide to setup virtual platform using VMWare player.

2.2. Install Spark on Master

I. Prerequisites

a. Add Entries in hosts file

Edit hosts file

```
sudo nano /etc/hosts
```

Now add entries of master and slaves

```
1. MASTER-IP master
2. SLAVE01-IP slave01
3. SLAVE02-IP slave02
```

(NOTE: In place of MASTER-IP, SLAVE01-IP, SLAVE02-IP put the value of the corresponding IP)

b. Install Java 7 (Recommended Oracle Java)

```
1. sudo apt-get install python-software-properties
2. sudo add-apt-repository ppa:webupd8team/java
3. sudo apt-get update
4. sudo apt-get install oracle-java7-installer
```

c. Install Scala

```
sudo apt-get install scala
```

d. Configure SSH

i. Install Open SSH Server-Client

```
sudo apt-get install openssh-server openssh-client
```

ii. Generate Key Pairs

```
ssh-keygen -t rsa -P ""
```

iii. Configure passwordless SSH

Copy the content of .ssh/id_rsa.pub (of master) to .ssh/authorized_keys (of all the slaves as well as master)

iv. Check by SSH to all the Slaves

```
1. ssh slave01
2. ssh slave02
```

II. Install Spark

a. Download Spark

You can download the latest version of spark from **http://spark.apache.org/downloads.html**.

b. Untar Tarball

tar xzf spark-2.0.0-bin-hadoop2.6.tgz

(Note: All the scripts, jars, and configuration files are available in newly created directory "spark-2.0.0-bin-hadoop2.6")

c. Setup Configuration

i. Edit .bashrc

Now edit .bashrc file located in user's home directory and add following environment variables:

```
1. export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib/jvm/java-7-oracle/)
2. export SPARK_HOME=<path-to-the-root-of-your-spark-installation> (eg: /home/dataflair/spark-2.0.0-bin-hadoop2.6/)
3. export PATH=$PATH:$SPARK_HOME/bin
```

(Note: After above step restart the Terminal/Putty so that all the environment variables will come into effect)

ii. Edit spark-env.sh

Now edit configuration file spark-env.sh (in $SPARK_HOME/conf/) and set following parameters:

Note: Create a copy of template of spark-env.sh and rename it:

```
cp spark-env.sh.template spark-env.sh
```

```
1. export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib/jvm/java-7-oracle/)
2. export SPARK_WORKER_CORES=8
```

iii. Add Salves

Create configuration file slaves (in $SPARK_HOME/conf/) and add following entries:

```
1. slave01
2. slave02
```

"Apache Spark has been installed successfully on Master, now deploy Spark on all the Slaves"

2.3. Install Spark On Slaves

I. Setup Prerequisites on all the slaves

Run following steps on all the slaves (or worker nodes):

- "1.1. Add Entries in hosts file"
- "1.2. Install Java 7"
- "1.3. Install Scala"

II. Copy setups from master to all the slaves

a. Create tarball of configured setup

```
tar czf spark.tar.gz spark-2.0.0-bin-hadoop2.6
```

NOTE: Run this command on Master

b. Copy the configured tarball on all the slaves

```
scp spark.tar.gz slave01:~
```

NOTE: Run this command on Master

```
scp spark.tar.gz slave02:~
```

NOTE: Run this command on Master

III. Un-tar configured spark setup on all the slaves

```
tar xzf spark.tar.gz
```

NOTE: Run this command on all the slaves

"Congratulations Apache Spark has been installed on all the Slaves. Now Start the daemons on the Cluster"

2.4. Start Spark Cluster

I. Start Spark Services

```
sbin/start-all.sh
```

Note: Run this command on Master

II. Check whether services have been started

a. Check daemons on Master

```
1. jps
2. Master
```

b. Check daemons on Slaves

```
1. jps
2. Worker
```

2.5. Spark Web UI

I. Spark Master UI

Browse the Spark UI to know about worker nodes, running application, cluster resources.

http://MASTER-IP:8080/

II. Spark application UI

http://MASTER-IP:4040/

2.6. Stop the Cluster

I. Stop Spark Services

Once all the applications have finished, you can stop the spark services (master and slaves daemons) running on the cluster

```
sbin/stop-all.sh
```

Note: Run this command on Master

After Apache Spark installation, I recommend learning Spark **RDD**, **DataFrame**, and **Dataset.** You can proceed further with **Spark shell commands** to play with Spark.

3. Conclusion

After installing the Apache Spark on the multi-node cluster you are now ready to work with Spark platform. Now you can play with the data, **create an RDD**, perform operations on those RDDs over multiple nodes and much more.

If you have any query to install Apache Spark, so, feel free to share with us. We will be happy to solve them.

See Also-

- SparkContext in Apache Spark
- RDD Transformations and Actions APIs

Reference:

http://spark.apache.org/

6.SPARK SHELL COMMANDS TO INTERACT WITH SPARK-SCALA

1. Objective

The shell acts as an interface to access the operating system's service. **Apache Spark** is shipped with an interactive shell/scala prompt with the interactive shell we can run different commands to process the data. This is an Apache Spark Shell commands guide with step by step list of basic spark commands/operations to interact with **Spark shell**.

Before starting you must have Spark installed. follow this guide to install Apache Spark.

After Spark installation, You can create **RDDs** and perform various transformations and actions like filter(), partitions(), cache(), count(), collect, etc. In this blog, we will also discuss the integration of Spark with Hadoop, how spark reads the data from HDFS and write to HDFS?.

2. Scala – Spark Shell Commands

Start the Spark Shell

Apache Spark is shipped with an interactive shell/scala prompt, as the spark is developed in **Scala**. Using the interactive shell we will run different commands (**RDD transformation/action**) to process the data.

The command to start the Apache Spark Shell:

```
$bin/spark-shell
```

2.1. Create a new RDD

a) Read File from local filesystem and create an RDD.

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

Note: sc is the object of SparkContext

Note: You need to create a file data.txt in Spark_Home directory

b) Create an RDD through Parallelized Collection

```
1. scala> val no = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
2. scala> val noData = sc.parallelize(no)
```

c) From Existing RDDs

```
scala> val newRDD = no.map(data => (data * 2))
```

These are **three methods to create the RDD**. We can use the first method, when data is already available with the external systems like a local filesystem, **HDFS**,**HBase**, **Cassandra, S3**, etc. One can create an RDD by calling a **textFile** method of **Spark Context** with path / URL as the argument. The second approach can be used with the existing collections and the third one is a way to create new RDD from the existing one.

2.2. Number of Items in the RDD

Count the number of items available in the RDD. To count the items we need to call an Action:

```
scala> data.count()
```

2.3. Filter Operation

Filter the RDD and create new RDD of items which contain word "DataFlair". To filter, we need to call transformation filter, which will return a new RDD with subset of items.

```
scala> val DFData = data.filter(line => line.contains("DataFlair"))
```

2.4. Transformation and Action together

For complex requirements, we can chain multiple operations together like filter transformation and count action together:

```
scala> data.filter(line => line.contains("DataFlair")).count()
```

2.5. Read the first item from the RDD

To read the first item from the file, you can use the following command:

```
scala> data.first()
```

2.6. Read the first 5 item from the RDD

To read the first 5 item from the file, you can use the following command:

```
scala> data.take(5)
```

2.7. RDD Partitions

An RDD is made up of multiple partitions, to count the number of partitions:

```
scala> data.partitions.length
```

*Note: Minimum no. of partitions in the RDD is 2 (by default). When we create RDD from **HDFS** file then a number of blocks will be equals to the number of partitions.*

2.8. Cache the file

Caching is the optimization technique. Once we **cache the RDD** in the memory all future computation will work on the **in-memory** data, which saves disk seeks and improve the performance.

```
scala> data.cache()
```

RDD will not be cached once you run above operation, you can visit the web UI: http://localhost:4040/storage, it will be blank. RDDs are not explicitly cached once we run **cache()**, rather RDDs will be cached once we run the Action, which actually needs data read from the disk.

Let's run some actions

```
scala> data.count()
```

```
scala> data.collect()
```

Now as we have run some actions on the data file, which needs to be read from the disk to perform those operations. During this process, Spark will cache the file, so that for all future operations will get the data from the memory (no need for any disk interaction). Now if we run any transformation or action it will be done in-memory and will be much faster.

2.9. Read Data from HDFS file

To read data from HDFS file we can specify complete hdfs URL like**hdfs://IP:PORT/PATH**

```
scala> var hFile = sc.textFile("hdfs://localhost:9000/inp")
```

2.10. Spark WordCount Program in Scala

One of the most popular operations of **MapReduce** – **Wordcount**. Count all the words available in the file.

```
scala> val wc = hFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```

Read the result on console

```
scala> wc.take(5)
```

It will display first 5 results

2.11 Write the data to HDFS file

To write the data from HFDS:

```
scala> wc.saveAsTextFile("hdfs://localhost:9000/out")
```

3. Conclusion

In conclusion, we can say that using Spark Shell commands we can create RDD (In three ways), read from RDD, and partition RDD. We can even cache the file, read and write data from and to HDFS file and perform various operation on the data using the Apache Spark Shell commands.

Now you can create your first Spark Scala project.

See Also-

- Apache Spark installation on multi-node cluster
- Spark Map vs FaltMap operation

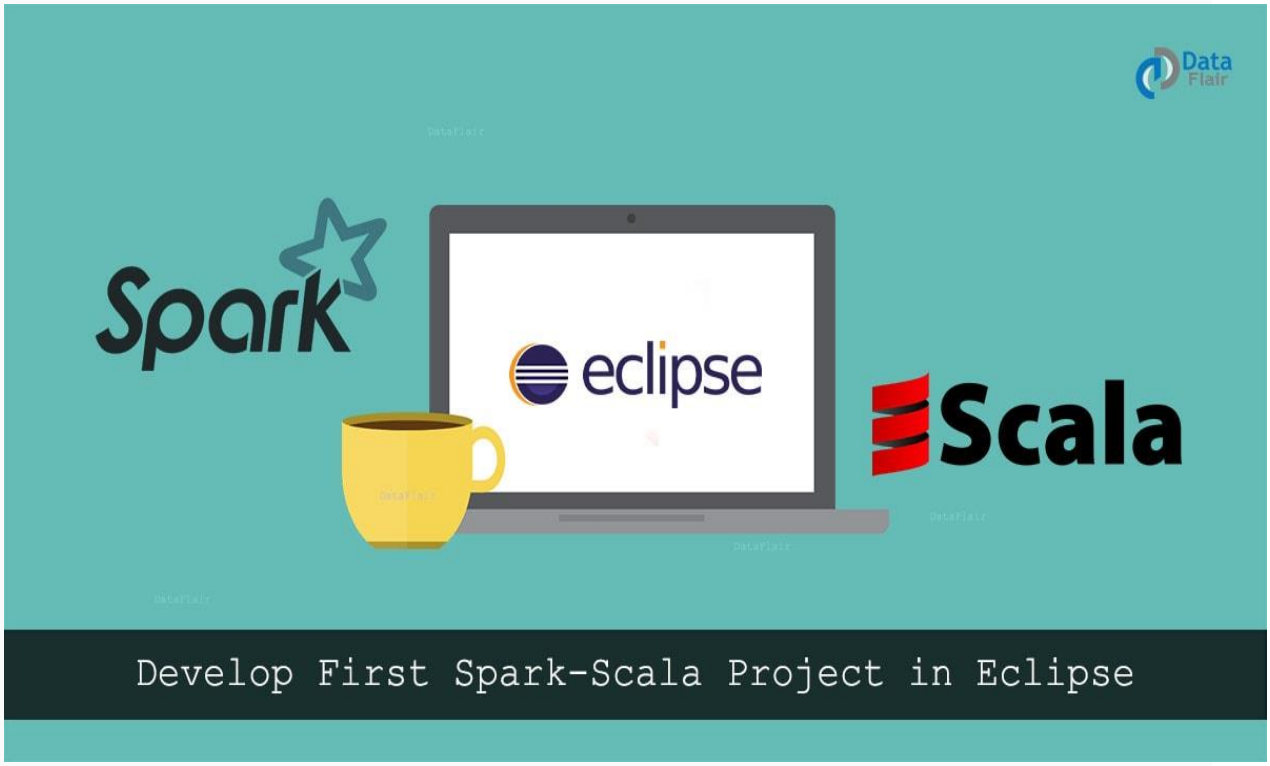
7.Create Spark project in Scala with Eclipse without Maven

1. Objective

This step by step tutorial will explain how to create a Spark project in Scala with Eclipse without Maven and how to submit the application after the

creation of jar. This Guide also briefs about the installation of **Scala** plugin in eclipse and setup spark environment in eclipse. Learn how to configure development environment for developing Spark applications in Scala in this tutorial.

If you are completely new to Apache Spark, I recommend you to read this **Apache Spark Introduction Guide.**



2. Steps to create Spark project in Scala

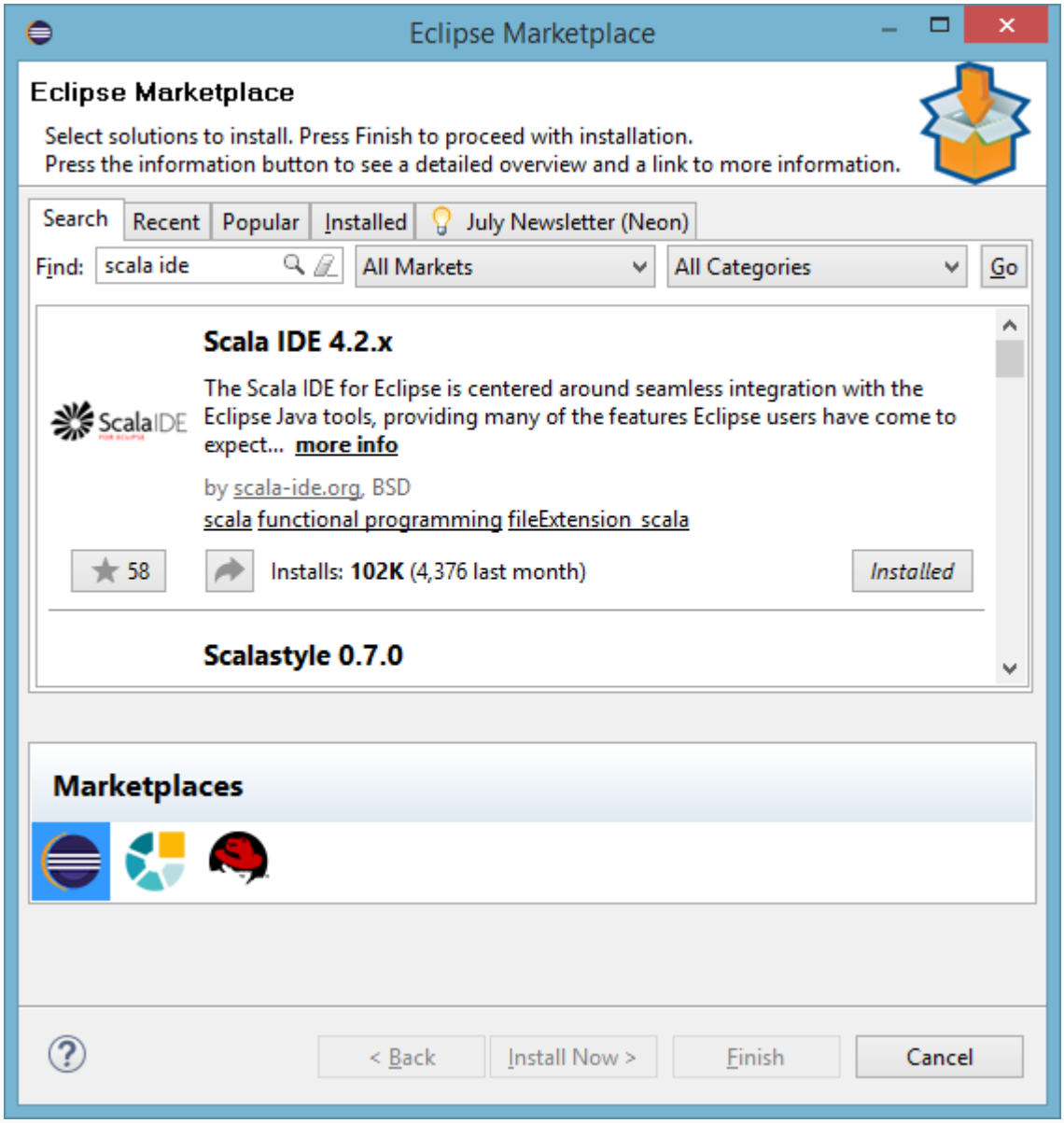
To create Spark Project in Scala with Eclipse without Maven follow the steps given below-

2.1. Platform Used / Required

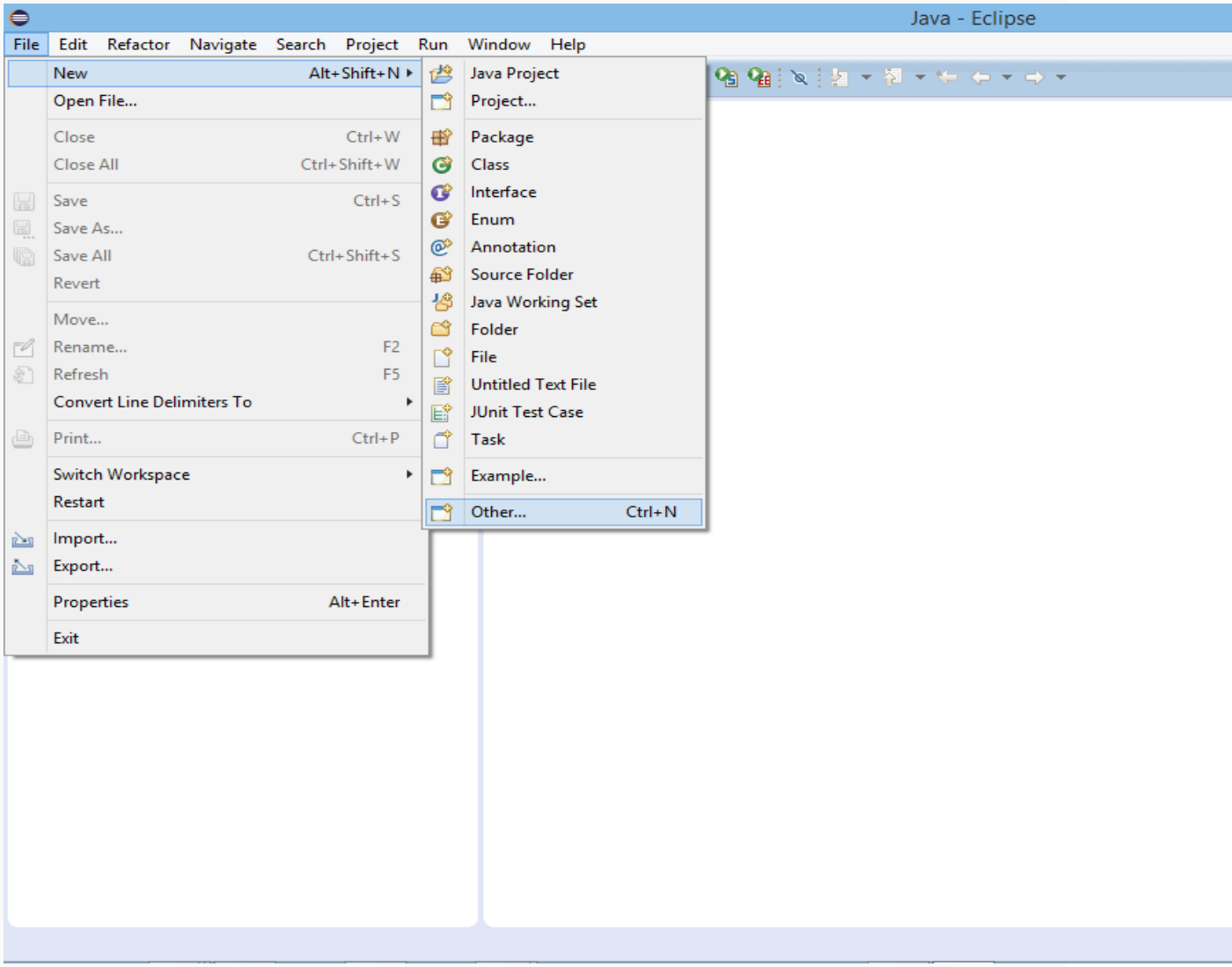
- **Operating System:** Windows / Linux / Mac
- **Java:** Oracle Java 7
- **Scala:** 2.11
- **Eclipse:** Eclipse Luna, Mars or later

2.2. Install Eclipse plugin for Scala

Open Eclipse Marketplace (**Help >> Eclipse Marketplace**) and search for "scala ide". Now install the Scala IDE. Alternatively, you can download Eclipse for Scala.


Eclipse Marketplace
Eclipse Marketplace
Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.
Search
Recent
Popular
Installed
July Newsletter (Neon)
Find:
scala ide
All Markets
All Categories
Go
Scala IDE 4.2.x
The Scala IDE for Eclipse is centered around seamless integration with the Eclipse Java tools, providing many of the features Eclipse users have come to expect... more info
by scala-ide.org, BSD
scala functional programming fileExtension .scala
58
Installs: 102K (4,376 last month)
Installed
Scalastyle 0.7.0
Marketplaces
< Back
Install Now >
Finish
Cancel

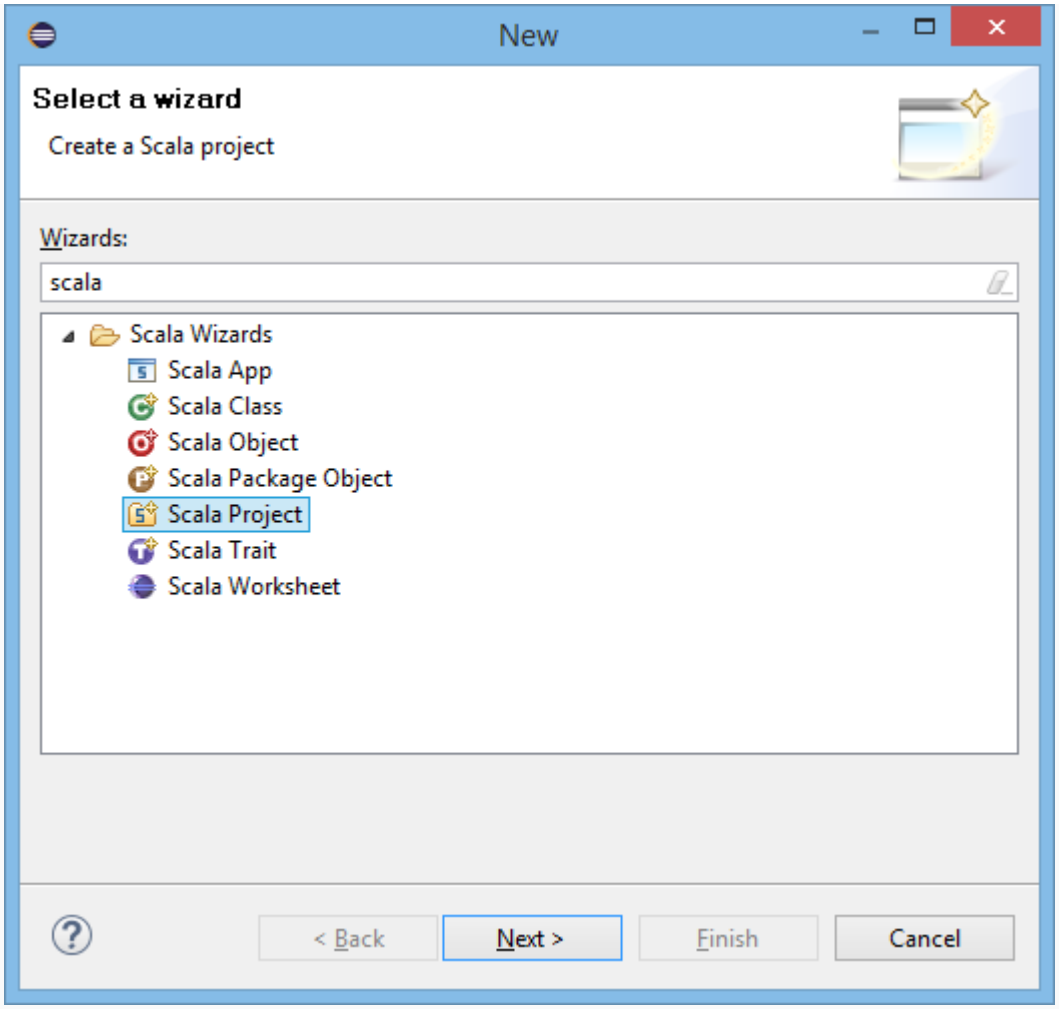
2.3. Create a New Spark Sc



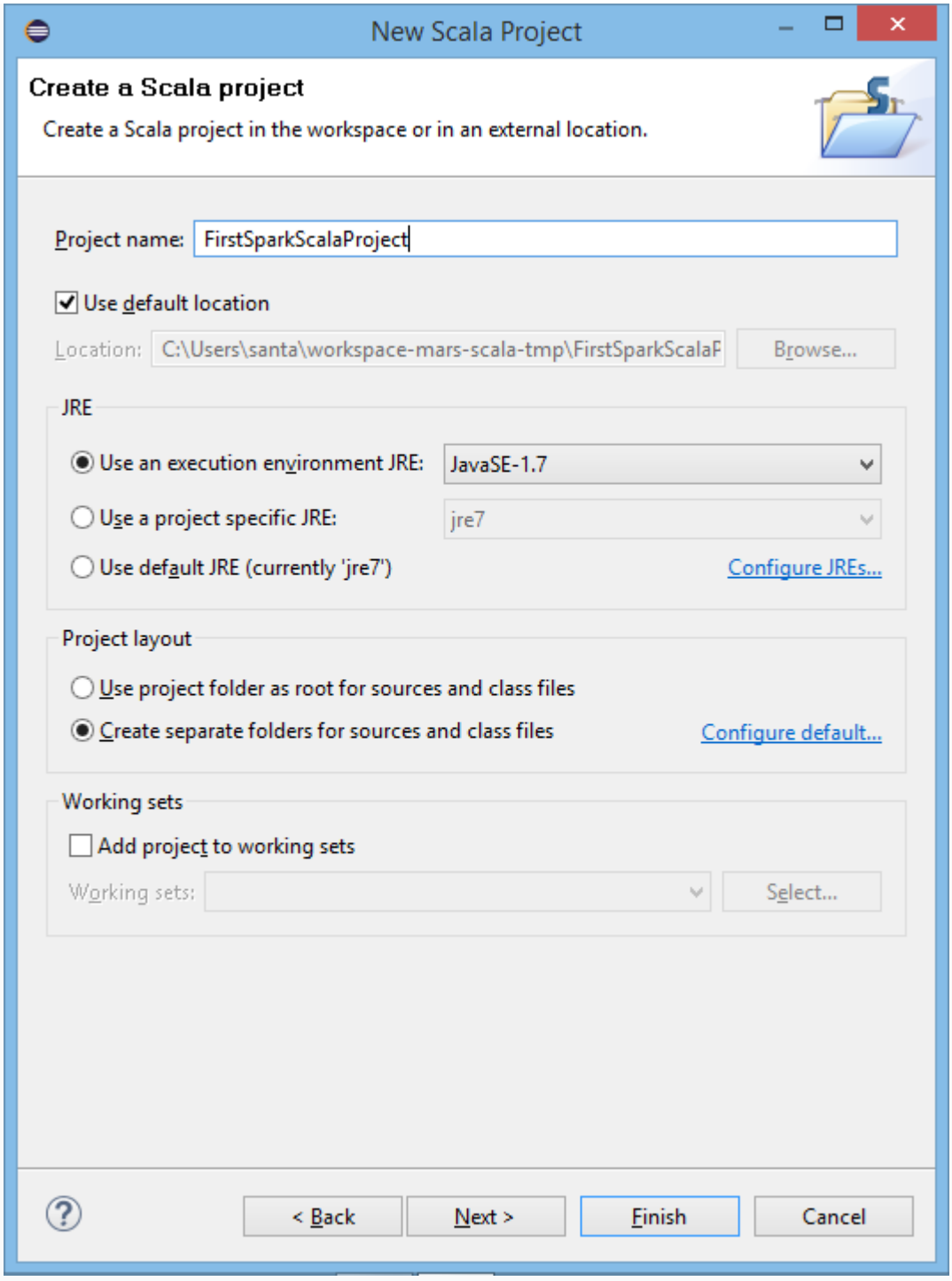
ala Project

To create a new Spark Scala project, click on **File >> New >> Other**

Select Scala Project:

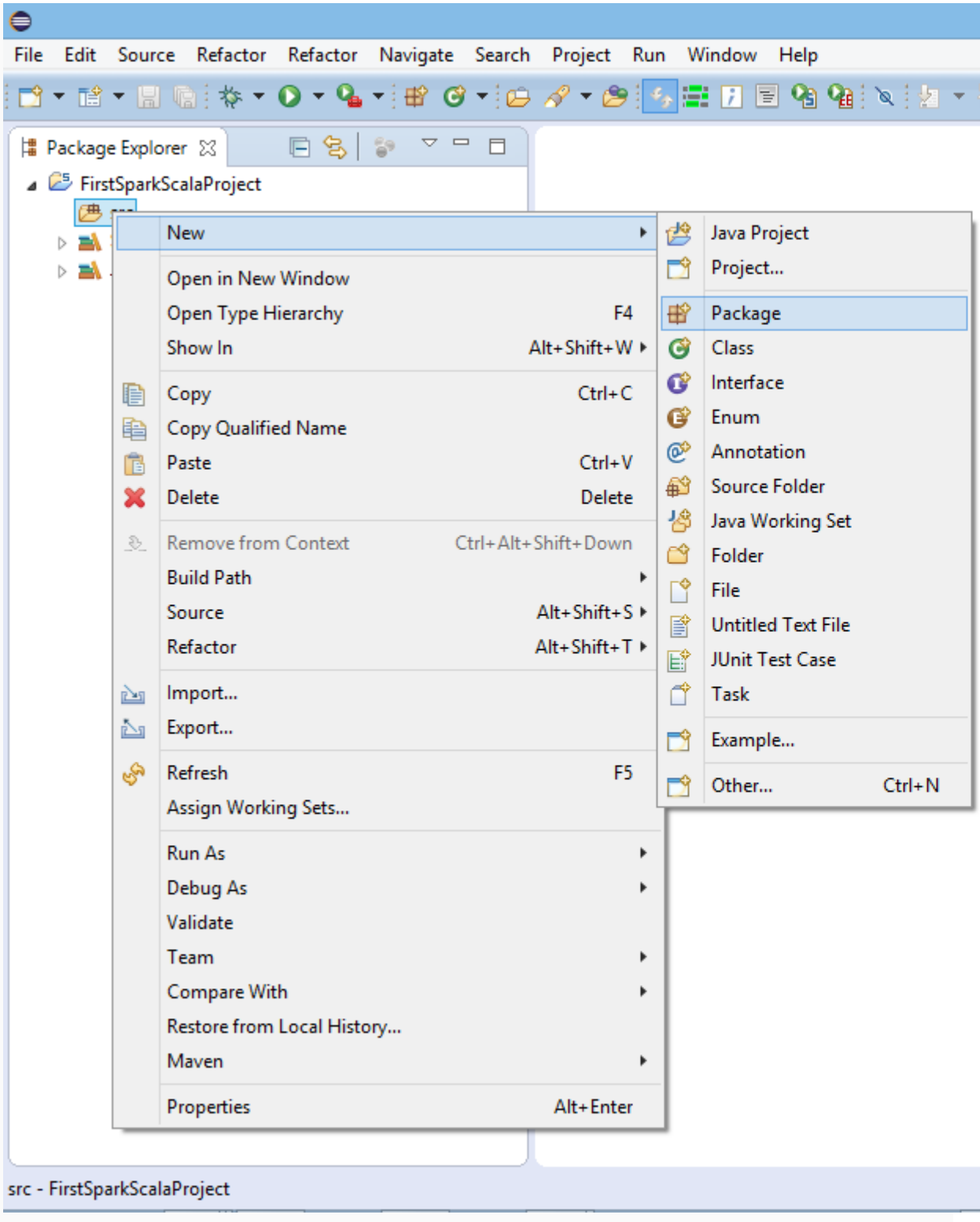


Supply Project Name:

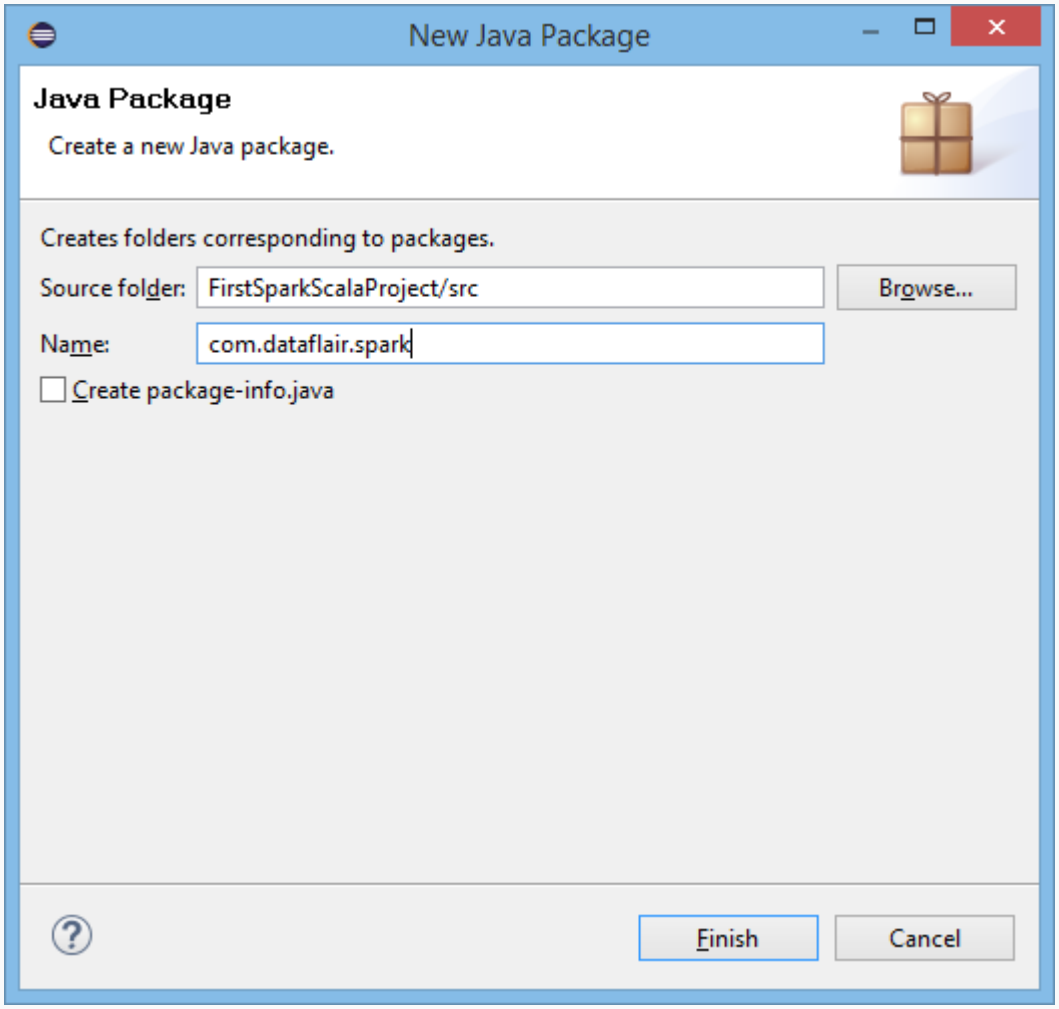


2.4. Create New Package

After creating the project, now create a new package.

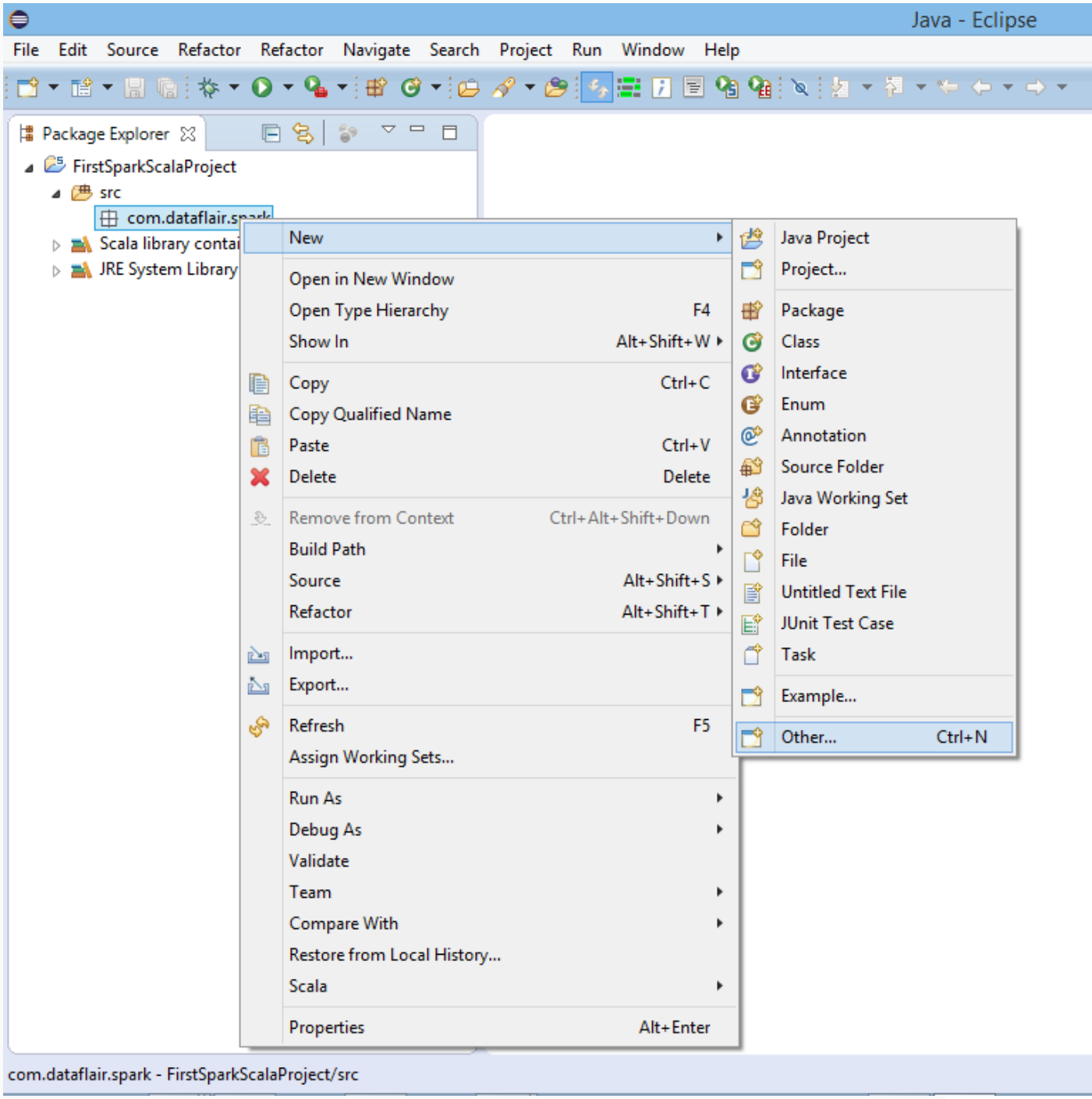

Java - Ec
File Edit Source Refactor Refactor Navigate Search Project Run Window Help
Package Explorer
FirstSparkScalaProject
New
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Assign Working Sets...
Run As
Debug As
Validate
Team
Compare With
Restore from Local History...
Maven
Properties Alt+Enter
Java Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Task
Example...
Other... Ctrl+N
src - FirstSparkScalaProject

Supply Package Name:

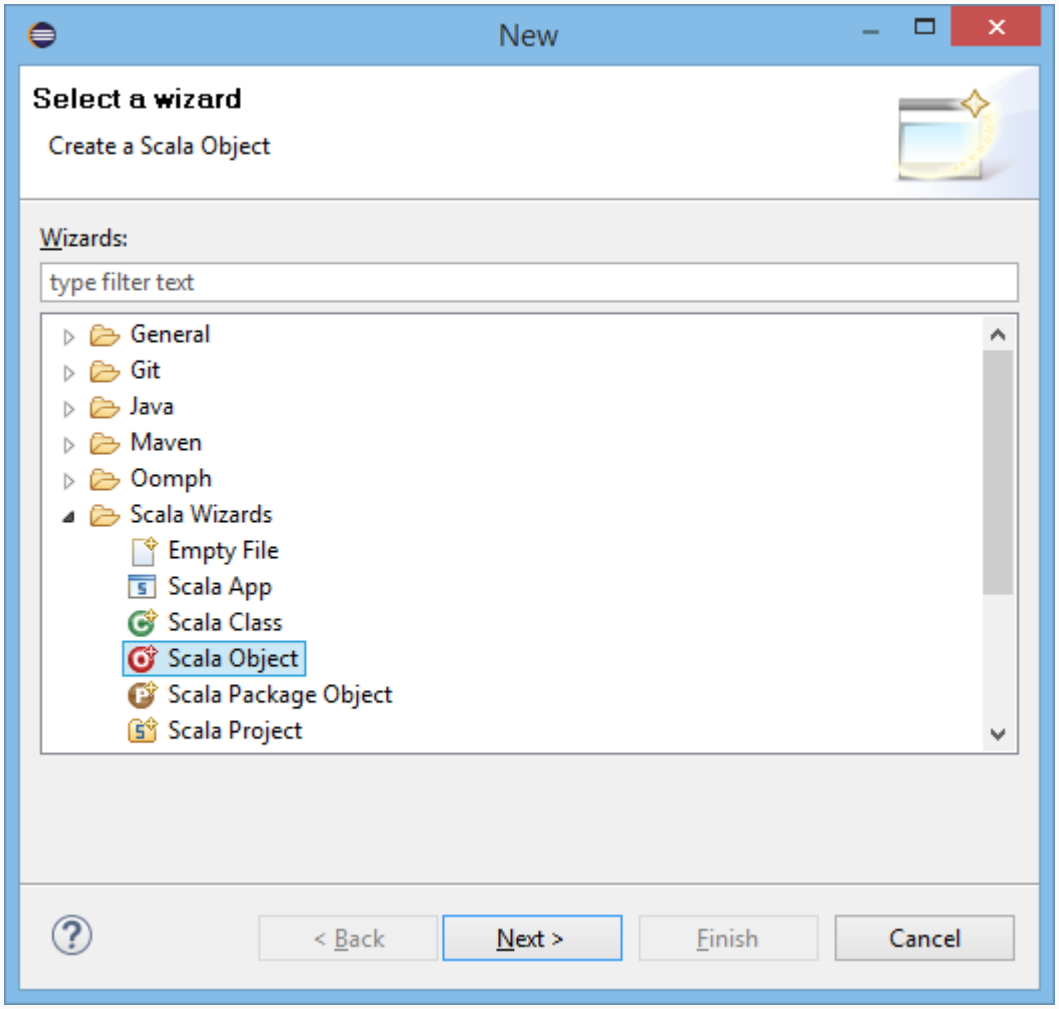


2.5. Create a New Scala Object

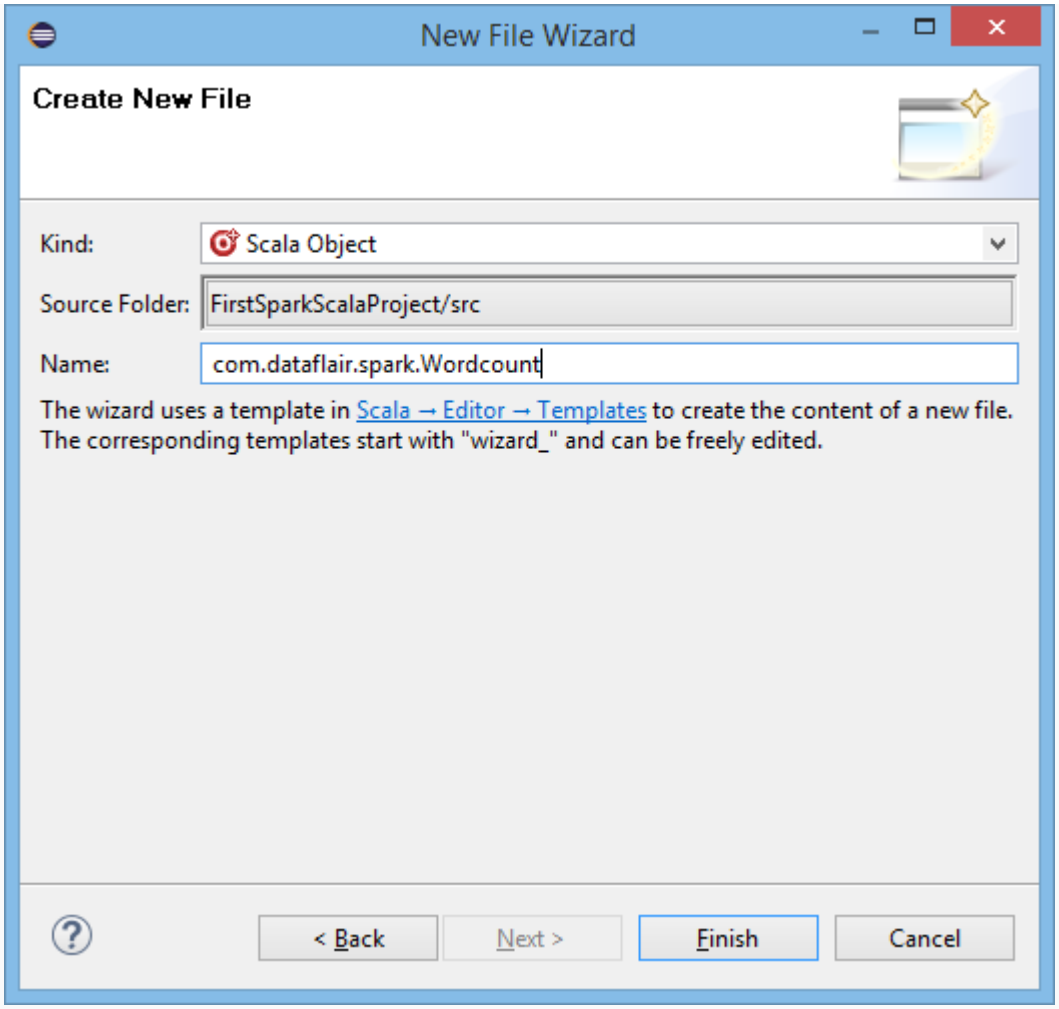
Now create a new Scala Object to develop Scala program for Spark application


Java - Eclipse
File Edit Source Refactor Refactor Navigate Search Project Run Window Help
Package Explorer
FirstSparkScalaProject
src
com.dataflair.spark
Scala library contai
JRE System Library
New
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Assign Working Sets...
Run As
Debug As
Validate
Team
Compare With
Restore from Local History...
Scala
Properties Alt+Enter
Java Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Task
Example...
Other... Ctrl+N
com.dataflair.spark - FirstSparkScalaProject/src

Select Scala Object:

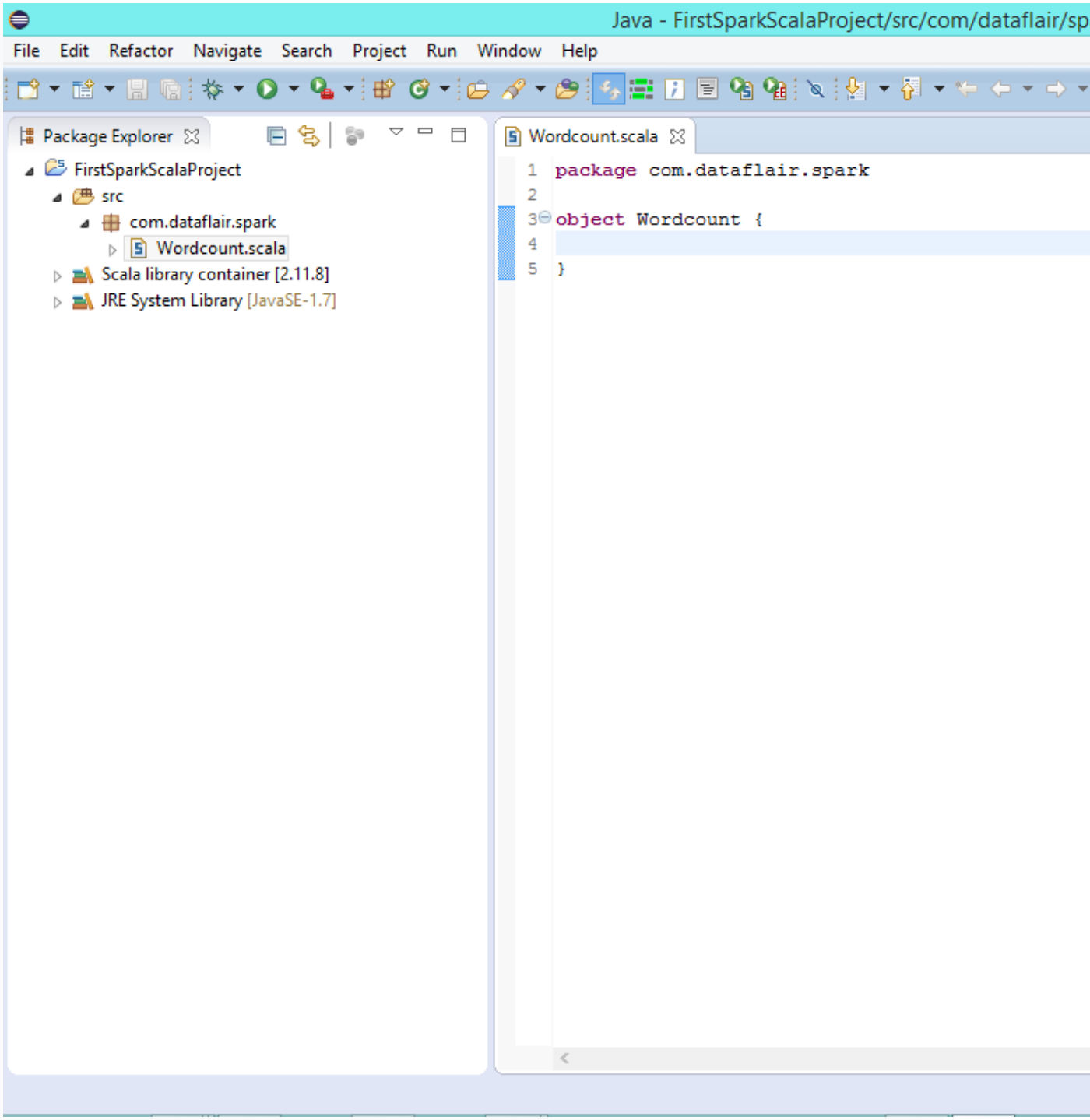


Supply Object Name:



2.6. New Scala Object in Editor

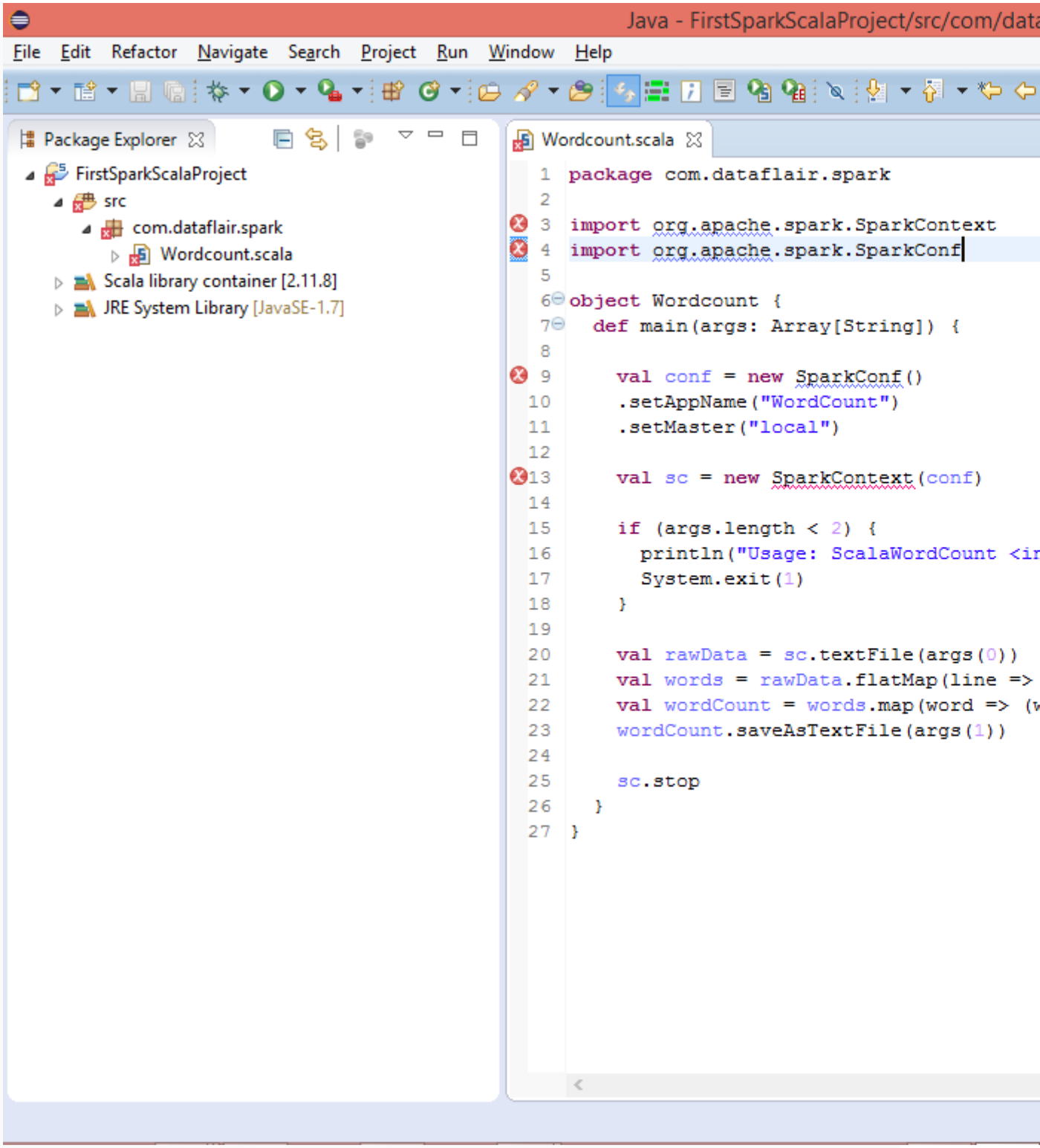
Scala object is ready now we can develop our Spark wordcount code in Scala-



2.7. Copy below Spark Scala Wordcount Code in Editor

```
1. package com.dataflair.spark
2.
3. import org.apache.spark.SparkContext
4. import org.apache.spark.SparkConf
5.
6. object Wordcount {
7. def main(args: Array[String]) {
8.
9. //Create conf object
```

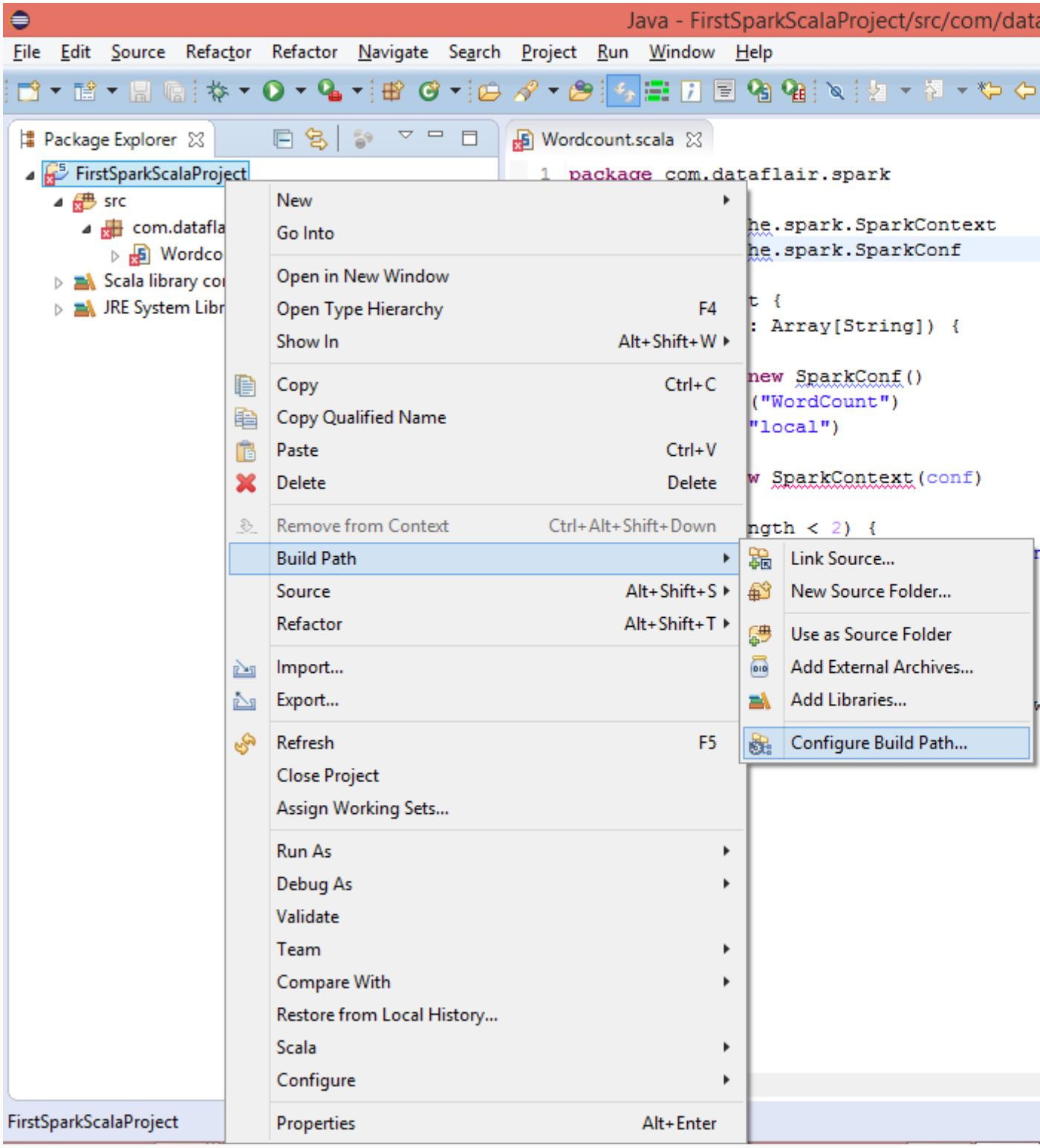
```
10. val conf = new SparkConf()
11. .setAppName("WordCount")
12.
13. //create spark context object
14. val sc = new SparkContext(conf)
15.
16. //Check whether sufficient params are supplied
17. if (args.length < 2) {
18. println("Usage: ScalaWordCount <input> <output>")
19. System.exit(1)
20. }
21. //Read file and create RDD
22. val rawData = sc.textFile(args(0))
23.
24. //convert the lines into words using flatMap operation
25. val words = rawData.flatMap(line => line.split(" "))
26.
27. //count the individual words using map and reduceByKey operation
28. val wordCount = words.map(word => (word, 1)).reduceByKey(_ + _)
29.
30. //Save the result
31. wordCount.saveAsTextFile(args(1))
32.
33. //stop the spark context
34. sc.stop
35. }
36. }
```



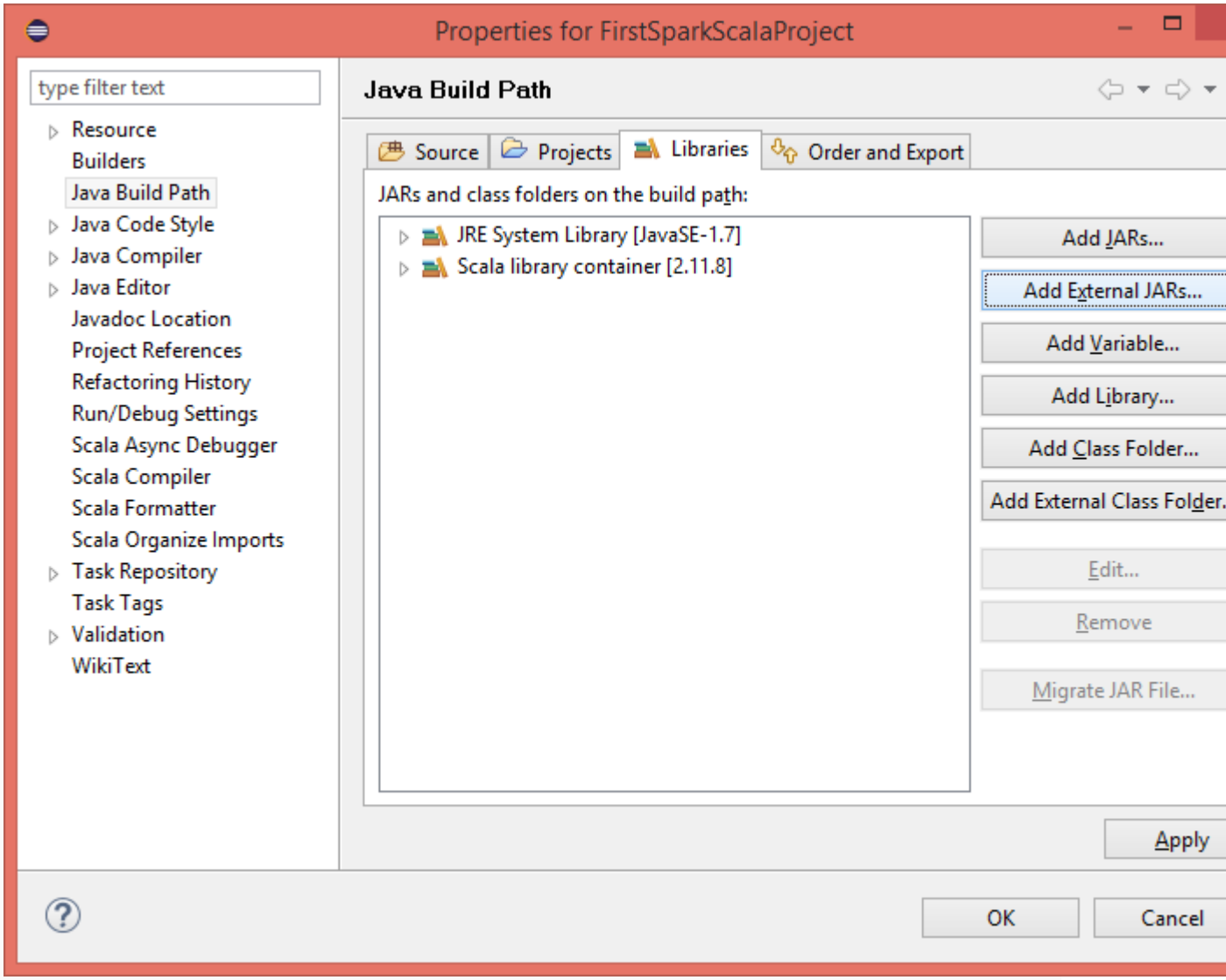
You will see lots of error due to missing libraries.

2.8. Add Spark Libraries

Configure Spark environment in Eclipse: Right click on project name >> build path >> Configure Build Path

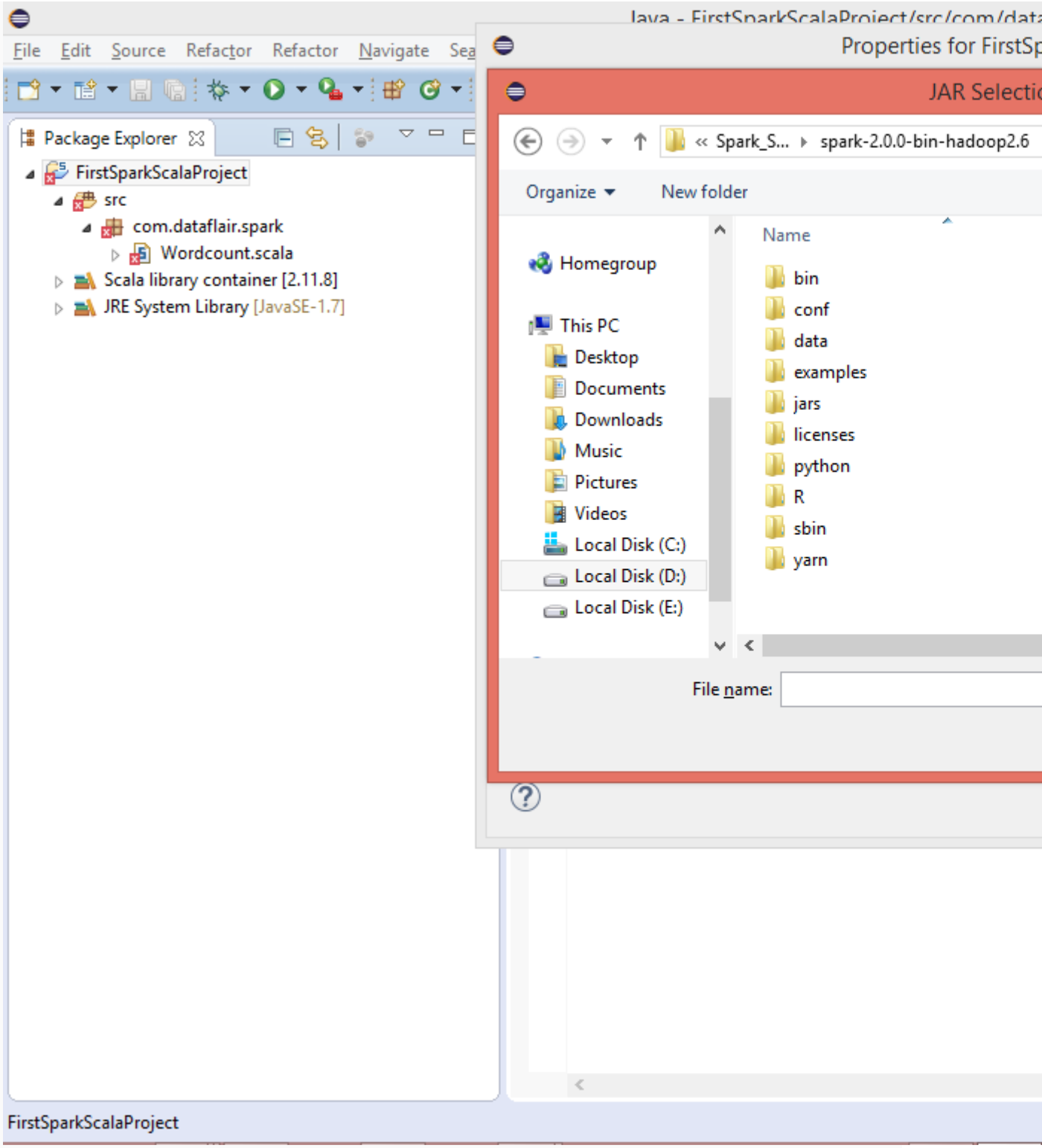


Add the External Jars:

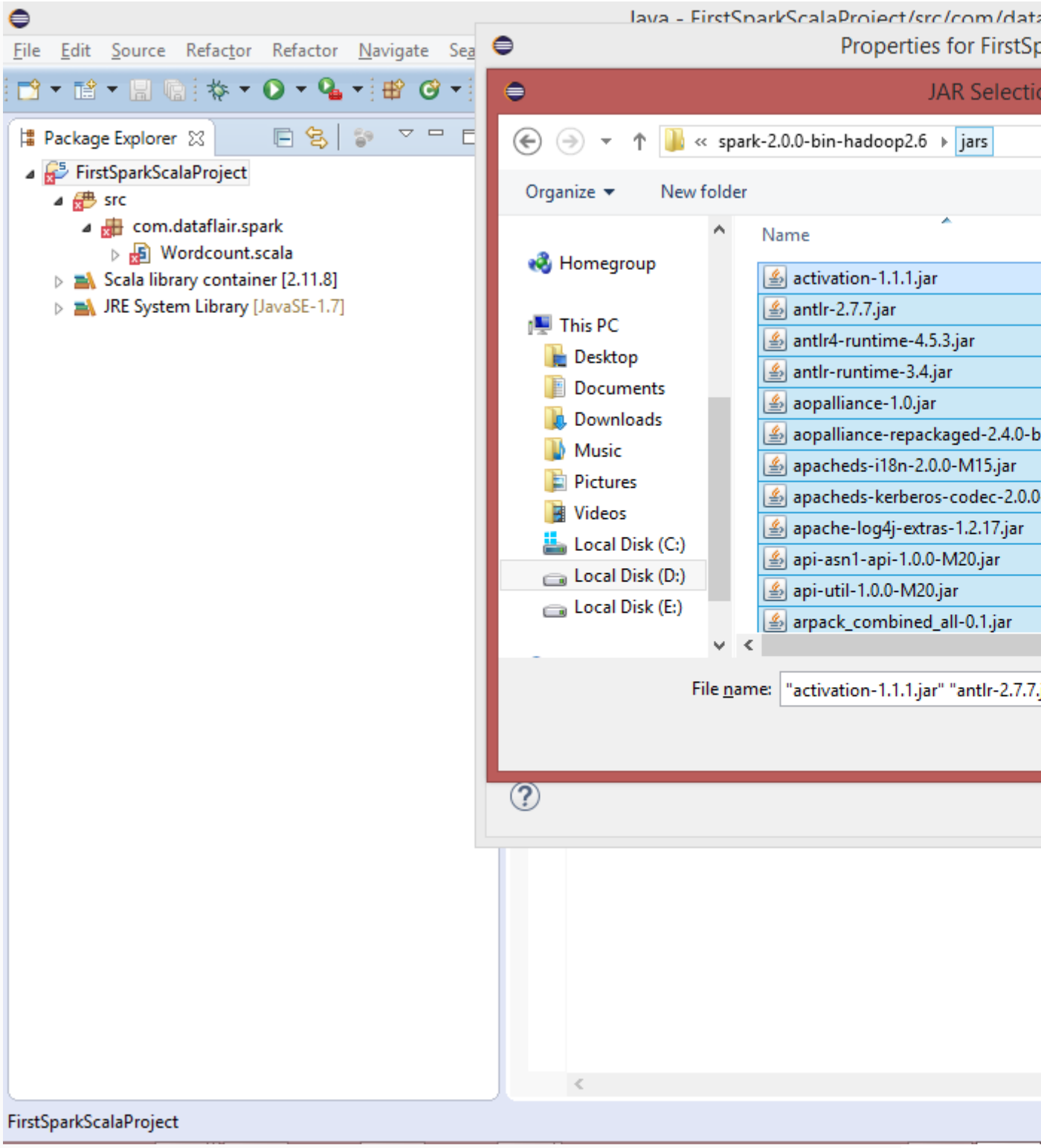


2.9. Select Spark Jars and insert

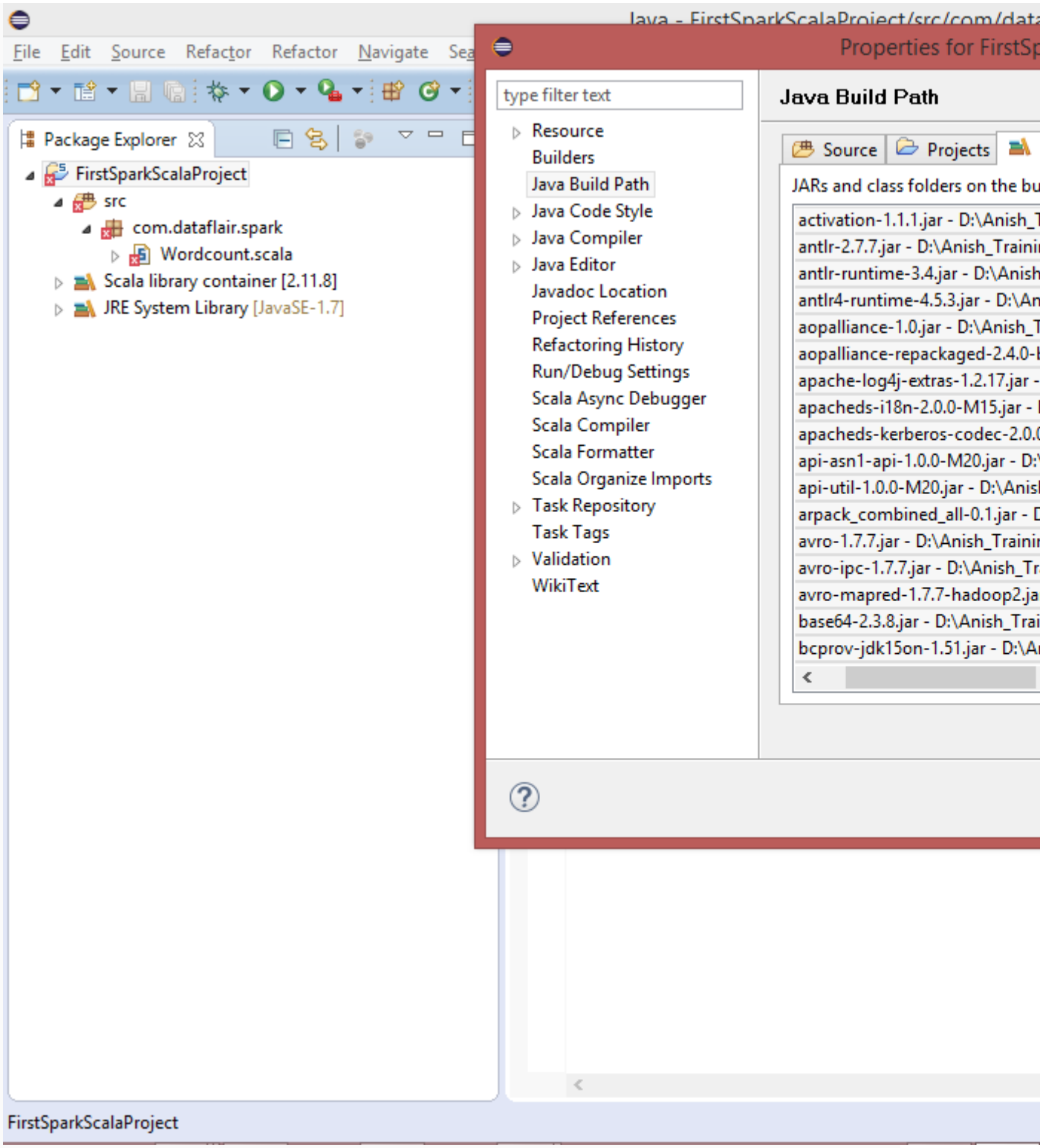
You should have spark setup available in developing environment, it will be needed for spark libraries.



Go to "**Spark-Home >> jars**" and select all the jars:

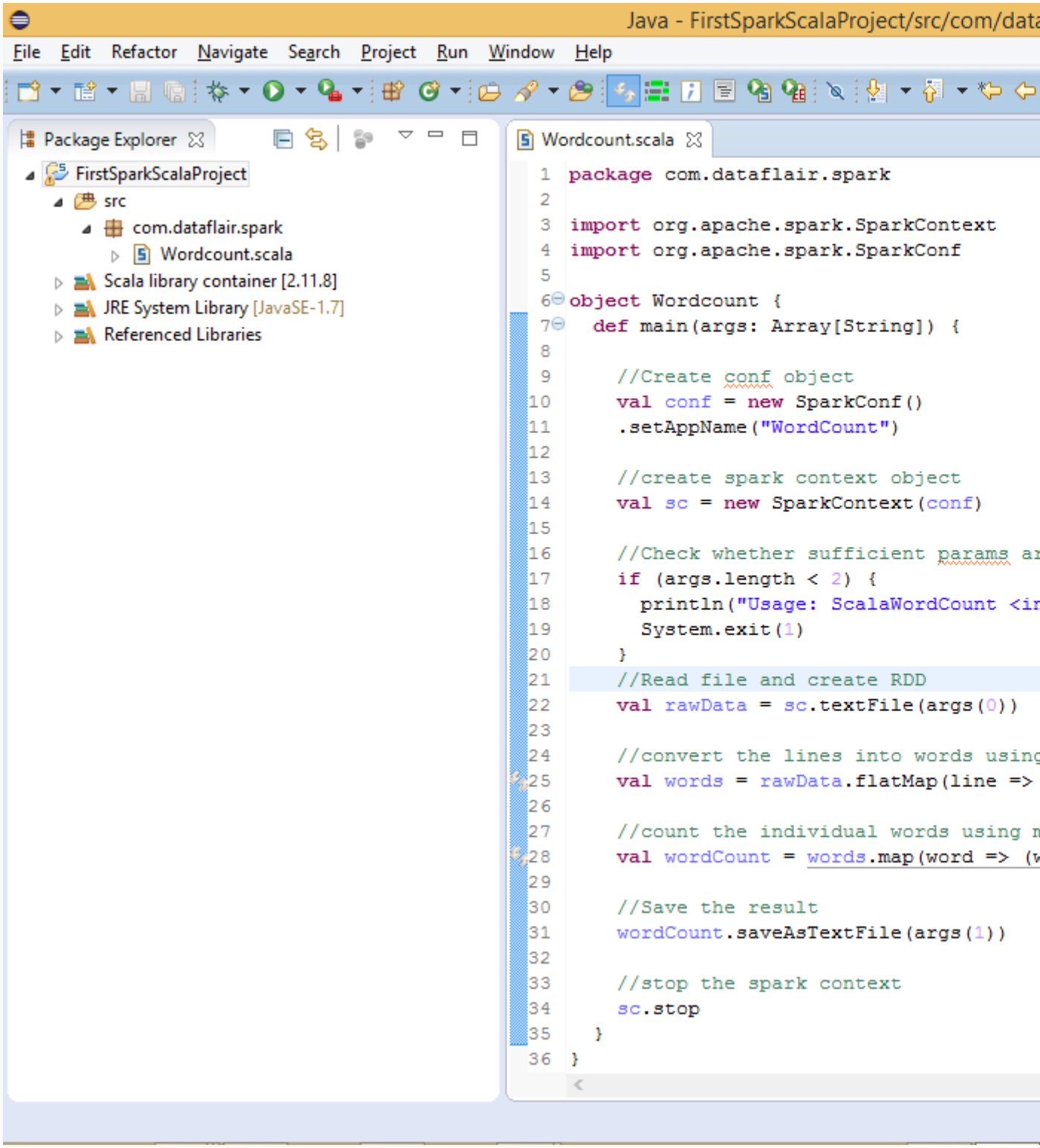


Import the selected jar:



2.10. Spark Scala Word Count Program

After importing the libraries all the errors will be removed.

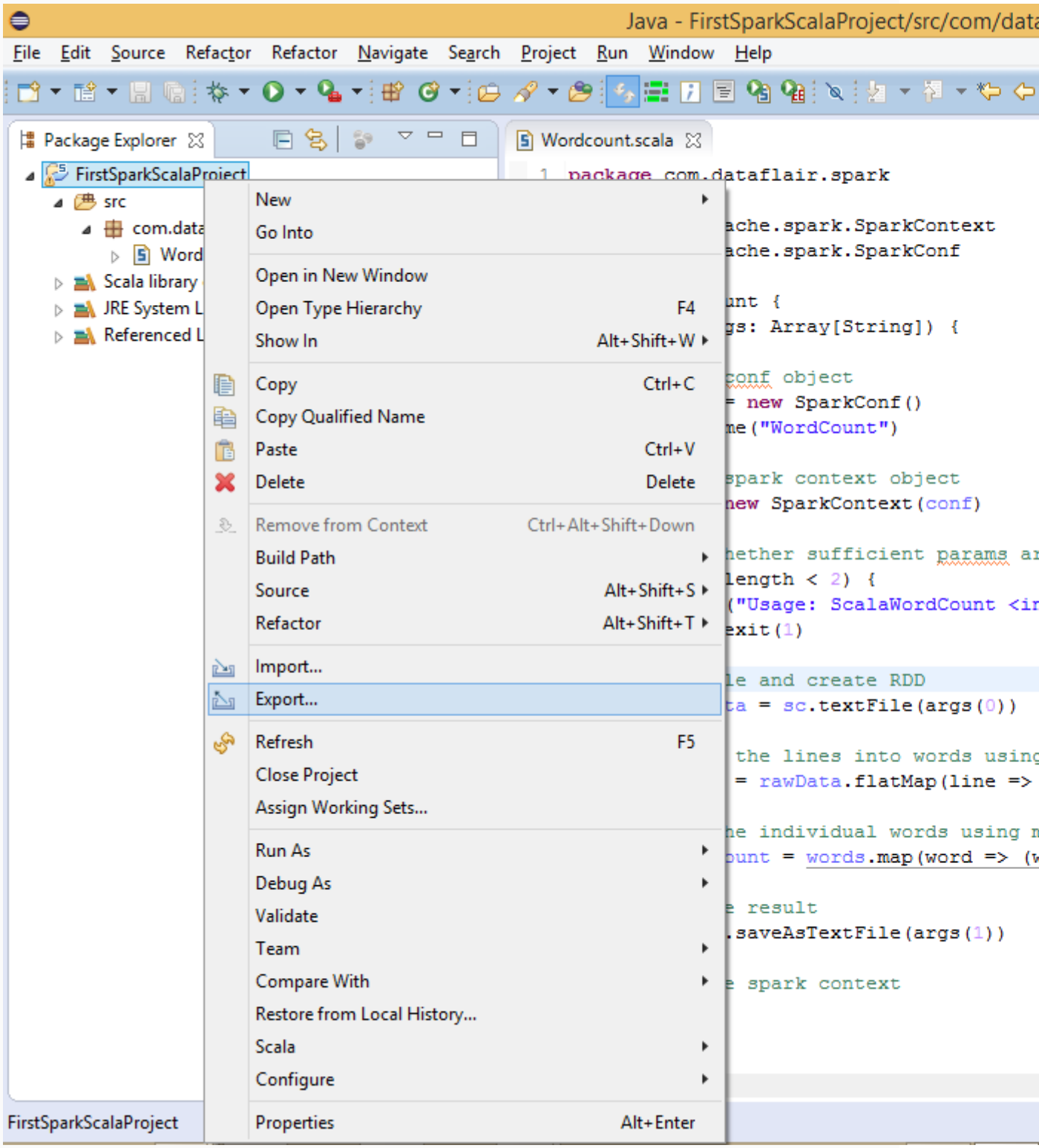


We have successfully created Spark environment in Eclipse and developed Spark Scala program. Now let's deploy the Spark job on Linux, before deploying/running the application you must have Spark Installed.

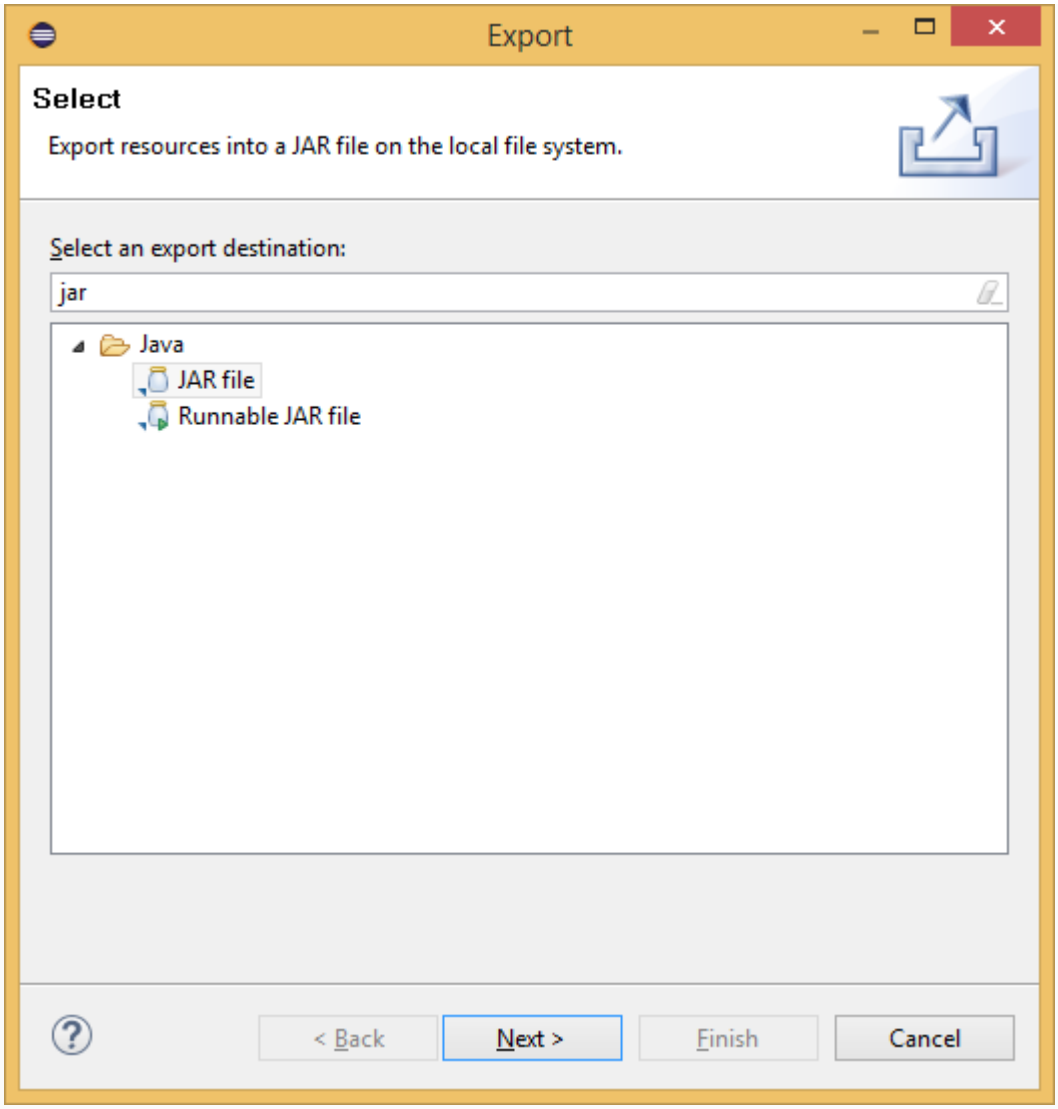
Follow this links to install Apache Spark on single node cluster or on the multi-node cluster.

2.11. Create the Spark Scala Program Jar File

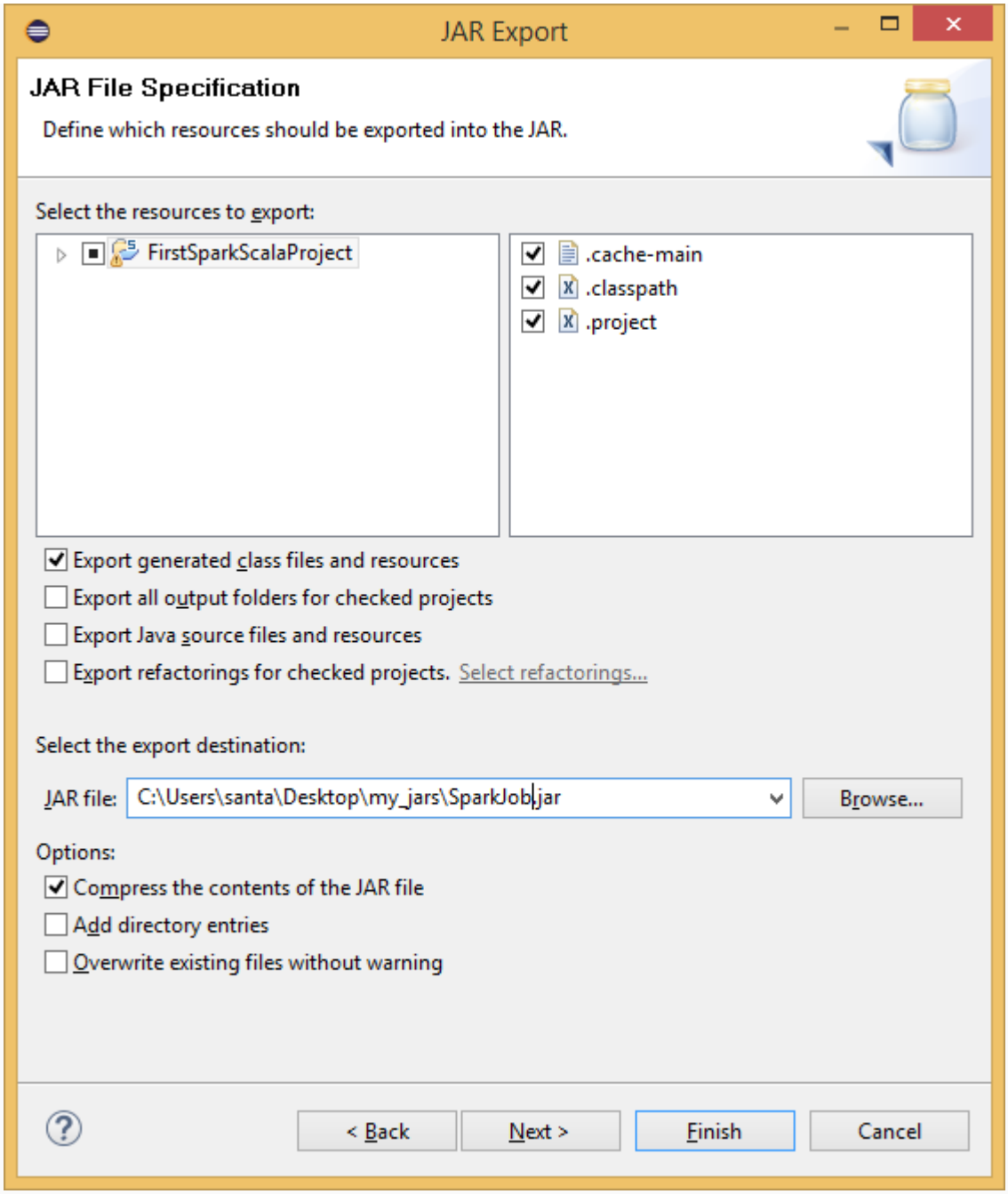
Before running created **Spark word count application** we have to create a jar file. Right click on **project >> export**



Select Jar-file Option to Export:



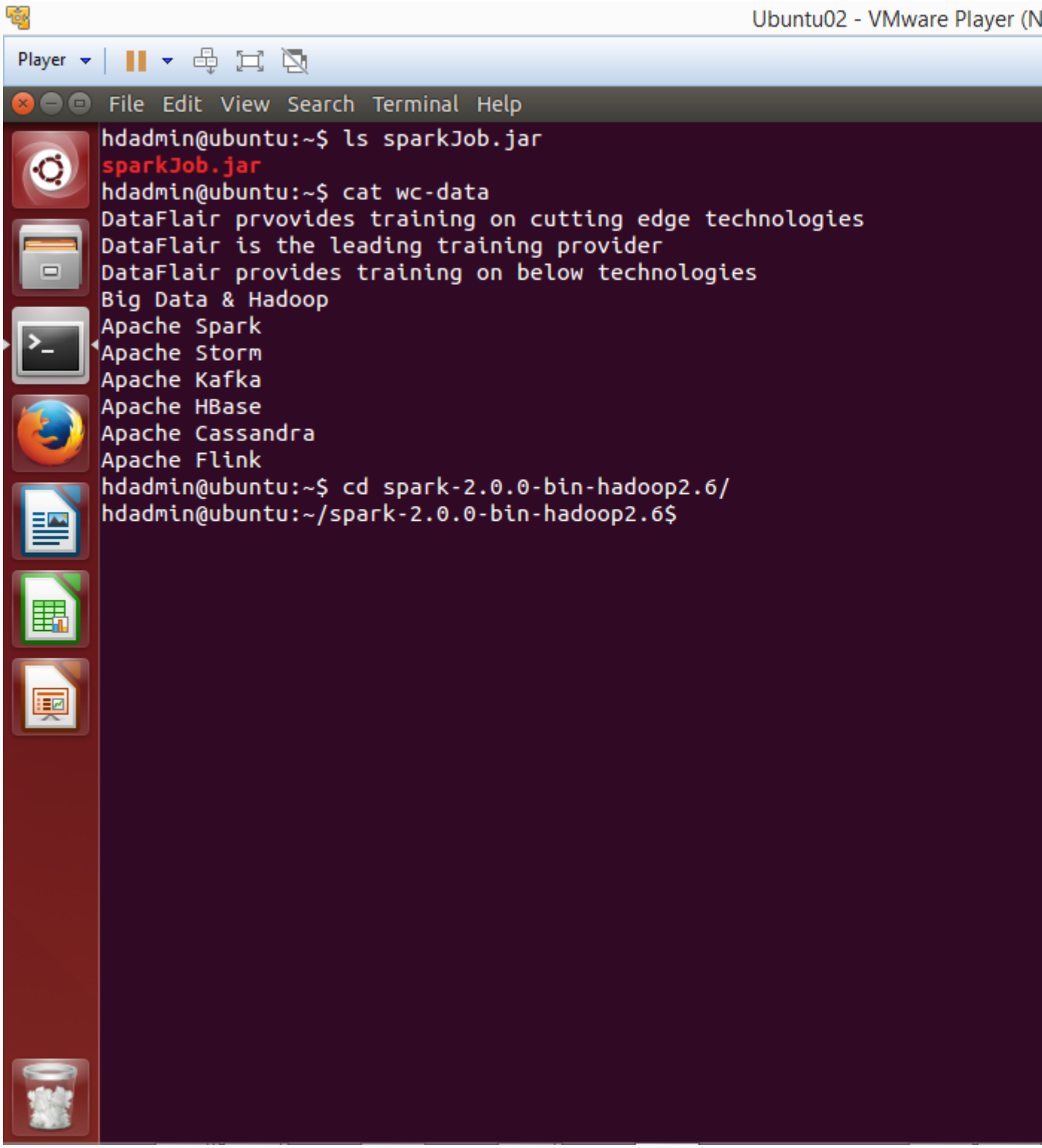
Create the Jar file:



The jar file for the Spark Scala application has been created, now we need to run it.

2.12. Go to Spark Home Directory

Login to Linux and open terminal. To run Spark Scala application we will be using Ubuntu Linux. Copy the jar file to Ubuntu and create one text file, which we will use as input for Spark Scala wordcount job.



2.13. Submit Spark Application using spark-submit script

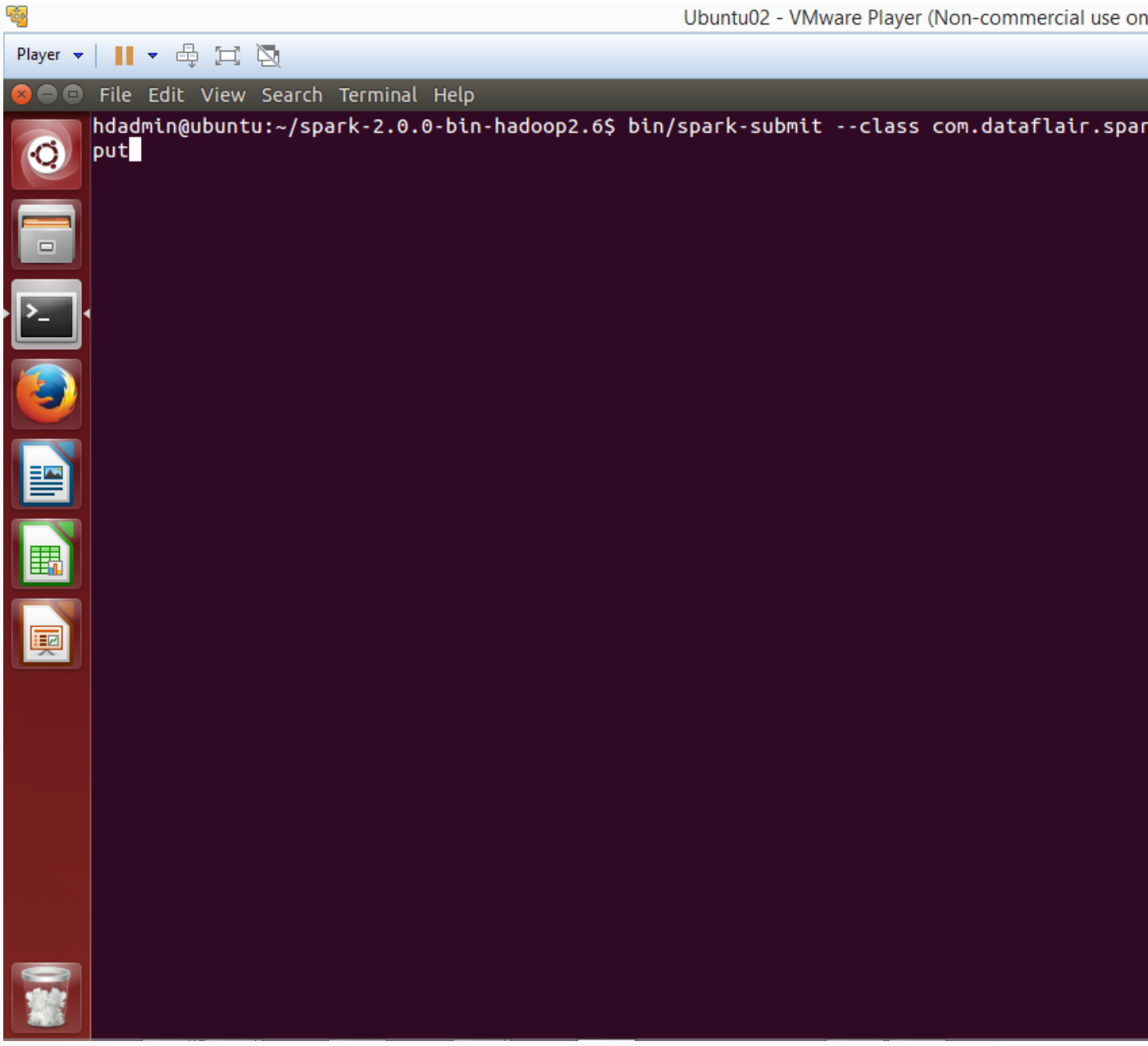
To submit the Spark application using below command:

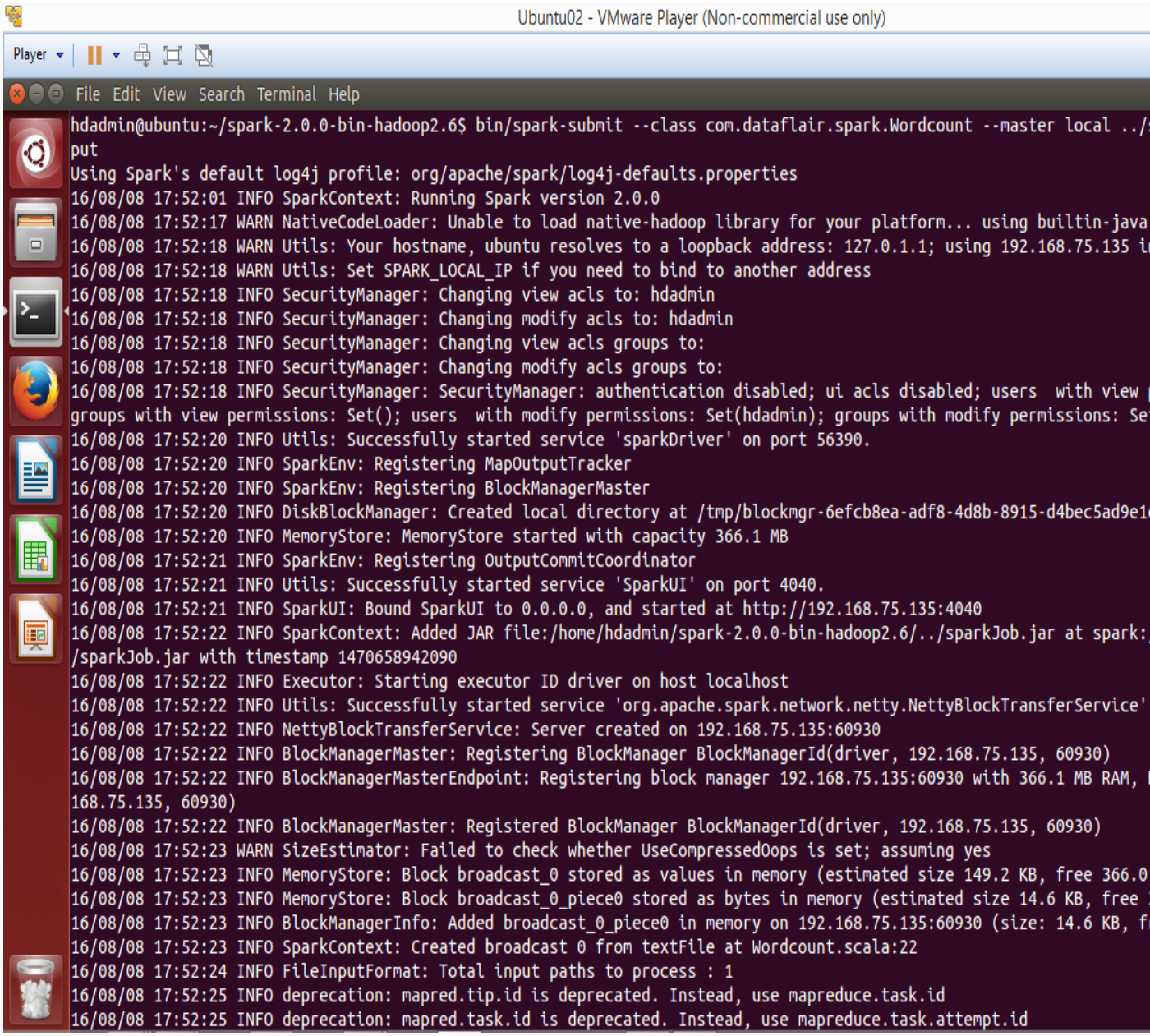
```
bin/spark-submit --class <Qualified-Class-Name> --master <Master> <Path-Of-Jar-File> <Input-Path> <Output-Path>
```

```
bin/spark-submit --class com.dataflair.spark.Wordcount --master local ../sparkJob.jar ../wc-data output
```

Let's understand above command:

- **bin/spark-submit:** To submit Spark Application
- **–class:** To specify the class name to execute
- **–master:** Master (local / <Spark-URI> / yarn)
- **<Jar-Path>:** The jar file of application
- **<Input-Path>:** Location from where input data will be read
- **<Output-Path>:** Location where Spark application will write output

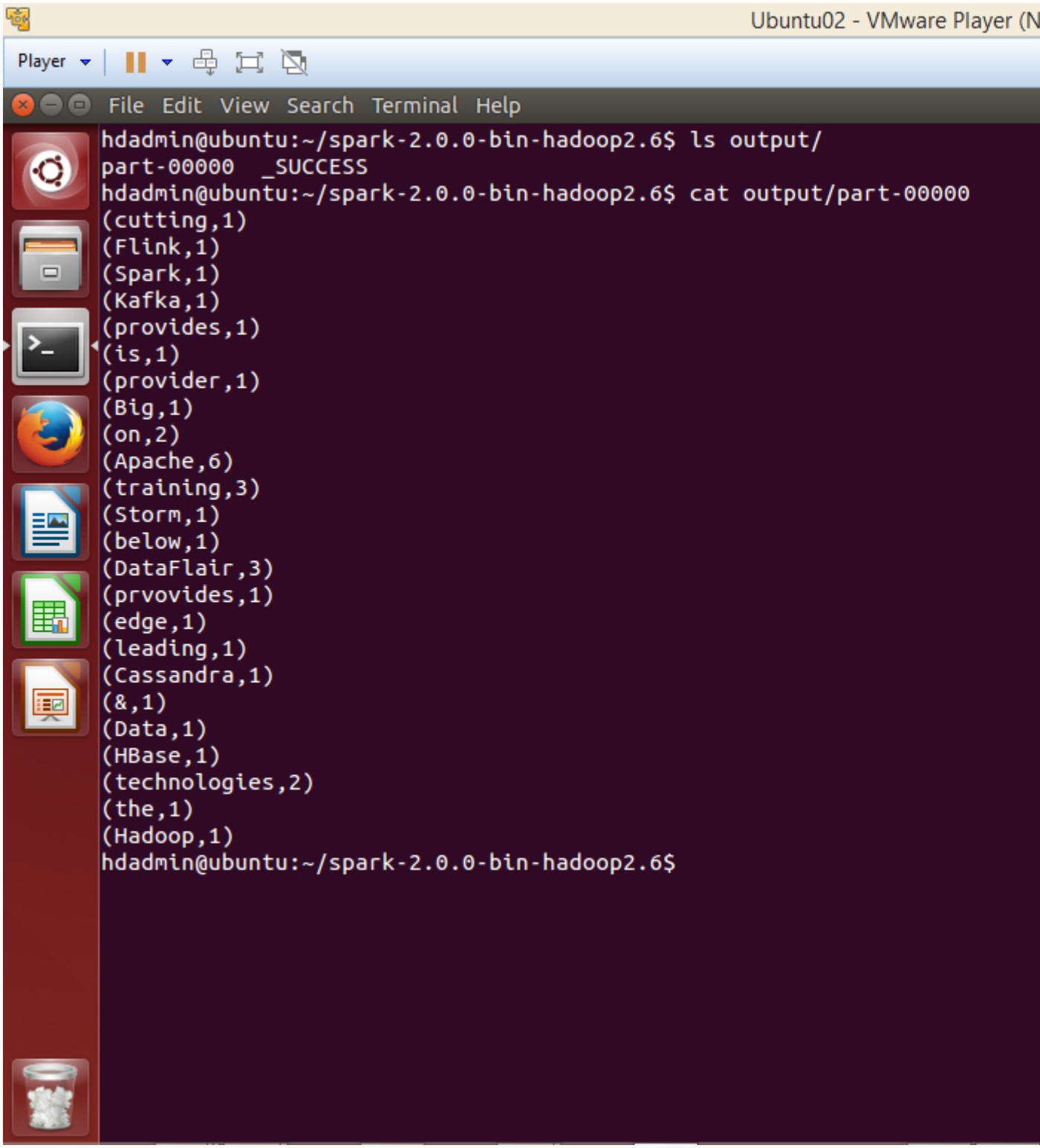




The application has been completed successfully, now browse the result.

2.14. Browse the result

Browse the output directory and open the file with name part-xxxxx which contains the output of the application.



We have successfully created Spark project in Scala and deployed on Ubuntu.

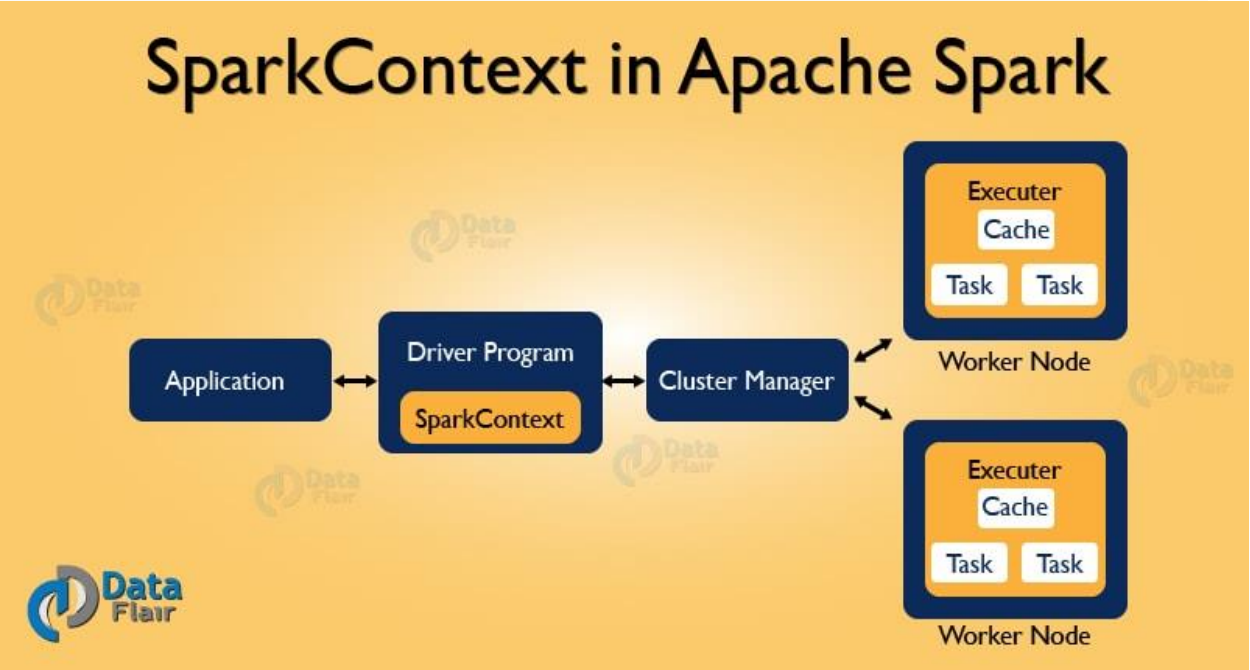
To play with Spark First learn **RDD**, **DataFrame**, **DataSet in Apache Spark** and then refer this **Spark shell commands tutorial** to practically implements Spark functionalities.

8.LEARN SPARKCONTEXT – INTRODUCTION AND FUNCTIONS

1. Objective

SparkContext is the entry gate of **Apache Spark** functionality. The most important step of any Spark driver application is to generate SparkContext. It allows your Spark Application to access Spark Cluster with the help of Resource Manager (**YARN/Mesos**). To create SparkContext, first **SparkConf** should be made. The SparkConf has a configuration parameter that our Spark driver application will pass to SparkContext.

In this Apache Spark tutorial, we will deeply understand what is SparkContext in Spark. How to create SparkContext Class in Spark with the help of Spark-Scala word count program. We will also learn various tasks of SparkContext and how to stop SparkContext in Apache Spark.



Learn how to install Apache Spark in standalone mode and Apache Spark installation in a multi-node cluster.

2. What is SparkContext in Apache Spark?

It is the entry point of Spark functionality. The most important step of any Spark driver application is to generate SparkContext. It allows your Spark Application to access Spark Cluster with the help of Resource Manager. The

resource manager can be one of these three- **Spark Standalone**, **YARN**, **Apache Mesos**.

3. How to Create SparkContext Class?

If you want to create SparkContext, first **SparkConf** should be made. The SparkConf has a configuration parameter that our Spark driver application will pass to SparkContext. Some of these parameter defines properties of Spark driver application. While some are used by Spark to allocate resources on the cluster, like the number, memory size, and cores used by executor running on the worker nodes.

In short, it guides how to access the Spark cluster. After the creation of a SparkContext object, we can invoke functions such as **textFile, sequenceFile, parallelize** etc. The different contexts in which it can run are local, yarn-client, Mesos URL and Spark URL.

Once the SparkContext is created, it can be used to create RDDs, broadcast variable, and accumulator, ingress Spark service and run jobs. All these things can be carried out until SparkContext is stopped.

4. Stopping SparkContext

Only one SparkContext may be active per JVM. You must stop the active it before creating a new one as below:

stop(): Unit

It will display the following message:

INFO SparkContext: Successfully stopped SparkContext

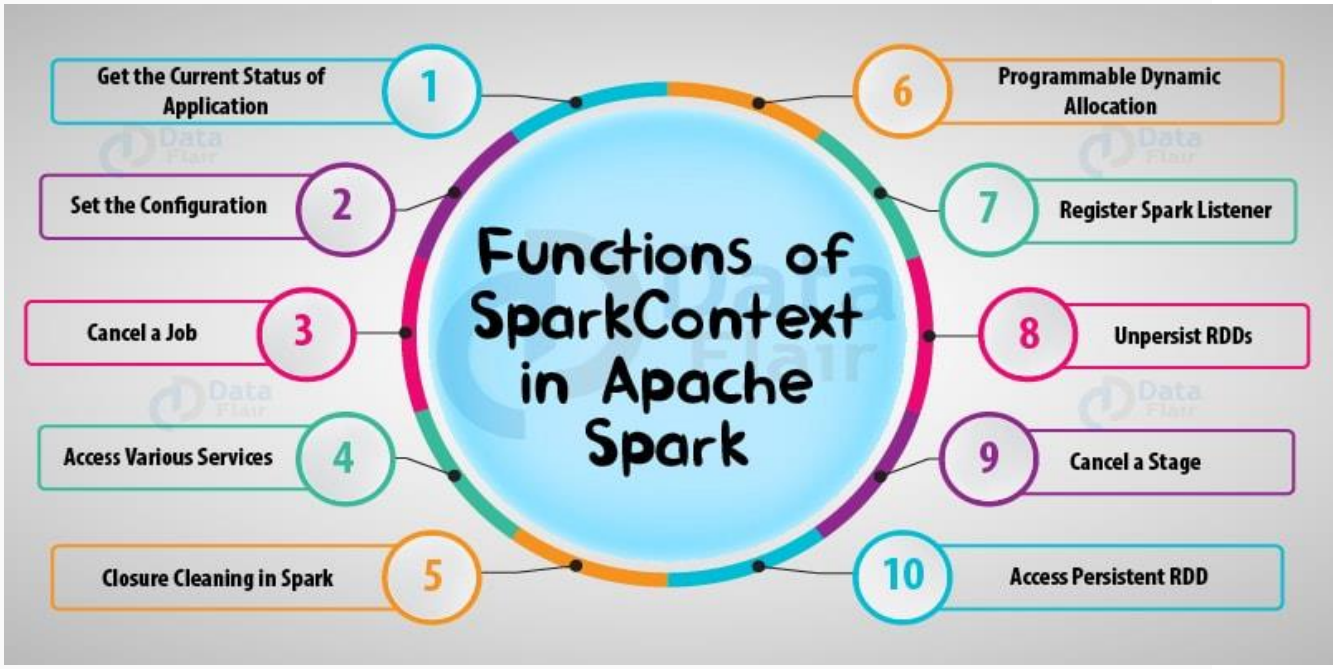
5. Spark Scala Word Count Example

Let's see how to create SparkContext using SparkConf with the help of Spark-Scala word count example-

```
1.  package com.dataflair.spark
2.
3.  import org.apache.spark.SparkContext
4.  import org.apache.spark.SparkConf
5.
6.  object Wordcount {
7.  def main(args: Array[String]) {
8.
9.  //Create conf object
10. val conf = new SparkConf()
11. .setAppName("WordCount")
12.
13. //create spark context object
14. val sc = new SparkContext(conf)
15.
16. //Check whether sufficient params are supplied
```

```
17. if (args.length < 2) {
18. println("Usage: ScalaWordCount <input> <output>")
19. System.exit(1)
20. }
21. //Read file and create RDD
22. val rawData = sc.textFile(args(0))
23.
24. //convert the lines into words using flatMap operation
25. val words = rawData.flatMap(line => line.split(" "))
26.
27. //count the individual words using map and reduceByKey operation
28. val wordCount = words.map(word => (word, 1)).reduceByKey(_ + _)
29.
30. //Save the result
31. wordCount.saveAsTextFile(args(1))
32.
33. //stop the spark context
34. sc.stop
35. }
36. }
```

6. Functions of SparkContext in Apache Spark



6.1. To get the current status of Spark Application

- **SpkEnv –** It is a runtime environment with Spark's public services. It interacts with each other to establish a distributed computing platform for Spark Application. A SparkEnv object that holds the required runtime

services for running Spark application with the different environment for the driver and executor represents the Spark runtime environment.

- **SparkConf –** The Spark Properties handles maximum applications settings and are configured separately for each application. We can also easily set these properties on a SparkConf. Some common properties like master URL and application name, as well as an arbitrary key-value pair, configured through the**set()** method.
- **Deployment environment (as master URL) –** Spark deployment environment are of two types namely local and clustered. **Local mode** is non-distributed single-JVM deployment mode. All the execution components – driver, executor, LocalSchedulerBackend, and master are present in same single JVM. Hence, the only mode where drivers are useful for execution is the local mode. For testing, debugging or demonstration purpose, the local mode is suitable because it requires no earlier setup to launch spark application. While in clustered mode, the Spark runs in distributive mode. Learn Spark Cluster Manager in detail.

6.2. To set the configuration

- **Master URL –** The master method returns back the current value of *spark.master* which is deployment environment in use.
- **Local properties-Creating Logical Job Groups –** The reason of local properties concept is to form logical groups of jobs by means of properties that create the separate job launched from different threads belong to a single logic group. We can set a local property which will affect Spark jobs submitted from a thread, such as the Spark fair scheduler pool.
- **Default Logging level –** It lets you set the root login level in a Spark application, for example, Spark Shell.

6.3. To Access various services

It also helps in accessing services like *TaskScheduler, LiveListenBus, BlockManager, SchedulerBackend, ShuffelManager* and the optional *ContextCleaner*.

6.4. To Cancel a job

cancleJob simply requests *DAGScheduler* to drop a Spark job.

Learn about Spark **DAG(Directed Acyclic Graph)** in detail.

6.5. To Cancel a stage

cancleStage simply requests *DAGScheduler* to drop a Spark stage.

6.6. For Closure cleaning in Spark

Spark cleanups the closure every time an Action occurs, i.e. the body of Action before it is serialized and sent over the wire to execute. The clean method in SparkContext does this. This, in turn, calls *ClosureClean.clean* method. It not only cleans the closure but also referenced closure is clean transitively. It assumes serializable until it does not explicitly reference unserializable objects.

6.7. To Register Spark listener

We can register a custom *SparkListenerInterface* with the help of *addSparkListener* method. We can also register custom listeners using the *spark.extraListeners* setting.

6.8. Programmable Dynamic allocation

It also provides the following method as the developer API for dynamic allocation of executors: *requestExecutors, killExecutors, requestTotalExecutors, getExecutorIds*.

6.9. To access persistent RDD

getPersistentRDDs gives the collection of **RDDs** that have marked themselves as persistent via cache.

6.10. To unpersist RDDs

From the master's Block Manager and the internal *persistentRdds* mapping, the unpersist removes the RDD.

7. Conclusion

Hence, SparkContext provides the various functions in Spark like get the current status of Spark Application, set the configuration, cancel a job, Cancel a stage and much more. It is an entry point to the Spark functionality. Thus, it acts a backbone.

If you have any query about this tutorial, So feel free to Share with us. We will be glad to solve them.

9.SPARK RDD – INTRODUCTION, FEATURES & OPERATIONS OF RDD

1. Spark RDD

RDD (Resilient Distributed Dataset) is the fundamental data structure of **Apache Spark** which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in **Spark RDD** is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

In this blog, we are going to get to know about what is RDD in Apache Spark. What are the features of RDD, What is the motivation behind RDDs, RDD vs DSM? We will also cover Spark RDD operation i.e. transformations and actions, various limitations of RDD in Spark and how RDD make Spark feature rich in this Spark tutorial.

To play with RDD first install Apache Spark in pseudo distributed mode or in a multi-node cluster.

2. What is Apache Spark RDD?

RDD stands for "**Resilient Distributed Dataset"**. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(**DAG**) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It posses self-recovery in the case of failure.

There are three **ways to create RDDs in Spark** such as – *Data in stable storage, other RDDs, and parallelizing already existing collection in driver program*. One can also operate Spark RDDs in parallel with a low-level API that offers *transformations* and *actions*. We will study these Spark RDD Operations later in this section.

Spark RDD can also be **cached** and **manually partitioned**. Caching is beneficial when we use RDD several times. And manual partitioning is important to correctly balance partitions. Generally, smaller partitions allow distributing RDD data more equally, among more executors. Hence, fewer partitions make the work easy.

Programmers can also call a **persist** method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs **in memory** by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist.

Refer this link to **Learn how to persist and cache RDD in Spark**.

3. Why do we need RDD in Spark?

The key motivations behind the concept of RDD are-

- Iterative algorithms.
- Interactive data mining tools.
- **DSM** (Distributed Shared Memory) is a very general abstraction, but this generality makes it harder to implement in an efficient and fault tolerant manner on commodity clusters. Here the need of RDD comes into the picture.
- In distributed computing system data is stored in intermediate stable distributed store such as **HDFS** or Amazon S3. This makes the computation of job slower since it involves many IO operations, replications, and serializations in the process.

In first two cases we keep data in-memory, it can improve performance by an order of magnitude.

The main challenge in designing RDD is defining a program interface that provides fault tolerance efficiently. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on **coarse-grained transformation** rather than **fine-grained** updates to shared state.

Spark exposes RDD through language integrated API. In integrated API each data set is represented as an object and transformation is involved using the method of these objects.

Apache Spark evaluates RDDs lazily. It is called when needed, which saves lots of time and improves efficiency. The first time they are used in an action so that it can pipeline the transformation. Also, the programmer can call a persist method to state which RDD they want to use in future operations.

4. RDD vs DSM (Distributed Shared Memory)

In this Spark RDD tutorial, we are going to get to know the difference between RDD and DSM which will take RDD in Apache Spark into the limelight.

i. Read

- **RDD –** The read operation in RDD is either coarse grained or fine grained. Coarse-grained meaning we can transform the whole dataset but not an individual element on the dataset. While fine-grained means we can transform individual element on the dataset.
- **DSM –** The read operation in Distributed shared memory is fine-grained.

ii. Write

- **RDD –** The write operation in RDD is coarse grained.
- **DSM –** The Write operation is fine grained in distributed shared system.

iii. Consistency

- **RDD –** The consistency of RDD is trivial meaning it is immutable in nature. Any changes on RDD is permanent i.e we can not realtor the content of RDD. So the level of consistency is high.
- **DSM –** In Distributed Shared Memory the system guarantees that if the programmer follows the rules, the memory will be consistent and the results of memory operations will be predictable.

iv. Fault-Recovery Mechanism

- **RDD –** The lost data can be easily recovered in Spark RDD using lineage graph at any moment. Since for each transformation, new RDD is formed and RDDs are immutable in nature so it is easy to recover.
- **DSM –** Fault tolerance is achieved by a checkpointing technique which allows applications to roll back to a recent checkpoint rather than restarting.

v. Straggler Mitigation

Stragglers, in general, are those that take more time to complete than their peers. This could happen due to many reasons such as load imbalance, I/O blocks, garbage collections, etc.

The problem with stragglers is that when the parallel computation is followed by synchronizations such as reductions. This would cause all the parallel tasks to wait for others.

- **RDD –** In RDD it is possible to mitigate stragglers using backup task.
- **DSM –** It is quite difficult to achieve straggler mitigation.

vi. Behavior if not enough RAM

- **RDD –** If there is not enough space to store RDD in RAM then the RDDs are shifted to disk**.**
- **DSM –** In this type of system, the performance decreases if the RAM runs out of storage.

5. Features of RDD in Spark

Several features of Apache Spark RDD are:

Features of Spark RDD

5.1. In-memory Computation

Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

5.2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program. Follow this guide for the deep study of **Spark Lazy Evaluation**.

5.3. Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself. Follow this guide for the deep study of **RDD Fault Tolerance**.

5.4. Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5.5. Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

5.6. Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

5.7. Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

5.8. Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAGScheduler** places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation. Follow this guide to learn What is DAG?

6. Spark RDD Operations

RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

6.1. Transformations

Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.

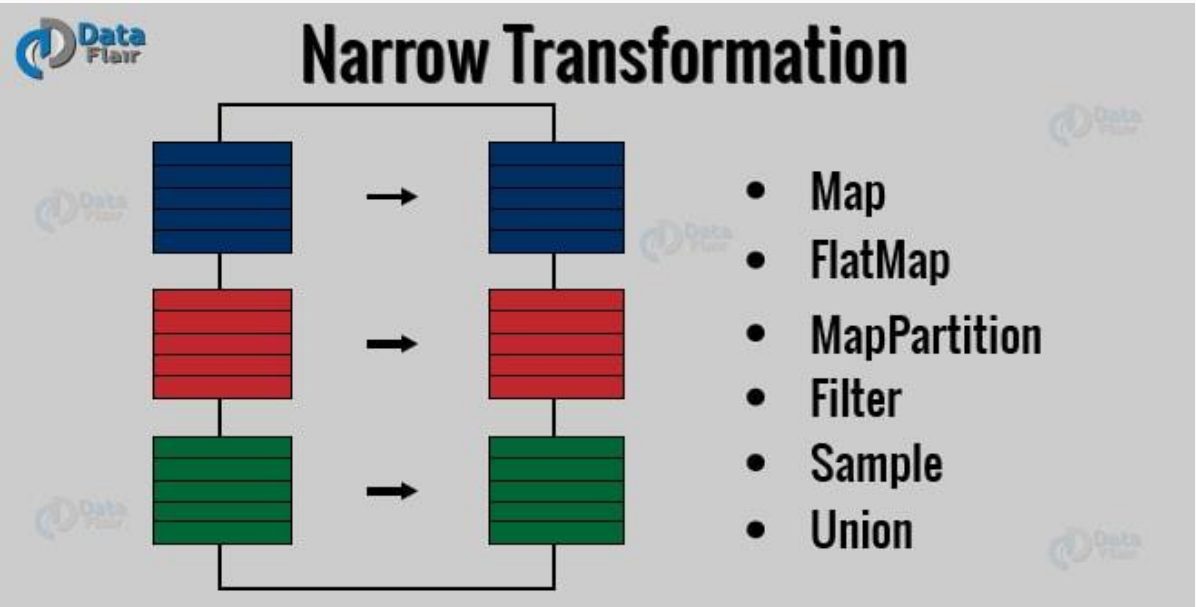
Transformations are **lazy** operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations. There are two kinds of transformations: narrow transformation, wide transformation.

6.1.1. Narrow Transformations

It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

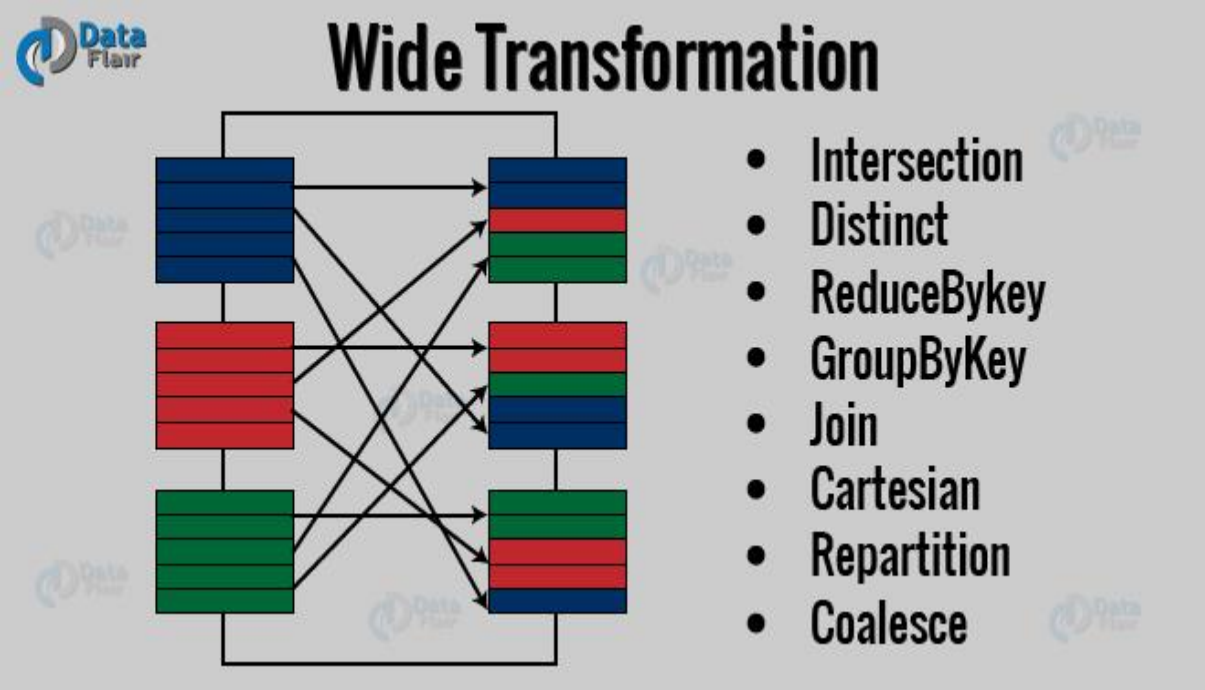
Spark groups narrow transformations as a stage known as **pipelining**.



Spark RDD – Narrow Transformation

6.1.2. Wide Transformations

It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.



Wide Transformation

6.2. Actions

An **Action** in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

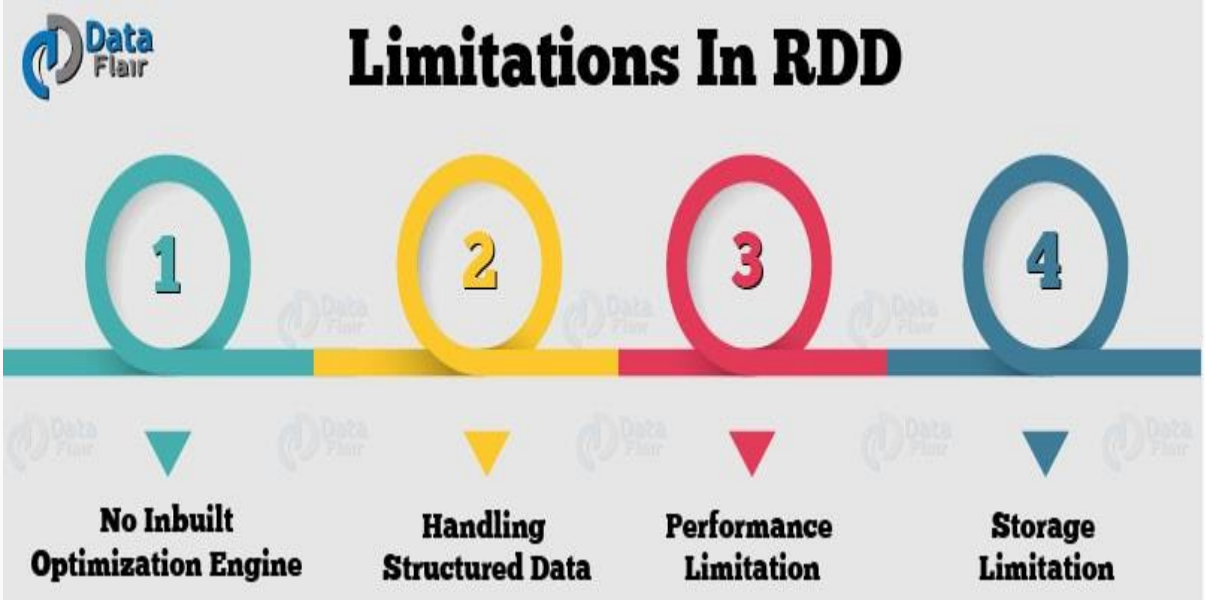
Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.

Using transformations, one can create RDD from the existing one. But when we want to work with the actual dataset, at that point we use Action. When the Action occurs it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.

Follow this link to **learn RDD Transformations and Actions APIs with examples.**

7. Limitation of Spark RDD

There is also some limitation of Apache Spark RDD. Let’s discuss them one by one-



Limitations of Apache Spark RDD

7.1. No inbuilt optimization engine

When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including **catalyst optimizer** and **Tungsten execution engine**. Developers need to optimize each RDD based on its attributes.

7.2. Handling structured data

Unlike **Dataframe** and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

7.3. Performance limitation

Being in-memory JVM objects, RDDs involve the overhead of Garbage Collection and Java Serialization which are expensive when data grows.

7.4. Storage limitation

RDDs degrade when there is not enough memory to store them. One can also store that partition of RDD on disk which does not fit in RAM. As a result, it will provide similar performance to current data-parallel systems.

8. Conclusion

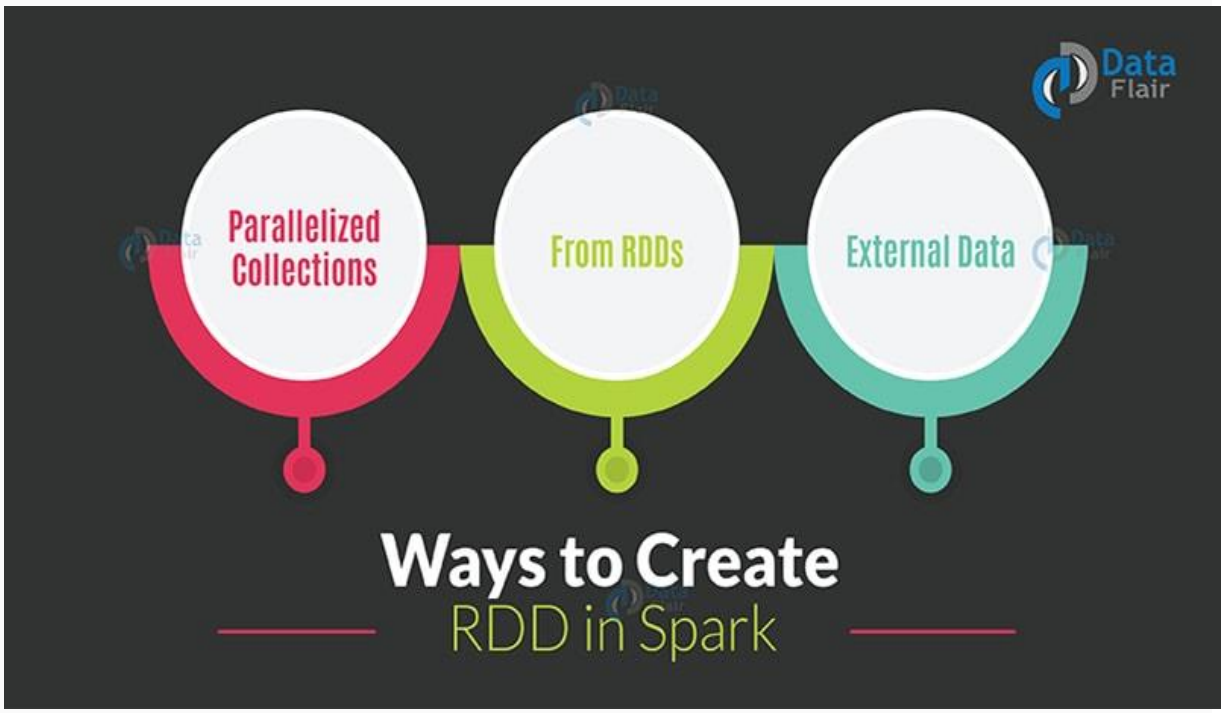
In conclusion to RDD, the shortcomings of Hadoop MapReduce was so high. Hence, it was overcome by Spark RDD by introducing in-memory processing, immutability etc. But there were some limitations of RDD. For example No inbuilt optimization, storage and performance limitation etc.

Because of the above-stated limitations of RDD to make spark more versatile DataFrame and Dataset evolved.

10.HOW TO CREATE RDDS IN APACHE SPARK

1. Objective

In this **Spark tutorial**, we are going to understand different ways of how to create RDDs in Apache Spark. We will understand Spark RDDs and 3 ways of creating RDDs in Spark – Using parallelized collection, from existing Apache Spark RDDs and from external datasets. We will also understand Spark RDDs creation with examples to get in-depth knowledge of how to create RDDs in Spark.



2. How to Create RDDs in Apache Spark?

Resilient Distributed Datasets (RDD) is the fundamental data structure of Spark. RDDs are immutable and fault tolerant in nature. These are distributed collections of objects. The datasets are divided into a logical partition, which is further computed on different nodes over the cluster. Thus, RDD is just the way of representing dataset distributed across multiple machines, which can be operated around in parallel. RDDs are called resilient because they have the ability to always re-compute an RDD. Let us revise Spark RDDs in depth here.

Now as we have already seen what is RDD in Spark, let us see how to create Spark RDDs.

There are three ways to create an RDD in Spark.

- Parallelizing already existing collection in driver program.
- Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
- Creating RDD from already existing RDDs.

Learn: RDD Persistence and Caching Mechanism in Apache Spark

Let us learn these in details below:

2.1. Parallelized collection (parallelizing)

In the initial stage when we learn Spark, **RDDs** are generally created by parallelized collection i.e. by taking an existing collection in the program and passing it to**SparkContext's** parallelize() method. This method is used in the initial stage of learning Spark since it quickly creates our own RDDs in Spark shell and performs operations on them. This method is rarely used outside testing and prototyping because this method requires entire dataset on one machine.

Consider the following example of sortByKey(). In this, the data to be sorted is taken through parallelized collection:

```
1. val data=spark.sparkContext.parallelize(Seq(("maths",52),("english",75),("science",82),
   ("computer",65),("maths",85)))
2. val sorted = data.sortByKey()
3. sorted.foreach(println)
```

The key point to note in parallelized collection is the number of partition the dataset is cut into. Spark will run one task for each partition of cluster. We require two to four partitions for each CPU in **cluster**. Spark sets number of partition based on our cluster. But we can also manually set the number of partitions. This is achieved by passing number of partition as second parameter to parallelize .

e.g. sc.parallelize(data, 10), here we have manually given number of partition as 10.

Consider one more example, here we have used parallelized collection and manually given the number of partitions:

```
1. val rdd1 = spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)
2. val result = rdd1.coalesce(2)
3. result.foreach(println)
```

2.2. External Datasets (Referencing a dataset)

In Spark, distributed dataset can be formed from any data source supported by Hadoop, including the local file system, **HDFS**, Cassandra, **HBase** etc. In this, the data is loaded from the external dataset. To create text file RDD, we can use SparkContext's textFile method. It takes URL of the file and read it as a collection of line. URL can be a local path on the machine or a hdfs://, s3n://, etc.

The point to jot down is that the path of the local file system and worker node should be same. The file should be present at same destinations both in local file system and worker node. We can copy the file to the worker nodes or use a network mounted shared file system.

Learn: Spark Shell Commands to Interact with Spark-Scala

DataFrameReader Interface is used to load a Dataset from external storage systems (e.g. file systems, key-value stores, etc). Use SparkSession.read to access an instance of DataFrameReader.DataFrameReader supports many file formats-

i) csv (String path)

It loads a CSV file and returns the result as a Dataset<Row>.

Example:

```
1. import org.apache.spark.sql.SparkSession
2. def main(args: Array[String]):Unit = {
3. object DataFormat {
4. val spark = SparkSession.builder.appName("AvgAnsTime").master("local").getOrCreate()
5. val dataRDD = spark.read.csv("path/of/csv/file").rdd
```

Note – Here .rdd method is used to convert Dataset<Row> to RDD<Row>.

ii) json (String path)

It loads a JSON file (one object per line) and returns the result as a Dataset<Row>

```
val dataRDD = spark.read.json("path/of/json/file").rdd
```

iii) textFile (String path)

It loads text files and returns a Dataset of String.

```
val dataRDD = spark.read.textFile("path/of/text/file").rdd
```

Learn: RDD lineage in Spark: ToDebugString Method

2.3. Creating RDD from existing RDD

Transformation mutates one RDD into another RDD, thus transformation is the way to create an RDD from already existing RDD. This creates difference between **Apache Spark and Hadoop MapReduce**. Transformation acts as a function that intakes an RDD and produces one. The input RDD does not get changed, because RDDs are immutable in nature but it produces one or more RDD by applying operations. Some of the operations applied on RDD are: filter, count, distinct, **Map, FlatMap** etc.

Example:

```
1. val words=spark.sparkContext.parallelize(Seq("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
   "dog"))
2. val wordPair = words.map(w => (w.charAt(0), w))
3. wordPair.foreach(println)
```

Note – In above code RDD "wordPair" is created from existing RDD "word" using map() transformation which contains word and its starting character together.

Read: Limitations of Spark RDD

4. Conclusion

Now as we have seen how to create RDDs in Apache Spark, let us learn **RDD transformations and Actions in Apache Spark** with the help of examples. If you like this blog or you have any query to create RDDs in Apache Spark, so let us know by leaving a comment in the comment box.

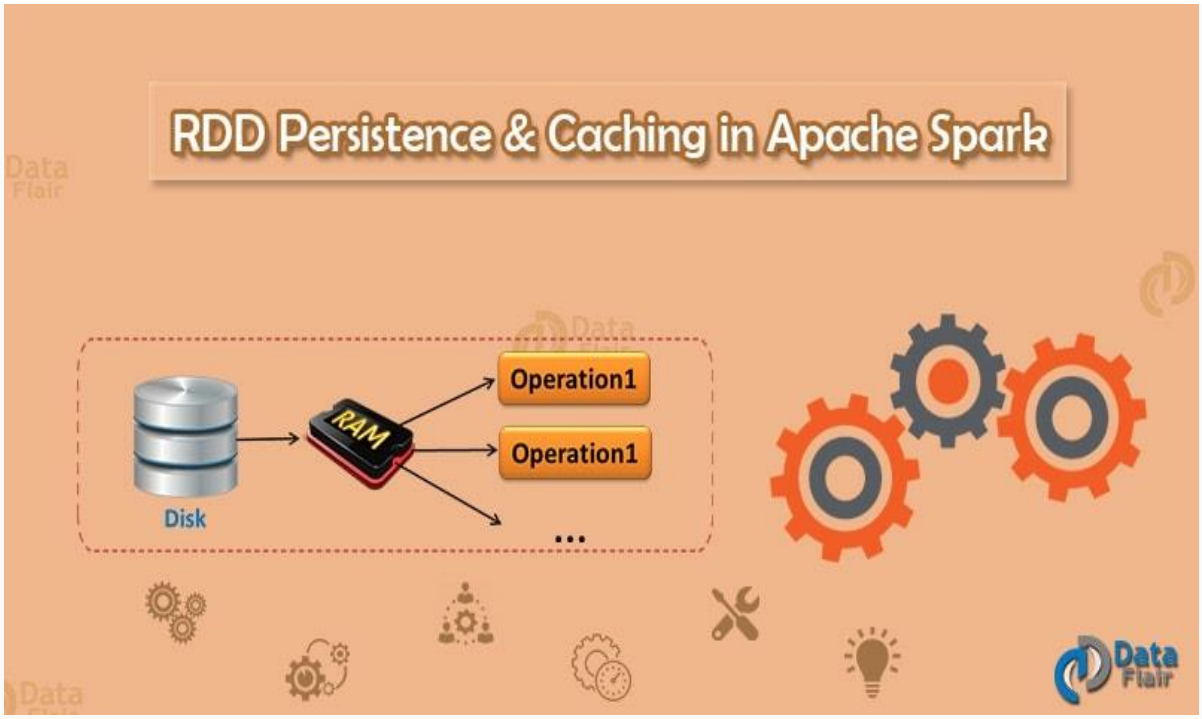
Reference:

http://spark.apache.org/

11.RDD PERSISTENCE AND CACHING MECHANISM IN APACHE SPARK

1. Objective

This blog covers the detailed view of **Apache Spark** RDD Persistence and Caching. This tutorial gives the answers for – What is RDD persistence, Why do we need to call ***cache*** or ***persist*** on an RDD, What is the Difference between Cache() and Persist() method in Spark, What are the different storage levels in spark to store the persisted RDD, How to Unpersist RDD? The need of persisting RDD and various advantages of persistence are also discussed in this Spark tutorial.



2. What is RDD Persistence and Caching in Spark?

Spark **RDD** persistence is an optimization technique in which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.

We can make persisted RDD through **cache()** and **persist()** methods. When we use the cache() method we can store all the RDD in-memory. We can persist the RDD in memory and use it efficiently across parallel operations.

The difference between cache() and persist() is that using cache() the default storage level is **MEMORY_ONLY** while using persist() we can use various storage levels (described below). It is a key tool for an interactive algorithm. Because, when we persist RDD each node stores any partition of it that it computes in memory and makes it reusable for future use. This process speeds up the further computation ten times.

When the RDD is computed for the first time, it is kept in memory on the node. The cache memory of the Spark is fault tolerant so whenever any partition of RDD is lost, it can be recovered by transformation Operation that originally created it.

3. Need of Persistence in Apache Spark

In Spark, we can use some RDD's multiple times. If honestly, we repeat the same process of **RDD evaluation** each time it required or brought into

action. This task can be time and memory consuming, especially for iterative algorithms that look at data multiple times. To solve the problem of repeated computation the technique of persistence came into the picture.

4. Benefits of RDD Persistence in Spark

There are some advantages of RDD caching and persistence mechanism in spark. It makes the whole system

- Time efficient
- Cost efficient
- Lessen the execution time.

5. Storage levels of Persisted RDDs

Using **persist()** we can use various storage levels to Store Persisted RDDs in Apache Spark. Let's discuss each RDD storage level one by one-

a. MEMORY_ONLY

In this storage level, RDD is stored as deserialized Java object in the JVM. If the size of RDD is greater than memory, It will not cache some partition and recompute them next time whenever needed. In this level the space used for storage is very high, the CPU computation time is low, the data is stored in-memory. It does not make use of the disk.

b. MEMORY_AND_DISK

In this level, RDD is stored as deserialized Java object in the JVM. When the size of RDD is greater than the size of memory, it stores the excess partition on the disk, and retrieve from disk whenever required. In this level the space used for storage is high, the CPU computation time is medium, it makes use of both in-memory and on disk storage.

c. MEMORY_ONLY_SER

This level of Spark store the RDD as serialized Java object (one-byte array per partition). It is more space efficient as compared to deserialized objects, especially when it uses fast serializer. But it increases the overhead on CPU. In this level the storage space is low, the CPU computation time is high and the data is stored in-memory. It does not make use of the disk.

d. MEMORY_AND_DISK_SER

It is similar to **MEMORY_ONLY_SER**, but it drops the partition that does not fits into memory to disk, rather than recomputing each time it is needed. In this storage level, The space used for storage is low, the CPU computation time is high, it makes use of both in-memory and on disk storage.

e. DISK_ONLY

In this storage level, RDD is stored only on disk. The space used for storage is low, the CPU computation time is high and it makes use of on disk storage.

Refer this guide for the detailed description of Spark in-memory computation.

6. How to Unpersist RDD in Spark?

Spark monitor the cache of each node automatically and drop out the old data partition in the LRU (least recently used) fashion. LRU is an algorithm which ensures the least frequently used data. It spills out that data from the cache. We can also remove the cache manually using **RDD.unpersist()** method.

7. Conclusion

Hence, **Caching** or **persistence** are the optimization techniques for interactive and iterative Spark computations. It helps to save intermediate results so we can reuse them in subsequent stages. These intermediate results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

What Next –

- Features of Apache Spark.
- Limitations of Apache Spark.
- How Apache Spark works.

12.11 SHINING FEATURES OF SPARK RDD YOU MUST KNOW

1. Objective

In this Spark tutorial, we will come across various twinkling **features of Spark RDD**. Before moving forward to this blog make yourself familiar with the concepts of **Bigdata** and Apache Spark. This blog also contains

the introduction to Apache SparkRDD and its operations along with the methods to create RDD.

To play with RDD learn Apache Spark Installation in standalone mode and Multi-cluster node.

2. Apache Spark RDD

RDD stands **Resilient Distributed Dataset**. RDDs are the fundamental abstraction of Apache Spark. It is an immutable distributed collection of the dataset. Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute These partitions. RDDs are a read-only partitioned collection of record. we can create RDD in three ways:

- **Parallelizing** already existing collection in driver program.
- **Referencing a dataset** in an external storage system (e.g. HDFS, Hbase, shared file system).
- Creating RDD **from already existing RDDs**.

There are two operations in RDD namely transformation and Action.

3. Sparkling Features of Spark RDD

There are several advantages of using RDD. Some of them are-

3.1. In-memory computation

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes. refer this comprehensive guide to Learn Spark in-memory computation in detail.

3.2. Lazy Evaluation

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do. Follow this guide to learn Spark lazy evaluation in great detail.

3.3. Fault Tolerance

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data. Learn Fault tolerance is Spark in detail.

3.4. Immutability

RDDS are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

3.5. Persistence

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function. Follow this guide for the detailed study of RDD persistence in Spark.

3.6. Partitioning

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

3.7. Parallel

Rdd, process the data parallelly over the cluster.

3.8. Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAGScheduler**places the partitions in such a way that task is close to data as much as possible. Thus speed up computation. Follow this guide to learn What is DAG?

3.9. Coarse-grained Operation

We apply coarse-grained transformations to RDD**.** Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the data set of RDD.

3.10. Typed

We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

3.11. No limitation

we can have any number of RDD. there is no limit to its number. the limit depends on the size of disk and memory.

4. Conclusion

Hence, using RDD we can recover the shortcoming of Hadoop MapReduce and can handle the large volume of data, as a result, it decreases the time complexity of the system. Thus the above-mentioned features of Spark RDD make them useful for fast computations and increase the performance of the system.

If you like this post or feel that I have missed some features of Spark RDD, please do leave a comment.

13.How to Overcome the Limitations of RDD in Apache Spark

1. Objective

This Tutorial on the limitations of RDD in Apache Spark, walk you through the Introduction to RDD in Spark, what is the need of DataFrame and

Dataset in Spark, when to use DataFrame and when to use DataSet in Apache Spark. To get the answer to these questions we will discuss various limitations of Apache Spark RDD and How we can use DataFrame and Dataset to overcome the Disadvantages of Spark RDD.

2. What is RDD in Apache Spark?

Before going to the disadvantages of RDD, let's have a brief introduction to Spark RDD.

RDD is the fundamental data structure of **Apache Spark**. RDD is Read only partition collection of records. It can only be created through deterministic operation on either: Data in stable storage, other RDDs, and parallelizing already existing collection in driver program(Follow this guide to learn the ways to create RDD in Spark). RDD is an immutable distributed collection of data, partitioned across nodes in the cluster that can be operated in parallel with a low-level API that offers transformations and actions.

3. What are the Limitations of RDD in Apache Spark?

In this Section of Spark tutorial, we will discuss the problems related to RDDs in Apache Spark along with their solution.

3.1. No input optimization engine

There is no provision in RDD for automatic optimization. It cannot make use of Spark advance optimizers like **catalyst optimizer** and **Tungsten execution engine**. We can optimize each RDD manually.

This limitation is overcome in **Dataset** and **DataFrame**, both make use of Catalyst to generate optimized logical and physical query plan. We can use same code optimizer for R, Java, Scala, or Python DataFrame/Dataset APIs. It provides space and speed efficiency.

3.2. Runtime type safety

There is no **Static typing** and **run-time type safety** in RDD. It does not allow us to check error at the runtime.

Dataset provides **compile-time type safety** to build complex data workflows. Compile-time type safety means if you try to add any other type of element to this list, it will give you compile time error. It helps detect errors at compile time and makes your code safe.

3.3. Degrade when not enough memory

The RDD degrades when there is not enough memory to store RDD **in-memory** or on disk. There comes storage issue when there is a lack of memory to store RDD. The partitions that overflow from RAM can be stored on disk and will provide the same level of performance. By increasing the size of RAM and disk it is possible to overcome this issue.

3.4. Performance limitation & Overhead of serialization & garbage collection

Since the RDD are in-memory JVM object, it involves the overhead of **Garbage Collection** and **Java serialization** this is expensive when the data grows.

Since the cost of garbage collection is proportional to the number of Java objects. Using data structures with fewer objects will lower the cost. Or we can persist the object in serialized form.

3.5. Handling structured data

RDD does not provide schema view of data. It has no provision for handling structured data.

Dataset and DataFrame provide the Schema view of data. It is a distributed collection of data organized into named columns.

4. Conclusion

As a result of RDD's limitations, the need of DataFrame and Dataset emerged. Thus made the system more friendly to play with a large volume of data.

If you are willing to work with Spark 1.6.0 then the DataFrame API is the most stable option available and offers the best performance. However, the Dataset API is very promising and provides a more natural way to code.

If you like this post and think that I have missed some limitations of RDD in Apache Spark, So, please leave a comment in the comment box.

See Also-

- Features of Apache Spark.
- Comparison between Apache Spark RDD vs DataFrame vs DataSet.

14.SPARK RDD OPERATIONS-TRANSFORMATION & ACTION WITH EXAMPLE

1. Spark RDD Operations

Two types of **Apache Spark** RDD operations are- Transformations and Actions. A **Transformation** is a function that produces new **RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation.

In this **Apache Spark** RDD operations tutorial we will get the detailed view of what is Spark RDD, what is the transformation in Spark RDD, various RDD transformation operations in Spark with examples, what is action in Spark RDD and various RDD action operations in Spark with examples.

2. Apache Spark RDD Operations

Before we start with Spark RDD Operations, let us deep dive into **RDD in Spark.**

Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

Now let us understand first what is Spark RDD Transformation and Action-

3. RDD Transformation

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

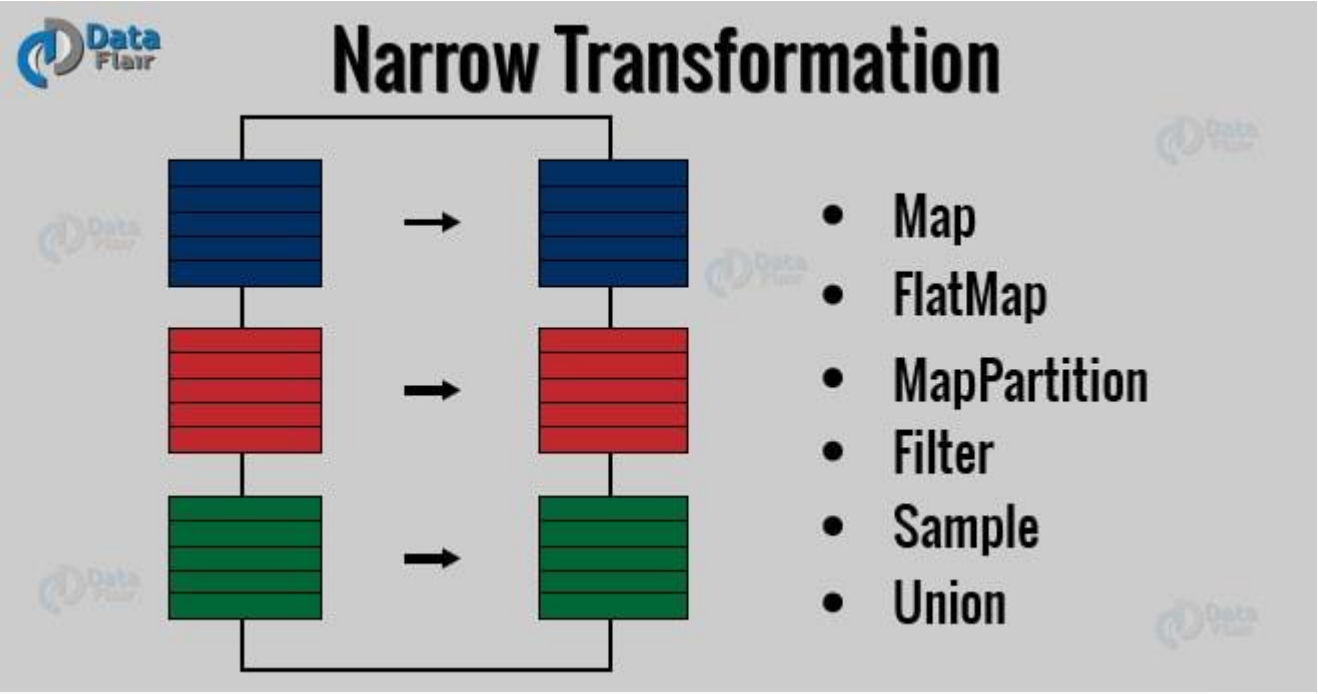
Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is Directed Acyclic Graph (**DAG**) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().

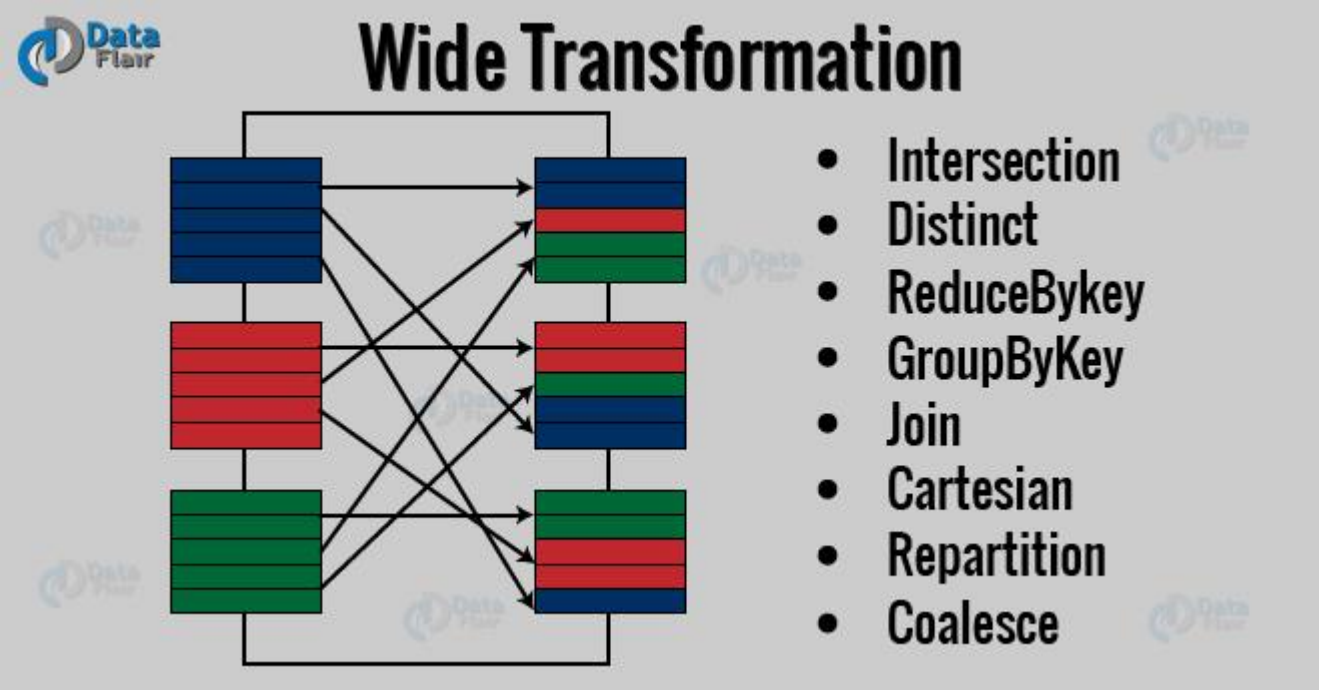
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).

There are two types of transformations:

- **Narrow transformation –** In *Narrow transformation*, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. *Narrow transformations* are the result of *map(), filter().*



- **Wide transformation –** In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*.



There are various functions in RDD transformation. Let us see RDD transformation with examples.

3.1. map(func)

The map function iterates over every line in RDD and split into new RDD. Using **map()** transformation we take in any function, and that function is applied to every element of RDD.

In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For example, in RDD {1, 2, 3, 4, 5} if we apply "rdd.map(x=>x+2)" we will get the result as (3, 4, 5, 6, 7).

Also Read: **How to create RDD**

Map() example:

```
1. import org.apache.spark.SparkContext
2. import org.apache.spark.SparkConf
3. import org.apache.spark.sql.SparkSession
4. object mapTest{
5. def main(args: Array[String]) = {
6. val spark = SparkSession.builder.appName("mapExample").master("local").getOrCreate()
7. val data = spark.read.textFile("spark_test.txt").rdd
8. val mapFile = data.map(line => (line,line.length))
9. mapFile.foreach(println)
10. }
11. }
```

spark_test.txt"

```
hello...user! this file is created to check the operations of spark.

?, and how can we apply functions on that RDD partitions?. All this will be done t
hrough spark programming which is done with the help of scala language support…
```

- ***Note –*** In above code, map() function map each line of the file with its length.

3.2. flatMap()

With the help of **flatMap()** function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.

Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key **difference between map() and flatMap()** is map() returns only one element, while flatMap() can return a list of elements.

flatMap() example:

```
1. val data = spark.read.textFile("spark_test.txt").rdd
2. val flatmapFile = data.flatMap(lines => lines.split(" "))
3. flatmapFile.foreach(println)
```

- ***Note –*** In above code, flatMap() function splits each line when space occurs.

3.3. filter(func)

Spark RDD **filter()** function returns a new RDD, containing only the elements that meet a predicate. It is a *narrow operation* because it does not shuffle data from one partition to many partitions.

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

Filter() example:

```
1. val data = spark.read.textFile("spark_test.txt").rdd
2. val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
3. println(mapFile.count())
```

- ***Note*** – In above code, flatMap function map line into words and then count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

Read: Apache Spark RDD vs DataFrame vs DataSet

3.4. mapPartitions(func)

The **MapPartition** converts each *partition* of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

3.5. mapPartitionWithIndex()

It is like mapPartition; Besides mapPartition it provides *func* with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.

Learn: Spark Shell Commands to Interact with Spark-Scala

3.6. union(dataset)

With the **union()** function, we get the elements of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

For example, the elements of **RDD1** are (Spark, Spark, **Hadoop**, **Flink**) and that of**RDD2** are (**Big data**, Spark, Flink) so the resultant ***rdd1.union(rdd2)*** will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

Union() example:

```
1. val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014)))
2. val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
3. val rdd3 = spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
4. val rddUnion = rdd1.union(rdd2).union(rdd3)
5. rddUnion.foreach(Println)
```

- ***Note –*** In above code union() operation will return a new dataset that contains the union of the elements in the source dataset (rdd1) and the argument (rdd2 & rdd3).

3.7. intersection(other-dataset)

With the **intersection()** function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant ***rdd1.intersection(rdd2)***will have elements (spark).

Intersection() example:

```
val rdd1 = spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014, (16,"feb",2014)))
val rdd2 = spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
val comman = rdd1.intersection(rdd2)
comman.foreach(Println)
```

- ***Note –*** The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

Learn to Install Spark on Ubuntu

3.8. distinct()

It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.

For example, if RDD has elements (Spark, Spark, Hadoop, Flink), then ***rdd.distinct()***will give elements (Spark, Hadoop, Flink).

Distinct() example:

```
val rdd1 = park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
val result = rdd1.distinct()
println(result.collect().mkString(", "))
```

- ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'"nov",2014).

3.9. groupByKey()

When we use **groupByKey()** on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.

Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory. Follow this link to **learn about RDD Caching and Persistence mechanism** in detail.

groupByKey() example:

```
val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)
```

- ***Note –*** The groupByKey() will group the integers on the basis of same key(alphabet). After that *collect()* action will return all the elements of the dataset as an Array.

3.10. reduceByKey(func, [numTasks])

When we use **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.

reduceByKey() example:

```
val words = Array("one","two","two","four","five","six","six","eight","nine","ten")
val data = spark.sparkContext.parallelize(words).map(w => (w,1)).reduceByKey(_+_)
data.foreach(println)
```

- ***Note –*** The above code will parallelize the Array of String. It will then map each word with count 1, then reduceByKey will merge the count of values having the similar key.

Read: Various Features of RDD

3.11. sortByKey()

When we apply the **sortByKey() function** on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.

sortByKey() example:

```
val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)
```

- ***Note –*** In above code, sortByKey() transformation sort the data RDD into Ascending order of the Key(String).

Read: Limitations of RDD

3.12. join()

The **Join** is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.

The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

Join() example:

```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
```

```
4. println(result.collect().mkString(","))
```

- ***Note*** – The join() transformation will join two different RDDs on the basis of Key.

Read: RDD lineage in Spark: ToDebugString Method

3.13. coalesce()

To avoid full shuffling of data we use coalesce() function. In **coalesce()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.

Coalesce() example:

```
1. val rdd1 = spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)
2. val result = rdd1.coalesce(2)
3. result.foreach(println)
```

- ***Note –*** The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

4. RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from *Executer* to the *driver.* Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

4.1. count()

Action **count()** returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.count()" will give the result 8.

Count() example:

```
1. val data = spark.read.textFile("spark_test.txt").rdd
```

```
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())
```

- ***Note –*** In above code *flatMap()* function maps line into words and count the word "Spark" using *count()* Action after filtering lines containing "Spark" from mapFile.

Learn: Spark Streaming

4.2. collect()

The action **collect()** is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.

Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

Collect() example:

```
val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2 =spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))
```

- ***Note –*** *join()* transformation in above code will join two RDDs on the basis of same key(alphabet). After that *collect()* action will return all the elements to the dataset as an Array.

4.3. take(n)

The action **take(n)** returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "take (4)" will give result { 2, 2, 3, 4}

Take() example:

```
val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
val twoRec = result.take(2)
twoRec.foreach(println)
```

- ***Note*** – The *take(2)* Action will return an array with the first *n* elements of the data set defined in the taking argument.

Learn: Apache Spark DStream (Discretized Streams)

4.4. top()

If ordering is present in our RDD, then we can extract top elements from our RDD using **top()**. Action *top()* use default ordering of data.

Top() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
res.foreach(println)
```

- ***Note*** – *map()* operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

4.5. countByValue()

The **countByValue()** returns, many times each element occur in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.countByValue()" will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

countByValue() example:

```
val data = spark.read.textFile("spark_test.txt").rdd
val result= data.map(line => (line,line.length)).countByValue()
result.foreach(println)
```

- ***Note –*** The *countByValue()* action will return a hashmap of (K, Int) pairs with the count of each key.

Learn: Apache Spark Streaming Transformation Operations

4.6. reduce()

The **reduce()** function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

Reduce() example:

```
val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)
```

- ***Note*** – The *reduce()* action in above code will add the elements of the source RDD.

4.7. fold()

The signature of the **fold()** is like *reduce().* Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the **condition with zero value** is that it should be the **identity element of that operation**. The key difference between *fold()* and *reduce()* is that, *reduce()* throws an exception for empty collection, but *fold()* is defined for empty collection.

For example, zero is an identity for addition; one is identity element for multiplication. The return type of *fold()* is same as that of the element of RDD we are operating on.

For example, rdd.fold(0)((x, y) => x + y).

Fold() example:

```
1. val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))
2. val additionalMarks = ("extra", 4)
3. val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2
4. ("total", add)
5. }
6. println(sum)
```

- ***Note –*** In above code *additionalMarks* is an initial value. This value will be added to the int value of each record in the source RDD.

Learn: Spark Streaming Checkpoint in Apache Spark

4.8. aggregate()

It gives us the flexibility to get data type different from the input type. The **aggregate()** takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

4.9. foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the *driver*. In this case, **foreach()** function is useful. For example, inserting a record into the database.

Foreach() example:

```
1. val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
```

```
val group = data.groupByKey().collect()
group.foreach(println)
```

- ***Note –*** *The foreach()* action run a function *(println)* on each element of the dataset group.

5. Conclusion

In conclusion, on applying a transformation to an RDD creates another RDD. As a result of this RDDs are immutable in nature. On the introduction of an action on an RDD, the result gets computed. Thus, this lazy evaluation decreases the overhead of computation and make the system more efficient.

If you have any query about Spark RDD Operations, So, feel free to share with us. We will be happy to solve them.

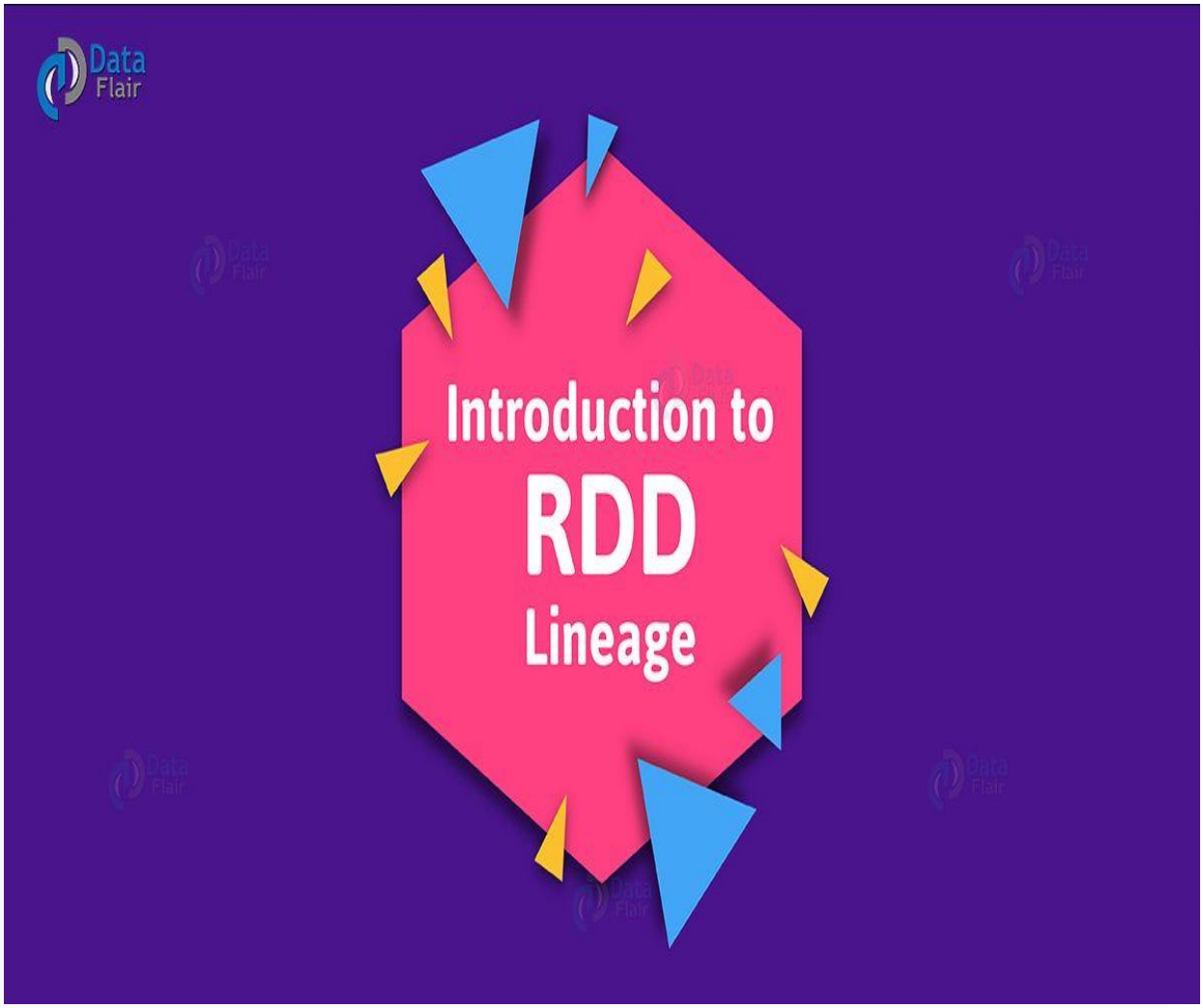
See Also-

- **Spark SQL Introduction**
- **Apache Spark SQL DataFrame**

15.RDD LINEAGE IN SPARK: TODEBUGSTRING METHOD

1. Objective

Basically, in **Spark** all the dependencies between the RDDs will be logged in a graph, despite the actual data. This is what we call as a lineage graph in Spark. This document holds the concept of RDD lineage in Spark logical execution plan. Moreover, we will get to know that how to get RDD Lineage Graph by toDebugString method in detail. Before all, let's also learn about Spark RDDs.



Introduction to Spark RDD Lineage

2. Introduction to Spark RDD

Spark RDD is nothing but an acronym for "Resilient Distributed Dataset". We can consider RDD as a fundamental data structure of Apache Spark. To be very specific, RDD is an immutable collection of objects in Apache Spark. That helps to compute on the different node of the cluster.

On decomposing the name of Spark RDD:

- **Resilient**

This means **fault-tolerant**. By using RDD lineage graph(DAG), we can recompute missing or damaged partitions due to node failures.

- **Distributed**

It means data resides on multiple nodes.

- **Dataset**

It is nothing but a record of the data you work with. Also, a user can load the dataset externally. For example, JSON file, CSV file, text file or database via JDBC with no specific data structure.

3. Introduction to RDD Lineage

Basically, evaluation of RDD is lazy in nature. It means a series of transformations are performed on an RDD, which is not even evaluated immediately.

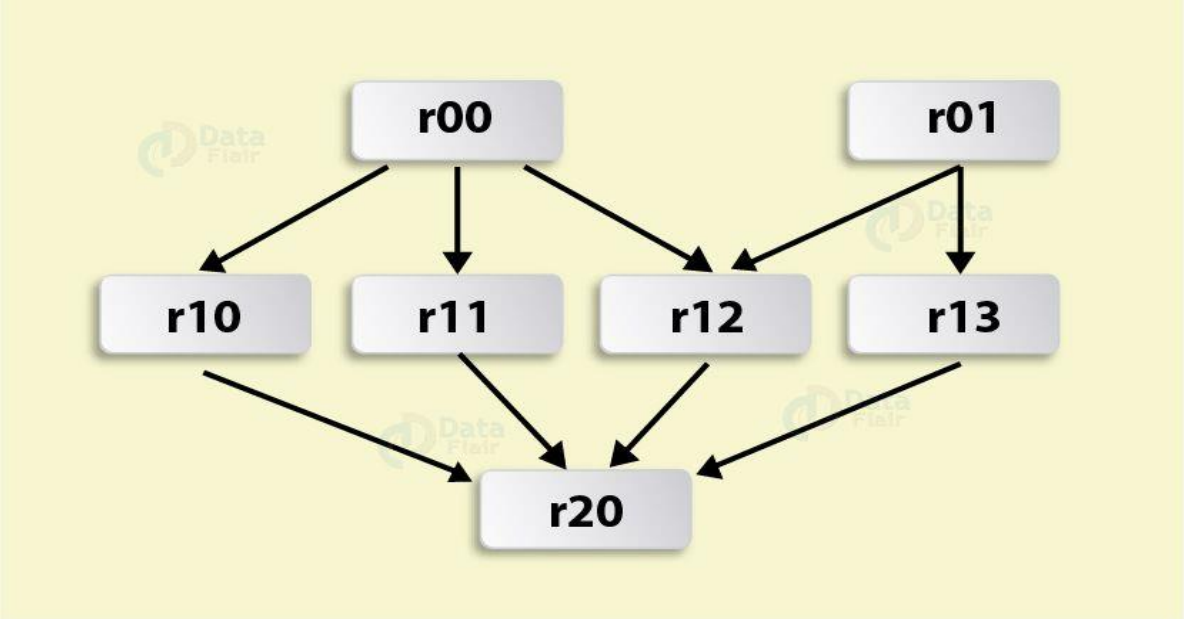
While **we create a new RDD** from an existing Spark RDD, that new RDD also carries a pointer to the parent RDD in Spark. That is same as all the dependencies between the RDDs those are logged in a graph, rather than the actual data. It is what we call as lineage graph.

RDD lineage is nothing but the graph of all the parent RDDs of an RDD. We also call it an RDD operator graph or RDD dependency graph. To be very specific, it is an output of applying transformations to the spark. Then, it creates a logical execution plan.

Also, physical execution plan or execution **DAG** is known as DAG of stages.

Let's start with one example of Spark RDD lineage by using Cartesian or zip to understand well. However, we can also use other operators to build an RDD graph in Spark.

For example



Introduction to RDD lineage in Apache Spark

Above figure depicts an RDD graph, which is the result of the following series of transformations:

val r00 = sc.parallelize(0 to 9)

val r01 = sc.parallelize(0 to 90 by 10)

val r10 = r00 cartesian df01

val r11 = r00.map(n => (n, n))

val r12 = r00 zip df01

val r13 = r01.keyBy(_ / 20)

val r20 = Seq(r11, r12, r13).foldLeft(r10)(_ union _)

After an action has been called, this is a graph of what transformations need to be executed.

In other words, whenever on the basis of the existing RDDs we create new RDDs, using lineage graph spark manage these dependencies. Basically, along with metadata about what type of relationship it has with the parent RDD, each RDD maintains a pointer to one or more parent.

For example,

if we say, on an

RDD val b=a.map().

Hence, RDD b keeps a reference to its parent RDD a. That is a sort of an RDD lineage.

4. Logical Execution Plan for RDD Lineage

Basically, logical execution plan gets initiated with earliest RDDs. Earliest RDDs are nothing but RDDs which are not dependent on other RDDs. To be very specific those are independent of reference cached data. Moreover, it ends with the RDD those produces the result of the action which has been called to execute.

We can also say, it is a DAG that is executed when **SparkContext** is requested to run a Spark job.

5. ToDebugString Method to get RDD Lineage Graph in Spark

Although there are several methods to get RDD lineage graph in spark, one of the methods is toDebugString method. Such as,

toDebugString: String

Basically, we can learn about an Spark RDD lineage graph with the help of this method.

```
scala> val wordCount1 = sc.textFile("README.md").flatMap(_.split("\\s+")).map((_, 1)).reduceByKey(_ + _)

wordCount1: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[21] at reduceByKey at <console>:24

scala> wordCount1.toDebugString

res13: String =

(2) ShuffledRDD[21] at reduceByKey at <console>:24 []

+-(2) MapPartitionsRDD[20] at map at <console>:24 []

|  MapPartitionsRDD[19] at flatMap at <console>:24 []

|  README.md MapPartitionsRDD[18] at textFile at <console>:24 []

|  README.md HadoopRDD[17] at textFile at <console>:24 []
```

Here for indication of shuffle boundary, this method " toDebugString method" uses indentations.

Basically, here H in round brackets refers, numbers that show the level of parallelism at each stage.

For example, (2) in the above output.

```
scala> wordCount1.getNumPartitions

res14: Int = 2
```

The toDebugString method is included when executing an action, With spark.logLineage property enabled.

```
$ ./bin/spark-shell –conf spark.logLineage=true

scala> sc.textFile("README.md", 4).count

…

15/10/17 14:46:42 INFO SparkContext: Starting job: count at <console>:25

15/10/17 14:46:42 INFO SparkContext: RDD's recursive dependencies:

(4) MapPartitionsRDD[1] at textFile at <console>:25 []

|  README.md HadoopRDD[0] at textFile at <console>:25 []
```

6. Conclusion

Hence, by this blog, we have learned the actual meaning of Apache Spark RDD lineage graph. Moreover, also we have tasted the flavor of the logical execution plan in Apache Spark. However, we have also seen toDebugString

method in detail. Therefore, we have covered all the concept of lineage graph in Apache **Spark RDD**.

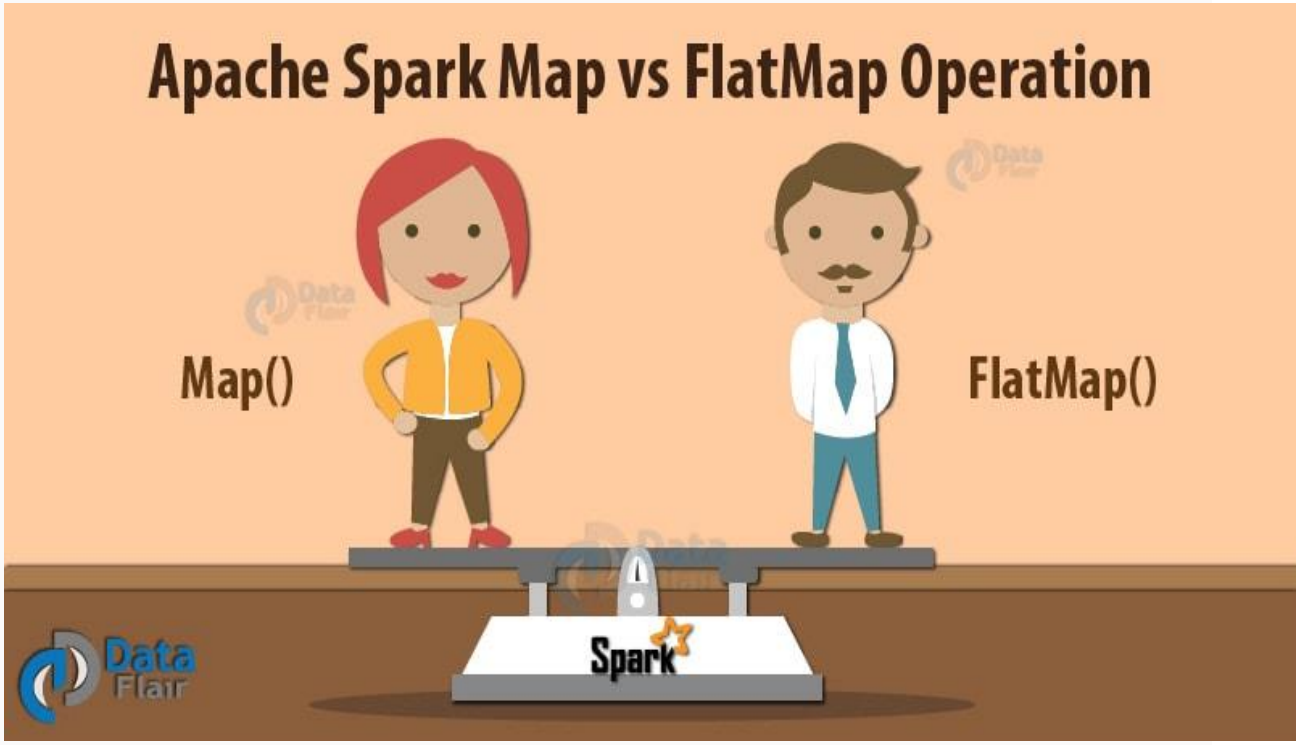
Furthermore, if you have any query, please ask in the comment section.

16.APACHE SPARK MAP VS FLATMAP OPERATION

1. Objective

In this **Apache Spark** tutorial, we will discuss the comparison between Spark Map vs FlatMap Operation. Map and FlatMap are the **transformation operations in Spark**. **Map()** operation applies to each element of **RDD** and it returns the result as new RDD. In the Map, operation developer can define his own custom business logic. While **FlatMap()** is similar to Map, but FlatMap allows returning 0, 1 or more elements from map function.

In this blog, we will discuss how to perform map operation on RDD and how to process data using FlatMap operation. This tutorial also covers what is map operation, what is a flatMap operation, the difference between map() and flatMap() transformation in Apache Spark with examples. We will also see Spark map and flatMap example in **Scala** and **Java** in this Spark tutorial.



Before starting to Map and FlatMap function, follow this guide to **install and configure Apache Spark** to practically implement these spark transformation operations on RDD.

2. Difference between Spark Map vs FlatMap Operation

This section of the Spark tutorial provides the details of Map vs FlatMap operation in Apache Spark with examples in **Scala** and **Java** programming languages.

2.1. Spark Map Transformation

A **map** is a transformation operation in **Apache Spark.** It applies to each element of**RDD** and it returns the result as new RDD. In the Map, operation developer can define his own custom business logic. The same logic will be applied to all the elements of RDD.

Spark Map function takes one element as input process it according to custom code (specified by the developer) and returns one element at a time. Map transforms an **RDD** of length N into another RDD of length N. The input and output RDDs will typically have the same number of records.Learn more on **Spark RDD Operations-Transformation & Action with Example**

2.1.1. Map Transformation Scala Example

a. Create RDD

```
val data = spark.read.textFile("INPUT-PATH").rdd
```

Above statement will create an RDD with name data. Follow this guide to learn more**ways to create RDDs** in Apache Spark**.**

b. Map Transformation-1

```
val newData = data.map (line => line.toUpperCase() )
```

Above the map, a transformation will convert each and every record of RDD to upper case.

c. Map Transformation-2

```
val tag = data.map {line => {
val xml = XML.loadString(line)
xml.attribute("Tags").get.toString()
}
}
```

Above the map, a transformation will parse XML and collect Tag attribute from the XML data. Overall the map operation is converting XML into a structured format. Follow this guide to **learn more Spark Transformation operations.**

2.1.2. Map Transformation Java Example

a. Create RDD

```
JavaRDD<String> linesRDD = spark.read().textFile("INPUT-PATH").javaRDD();
```

Above statement will create an RDD with name linesRDD.

b. Map Transformation

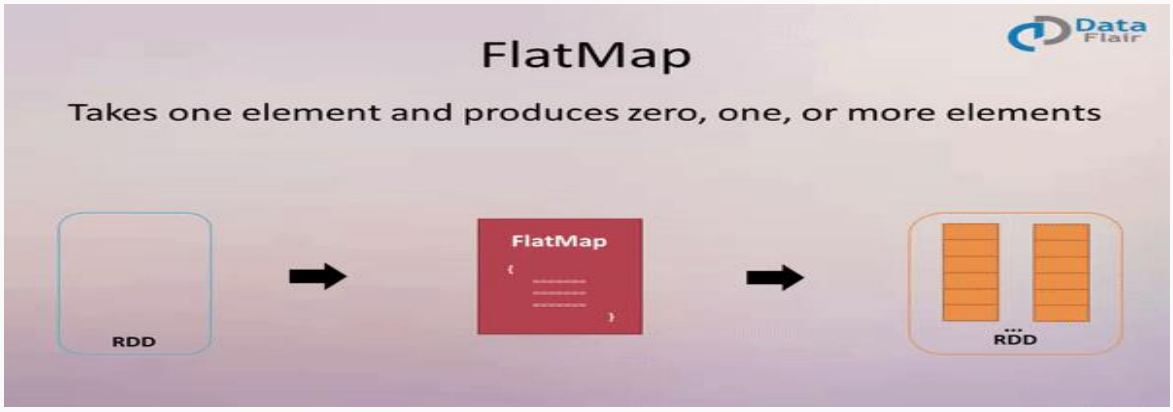
```
JavaRDD<String> newData = linesRDD.map(new Function<String, String>() {

public String call(String s) {
String result = s.trim().toUpperCase();
return result;
}
});
```

Learn More on Spark Shell Commands to Interact with Spark-Scala

2.2. Spark FlatMap Transformation Operation

Let's now discuss flatMap() operation in Apache Spark-



A **flatMap** is a transformation operation. It applies to each element of RDD and it returns the result as new RDD. It is similar to Map, but FlatMap allows returning 0, 1 or more elements from map function. In the FlatMap operation, a developer can define his own custom business logic. The same logic will be applied to all the elements of the RDD.

A FlatMap function takes one element as input process it according to custom code (specified by the developer) and returns 0 or more element at a time. flatMap() transforms an RDD of length N into another RDD of length M.

2.2.1. FlatMap Transformation Scala Example

```
val result = data.flatMap (line => line.split(" ") )
```

Above flatMap transformation will convert a line into words. One word will be an individual element of the newly created RDD.

Learn to Create Spark project in Scala with Eclipse

2.2.2. FlatMap Transformation Java Example

```
1. JavaRDD<String> result = data.flatMap(new FlatMapFunction<String, String>() {
2. public Iterator<String> call(String s) {
3. return Arrays.asList(s.split(" ")).iterator();
4. } });
```

Above flatMap transformation will convert a line into words. One word will be an individual element of the newly created RDD.

3. Conclusion

Hence, from the comparison between Spark map() vs flatMap(), it is clear that Spark map function expresses a one-to-one transformation. It transforms each element of a collection into one element of the resulting collection. While Spark flatMap function expresses a one-to-many transformation. It transforms each element to 0 or more elements.

Please leave a comment if you like this post or have any query about Apache Spark map vs flatMap function.

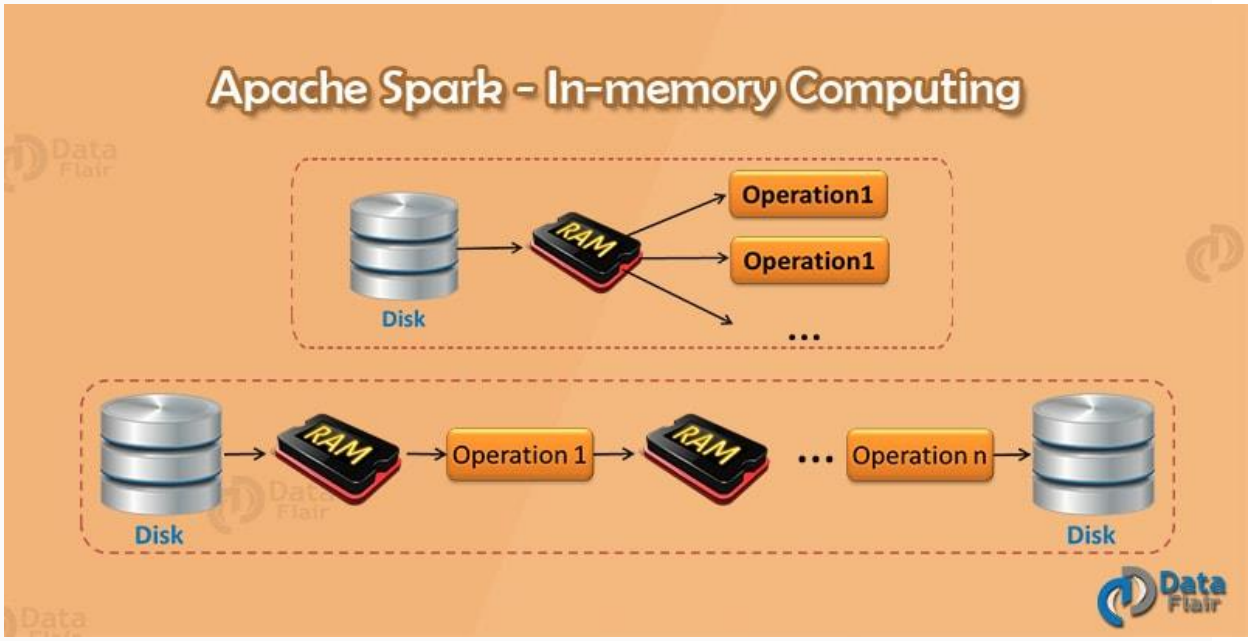
See Also-

- **Apache Spark RDD vs Dataframe vs Dataset**
- **How to Create RDDs in Apache Spark?**
- **Apache Spark compatibility with Hadoop**
- **How to Overcome the Limitations of RDD in Apache Spark?**
- **Spark Streaming Tutorial for Beginners**

17.SPARK IN-MEMORY COMPUTING – A BEGINNERS GUIDE

1. Objective

This tutorial on **Apache Spark** in-memory computing will provide you the detailed description of what is in memory computing? Introduction to Spark in-memory processing and how does Apache Spark process data that does not fit into the memory? This tutorial will also cover various storage levels in Spark and benefits of in-memory computation.



2. What is In-memory Computing?

In in-memory computation, the data is kept in *random access memory(**RAM**)* instead of some slow disk drives and is processed in parallel. Using this we can detect a pattern, analyze large data. This has become popular because it reduces the cost of memory. So, in-memory processing is economic for applications. The two main columns of in-memory computation are-

- RAM storage
- Parallel distributed processing.

3. Introduction to Spark In-memory Computing

Keeping the data in-memory improves the performance by an order of magnitudes. The main abstraction of Spark is its **RDDs.** And the RDDs are cached using the **cache()** or **persist()** method.Follow this link to learn Spark RDD persistence and caching mechanism.

When we use *cache()* method, all the RDD are stored in-memory. When RDD stores the value in memory, the data that does not fit in memory is either recalculated or the excess data is sent to disk. Whenever we want RDD, it can be extracted without going to disk. This reduces the space – time complexity and overhead of disk storage.

The in-memory capability of Spark is good for *machine learning* and *micro-batch processing*. It provides faster execution for iterative jobs.

When we use *persist()* method the RDDs can also be stored in-memory, we can use it across parallel operations. The difference between cache() and persist() is that using cache() the default storage level is **MEMORY_ONLY** while using persist() we can use various storage levels.

4. Storage levels of RDD Persist() in Spark

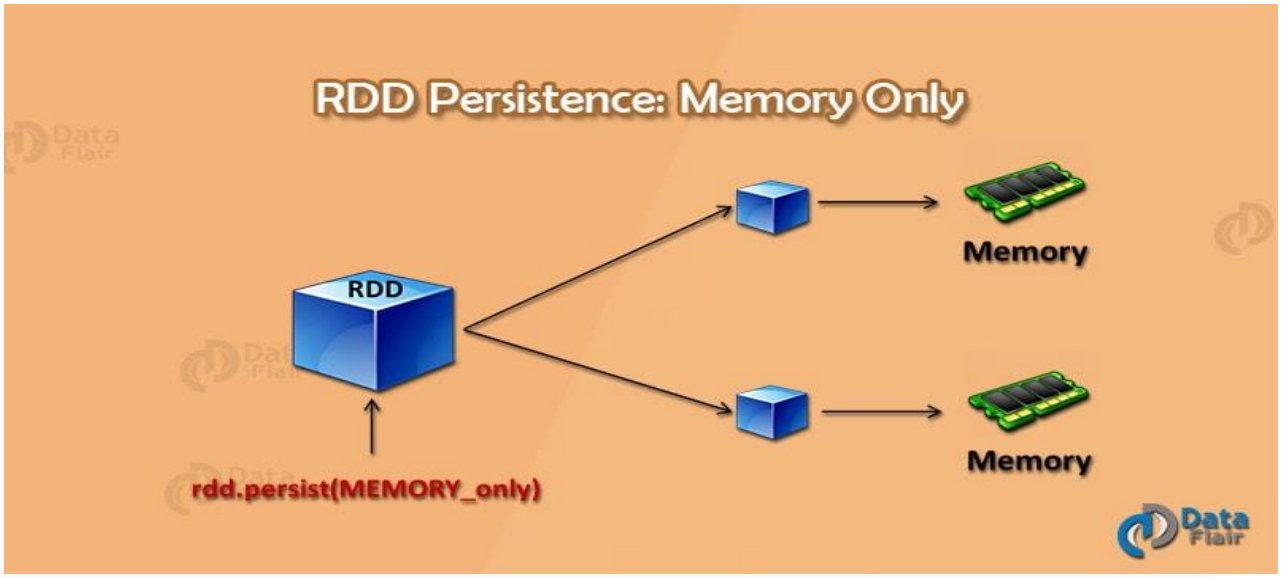
The various storage level of persist() method in Apache Spark RDD are:

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY
- MEMORY_ONLY_2 and MEMORY_AND_DISK_2

Let’s discuss the above mention Apache Spark storage levels one by one –

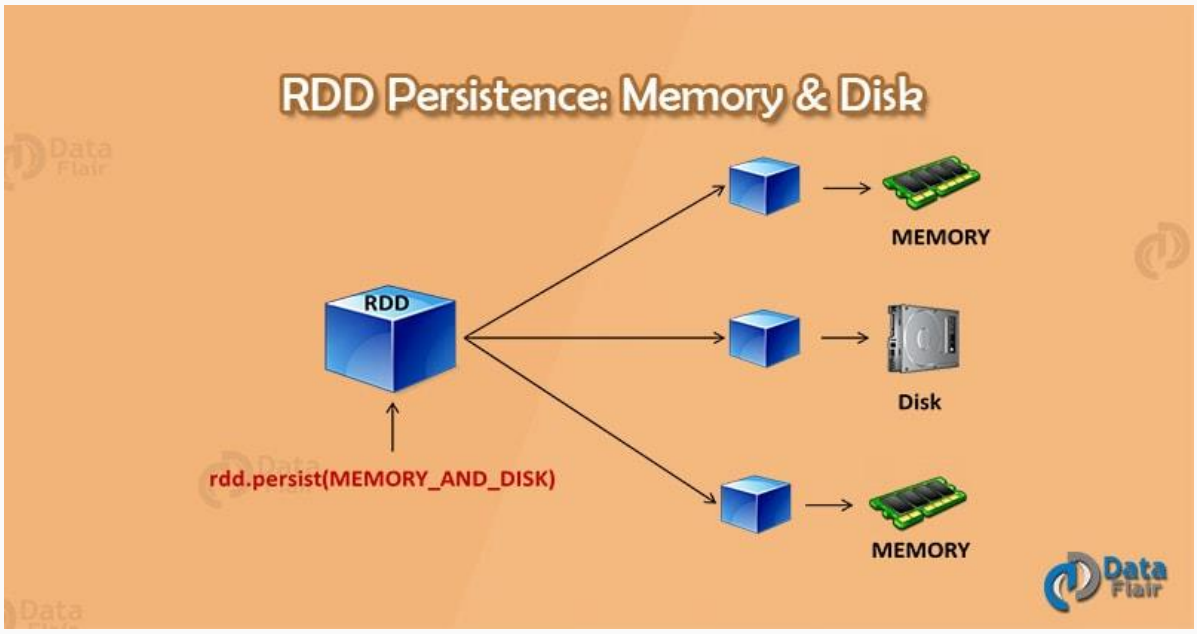
4.1. MEMORY_ONLY

In this storage level Spark, RDD is stored as deserialized JAVA object in JVM. If RDD does not fit in memory, then the remaining will be recomputed each time they are needed.



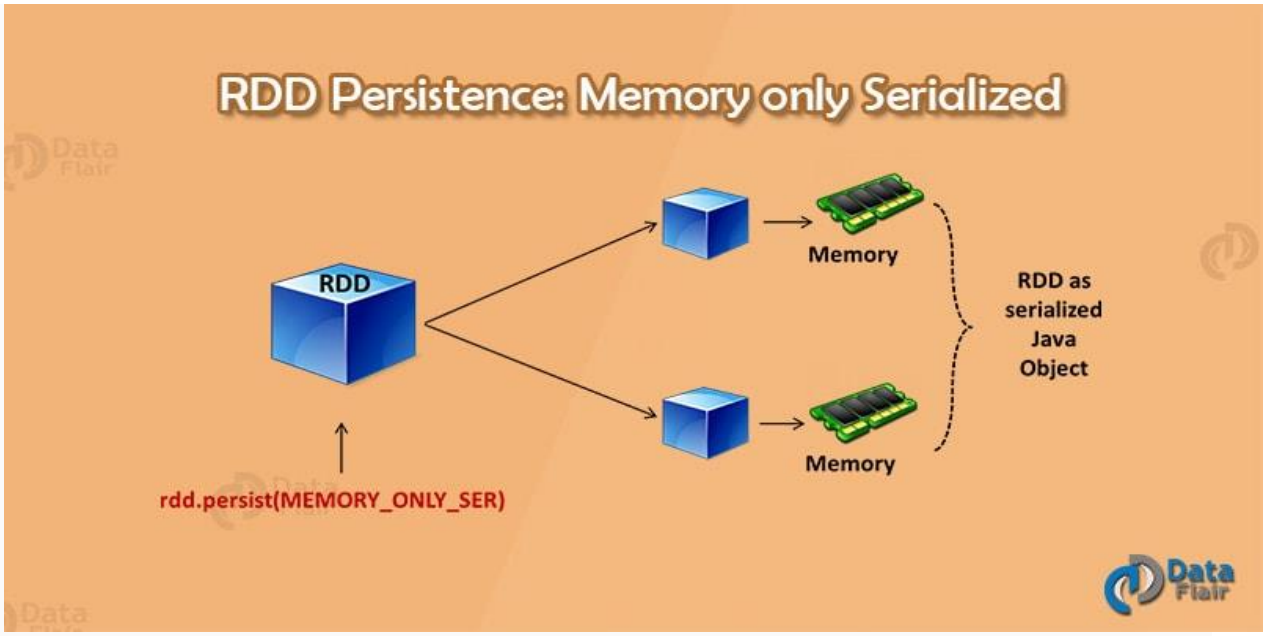
4.2. MEMORY_AND_DISK

In this level, RDD is stored as deserialized JAVA object in JVM. If the full RDD does not fit in memory then the remaining partition is stored on disk, instead of recomputing it every time when it is needed.



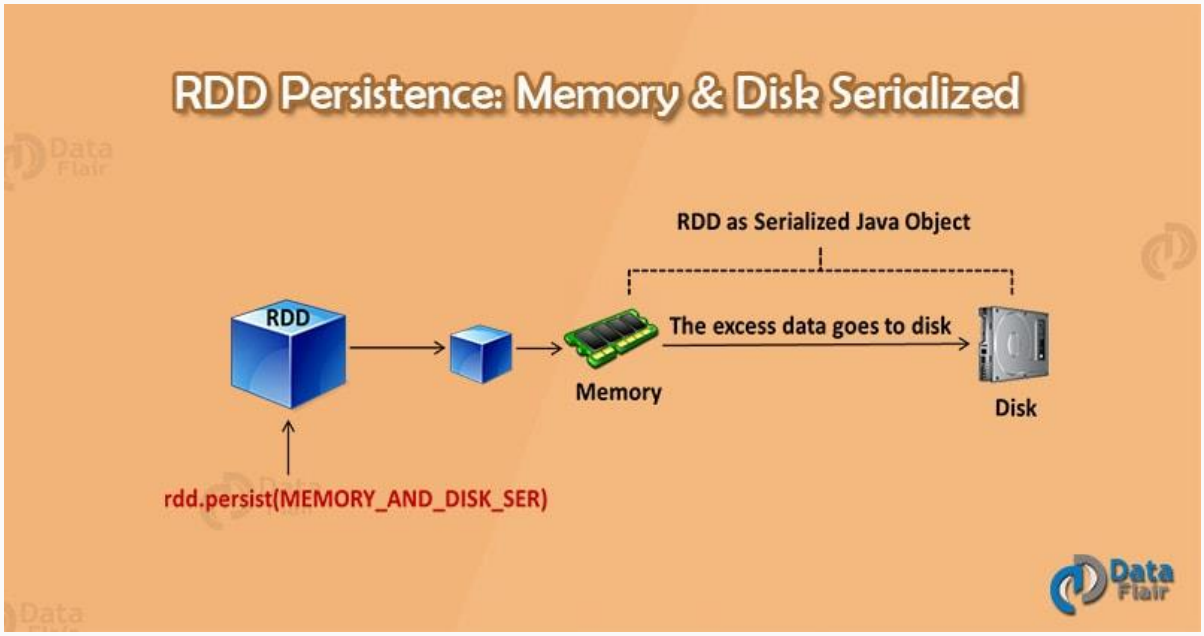
4.3. MEMORY_ONLY_SER

This level stores RDDs as serialized JAVA object. It stores one-byte array per partition. It is like *MEMORY_ONLY* but is more space efficient especially when we use fast serializer.



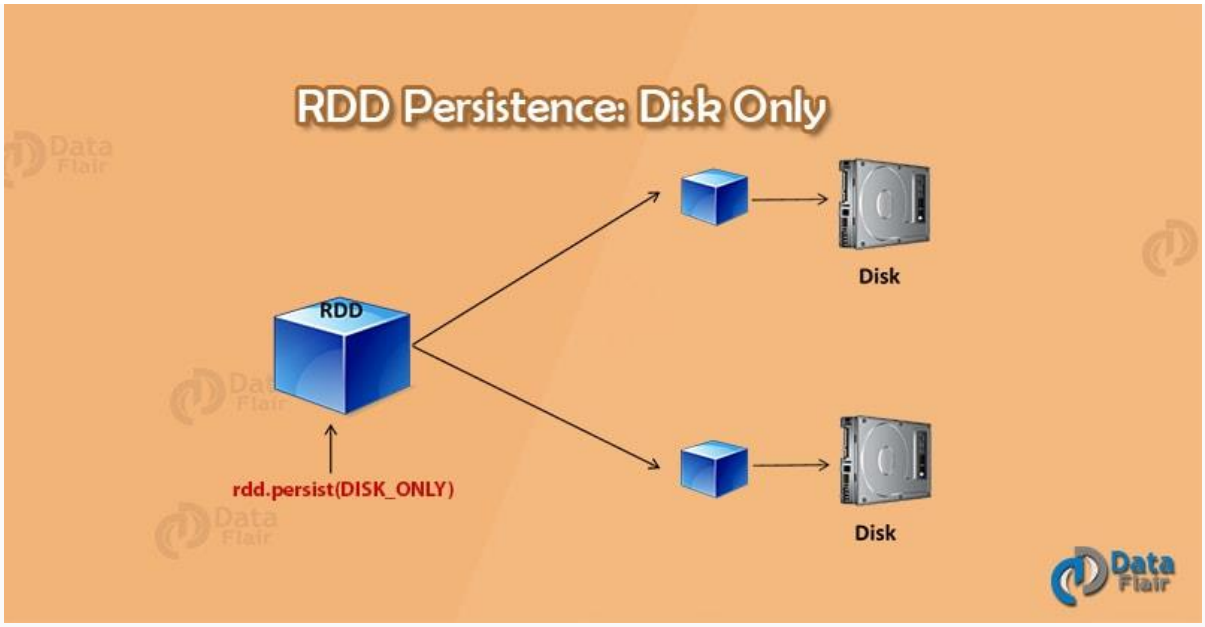
4.4. MEMORY_AND_DISK_SER

This level stores RDD as serialized JAVA object. If the full RDD does not fit in the memory then it stores the remaining partition on the disk, instead of recomputing it every time when we need.



4.5. DISK_ONLY

This storage level stores the RDD partitions only on disk.



4.6. MEMORY_ONLY_2 and MEMORY_AND_DISK_2

It is like *MEMORY_ONLY* and MEMORY_AND_DISK. The only difference is that each partition gets replicate on two nodes in the cluster.

Follow this link to learn more about Spark terminologies and concepts in detail.

5. Advantages of In-memory Processing

After studying Spark in-memory computing introduction and various storage levels in detail, let's discuss the advantages of in-memory computation-

1. When we need a data to analyze it is already available on the go or we can retrieve it easily.
2. It is good for real-time risk management and fraud detection.
3. The data becomes highly accessible.
4. The computation speed of the system increases.
5. Improves complex event processing.
6. Cached a large amount of data.
7. It is economic, as the cost of RAM has fallen over a period of time.

6. Conclusion

In conclusion, **Apache Hadoop** enables users to store and process huge amounts of data at very low costs. However, its relies on persistent storage

to provide fault tolerance and its one-pass computation model makes **MapReduce** a poor fit for low-latency applications and iterative computations, such as machine learning and graph algorithms.

Hence, Apache Spark solves these **Hadoop drawbacks** by generalizing the MapReduce model. It improves the performance and ease of use.

If you like this post or have any query related to Apache Spark In-Memory Computing, so, do let us know by leaving a comment.

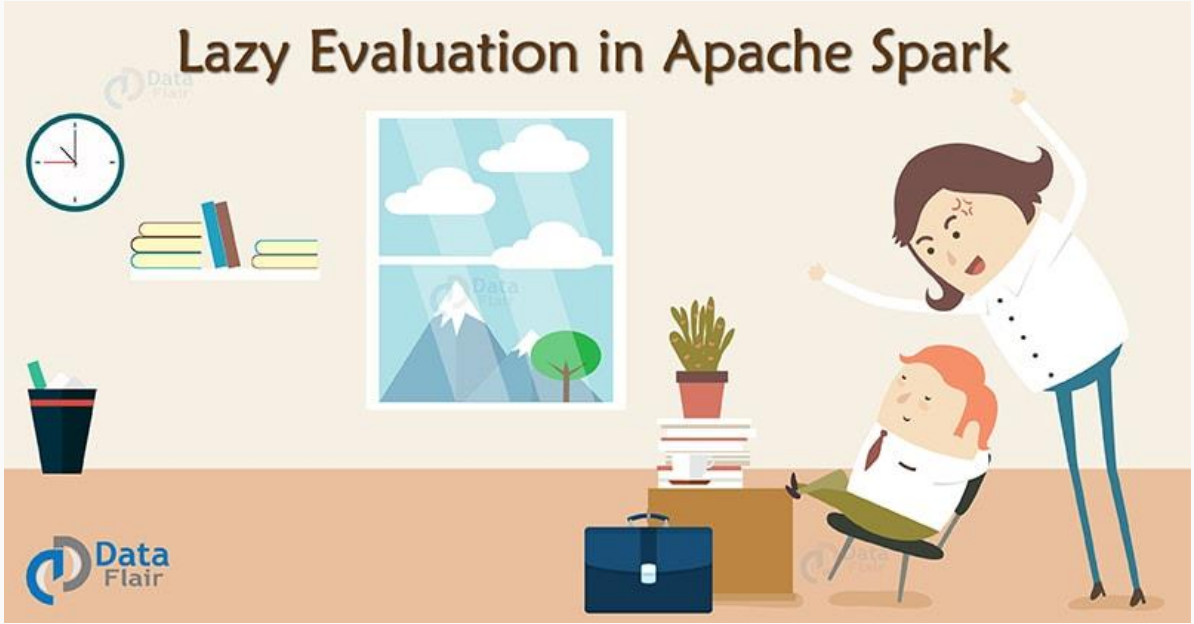
See Also-

- Features of Apache Spark.
- Limitations Of Apache Spark.

18.LAZY EVALUATION IN APACHE SPARK – A QUICK GUIDE

1. Objective

In this **Apache Spark lazy evaluation** tutorial, we will understand what is lazy evaluation in Apache Spark, How Spark manages the lazy evaluation of **Spark RDD**data transformation, the reason behind keeping Spark lazy evaluation and what are the advantages of lazy evaluation in Spark transformation.



2. What is Lazy Evaluation in Apache Spark?

Before starting with lazy evaluation in Spark, let us revise **Apache Spark concepts**.

As the name itself indicates its definition, **lazy evaluation** in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.

Transformations are lazy in nature meaning when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called(Through **DAG**). We can think **Spark RDD** as the data, that we built up through transformation. Since transformations are lazy in nature, so we can execute operation any time by calling an action on data. Hence, in lazy evaluation data is not loaded until it is necessary.

In **MapReduce**, much time of developer wastes in minimizing the number of MapReduce passes. It happens by clubbing the operations together. While in Spark we do not create the single execution graph, rather we club many simple operations. Thus it creates the difference between Hadoop MapReduce vs Apache Spark.

In Spark, *driver program* loads the code to the cluster. When the code executes after every operation, the task will be time and memory consuming. Since each time data goes to the cluster for evaluation.

3. Advantages of Lazy Evaluation in Spark Transformation

There are some benefits of Lazy evaluation in Apache Spark-

a. Increases Manageability

By lazy evaluation, users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.

b. Saves Computation and increases Speed

Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get compute. It saves the trip between driver and cluster, thus speeds up the process.

c. Reduces Complexities

The two main complexities of any operation are *time* and *space* complexity. Using Apache Spark lazy evaluation we can overcome both. Since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.

d. Optimization

It provides optimization by reducing the number of queries. Learn more about Apache Spark Optimization.

4. Conclusion

Hence, Lazy evaluation enhances the power of Apache Spark by reducing the execution time of the **RDD operations**. It maintains the lineage graph to remember the operations on RDD. As a result, it Optimizes the performance and achieves **fault tolerance**.

If you like this blog or have any query so please leave a comment.

See Also-

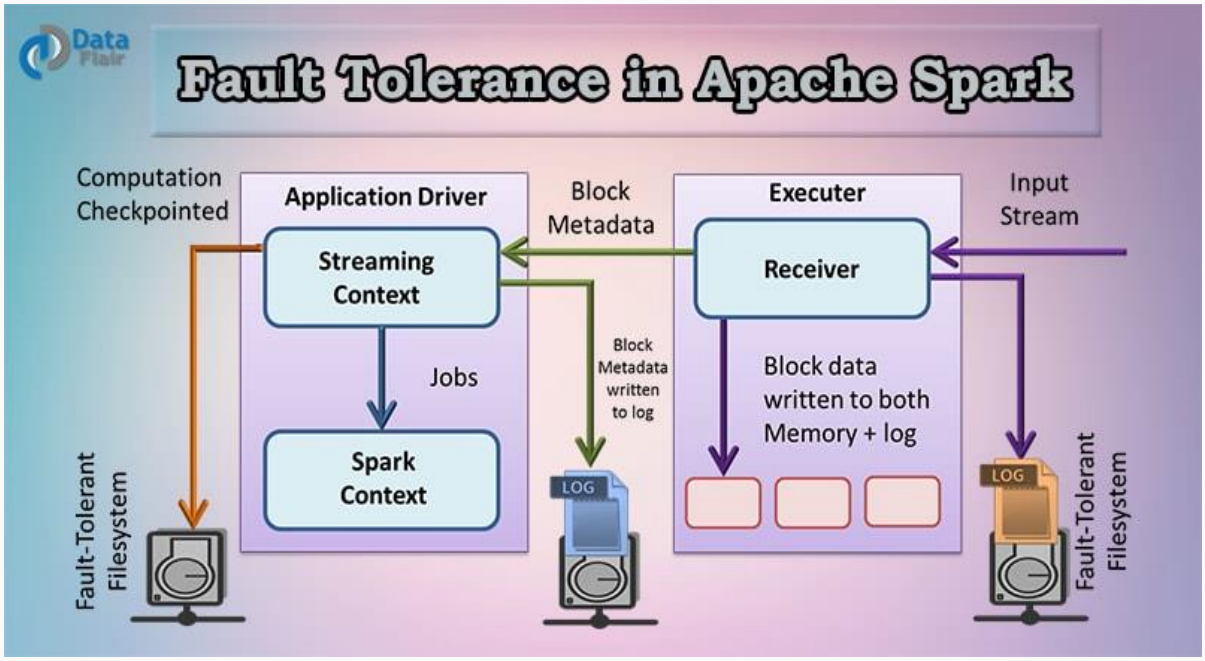
- Features of Apache Spark
- Limitations of Apache Spark

20.FAULT TOLERANCE IN APACHE SPARK – RELIABLE SPARK STREAMING

1. Objective

In this Spark fault tolerance tutorial, we will learn what do you mean by fault tolerance and how Apache Spark handles fault tolerance. We will see fault-

tolerant stream processing with Spark Streaming and **Spark RDD** fault tolerance. We will also learn what is Spark Streaming write ahead log, Spark streaming driver failure, Spark streaming worker failure to understand how to achieve fault tolerance in Apache Spark.



2. Introduction to Fault Tolerance in Apache Spark

Before we start with learning what is fault tolerance in Apache Spark, let us revise concepts of Apache Spark for beginners.

Now let's understand what is fault and how Spark handles fault tolerance.

Fault refers to failure, thus fault tolerance in Apache Spark is the capability to operate and to recover loss after a failure occurs. If we want our system to be fault tolerant, it should be redundant because we require a redundant component to obtain the lost data. The faulty data is recovered by redundant data.

3. Spark RDD Fault Tolerance

Let us firstly see how to create RDDs in Spark? Spark operates on data in fault-tolerant file systems like **HDFS** or **S3**. So all the RDDs generated from fault tolerant data is fault tolerant. But this does not set true for streaming/live data (data over the network). So the key need of fault tolerance in Spark is for this kind of data. The basic fault-tolerant semantic of Spark are:

- Since Apache Spark RDD is an immutable dataset, each Spark RDD remembers the lineage of the deterministic operation that was used on fault-tolerant input dataset to create it.
- If due to a worker node failure any partition of an RDD is lost, then that partition can be re-computed from the original fault-tolerant dataset using the lineage of operations.

- Assuming that all of the **RDD transformations** are deterministic, the data in the final transformed RDD will always be the same irrespective of failures in the Spark cluster.

To achieve fault tolerance for all the generated RDDs, the achieved data is replicated among multiple Spark executors in worker nodes in the cluster. This results in two types of data that needs to be recovered in the event of failure:

- **Data received and replicated –** In this, the data gets replicated on one of the other nodes thus the data can be retrieved when a failure.
- **Data received but buffered for replication –** The data is not replicated thus the only way to recover fault is by retrieving it again from the source.

Failure also occurs in worker as well as driver nodes.

- **Failure of worker node –** The node which runs the application code on the **Spark cluster** is Spark worker node. These are the slave nodes. Any of the worker nodes running executor can fail, thus resulting in loss of **in-memory** If any receivers were running on failed nodes, then their buffer data will be lost.
- **Failure of driver node –** If there is a failure of the driver node that is running the Spark Streaming application, then SparkContent is lost and all executors with their in-memory data are lost.

Apache Mesos helps in making the Spark master fault tolerant by maintaining the backup masters. It is open source software residing between the application layer and the operating system. It makes easier to deploy and manage applications in large-scale clustered environment. Executors are relaunched if they fail. Post failure, executors are relaunched automatically and spark streaming does parallel recovery by recomputing Spark RDD's on input data. Receivers are restarted by the workers when they fail.

4. Fault Tolerance with Receiver-based sources

For input sources based on receivers, the fault tolerance depends on both- **the failure scenario** and the **type of receiver**. There are two types of receiver:

- **Reliable receiver –** Once it is ensured that the received data has been replicated, the reliable sources are acknowledged. If the receiver fails, the source will not receive acknowledgment for the buffered data. So, the next time the receiver is restarted, the source will resend the data. Hence, no data will be lost due to failure.
- **Unreliable Receiver –** Due to the worker or driver failure, the data can be lost since receiver does not send an acknowledgment.
- **If the worker node fails**, and the receiver is reliable there will be no data loss. But in the case of unreliable receiver data loss will occur. With the unreliable receiver, data received but not replicated can be lost.

5. Spark Streaming write ahead logs

If the driver node fails, all the data that was received and replicated in memory will be lost. This will affect the result of the stateful transformation. To avoid the loss of data, Spark 1.2 introduced **write ahead logs,** which save received data to fault-tolerant storage. All the data received is written to write ahead logs before it can be processed to Spark Streaming.

Write ahead logs are used in database and file system. It ensure the durability of any data operations. It works in the way that first the intention of the operation is written down in the durable log. After this, the operation is applied to the data. This is done because if the system fails in the middle of applying the operation, the lost data can be recovered. It is done by reading the log and reapplying the data it has intended to do.

Deployment Scenario	**Worker Failure**	**Driver Failure**
Spark 1.1 or earlier	Buffered data lost with unreliable receivers	Buffered data lost with the unreliable receiver.
*Spark 1.2 or later **without** write ahead logs*	Zero data loss with reliable receivers, At-least-once semantics	Past data lost with all receivers, Undefined semantics

*Spark 1.2 or later **with** write ahead logs*	Zero data loss with reliable receivers, At-least-once semantics	Zero data loss with reliable receivers and files, At-least-once semantics

6. High Availability

Any system is said to be highly available if its downtime is tolerable. This time depends on how critical the application is. Zero down time is an imaginary term for any system. Consider any machine has an uptime of 97.7%, so its probability to go down will be 0.023. If we have similar two machines, then the probability of both of them going down will be (0.023*0.023). in most high availability environment we have three machines in use, in that case, the probability of going down is (0.023*0.023*0.023) i.e. 0.000012167, which guarantees an uptime of system to be 99.9987833% which is highly acceptable uptime guarantee. 6. Apache Spark High Availability

7. Conclusion

Hence we have studied Fault Tolerance in Apache Spark. I hope this blog helps you a lot to understand How Apache Spark is Fault Tolerant Framework. if Still you have doubt related to Fault Tolerance in Apache Spark, so leave a comment in a section given below

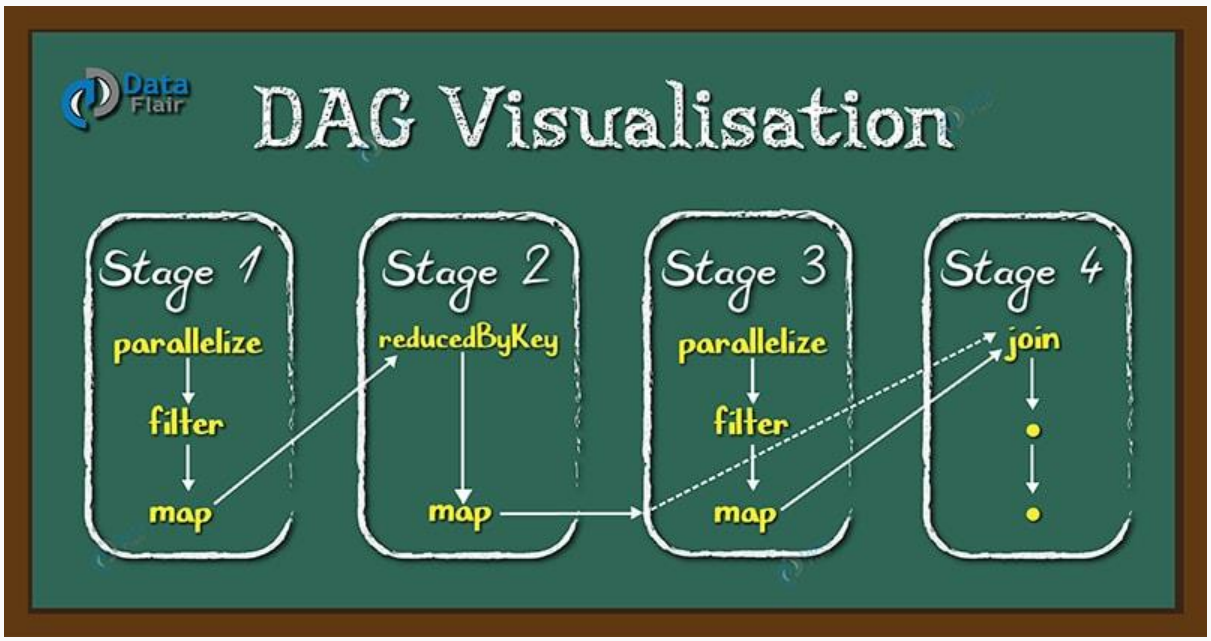
21.DIRECTED ACYCLIC GRAPH DAG IN APACHE SPARK

1. Objective

(Directed Acyclic Graph) DAG in **Apache Spark** is a set of **Vertices** and **Edges**, where *vertices* represent the **RDDs** and the *edges* represent the **Operation to be applied on RDD**. In Spark DAG, every edge is directed from earlier to later in the sequence. On calling of *Action*, the created DAG is submitted to **DAG Scheduler**which further splits the graph into the **stages** of the **task**.

In this Apache Spark tutorial, we will understand what is DAG in Apache Spark, what is DAG Scheduler, what is the need of directed acyclic graph in Spark, how DAG is created in Spark and how it helps in achieving fault tolerance. We will also learn how DAG works in RDD, the advantages of DAG

in Spark which creates the difference between Apache Spark and Hadoop MapReduce.



2. What is DAG in Apache Spark?

DAG is a finite directed graph with no directed cycles. There are finitely many *vertices* and *edges,* where each edge directed from one vertex to another. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence. It is a strict generalization of **MapReduce** model. DAG operations can do better global optimization than other systems like MapReduce. The picture of DAG becomes clear in more complex jobs.

Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage. In the *stage view,* the details of all **RDDs** belonging to that stage are expanded. The Scheduler splits the Spark RDD into **stages** based on various transformation applied. (You can refer this link to learn RDD Transformations and Actions in detail) Each stage is comprised of **tasks**, based on the partitions of the RDD, which will perform same computation in parallel. *The graph here refers to navigation, and directed and acyclic refers to how it is done.*

3. Need of Directed Acyclic Graph in Spark

The limitations of Hadoop MapReduce became a key point to introduce DAG in Spark. The computation through MapReduce is carried in three steps:

- The data is read from HDFS.
- Map and Reduce operations are applied.

- The computed result is written back to HDFS.

Each MapReduce operation is independent of each other and **HADOOP** has no idea of which Map reduce would come next. Sometimes for some iteration, it is irrelevant to read and write back the immediate result between two map-reduce jobs. In such case, the memory in stable storage **(HDFS**) or disk memory gets wasted.

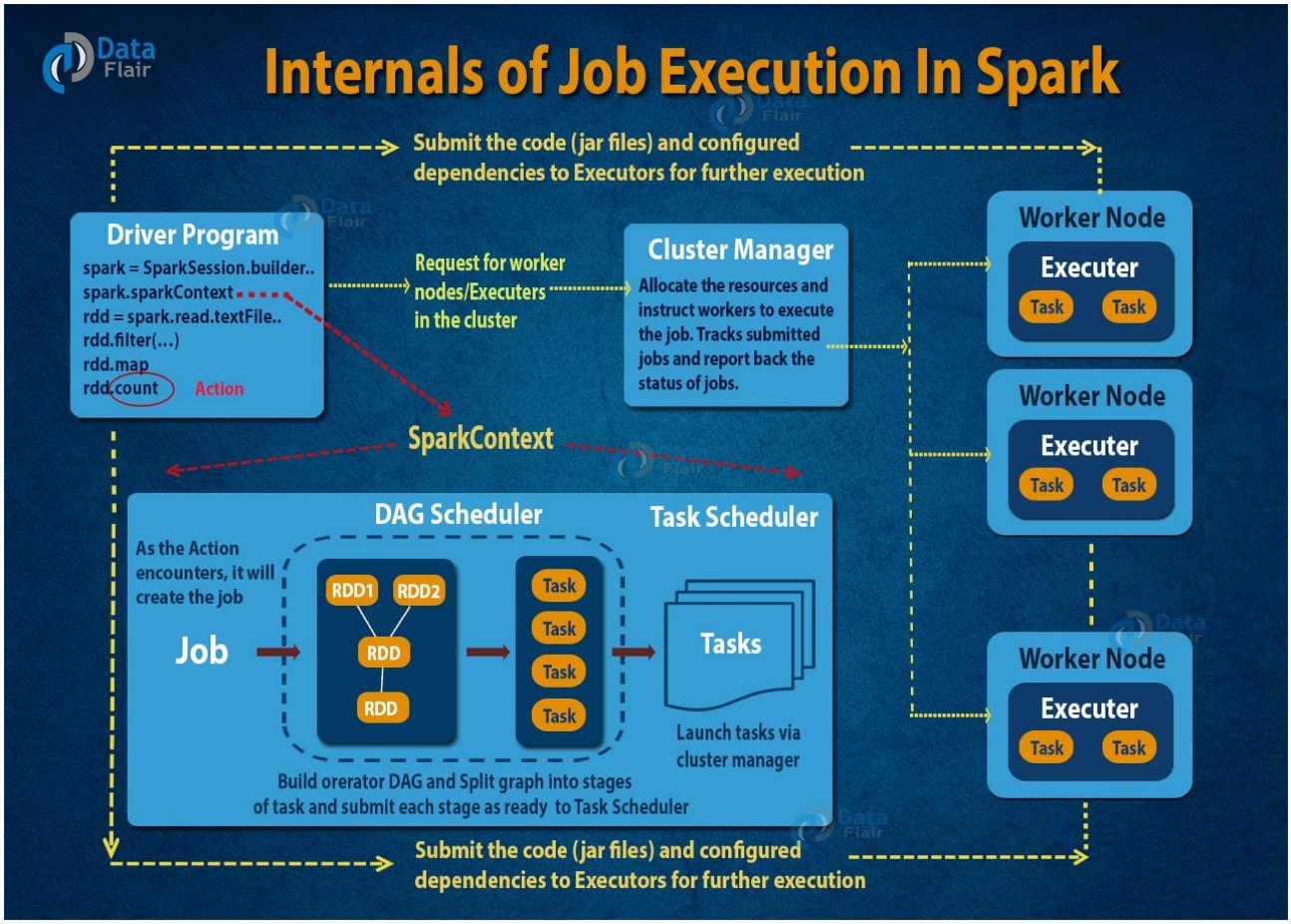
In multiple-step, till the completion of the previous job all the jobs are blocked from the beginning. As a result, complex computation can require a long time with small data volume.

While in Spark, a DAG (Directed Acyclic Graph) of consecutive computation stages is formed. In this way, the execution plan is optimized, e.g. to minimize shuffling data around. In contrast, it is done manually in MapReduce by tuning each MapReduce step.

4. How DAG works in Spark?

- The interpreter is the first layer, using a Scala interpreter, Spark interprets the code with some modifications.
- Spark creates an operator graph when you enter your code in Spark console.
- When an **Action** is called on Spark RDD at a high level, Spark submits the operator graph to the **DAG Scheduler.**
- Operators are divided into **stages** of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators are scheduled in a single stage.
- The stages are passed on to the **Task Scheduler**. It launches task through **cluster manager**. The dependencies of stages are unknown to the task scheduler.
- The **Workers** execute the task on the slave.

The image below briefly describes the steps of How DAG works in the Spark job execution.



At higher level, two type of RDD transformations can be applied: **narrow transformation** (e.g. map(), filter() etc.) and **wide transformation** (e.g. reduceByKey()). *Narrow transformation* does not require the shuffling of data across a partition, the narrow transformations will be grouped into single stage while in *wide transformation* the data is shuffled. Hence, Wide transformation results in stage boundaries.

Each RDD maintains a pointer to one or more parent along with metadata about what type of relationship it has with the parent. For example, if we call *val b=a.map()* on an RDD, the RDD *b* keeps a reference to its parent RDD *a*, that's an **RDD lineage**.

5. How is Fault tolerance achieved through DAG?

RDD is split into the partition and each node is operating on a partition at any point in time. Here, the series of **Scala function** is executing on a partition of the RDD. These operations are composed together and Spark execution engine view these as DAG (Directed Acyclic Graph).

When any node crashes in the middle of any operation say O3 which depends on operation O2, which in turn O1. The *cluster manager* finds out the node is dead and assign another node to continue processing. This node will operate

on the particular partition of the RDD and the series of operation that it has to execute (O1->O2->O3). Now there will be no data loss.

You can refer this link to learn Fault Tolerance in Apache Spark.

6. Working of DAG Optimizer in Spark

The DAG in Apache Spark is optimized by rearranging and combining operators wherever possible. For, example if we submit a spark job which has a **map() operation** followed by a **filter operation**. The **DAG Optimizer** will rearrange the order of these operators since filtering will reduce the number of records to undergo map operation.

7. Advantages of DAG in Spark

There are multiple advantages of Spark DAG, let's discuss them one by one:

- The lost RDD can be recovered using the Directed Acyclic Graph.
- Map Reduce has just two queries the map, and reduce but in DAG we have multiple levels. So to execute SQL query, DAG is more flexible.
- DAG helps to achieve fault tolerance. Thus the lost data can be recovered.
- It can do a better global optimization than a system like Hadoop MapReduce.

8. Conclusion

DAG in Apache Spark is an alternative to the MapReduce. It is a programming style used in distributed systems. In MapReduce, we just have two functions (map and reduce), while DAG has multiple levels that form a tree structure. Hence, DAG execution is faster than MapReduce because intermediate results are not written to disk.

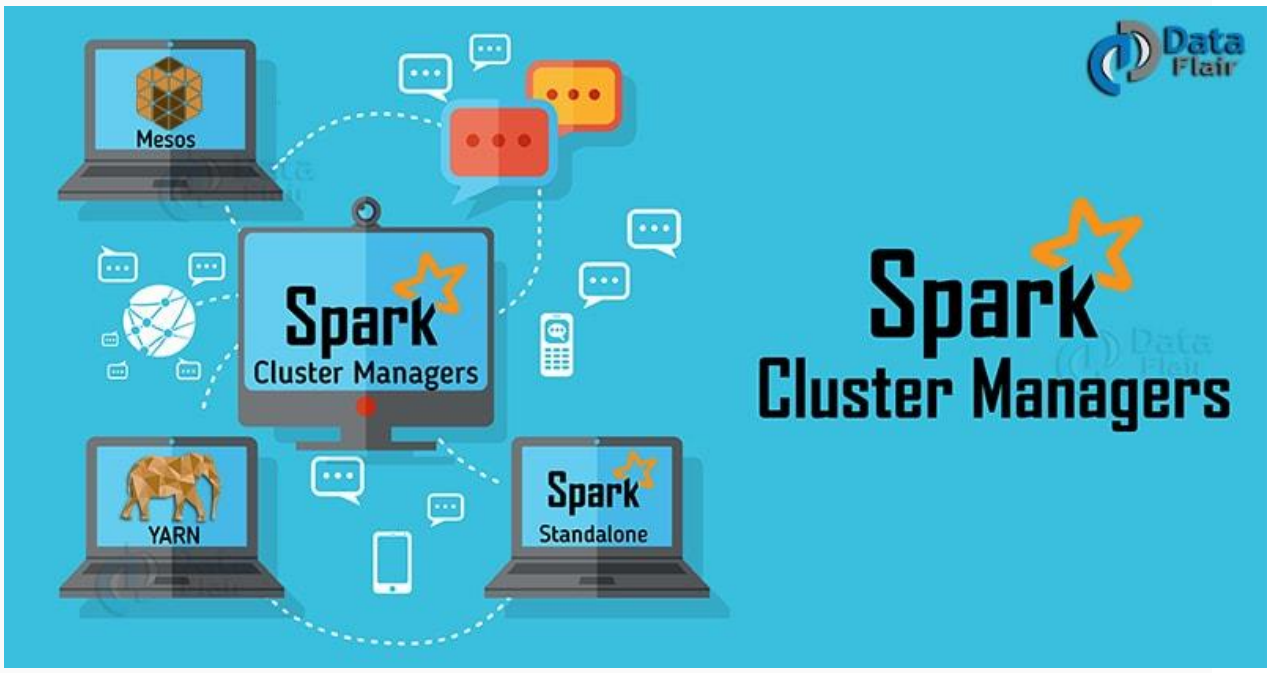
If in case you have any confusion about DAG in Apache Spark, then feel free to share with us. We will be glad to solve your queries.

22.APACHE SPARK CLUSTER MANAGERS – YARN, MESOS & STANDALONE

1. Objective

In this tutorial on **Apache Spark** cluster managers, we are going to learn what Cluster Manager in Spark is. Various types of cluster managers-**Spark Standalone cluster, YARN mode, and Spark Mesos**. We will learn how

Apache Spark cluster managers work. The difference between Spark Standalone vs YARN vs Mesos is also covered in this blog.



2. Introduction to Apache Spark Cluster Managers

Apache Spark is an engine for **Big Data** processing. One can run Spark on distributed mode on the cluster. In the cluster, there is master and n number of workers. It schedules and divides resource in the host machine which forms the cluster. The prime work of the cluster manager is to divide resources across applications. It works as an external service for acquiring resources on the cluster.

The cluster manager dispatches work for the cluster. Spark supports pluggable cluster management. The cluster manager in Spark handles starting executor processes. Refer this link to learn Apache Spark terminologies and concepts.

Apache Spark system supports three types of cluster managers namely-

a) Standalone Cluster Manager

b) Hadoop YARN

c) Apache Mesos

Let's discuss these Apache Spark Cluster Managers in detail.

2.1. Apache Spark Standalone Cluster Manager

Standalone mode is a simple cluster manager incorporated with Spark. It makes it easy to setup a cluster that Spark itself manages and can run

on **Linux**, Windows, or Mac OSX. Often it is the simplest way to run Spark application in a clustered environment. Learn, how to install Apache Spark On Standalone Mode.

a. How does Spark Standalone Cluster Works?

It has ***masters*** and number of ***workers*** with configured amount of memory and CPU cores. In Spark standalone cluster mode, Spark allocates resources based on the core. By default, an application will grab all the cores in the cluster.

In standalone cluster manager, **Zookeeper quorum** recovers the master using *standby master.* Using the file system, we can achieve the manual recovery of the master. Spark supports authentication with the help of shared secret with entire cluster manager. The user configures each node with a shared secret. For communication protocols, Data encrypts using SSL. But for block transfer, it makes use of data SASL encryption.

To check the application, each Apache Spark application has a Web User Interface. The Web UI provides information of executors, storage usage, running task in the application. In this cluster manager, we have Web UI to view cluster and job statistics. It also has detailed log output for each job. If an application has logged event for its lifetime, Spark Web UI will reconstruct the application's UI after the application exits.

2.2. Apache Mesos

Mesos handles the workload in distributed environment by dynamic resource sharing and isolation. It is healthful for deployment and management of applications in large-scale cluster environments. Apache Mesos clubs together the existing resource of the machines/nodes in a cluster. From this, a variety of workloads may use. This is node abstraction, thus it decreases an overhead of allocating a specific machine for different workloads. It is resource management platform for **Hadoop** and **Big Data**cluster. Companies such as *Twitter*, *Xogito*, and *Airbnb* use Apache Mesos as it can run on Linux or Mac OSX.

In some way, Apache Mesos is the reverse of *virtualization*. This is because in virtualization one physical resource divides into many virtual resources. While in Mesos many physical resources are club into a single virtual resource. Refer this link to learn Apache Mesos in detail.

a. How do Apache Mesos works?

The three components of Apache Mesos are *Mesos masters*, *Mesos slave*, *Frameworks*.

Mesos Master is an instance of the cluster. A cluster has many Mesos masters that provide fault tolerance. Here one instance is the leading master. **Mesos Slave** is Mesos instance that offers resources to the cluster. Mesos Master assigns the task to the slave. **Mesos Framework** allows

applications to request the resources from the cluster. Thus, the application can perform the task.

Some other Frameworks by Mesos
are *Chronos*, *Marathon*, *Aurora*, *Hadoop*, *Spark*, *Jenkins* etc.

Mesos authentication includes:

- The slave's registration with the master.
- Frameworks (that is, applications) submission to the cluster.
- Operators using endpoints such as HTTP endpoints.

Get the best Apache Mesos books to master Mesos.

2.3. Hadoop YARN

YARN became the sub-project of Hadoop in the year 2012. It is also known as **MapReduce 2.0.** YARN bifurcate the functionality of *resource manager* and *job scheduling* into different daemons. The plan is to get a Global Resource Manager (RM) and per-application Application Master (AM). An application is either a **DAG** of graph or an individual job. To learn YARN is great detail follow this Yarn tutorial.

a. How do Hadoop YARN works?

YARN data computation framework is a combination of
the *ResourceManager*, the *NodeManager*. It can run on Linux and Windows.

The Yarn Resource Manager manages resources among all the applications in the system. The Resource Manager has scheduler and Application Manager. **The Scheduler** allocates resource to the various running application. It is pure Scheduler, performs monitoring or tracking of status for the application. **The Application Manager** manages applications across all the nodes.

Yarn Node Manager contains Application Master and container. **A container** is a place where a unit of work happens. Each task of MapReduce runs in one container. The per-application **Application Master** is a framework specific library. It aims to negotiate resources from the Resource Manager. It continues with Node Manager(s) to execute and watch the tasks.

The application or job requires one or more containers. Node Manager handles monitoring containers, resource usage (CPU, memory, disk, and network). It reports this to the Resource Manager. YARN provides security for authentication, service level authorization. It also provides authentication for Web consoles and data confidentiality. Hadoop authentication
uses *Kerberos* to verify that each user and service has authentication.

3. Spark Standalone mode vs YARN vs Mesos

In this tutorial of Apache Spark Cluster Managers, features of 3 modes of Spark cluster have already present. Let us now see the comparison between Standalone mode vs YARN cluster vs Mesos Cluster in Apache Spark in details. It will help you to understand which Apache Spark Cluster Managers type one should choose for Spark.

3.1. High Availability

The **standalone cluster**: with *ZooKeeper Quorum* it supports an automatic recovery of the master. One can achieve manual recovery using the file system. The cluster tolerates the worker failure despite the recovery of the Master is enabling or not.

The **Apache Mesos**: using Apache ZooKeeper it supports an automatic recovery of the master. In the case of failover, tasks which are currently executing, do not stop their execution.

Apache Hadoop YARN: using a command line utility it supports manual recovery. And use Zookeeper-based *ActiveStandbyElector* embedded in the ResourceManager for automatic recovery. Thus, there is no need to run a separate ZooKeeper Failover Controller.

3.2. Security

Spark supports authentication via a shared secret with all the cluster managers. The **standalone manager** requires the user to configure each of the nodes with the shared secret. Data can be encrypted using SSL for the communication protocols. SASL encryption is supported for block transfers of data. Other options are also available for encrypting data. Access to Spark applications in the Web UI can be controlled via access control lists.

Mesos: For any entity interacting with the cluster Mesos provides authentication. This includes the slaves registering with the master, frameworks submitted to the cluster, and operators using endpoints such as HTTP endpoints. These entities can be enabling to use authentication or not. Custom module can replace Mesos' default authentication module, Cyrus SASL.

To allow access to services in Mesos Access, it makes use of control lists. By default, communication between the modules in Mesos is unencrypted. SSL/TLS can be enabled to encrypt communication. Mesos WebUI supports HTTPS.

Hadoop YARN: It contains security for authentication, service level authorization, authentication for Web consoles and data confidentiality. Using Service level authorization it ensures that client using Hadoop services has

authority. Using access control lists Hadoop services can be controlled. Additionally, using SSL data and communication between clients and services is encrypted. The data transferred between the Web console and clients with HTTPS.

3.4. Monitoring

Spark's standalone cluster manager: To view cluster and job statistics it has a Web UI. It also has detailed log output for each job. the Spark Web UI will reconstruct the application's UI after the application exists if an application has logged events for its lifetime.

Apache Mesos: It supports per container network monitoring and isolation. It provides many metrics for *master* and *slave* nodes accessible with URL. These metrics include percentage and number of allocated CPU's, memory usage etc.

Hadoop YARN has a Web UI for the *ResourceManager* and the *NodeManager*. The ResourceManager UI provides metrics for the cluster. And the NodeManager provides information for each node, the applications and containers running on the node.

4. Conclusion

In conclusion to Apache Spark Cluster Managers, we can say Standalone mode is easy to set up among all. It will provide almost all the same features as the other cluster managers.

To use richer resource scheduling capabilities (e.g. queues), both YARN and Mesos provide these features. Of these, YARN allows you to share and configure the same pool of cluster resources between all frameworks that run on YARN. It is likely to be pre-installed on Hadoop systems.

One advantage of Mesos over both YARN and the standalone mode is its fine-grained sharing option. This lets interactive applications (Spark shell) scale down their CPU allocation between commands. This makes it attractive in environments where many users are running interactive shells.

If you like this blog or have any query about Apache Spark Cluster Managers, so, do let us know by leaving a comment.

23.HOW APACHE SPARK WORKS – RUN-TIME SPARK ARCHITECTURE

1. Objective

Here we are going to learn how Apache Spark Works? In **Apache Spark**, the central coordinator is called the driver. When you enter your code in spark, SparkContext in the driver program creates the job when an Action is called.

This job is submitted to DAG Scheduler which creates the operator graph and then submits it to task Scheduler. Task Scheduler launches the task via cluster manager. Thus, with the help of cluster manager, a Spark Application is launched on a set of machines.

This Apache Spark tutorial will explain the run-time architecture of Apache Spark along with key **Spark terminologies** like Apache SparkContext, Spark shell, Apache Spark application, task, job and stages in Spark. We will also learn about the components of Spark run time architecture like the Spark driver, cluster manager & Spark executors. At last, we will see how Apache spark works using these components.

2. Internals of How Apache Spark works?

Apache Spark is an open source, general-purpose distributed computing engine used for processing and analyzing a large amount of data. Just like **HadoopMapReduce**, it also works with the system to distribute data across the cluster and process the data in parallel. Spark uses master/slave architecture i.e. one central coordinator and many distributed workers. Here, the central coordinator is called the**driver**.

The driver runs in its own Java process. These drivers communicate with a potentially large number of distributed workers called **executor**s. Each executor is a separate java process. A **Spark Application** is a combination of driver and its own executors. With the help of cluster manager, a Spark Application is launched on a set of machines. **Standalone Cluster Manager** is the default built in **cluster manager of Spark**. Apart from its built-in cluster manager, Spark also works with some open source cluster manager like **Hadoop Yarn**, **Apache Mesos** etc.

3. Terminologies of Spark

3.1. Apache SparkContext

SparkContext is the heart of Spark Application. It establishes a connection to the Spark Execution environment. It is used to create **Spark RDDs**, accumulators, and broadcast variables, access Spark services and run jobs. SparkContext is a client of Spark execution environment and acts as the master of Spark application. The main works of Spark Context are:

- Getting the current status of spark application
- Canceling the job
- Canceling the Stage
- Running job synchronously
- Running job asynchronously
- Accessing persistent RDD
- Unpersisting RDD
- Programmable dynamic allocation

Read about SparkContext in detail.

3.2. Apache Spark Shell

Spark Shell is a Spark Application written in **Scala**. It offers command line environment with auto-completion. It helps us to get familiar with the **features of Spark**, which help in developing our own Standalone Spark Application. Thus, this tool helps in exploring Spark and is also the reason why Spark is so helpful in processing data set of all size.

3.3. Spark Application

The Spark application is a self-contained computation that runs user-supplied code to compute a result. A Spark application can have processes running on its behalf even when it's not running a job.

3.4. Task

A **task** is a unit of work that is sent to the executor. Each stage has some task, one task per partition. The Same task is done over different partitions of RDD.

Learn: Spark Shell Commands to Interact with Spark-Scala

3.5. Job

The **job** is parallel computation consisting of multiple tasks that get spawned in response to **actions in Apache Spark.**

3.6. Stage

Each job gets divided into smaller sets of tasks called **stages** that depend on each other. Stages are classified as computational boundaries. All computation cannot be done in single stage. It is achieved over many stages.

4. Components of Spark Run-time Architecture of Spark

4.1. Apache Spark Driver

The **main()** method of the program runs in the driver. The driver is the process that runs the user code that **creates RDDs**, and performs **transformation and action**, and also creates SparkContext. When the Spark Shell is launched, this signifies that we have created a driver program. On the termination of the driver, the application is finished.

The driver program splits the Spark application into the task and schedules them to run on the executor. The task scheduler resides in the driver and distributes task among workers. The two main key roles of drivers are:

- Converting user program into the task.
- Scheduling task on the executor.

The structure of Spark program at a higher level is: RDDs are created from some input data, derive new RDD from existing using various transformations, and then after it performs an action to compute data. In Spark Program, the **DAG (directed acyclic graph)** of operations are created implicitly. And when the driver runs, it converts that Spark DAG into a physical execution plan.

4.2. Apache Spark Cluster Manager

Spark relies on cluster manager to launch executors and in some cases, even the drivers are launched through it. It is a pluggable component in Spark. On the cluster manager, jobs and action within a spark application are scheduled by Spark Scheduler in a FIFO fashion. Alternatively, the scheduling can also be done in Round Robin fashion. The resources used by a Spark application can be dynamically adjusted based on the workload. Thus, the application can free unused resources and request them again when there is a demand. This is available on all coarse-grained cluster managers, i.e. standalone mode, YARN mode, and Mesos coarse-grained mode.

Learn: Spark RDD – Introduction, Features & Operations of RDD

4.3. Apache Spark Executors

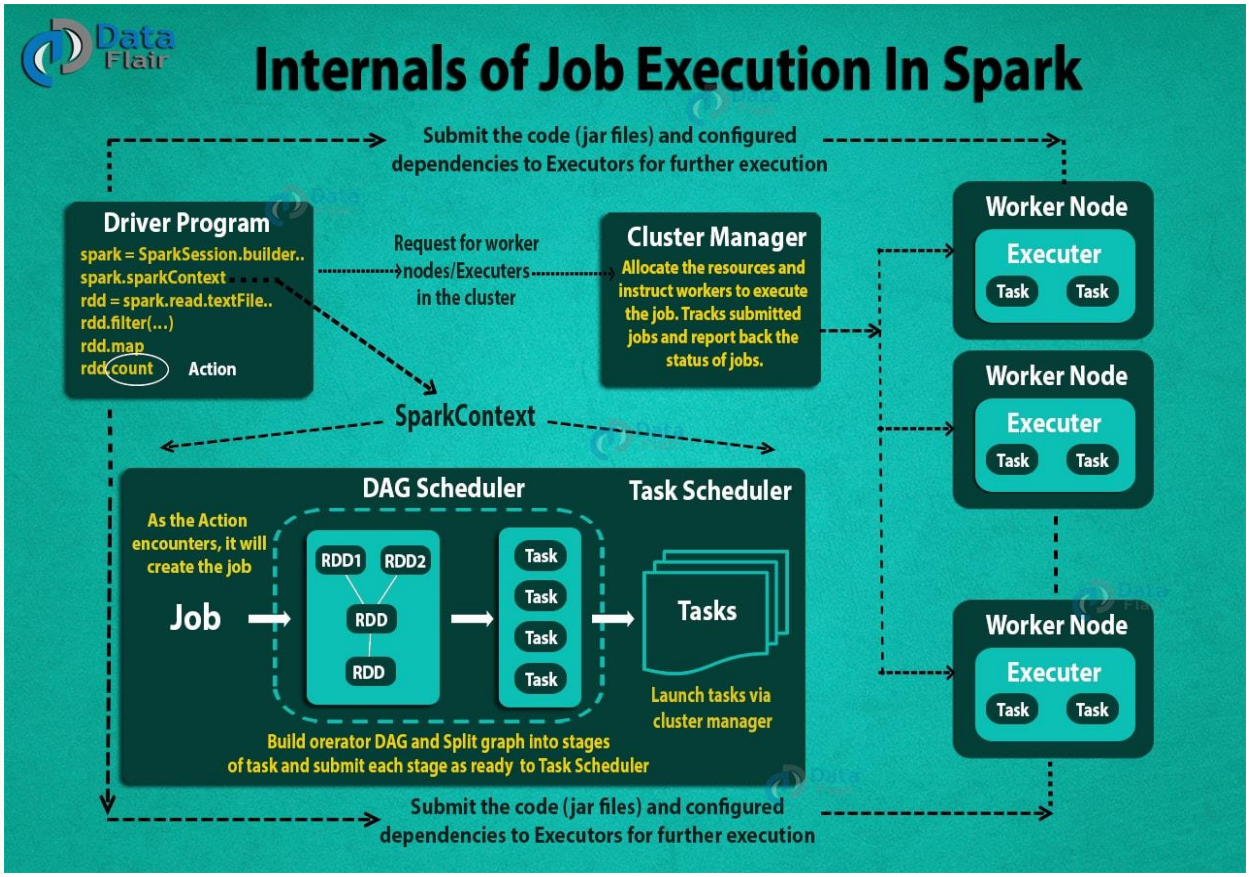
The individual task in the given Spark job runs in the Spark executors. Executors are launched once in the beginning of Spark Application and then they run for the entire lifetime of an application. Even if the Spark executor fails, the Spark application can continue with ease. There are two main roles of the executors:

- Runs the task that makes up the application and returns the result to the driver.
- Provide **in-memory** storage for RDDs that are cached by the user.

5. How to launch a Program in Spark?

Despite using any cluster manager, Spark comes with the facility of a single script that can be used to submit a program, called as **spark-submit**. It launches the application on the cluster. There are various options through which spark-submit can connect to different cluster manager and control how many resources our application gets. For some cluster managers, spark-submit can run the driver within the cluster (e.g., on a **YARN** worker node), while for others, it can run only on your local machine.

6. How to Run Apache Spark Application on a cluster



- Using spark-submit, the user submits an application.
- In spark-submit, the main() method specified by the user is invoked. It also launches the driver program.
- The driver program asks for the resources to the cluster manager that is required to launch executors.
- The cluster manager launches executors on behalf of the driver program.
- The driver process runs with the help of user application. Based on the **actions and transformation on RDDs**, the driver sends work to executors in the form of tasks.
- The executors process the task and the result is sent back to the driver through cluster manager.

If you have any query about Apache Spark job execution flow, so feel free to share with us. We will be happy to solve them.

24.6 IMPORTANT REASONS TO LEARN APACHE SPARK

1. Objective

With the increasing size of data getting generated every second, it has become important to analyze this data to get important business insights in a lesser time. Several **Big data** options
like **Hadoop**, **Storm**, **Spark**, **Flink** etc. have made this possible. But out of hundreds of choices available, why should you learn Apache Spark, how Apache Spark has replaced Hadoop and became most popular Big Data Engine and why is industry running behind Spark is a major concern. So let's see the reasons to learn Apache Spark.

2. Top 6 Reasons to Learn Apache Spark?

Let's now see the reason for why should you learn Apache Spark?

Apache Spark is a next Gen Big Data Tool. It provides both batch and streaming processing capabilities for faster data processing. 9 Out of 10 Companies have started using Apache Spark for their data processing. Because of its wide range of applications and ease of use to work with, Spark

is also called the Swiss army knife of **Big Data Analytics**. To learn more about Apache Spark follow this comprehensive guide. Here are few reasons to learn Apache Spark now and keep yourself moving technically ahead of others:

2.1. High compatibility with Hadoop

When Hadoop came into the picture, companies started investing in this technology. Even professionals from varied domains started learning it quickly. By the time Apache Spark was launched companies have already invested hugely in Hadoop (especially hardware and resources), it is not feasible to invest again for Spark.

Hence, Spark has come up with compatibility with Hadoop: Spark can be deployed on the same hardware of Hadoop and can use its resource management layer – **Yarn**apart from this Spark can process the data stored in **HDFS (Hadoop Distributed File System)**. If you are a professional with Hadoop knowledge, learning Spark would be advantageous as companies are now looking for Spark experts rather than Hadoop alone.

2.2. Hadoop is dwindling while Spark is sparking

Spark is **100 times faster** than **MapReduce**. It is easier to program in Spark as compared to Map Reduce. This has made Spark one of the top Apache Projects. Apache Spark is an **in-memory data processing** framework with all **Hadoop capabilities**. With the coming of Apache Spark, it has been projected for the possibility of the end of Map Reduce era. Hadoop was limited to just MapReduce, but Spark is a generalized framework to process the huge volume of data. Learn more about the differences between Hadoop vs Spark.

2.3. Increased access to Big Data

In today's leading world we are generating multi-terabytes of data. The amount of data generation is increasing day by day and this huge volume of data can't be accessed by traditional methods. So for eliminating the problem of **Big Data**, **Hadoop** emerged, but it has some limitations which are eliminated by Apache Spark.

Spark is more efficient than Hadoop due to its real time processing. Most of the **data scientist** prefer to work with Spark as it less complex and because of its fast speed. Data scientist prefer works with Spark mainly because of as Spark has the ability to store data resident in memory that helps speed up machine learning workloads.

2.4. High demand for Spark professionals

Spark is fastly becoming an ecosystem itself. Spark toolset is also constantly expanding which is attracting growing third-party interest. According to John Trippier, Alliances and Ecosystem Lead at Databricks, "*The adoption of Apache Spark by businesses large and small is growing at an incredible rate across a wide range of industries, and the demand for developers with certified expertise is quickly following suit*". So you can give a boost to your career and salary by learning Apache Spark.

2.5. Diverse

Spark enables users to write applications in **Java**, **Scala**, **Python**, **R**. This helps them to create and run their applications on programming languages they are comfortable with. Spark also scores on many senses. You can write a custom Spark big data app, use **Spark SQL** and do data analysis using SQL, set up ETL pipelines using Spark, use **Spark streaming** and make it part of real-time data pipeline, use MLlib **machine learning** library and run Analytics… Or even do graph processing with **GraphX**. On top of this, Scala is supported by Java which helps write concise code. It can replace a 50 line java map-reduce code with a 2-3 line Scala Spark code.

2.6. Apache Spark to make Big Money

Spark is the tool with the greatest coefficient. According to indeed.com, the average salary for a Spark Developer in San Francisco is **$128,000** as of December 16, 2015.

According to the O'Reilly 2014 Data Science Salary Survey, Spark developers earn the highest median salary among developers using the 10 most widely used Big Data development tools. In 2015 **Data Science** Salary Survey, O'Reilly found strong correlations between those who used Apache Spark and Scala and those who were paid more money.

In one of its models, using Spark added more than **$11,000** to the median salary, while Scala had about a **$4,000** impact to the bottom line. O'Reilly says in its report. According to it, "*learning Spark could apparently have more impact on salary than getting a Ph.D. Scala is another bonus: those who use both are expected to earn over **$15,000** more than an otherwise equivalent data professional.*"

So what are you waiting for then when Apache Spark has so many additional features for you. Learn now and grab the opportunity in the market.

25.APACHE SPARK COMPATIBILITY WITH HADOOP

1. Objective

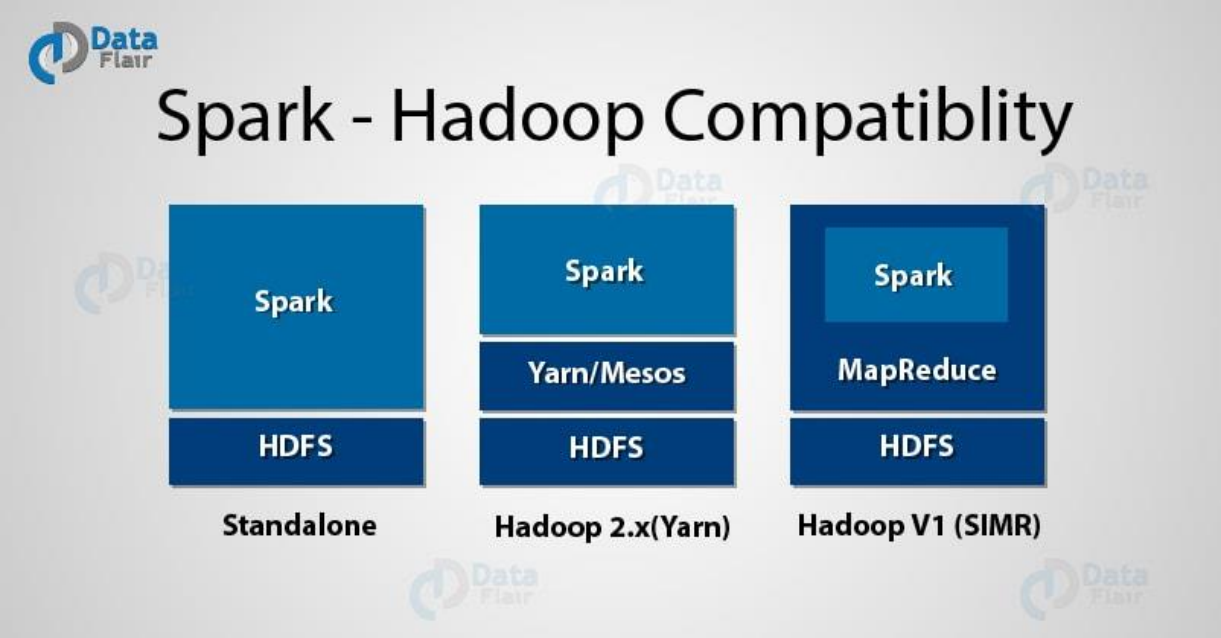
In this tutorial on **Apache Spark** compatibility with Hadoop, we will discuss how Spark is compatible with Hadoop? This tutorial covers three ways to use Apache Spark over Hadoop i.e. **Standalone**, **YARN**, **SIMR(Spark In MapReduce)**. We will also discuss the steps to launch Spark application in standalone mode, Launch Spark on YARN, Launch Spark in MapReduce (SIMR), and how SIMR works in this Spark Hadoop compatibility tutorial.

2. How is Spark compatible with Hadoop?

It is always mistaken that Spark replaces **Hadoop**, rather it influences the **functionality of Hadoop**. Right from the starting Spark read data from and write data to HDFS (**Hadoop Distributed File System**). Thus we can say that Apache Spark is Hadoop-based data processing engine; it can take over batch and streaming data overheads. Hence, running Spark over Hadoop provides enhanced and more functionality.

3. Apache Spark Compatibility with Hadoop

In three ways we can use Spark over Hadoop:



- **Standalone –** In this deployment mode we can allocate resource on all machines or on a subset of machines in **Hadoop Cluster**. We can run Spark side by side with Hadoop **MapReduce.**
- **YARN –** We can run Spark on **YARN** without any pre-requisites. Thus, we can also integrate Spark in Hadoop stack and take an **advantage and facilities of Spark.**
- **SIMR (Spark in MapReduce) –** Another way to do this is by launching Spark job inside Map reduce. With SIMR we can use Spark shell in few minutes after downloading it. Hence, this reduces the overhead of deployment, and we can play with Apache Spark.

Let's discuss these three ways of Apache Spark compatibility with Hadoop one by one in detail.

3.1. Launching Spark Application in Standalone Mode

Before Launching Spark application in standalone mode refer this guide to learn how to install Apache Spark in Standalone Mode(single node cluster)?

Spark support two deployment modes for standalone cluster namely the **cluster mode** and the **client mode**. In client mode, the driver is launched in the same process in which client submits the application. In cluster mode, the driver is launched from one of worker node process inside the cluster, the client process exit as it submits the application without waiting for the application to finish.

a. Adding the jar

If we launch the application through **Spark submit**, application jar gets automatically distributed to all worker nodes. For any additional jar specify it through *–jars* flag, use comma as a delimiter. If the application exits with non-zero exit code, the standalone cluster mode will restart your application.

b. Running application in Standalone Mode

If we want to run Spark application in standalone mode by taking input from **HDFS**use the code:

```
$ ./bin/spark-submit --class MyApp.class --master MyApp.jar --input hdfs://localhost:9000/input-file-path --
output output-file-path
```

3.2. Launching Spark on YARN

Apache Spark running on YARN was added in version 0.6.0 and was improved in later releases.

If we want to run Spark job on Hadoop YARN cluster(Learn to configure Hadoop with yarn in pseudo distributed mode) we first need to merge Spark JAR. It merges all the essential and required dependencies. We can achieve this by setting Hadoop version and SPARK_YARN environment variable as:

SPARK_HADOOP_VERSION=2.0.5-alpha SPARK_YARN=true sbt/sbt assembly.

Make sure YARN_CONF_DIR or HADOOP_CONF_DIR indicates those directories which have a configuration file for **Hadoop cluster**. Using these configurations we write to HDFS and connect to **YARN Resource Manager**. The configurations that are contained in this directory are shared among YARN cluster so that all the **containers**that are used by applications use the same configuration.

To launch Spark application on YARN there are two deployment modes namely: the *cluster mode* and the *client mode*.

i. Cluster Mode – In cluster mode, the **Spark driver** runs inside Application Master Process and this is managed by YARN on the cluster.

ii. Client Mode – In this mode, the driver runs in client process and Application Master is used only for requesting a resource from YARN and providing it to the driver program.

In YARN mode the address of ResourceManager is taken from Hadoop Configuration. So, here the –master parameter is yarn.

If we want to launch Spark application in cluster mode use the command:

```
$ ./bin/spark-submit --class path.to.your.Class --master yarn --deploy-mode cluster [options] <app jar> [app
options]
```

If we want to launch Spark application in client mode use the command (replace cluster in above by client)

```
$ ./bin/spark-submit --class path.to.your.Class --master yarn --deploy-mode client
```

a. Adding other JARs

When we run in cluster mode, the driver runs on a different machine as that of the client, so to make available the files that are on the client to **SparkContext.addJar**, add those files with –jars option in the launch command.

For Example:

```
$ ./bin/spark-submit --class my.main.Class \
--master yarn \
--deploy-mode cluster \
--jars my-other-jar.jar,my-other-other-jar.jar \
my-main-jar.jar \
app_arg1 app_arg2
```

If we want that Spark runtime jars to be accessible from YARN side, specify-

spark.yarn.archive or spark.yarn.jars.

If we do not specify these then Spark will form a zip file with all jar under $SPARK_HOME/jars and upload it to the distributed cache.

b. Debugging Application on YARN

In YARN, both the application master and executors run inside the "**containers**". Once the application has completed YARN has two modes to handle container log.

i. If log aggregation is turned on

In the case where the log aggregation is turned on using the YARN.log-aggregation-enable config the container logs are copied to **HDFS** and are deleted from the local machine. And later if we want to view this file from anywhere on cluster use the command:

```
yarn logs -applicationId <app ID>
```

This command will print out the content from all log files from all containers from given application.

We can also see the container log files in HDFS directly by using HDFS Shell or API. If we want to find the directory where the log file is present, use this command:

yarn.nodemanager.remote-app-log-dir and *yarn.nodemanager.remote-app-log-dir-suffix*

ii. If log aggregation is not turned on

in this case, logs are kept locally on each machine under *yarn_app_logs_dir*, which is generally configured to /tmp/logs or $HADOOP_HOME/logs/userlogs depending on the Hadoop version and installation.

If we want to view log from a container, we must first go to the host that contains it and look in the directory. Further, the sub-directory maintains log files by application ID and container ID.

3.3. Launching Spark in MapReduce (SIMR)

It is an easy way for Hadoop MapReduce 1 user to use Apache Spark. Using this we can run Spark job and Spark Shell without **installing Spark** or **Scala**, or have administrative rights. The only pre-requisites are HDFS access and MapReduce v1. SIMR is open-sourced and is a joint work of Databricks and UC Berkeley AMPLab. Once the SIMR is downloaded, we can try it by typing

```
./simr --shell
```

To use this user only need to download package of SIMR that matches **Hadoop cluster.** The package of SIMR contains 3 files:

SIMR runtime script: simr

simr-<hadoop-version>.jar

spark-assembly-<hadoop-version>.jar

To get usage information place all three in the directory and execute SIMR. the job SIMR is to launch MapReduce jobs with required number of map slots and makes sure that Spark/Scala and jobs are sent to all those nodes. One of the **mappers** is set as master and inside that mapper, Spark driver is made to run. On the remaining mapper SMIR launch the Spark executor, these executors will execute a task on behalf of the driver.

The master is selected by leader election by **writing to HDFS**, the mapper which writes first in the HDFS is set as the master mapper. And the remaining mapper finds the driver URL by **reading a specific file from HDFS**. Thus, in place of cluster manager SIMR uses MapReduce and HDFS.

a. How does SIMR work?

SIMR allows the user to interact with the driver program. On the master mapper, the SIMR runs the relay server and the relay client is run on the machine that launches SIMR. The input to the client and the output from the driver goes to and fro between the client and the master mapper. Hence, to achieve all this HDFS is extensively used.

4. Conclusion

In conclusion to Apache Spark compatibility with Hadoop, we can say that Spark is a Hadoop-based data processing framework; it can take over batch and streaming data overheads. Hence, running Spark over Hadoop provides enhanced and extra functionality. After studying Hadoop Spark compatibility follow this guide to **learn how Apache Spark works?**

If you feel any query about this post on Apache Spark Compatibility with Hadoop, please feel free to share with us. Hope we will solve them.

26.SPARK PERFORMANCE TUNING-LEARN TO TUNE APACHE SPARK JOB

1. Objective

The Spark Performance Tuning is the process of adjusting settings to record for memory, cores, and instances used by the system. This process guarantees that the Spark has optimal performance and prevents resource bottlenecking in Spark.

In this Tutorial of Performance tuning in **Apache Spark**, we will provide you complete details about How to tune your Apache Spark jobs? This Spark Tutorial covers performance tuning introduction in Apache Spark, Spark Data Serialization libraries such as **Java serialization** & **Kryo serialization**, Spark Memory tuning. We will also learn about Spark Data Structure Tuning, Spark Data Locality and Garbage Collection Tuning in Spark in this Spark performance tuning and Optimization tutorial.

Refer this guide to learn the Apache Spark installation in the Standalone mode.

2. What is Performance Tuning in Apache Spark?

The process of adjusting settings to record for memory, cores, and instances used by the system is termed **tuning**. This process guarantees that the

Spark has optimal performance and prevents resource bottlenecking. Effective changes are made to each property and settings, to ensure the correct usage of resources based on system specific setup. **Apache Spark** has **in-memory computation** nature. As a result resources in the cluster (CPU, memory etc.) may get bottlenecked.

Sometimes to decrease memory usage **RDDs** are stored in serialized form. Data serialization plays important role in good network performance and can also help in reducing memory usage, and memory tuning.

If used properly, tuning can:

- Ensure proper use of all resources in an effective manner.
- Eliminates those jobs that run long.
- Improves the performance time of the system.
- Guarantees that jobs are on correct execution engine.

3. Data Serialization in Spark

It is the process of converting the in-memory object to another format that can be used to store in a file or send over the network. It plays a distinctive role in the performance of any distributed application. The computation gets slower due to formats that are slow to serialize or consume a large number of files. **Apache Spark**gives two serialization libraries:

- Java serialization
- Kryo serialization

Java serialization – Objects are serialized in Spark using an **ObjectOutputStream**framework, and can run with any class that implements **java.io.Serializable.** The performance of serialization can be controlled by extending **java.io.Externalizable.** It is flexible but slow and leads to large serialized formats for many classes.

Kryo serialization – To serialize objects, Spark can use the Kryo library (Version 2). Although it is more compact than Java serialization, it does not support all Serializable types. For better performance, we need to register the classes in advance. We can switch to **Karyo** by initializing our job with *SparkConf* and calling-

conf.set("spark.serializer", "org.apache.spark.serializer.KyroSerializer")

We use the *registerKryoClasses* method, to register our own class with Kryo. In case our objects are large we need to increase *spark.kryoserializer.buffer* config. The value should be large so that it can hold the largest object we want to serialize.

4. Memory Tuning in Spark

Consider the following three things in tuning memory usage:

- Amount of memory used by objects (the entire dataset should fit in-memory)
- The cost of accessing those objects
- Overhead of garbage collection.

The Java objects can be accessed but consume 2-5x more space than the raw data inside their field. The reasons for such behavior are:

- Every distinct Java object has an "object header". The size of this header is 16 bytes. Sometimes the object has little data in it, thus in such cases, it can be bigger than the data.
- There are about 40 bytes of overhead over the raw string data in Java String. It stores each character as two bytes because of String's internal usage of UTF-16 encoding. If there are 10 characters String, it can easily consume 60 bytes.
- Common collection classes like ***HashMap*** and ***LinkedList*** make use of linked data structure, there we have "wrapper" object for every entry. This object has both header and pointer (8 bytes each) to the next object in the list.
- Collections of primitive types often store them as "boxed objects". For example, *java.lang.Integer.*

a. Spark Data Structure Tuning

By avoiding the Java features that add overhead we can reduce the memory consumption. There are several ways to achieve this:

- Avoid the nested structure with lots of small objects and pointers.
- Instead of using strings for keys, use numeric IDs or enumerated objects.
- If the RAM size is less than 32 GB, set JVM flag to ***–xx:+UseCompressedOops***to make a pointer to four bytes instead of eight.

b. Spark Garbage Collection Tuning

JVM garbage collection is problematic with large churn **RDD** stored by the program. To make room for new objects, Java removes the older one; it traces all the old objects and finds the unused one. But the key point is that cost of garbage collection in Spark is proportional to a number of Java objects. Thus, it is better to use a data structure in Spark with lesser objects.

One more way to achieve this is to persist objects in serialized form. As a result, there will be only one object per RDD partition.

5. Memory Management in Spark

We consider Spark memory management under two categories: **execution** and **storage**. The memory which is for computing in shuffles, Joins, aggregation is *Execution memory*. While the one for **caching** and propagating internal data in the cluster is storage memory. Both execution and storage share a unified region M. When the execution memory is not in use, the storage can use all the memory. The same case lies true for Storage memory. Execution can drive out the storage if necessary. This is done only until storage memory usage falls under certain threshold R.

We can get several properties by this design. First, the application can use entire space for execution if it does not use caching. While the applications that use caching can reserve a small storage (R), where data blocks are immune to evict.

Even though we have two relevant configurations, the users need not adjust them. Because default values are relevant to most workloads:

- *memory.fraction* describes the size of M as a fraction of the (JVM heap space-300MB)(default 0.6). The remaining 40% is stored in user data structure, internal metadata in Spark and safeguarding against OOM error in case of Sparse and large records.
- *memory.storageFraction* shows the size of R as the fraction of M (default 0.5).

Learn How Fault Tolerance is achieved in Apache Spark.

6. Determining Memory Consumption in Spark

If we want to know the size of Spark memory consumption a dataset will require to **create an RDD,** put that RDD into the cache and look at "Storage" page in Web UI. This page will let us know the amount of memory RDD is occupying.

If we want to know the memory consumption of particular object, use SizeEstimator'S estimate method.

7. Spark Garbage Collection Tuning

In garbage collection, tuning in Apache Spark, the first step is to gather statistics on how frequently garbage collection occurs. It also gathers the

amount of time spent in garbage collection. Thus, can be achieved by adding **-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps** to Java option. The next time when Spark job run, a message will display in workers log whenever garbage collection occurs. These logs will be in worker node, not on drivers program.

Java heap space divides into two regions Young and Old. The young generation holds short-lived objects while Old generation holds objects with longer life. The garbage collection tuning aims at, long-lived RDDs in the old generation. It also aims at the size of a young generation which is enough to store short-lived objects. With this, we can avoid full garbage collection to gather temporary object created during task execution. Some steps that may help to achieve this are:

- If full garbage collection is invoked several times before a task is complete this ensures that there is not enough memory to execute the task.
- In garbage collection statistics, if OldGen is near to full we can reduce the amount of memory used for caching. This can be achieved by lowering spark.memory.fraction. the better choice is to cache fewer objects than to slow down task execution. Or we can decrease the size of young generation i.e., lowering –Xmn.

The effect of **Apache Spark garbage collection tuning** depends on our application and amount of memory used.

8. Other consideration for Spark Performance Tuning

a. Level of Parallelism

To use the full cluster the level of parallelism of each program should be high enough. According to the size of the file, Spark sets the number of "Map" task to run on each file. The level of parallelism can be passed as a second argument. We can set the config property ***spark.default.parallelism*** to change the default.

b. Memory Usage of Reduce Task in Spark

Although RDDs fit in our memory many times we come across a problem of ***OutOfMemoryError***. This is because the working set of our task say *groupByKey* is too large. We can fix this by increasing the level of parallelism so that each task's input set is small. We can increase the number of *cores* in our cluster because Spark reuses one executor JVM across many tasks and has low task launching cost.

Learn about groupByKey and other Transformations and Actions API in Apache Spark with examples.

c. Broadcasting Large Variables

The size of each serialized task reduces by using broadcast functionality in **SparkContext.** If a task uses a large object from driver program inside of them, turn it into the broadcast variable. Generally, it considers the tasks that are about 20 Kb for optimization.

d. Data Locality in Apache Spark

Data locality plays an important role in the performance of Spark Jobs. The case in which the data and code that operates on that data are together, the computation is faster. But if the two are separate, then either the code should be moved to data or vice versa. It is faster to move serialized code from place to place then the chunk of data because the size of the code is smaller than the data.

Based on data current location there are various levels of locality. The order from closest to farthest is:

- The best possible locality is that the PROCESS_LOCAL resides in same JVM as the running code.
- NODE_LOCAL resides on the same node in this. It is because the data travel between processes is quite slower than PROCESS_LOCAL.
- There is no locality preference in NO_PREF data is accessible from anywhere.
- RACK_LOCAL data is on the same rack of the server. Since the data is on the same rack but on the different server, so it sends the data in the network, through a single switch.
- ANY data resides somewhere else in the network and not in the same rack.

9. Conclusion

Consequently, to increase the performance of the system performance tuning plays the vital role. Serializing the data plays an important role in tuning the system. Spark employs a number of optimization techniques to cut the processing time. Thus, Performance Tuning guarantees the better performance of the system.

After learning performance tuning in Apache Spark, Follow this guide to learn How Apache Spark works in detail.

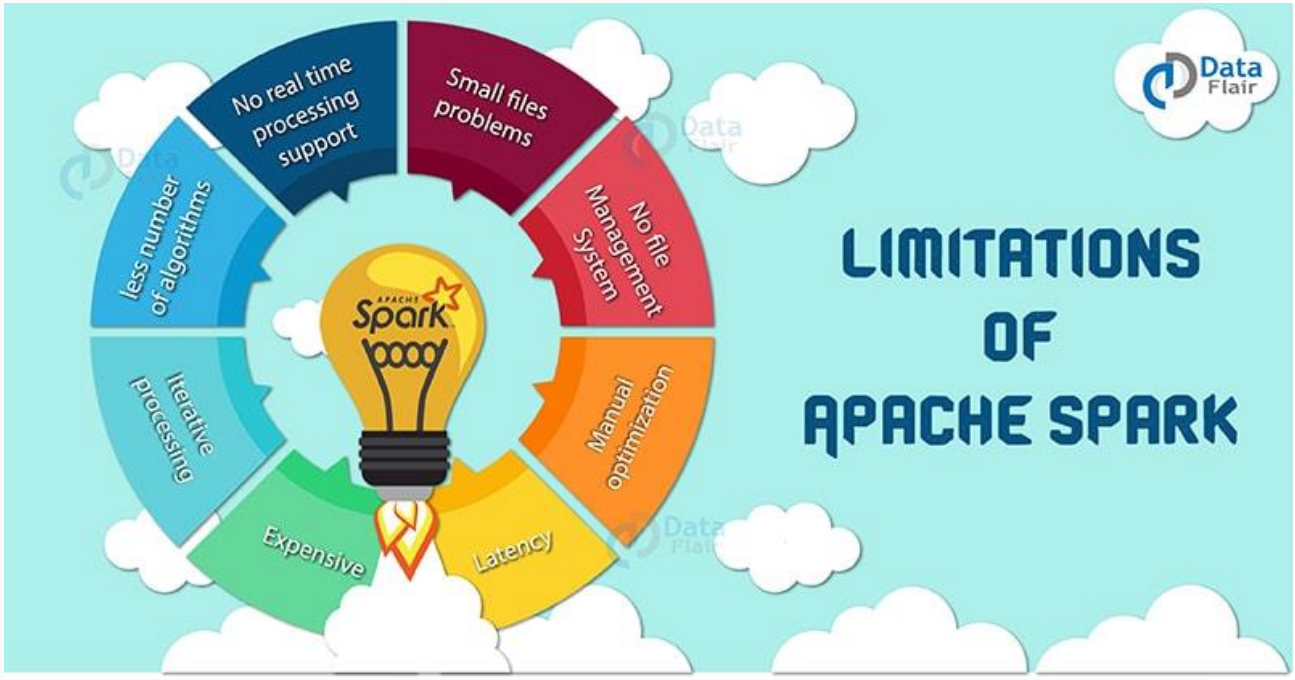
You can share your queries about Spark performance tuning, by leaving a comment. We will be happy to solve them.

27.LIMITATIONS OF APACHE SPARK – WAYS TO OVERCOME SPARK DRAWBACKS

1. Objective

Some of the drawbacks of **Apache Spark** are there is no support for real-time processing, Problem with small file, no dedicated File management system, Expensive and much more due to these limitations of Apache Spark, industries have started shifting to **Apache Flink**– 4G of **Big Data.**

In this Apache Spark limitations tutorial, we will discuss these Apache Spark disadvantages and how to overcome these limitations of Apache Spark.



2. Limitations of Apache Spark

As we know **Apache Spark** is the next Gen Big data tool that is being widely used by industries but there are certain limitations of Apache Spark due to which industries have started shifting to **Apache Flink**– **4G of Big Data**. Before we learn what are the disadvantages of Apache Spark, let us learn the advantages of Apache Spark.

So let us now understand Apache Spark problems and when not to use Spark.

a. No Support for Real-time Processing

In **Spark Streaming**, the arriving live stream of data is divided into batches of the pre-defined interval, and each batch of data is treated like **Spark**

Resilient Distributed Database (RDDs). Then these RDDs are processed using the operations like map, reduce, join etc. The result of these operations is returned in batches. Thus, it is not real time processing but Spark is near real-time processing of live data. Micro-batch processing takes place in Spark Streaming.

b. Problem with Small File

If we use Spark with **Hadoop**, we come across a problem of a small file. **HDFS**provides a limited number of large files rather than a large number of small files. Another place where Spark legs behind is we store the data gzipped in **S3**. This pattern is very nice except when there are lots of small gzipped files. Now the work of the Spark is to keep those files on network and uncompress them. The gzipped files can be uncompressed only if the entire file is on one core. So a large span of time will be spent in burning their core unzipping files in sequence.

In the resulting **RDD**, each file will become a partition; hence there will be a large amount of tiny partition within an RDD. Now if we want efficiency in our processing, the RDDs should be repartitioned into some manageable format. This requires extensive shuffling over the network.

c. No File Management System

Apache Spark does not have its own file management system, thus it relies on some other platform like **Hadoop** or another cloud-based platform which is one of the Spark known issues.

d. Expensive

In-memory capability can become a bottleneck when we want cost-efficient processing of big data as keeping data in memory is quite expensive, the memory consumption is very high, and it is not handled in a user-friendly manner. Apache Spark requires lots of RAM to run in-memory, thus the cost of Spark is quite high.

e. Less number of Algorithms

Spark MLlib lags behind in terms of a number of available algorithms like Tanimoto distance.

f. Manual Optimization

The Spark job requires to be manually optimized and is adequate to specific datasets. If we want to partition and **cache in Spark** to be correct, it should be controlled manually.

g. Iterative Processing

In Spark, the data iterates in batches and each iteration is scheduled and executed separately.

h. Latency

Apache Spark has higher latency as compared to **Apache Flink**.

i. Window Criteria

Spark does not support record based window criteria. It only has time-based window criteria.

j. Back Pressure Handling

Back pressure is build up of data at an input-output when the buffer is full and not able to receive the additional incoming data. No data is transferred until the buffer is empty. Apache Spark is not capable of handling pressure implicitly rather it is done manually.

These are some of the major pros and cons of Apache Spark. We can overcome these limitations of Spark by using Apache Flink – 4G of Big Data.

3. Conclusion

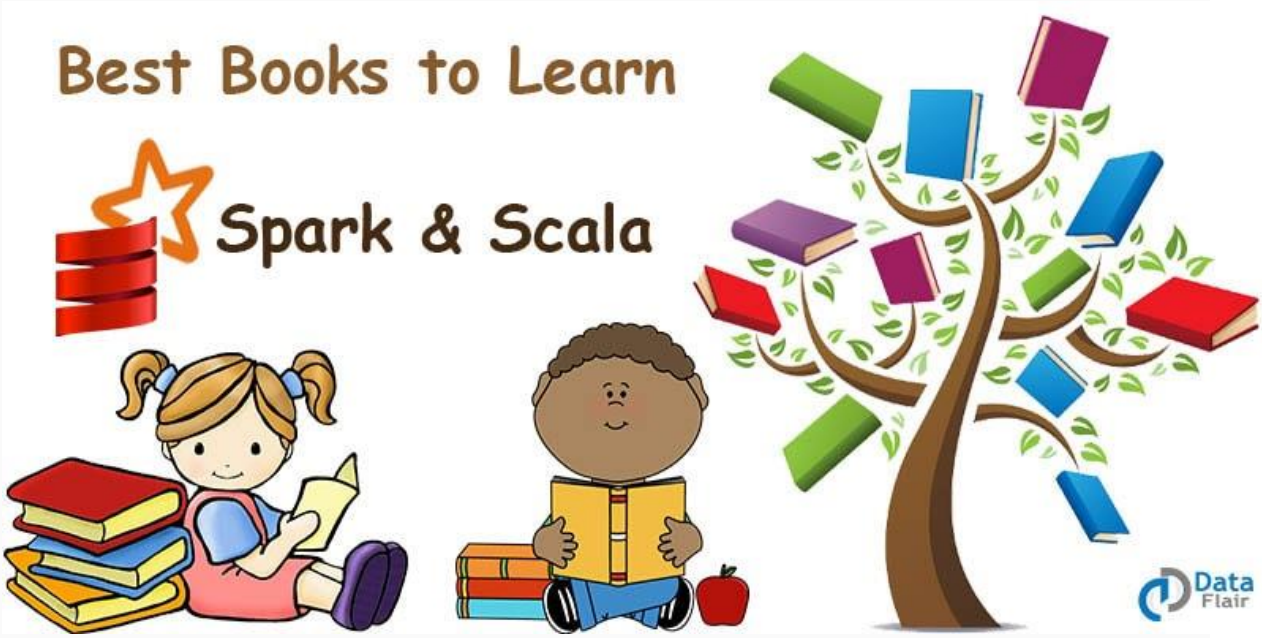
Although Spark has many drawbacks, it is still popular in the market for **big data** solution. But there are various technologies that are overtaking Spark. Like stream processing is much better using Flink then Spark as it is real time processing. Learn feature wise differences between **Apache Spark vs Apache Flink** to understand which is better and how.

27.BEST APACHE SPARK AND SCALA BOOKS FOR MASTERING SPARK SCALA

1. Objective

This blog on **Apache Spark** and **Scala books** give the list of best books of Apache Spark that will help you to learn **Apache Spark**. "Because to become a master in some domain good books are the key". It also gives the list of best books of **Scala** to start programming in Scala. Some of these books are for beginners to learn Scala Spark and some of these are for advanced level Spark Scala learning.

In addition to the name, this blog also contains a brief description of each book.



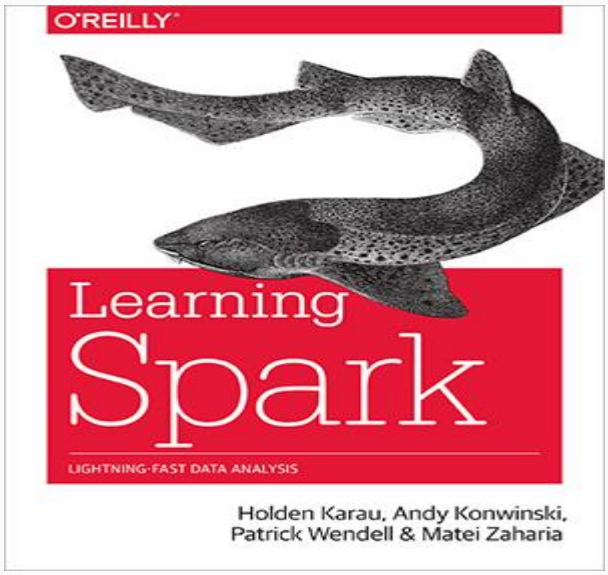
2. 10 Ultimate Apache Spark And Scala Books

Before we start learning Spark Scala from books, first of all understand what is Apache Spark and Scala programming language.

So, let's have a look at the list of Apache Spark and Scala books-

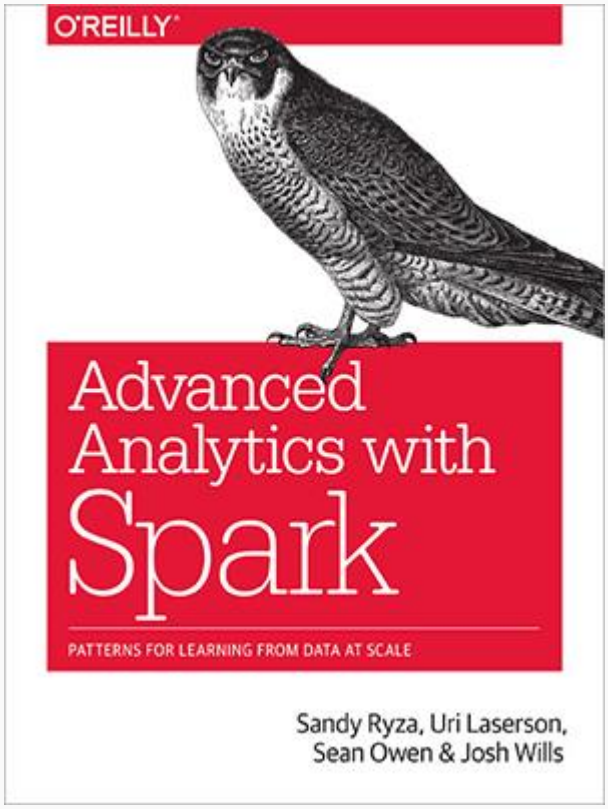
2.1. Apache Spark Books

1) Learning Spark by Matei Zaharia, Patrick Wendell, Andy Konwinski, Holden Karau



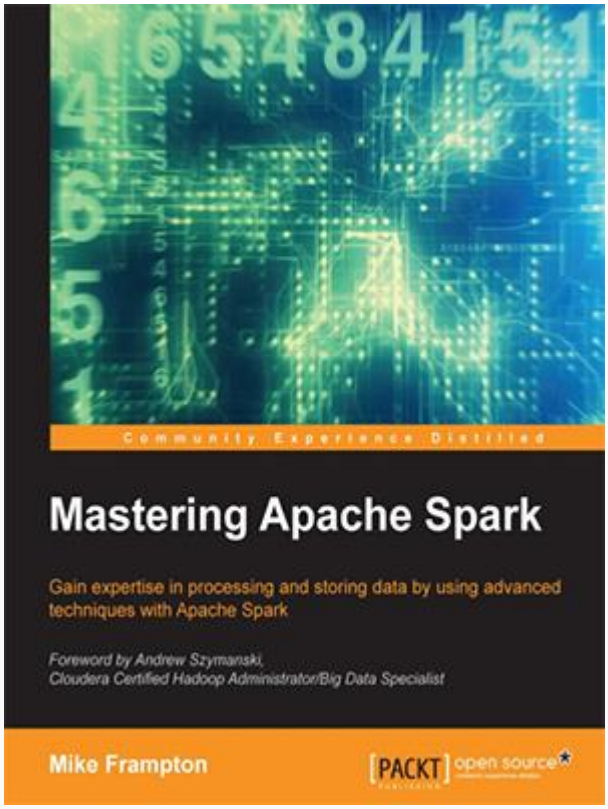
It is a learning guide for those who are willing to learn Spark from basics to advance level. It covers all key concepts like **RDD**, ways to create RDD, different transformations and actions, Spark SQL, Spark streaming, etc and has examples in all 3 languages *Java, Python, and Scala*. So, it provides a learning platform for all those who are from java or python or Scala background and want to learn Apache Spark.

2) Advanced Analytics with Spark by Sandy Ryza, Uri Laserson, Sean Owen and Josh Wills



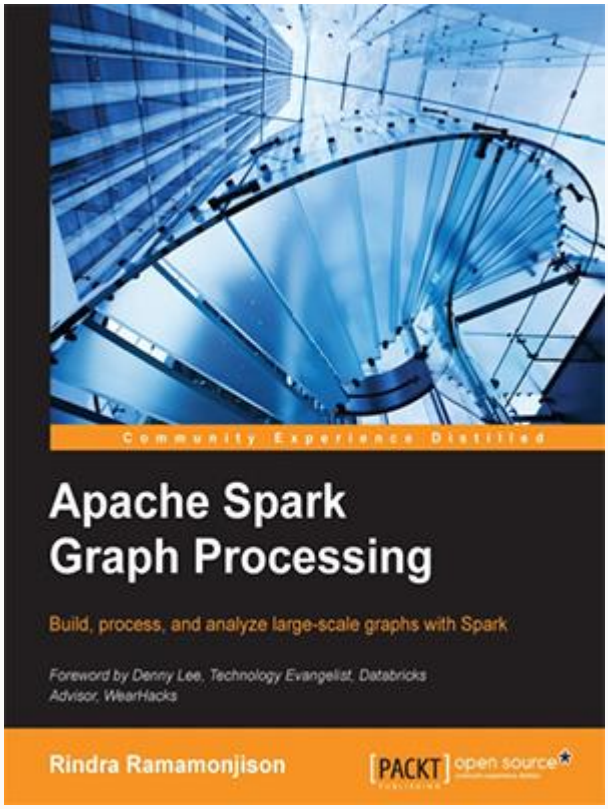
This book is meant for those who have basic knowledge on Spark and want to raise their Spark knowledge further. It covers how Spark is used to deal with large-scale data analytics. How to install Spark on single node cluster and Spark installation on a multi-node cluster. It also teaches how to approach analytics problem using statistical methods to make you Spark expert.

3) Mastering Apache Spark by Mike Frampton



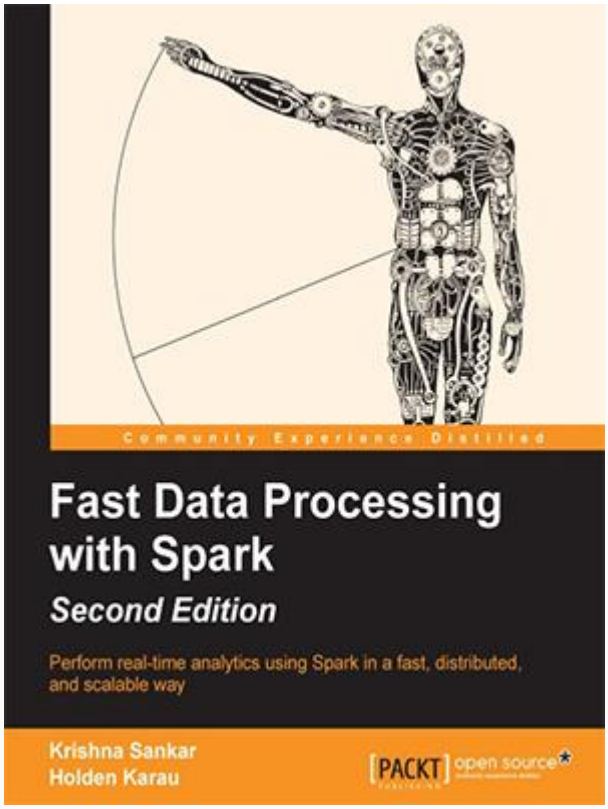
This book is especially for those readers who know basics about Spark and want to gain advanced programming knowledge with the help of **Spark use cases.** It assumes that the reader has basic knowledge about **Hadoop**, **Linux**, **Spark**, and **Scala**. It covers Spark integration with Databricks, Titan, H2O etc and other Spark features like **MLlib**, **Spark Stream Processing**, **Spark SQL** and Spark **GraphX.**

4) Apache Spark Graph Processing by Rindra Ramamonjison



This book will help the user to do **graphical programming in Spark** and also help them in building, processing and analyze large-scale graph data with Spark effectively. The later chapters of this book cover advanced topics like clustering graphs, implementing graph-parallel iterative algorithms and learning methods from graph data.

5) Fast Data Processing with Spark by Holden Karau and Krishna Sankar

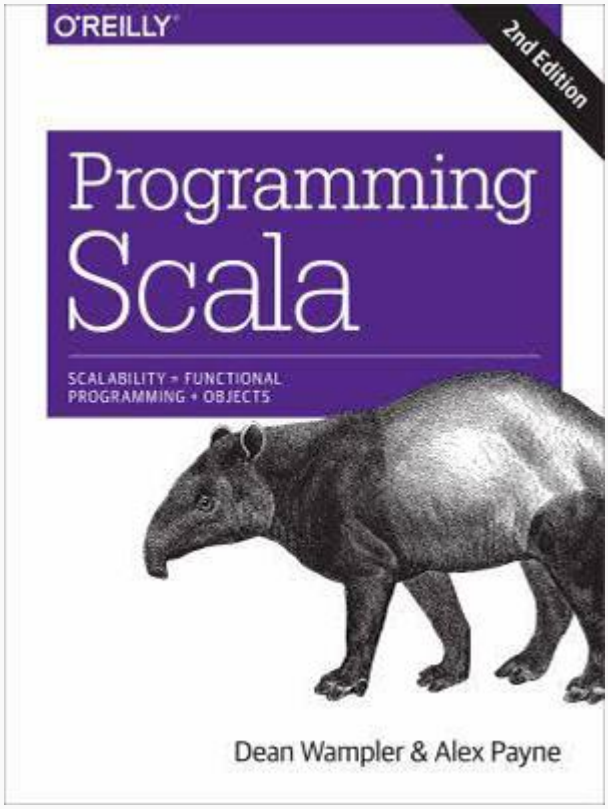


This book is for software developers who are willing to learn how to write distributed programs with Spark. It will also guide you through steps needed to write effectively distributed programs starting from setting up your cluster and exploring the API, deploying your job to the cluster, and tuning it. It also covers how **Hive** can be used with Spark for **SQL**-like query syntax with **Spark-SQL**, as well as manipulating **resilient distributed datasets (RDDs)**.

2.2. Best Scala Books

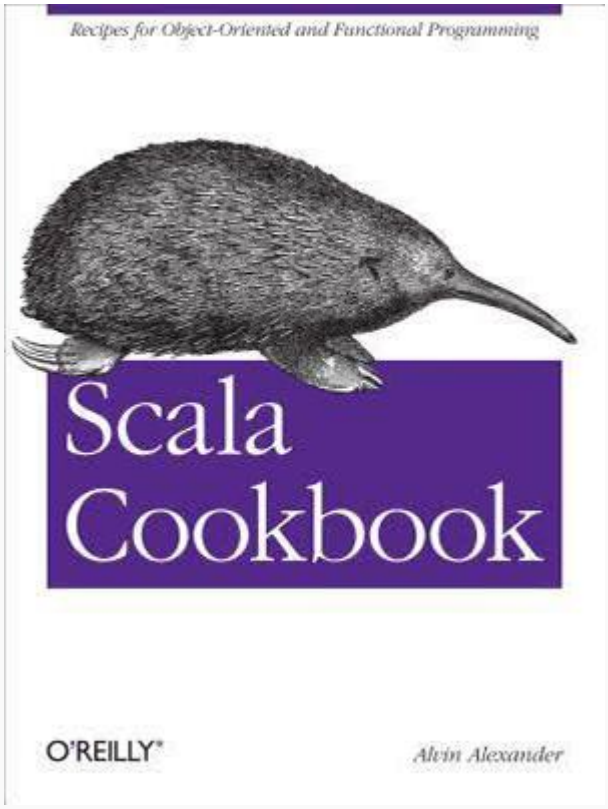
Now that you have started learning Spark, You have understood that **Scala** and **Java**are programming languages that are being used with Spark. Furthermore, if you are not comfortable with Java but would like to learn Scala, below are some of the best books to learn Scala and start your career in Spark and Scala.

1) Programming Scala by Dean Wampler, Alex Payne



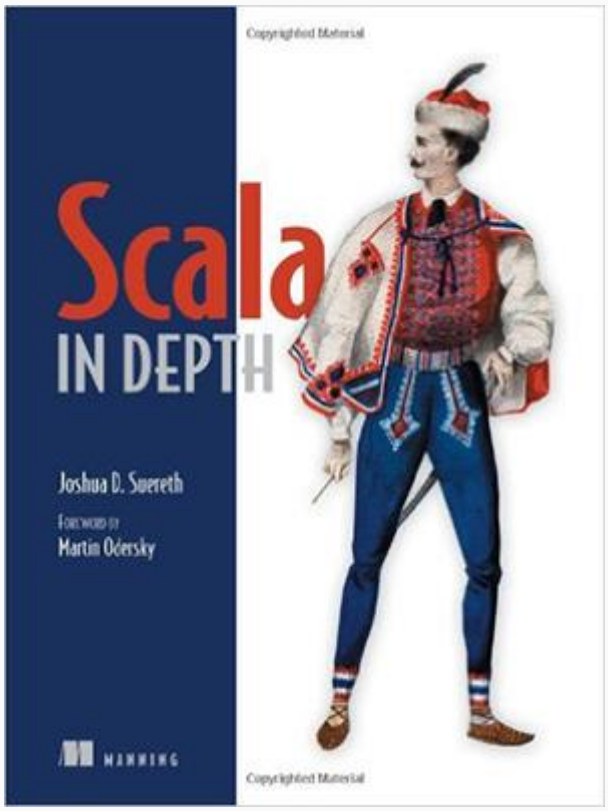
It explains Scala concepts as JVM language and how Scala turns out to be the best development option. Scala can also be learned using the test-driven approach from basics to hands-on the level through this book.

2) Scala Cookbook by Alvin Alexander



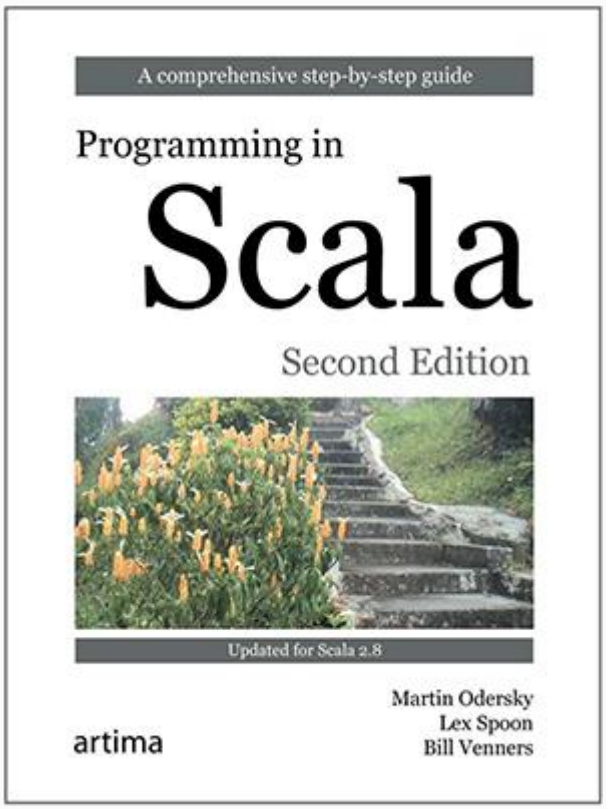
Here the author has used his experience to highlight **Scala features** in an efficient manner. It covers some of the best features of Scala like *FlatMap* and provides answers to questions that a new Scala learner would have.

3) Scala in depth by Joshua D. Suereth



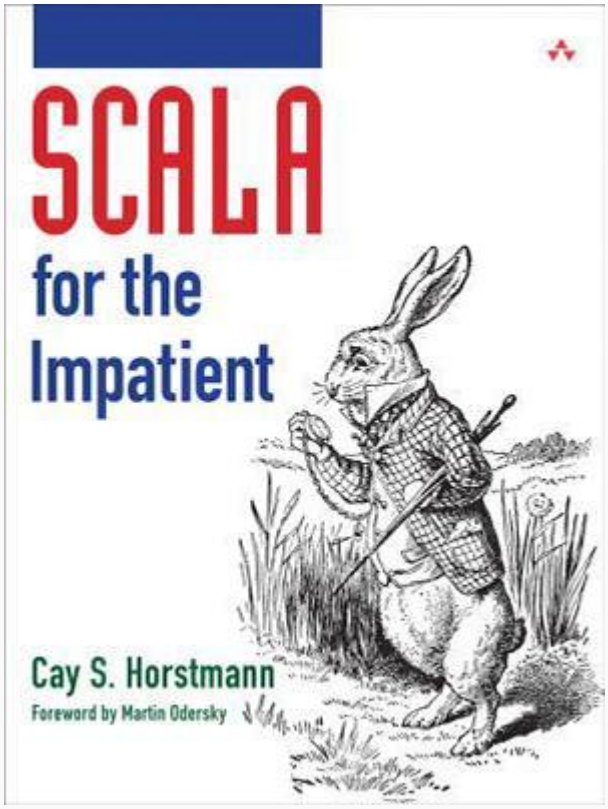
This book helps you learn how to integrate Scala effectively into the development process and powerful techniques with the help of examples to start your career in Scala. Due to this, it targets programmers willing to improve their programming Scala skills. It also helps you in learning best practices for creating Scala applications.

4) Programming in Scala by Martin Odersky, Lex Spoon, Bill Venners



This book explains the benefits of using Scala for development and starts from basic concepts to higher-end programming features. It also describes advanced **features of Scala** that you need to learn to become a better and productive developer.

5) Scala for the Impatient by Cay S. Horstmann



This book is designed especially for experienced programmers. Here the author explains what Scala can do and how coding can be dome effectively in Scala. It also covers Scala concepts and techniques and provides practical knowledge to readers for becoming Scala master.

I hope this blog helped you to find perfect Apache Spark and Scala Books. Also, leave a comment if you find other good books for Apache Spark and Scala.

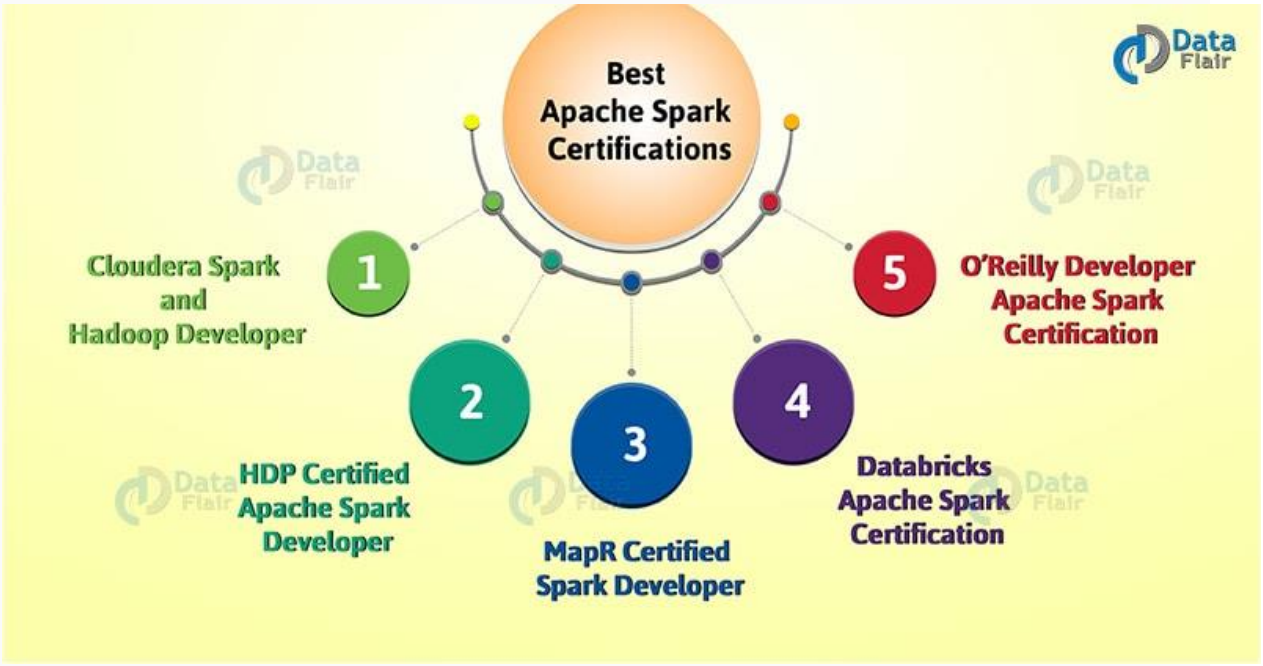
28.TOP 5 APACHE SPARK CERTIFICATIONS FOR YOUR SPARK CAREER

1. Objective

Recently, Apache Spark is extremely popular in the big data world. Moreover, there are hundreds of online resources to learn Spark. Basically, This popularity totally defines the demand for Apache Spark developers. And, to be one is possible though Apache Spark Certifications.

Apart from your practical knowledge of Spark, companies prefer certified candidates to hire. There are plenty of Apache Spark Certifications available. you can do any of them to become eligible for Spark related jobs.

Here, we will discuss some good Apache Spark Certifications. Before that, we will also learn about the reasons to do Apache Spark Certifications.



2. Some Reasons to Become a Certified Spark Developer

- While you think to start a career in the big data world, then this should be your First step to get the best spark certification. It will definitely, give your career a kickstart.
- As soon as you get certified through spark, you have the validation of your Spark skills. That will help as almost all the companies are looking for it.
- It is very easy to acquire Spark certification preparation. Since there are multiple ways you can get certified.
- The most important reason behind Getting certified is it gives you a distinct edge over your peers. Since there is tough competition outside.

3. Best Apache Spark Certifications

1. Cloudera Spark and Hadoop Developer

The feature which separates this certification process is involvement of Hadoop technology. Basically, It is best for those who want to work on both simultaneously. In addition, This certification includes some diverse number of topics. For example Flume, HDFS, Spark with Scala and Python, Avro, Sqoop, Avro, and Impala. Moreover, the questions asked in the certification

test based on programming aptitude. Furthermore, That can be anywhere within the range of 10 to 15.

Time Duration: 2hrs.

Cost of Exam: $295

Follow the below-mentioned link to Official Page:

https://www.cloudera.com/more/training/certification/cca-spark.html

2. HDP Certified Apache Spark Developer

One of the best certifications that you can get in Spark is Hortonworks HDP certified Apache Spark developer. Basically, they will test your Spark Core knowledge as well as Spark Data Frames in this certification. In addition, Those who are considering it very easy, it is not a simple multiple-choice question exam. Furthermore, This exam will check your skills to perform programming tasks on Spark cluster thoroughly.

Time Duration: 2 hours

Cost of Exam: $250

Follow the below-mentioned link to Official Page:

https://hortonworks.com/services/training/certification/hdp-certified-spark-developer/

3. MapR Certified Spark Developer

This is one of the best certification because it is designed for all who are interested to work with Spark. There is no condition to be a programmer, an engineer or a developer for this. Also, helps you to evaluate your Spark skills. With a focus on programming related tasks, there are 60-80 questions in the exam using production level Spark. Furthermore, The only requirement for this certification is Java and Scala programming experience.

Time Duration: 2 hrs.

Cost of Exam: $250

Follow the below-mentioned link to Official Page:

https://mapr.com/training/certification/mcsd/

4. Databricks Apache Spark Certifications

First and Foremost, you need to know either Scala or Python for this certification. On comparing to the HDP certification, Databricks certification is relatively different. Moreover, This tests only your programming Skills in

Spark. Since entire exam covers programming section only. Ultimately, your overall knowledge will be tested here.

Time Duration: 1 hr. 30 mins.

Cost of Exam: $300

Follow the below mentioned link to Official Page:

5. O'Reilly Developer Apache Spark Certifications

The best part of this certification process is the collaboration of Databricks and O'Reilly. Somehow, It seems very similar to the Databricks certification. Although, there is also some input from the O'Reilly media editorial team. Basically, we have seen one thing that if you want to stand out of the crowd, It is a good choice.

Time Duration: 1 hr. 30 mins.

Cost of Exam: $300

Follow the below mentioned link to Official Page:

http://www.oreilly.com/data/sparkcert

4. Conclusion

As a result, we have mentioned all the best Apache Spark Certifications on this blog. Moreover, we have also covered the reasons to do Spark Certifications. Hence, Spark certifications will give a boost to your Career. Still, if you feel any queries, feel free to ask in the comment section.

30.SPARK SQL – AN INTRODUCTORY GUIDE FOR BEGINNERS

1. Objective

Apache Spark **SQL** is a Spark module to simplify working with structured data using DataFrame and DataSet abstractions in Python, Java, and **Scala**. These abstractions are the distributed collection of data organized into named columns. It provides a good optimization technique. Using Spark SQL we can query data, both from inside a Spark program and from external tools that connect through standard database connectors (JDBC/ODBC) to Spark SQL.

This tutorial covers the components of Spark SQL architecture like **DataSets** and **DataFrames**, Apache Spark SQL Catalyst optimizer. We will also learn what is the need of Spark SQL in Apache Spark, Spark SQL advantage, and disadvantages.

2. Apache Spark SQL Tutorial

2.1. Spark SQL Introduction

Apache Spark SQL is a module for structured data processing in Spark. Using the interface provided by Spark SQL we get more information about the structure of the data and the computation performed. With this extra information, one can achieve extra optimization in Apache Spark. We can interact with Spark SQL in various ways like **DataFrame** and the **Dataset API**. The Same execution engine is used while computing a result, irrespective of which API/language we use to express the computation. Thus, the user can easily switch back and forth between different APIs, it provides the most natural way to express a given transformation.

In Apache Spark SQL we can use structured and semi-structured data in three ways:

- To simplify working with structured data it provides DataFrame abstraction in *Python, Java, and **Scala***. DataFrame is a distributed collection of data organized into named columns. It provides a good optimization technique.
- The data can be read and written in a variety of structured formats. For example, JSON, **Hive** Tables, and Parquet.
- Using SQL we can query data, both from inside a Spark program and from external tools. The external tool connects through standard database connectors (JDBC/ODBC) to Spark SQL.
- The best way to use Spark SQL is inside a Spark application. This empowers us to load data and query it with SQL. At the same time, we can also combine it with "regular" program code in Python, Java or Scala.

Get Best Scala books to become a master of Scala programming language.

When SQL run from the other programming language the result will be a **Dataset/DataFrame**. The interaction with SQL interface is made using the command line or over JDBC/ODBC.

2.2. Spark SQL DataFrames

There were some **limitations with RDDs.** When working with structured data, there was no inbuilt optimization engine. On the basis of attributes, the developer optimized each **RDD.** Also, there was no provision to handle structured data. The DataFrame in Spark SQL overcomes these limitations of RDD. **Spark DataFrame** is Spark 1.3 release. It is a distributed collection of data ordered into named columns. Concept wise it is equal to the table in a relational database or a data frame in **R**/Python. We can create DataFrame using:

- Structured data files
- Tables in Hive
- External databases
- Using existing RDD

2.3. Spark SQL Datasets

Spark Dataset is an interface added in version Spark 1.6. it is a distributed collection of data. Dataset provides the **benefits of RDDs** along with the benefits of Apache Spark SQL's optimized execution engine. Here an encoder is a concept that does conversion between JVM objects and tabular representation.

A Dataset can be made using JVM objects and after that, it can be manipulated using functional transformations (map, filter etc.). The Dataset API is accessible in *Scala*and *Java*. Dataset API is not supported by Python. But because of the dynamic nature of Python, many benefits of Dataset API are available. The same is the case with **R**. Using a Dataset of rows we represent DataFrame in Scala and Java. Follow this comparison guide to learn the comparison between Java vs Scala.

2.4. Spark Catalyst Optimizer

The *optimizer* used by Spark SQL is **Catalyst optimizer**. It optimizes all the queries written in Spark SQL and DataFrame DSL. The optimizer helps us to run queries much faster than their counter RDD part. This increases the performance of the system.

Spark Catalyst is a library built as a rule-based system. And each rule focusses on the specific optimization. For example, *ConstantFolding* focus on eliminating *constant expression* from the query.

2.5. Uses of Apache Spark SQL

- It executes SQL queries.
- We can read data from existing **Hive installation** using SparkSQL.
- When we run SQL within another programming language we will get the result as Dataset/DataFrame.

2.6. Functions defined by Spark SQL

a. Built-In function

It offers a built-in function to process the column value. We can access the inbuilt function by importing the following command: ***Import org.apache.spark.sql.functions***

b. User Defined Functions(UDFs)

UDF allows you to create the user define functions based on the user defined functions in scala. Refer this guide to learn the features of Scala.

c. Aggregate functions

These operate on a group of rows and calculate a single return value per group.

d. Windowed Aggregates(Windows)

These operate on a group of rows and calculate a single return value for each row in a group.

2.7. Advantages of Spark SQL

In this section of, we will discuss various advantages of Apache Spark SQL-

a. Integrated

Apache Spark SQL mixes SQL queries with Spark programs. With the help of Spark SQL, we can query structured data as a distributed dataset (RDD). We can run SQL queries alongside complex analytic algorithms using tight integration property of Spark SQL.

b. Unified Data Access

Using Spark SQL, we can load and query data from different sources. The Schema-RDDs lets single interface to productively work structured data. For example, Apache Hive tables, parquet files, and JSON files.

c. High compatibility

In Apache Spark SQL, we can run unmodified Hive queries on existing warehouses. It allows full compatibility with existing Hive data, queries and UDFs, by using the Hive fronted and MetaStore.

d. Standard Connectivity

It can connect through JDBC or ODBC. It includes server mode with industry standard JDBC and ODBC connectivity.

e. Scalability

To support mid-query fault tolerance and large jobs, it takes advantage of RDD model. It uses the same engine for interactive and long queries.

f. Performance Optimization

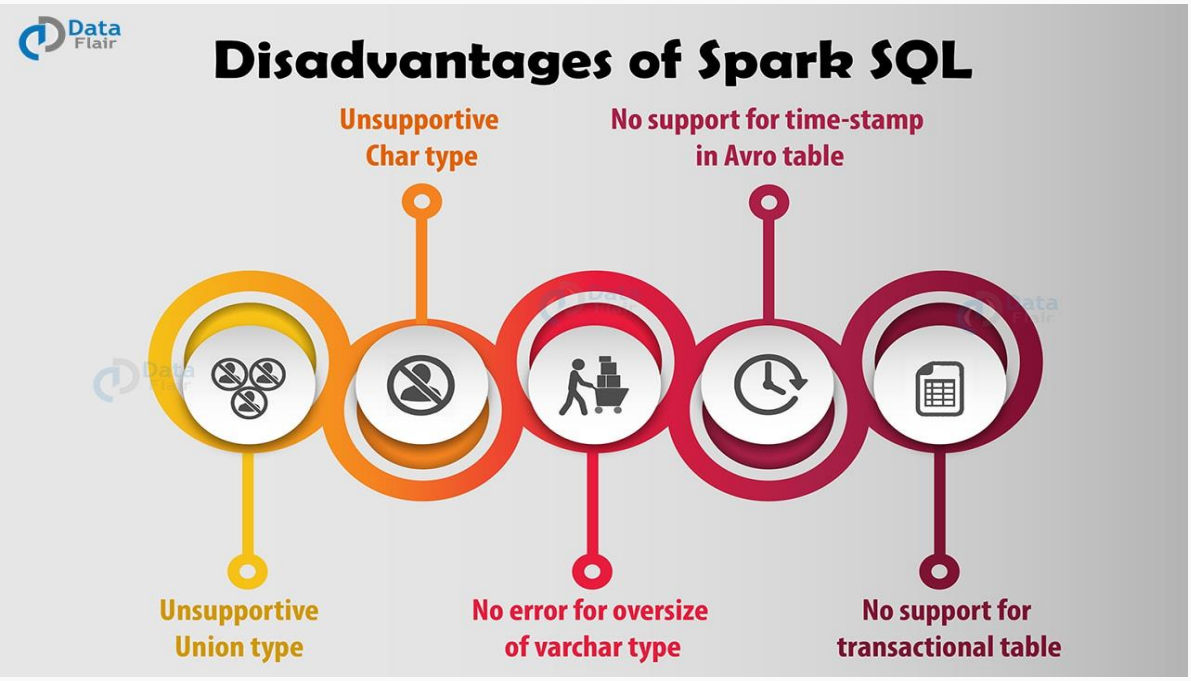
The query optimization engine in Spark SQL converts each SQL query to a logical plan. Further, it converts to many physical execution plans. Among the entire plan, it selects the most optimal physical plan for execution. Read more about Apache Spark performance tuning techniques in detail.

g. For batch processing of Hive tables

We can make use of Spark SQL for fast batch processing of Hive tables.

2.8. Disadvantages of Spark SQL

Apart from features, there are also some disadvantages of Spark SQL. Some of them are listed below-



a. Unsupportive Union type

Using Spark SQL, we cannot create or read a table containing union fields.

b. No error for oversize of varchar type

Even if the inserted value exceeds the size limit, no error will occur. The same data will truncate if read from Hive but not if read from Spark. SparkSQL will consider varchar as a string, meaning there is no size limit.

c. No support for transactional table

Hive transactions are not supported by Spark SQL.

d. Unsupportive Char type

Char type (fixed length strings) are not supported. Like the union, we cannot read or create a table with such fields.

e. No support for time-stamp in Avro table.

3. Conclusion

In conclusion to Spark SQL, it is a module of Apache Spark that analyses the structured data. It provides Scalability, it ensures high compatibility of the system. It has standard connectivity through JDBC or ODBC. Thus, it provides the most natural way to express the Structured Data.

If you have a query related to this blog, so please leave a comment in a section below.

31.SPARK SQL FEATURES

1. Objective

In this document, we will see various shining **Spark SQL** features. There are many features Like Unified Data Access, High Compatibility and many more. We will focus on each feature in detail. But, before learning features of Spark SQL, we will also study brief introduction to Spark SQL.

2. Introduction to Spark SQL

In **Apache Spark**, Spark SQL is a module for working with structured data. Spark SQL supports distributed **in-memory computations** on a huge scale. It divulges the information about the structure of both computations as well as data. To perform extra optimizations, this extra information turns very helpful. We can easily execute SQL queries through it.

In addition, to read data from an existing **Hive installation**, we can use Spark SQL. The results come as **Dataset**/**DataFrame** When SQL run in

another programming language. We can interact with the SQL interface, by using the command-line or over JDBC/ODBC.

The 3 main capabilities of using structured and semi-structured data, by Spark SQL. Such as:

- It grants a DataFrame abstraction in Scala, Java, as well as **Python**. Also, simplifies working with structured datasets. Here, DataFrames are similar to tables in a relational database.
- In various structured formats, Spark SQL can read and write data. For Example Hive Tables, JSON and Parquet.
- We can query the data by using Spark SQL. Both inside a Spark program and from external tools that connect to Spark SQL.

In Spark SQL, developers can switch back and forth between different APIs, as same as Spark. Therefore, it bestows the most natural way to express the given **Transformations**.

3. Spark SQL features

a. Integrated

Integrate is simply defined as combining or merge. Here, Spark SQL queries are integrated with Spark programs. Through Spark SQL we are allowed to query structured data inside Spark programs. This is possible by using SQL or a DataFrame that can be used in Java, Scala.

We can run streaming computation through it. Developers write a batch computation against the DataFrame / Dataset API to run it. After that to run it in a streaming fashion Spark itself increments the computation. Developers leverage the advantage of it that they don't have to manage state, failures on own. Even no need keep the application in sync with batch jobs. Despite, the streaming job always gives the same answer as a batch job on the same data.

b. Unified Data Access

To access a variety of data sources DataFrames and SQL support a common way. Data Sources like Hive, Avro, Parquet, ORC, JSON, as well as JDBC. It helps to join the data from these sources. To accommodate all the existing users into Spark SQL, it turns out to be very helpful.

c. High compatibility

We are allowed to run unmodified Hive queries on existing warehouses in Spark SQL. With existing Hive data, queries and UDFs, Spark SQL offers full compatibility. Also, rewrites the MetaStore and Hive frontend.

d. Standard Connectivity

We can easily connect Spark SQL through JDBC or ODBC. For connectivity for business intelligence tools, Both turned as industry norms. Also, includes industry standard JDBC and ODBC connectivity with server mode.

e. Scalability

It takes advantage of RDD model, to support large jobs and mid-query fault tolerance. For interactive as well as long queries, it uses the same engine.

f. Performance Optimization

In Spark SQL, query optimization engine converts each SQL query into a logical plan. Afterwards, it converts to many physical execution plans. At the time of execution, it selects the most optimal physical plan, among the entire plan. It ensures fast execution of HIVE queries.

g. For batch processing of Hive tables

While working with Hive tables, we can use Spark SQL for Batch Processing in them.

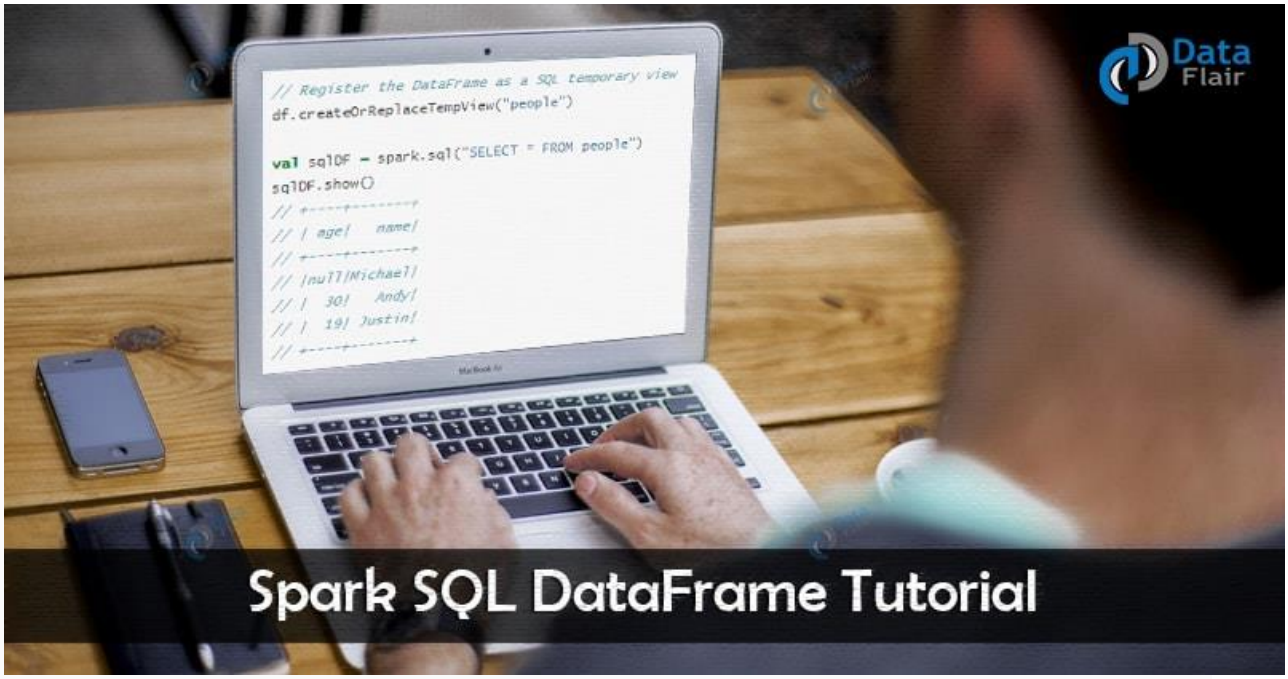
4. Conclusion

Hence, we have seen all Spark SQL features in detail. As a result, we have learned, Spark SQL is a module of Spark that analyses structured data. Basically, it offers scalability and ensures high compatibility of the system. Also, allow standard connectivity through JDBC or ODBC. Therefore, it bestows the most natural way to express the structured data. Moreover, it enhances its working efficiency with above-mentioned Spark SQL features.

32.SPARK SQL DATAFRAME TUTORIAL – AN INTRODUCTION TO DATAFRAME

1. Objective

In this **Spark SQL DataFrame** tutorial, we will learn what is DataFrame in **Apache Spark** and the need of Spark Dataframe. The tutorial covers the limitation of Spark RDD and How DataFrame overcomes those limitations. How to create DataFrame in Spark, Various Features of DataFrame like **Custom** Memory Management, Optimized Execution plan, and its limitations are also covers in this Spark tutorial.



2. Introduction to Spark SQL DataFrame

DataFrame appeared in Spark Release 1.3.0. We can term DataFrame as Dataset organized into named columns. DataFrames are similar to the table in a relational database or data frame in R /Python. It can be said as a relational table with good optimization technique.

The idea behind DataFrame is it allows processing of a large amount of structured data. DataFrame contains rows with Schema. The **schema** is the illustration of the structure of data.

DataFrame in Apache Spark prevails over RDD but contains the features of RDD as well. The features common to RDD and DataFrame are **immutability**, **in-memory**, resilient, distributed computing capability. It allows the user to impose the structure onto a distributed collection of data. Thus provides higher level abstraction.

We can build DataFrame from different data sources. For Example structured data file, tables in Hive, external databases or existing RDDs. The Application Programming Interface (APIs) of DataFrame is available in various languages. Examples include Scala, Java, Python, and R.

Both in Scala and Java, we represent DataFrame as Dataset of rows. In the Scala API, DataFrames are type alias of Dataset[Row]. In Java API, the user uses Dataset<Row> to represent a DataFrame.

3. Why DataFrame?

DataFrame is one step ahead of **RDD.** Since it provides memory management and optimized execution plan.

a. Custom Memory Management: This is also known as Project **Tungsten.** A lot of memory is saved as the data is stored in off-heap memory in binary format. Apart from this, there is no Garbage Collection overhead. Expensive Java serialization is also avoided. Since the data is stored in binary format and the schema of memory is known.

b. Optimized Execution plan: This is also known as the **query optimizer**. Using this, an optimized execution plan is created for the execution of a query. Once the optimized plan is created final execution takes place on RDDs of Spark.

4. Features of Apache Spark DataFrame

Some of the limitations of Spark RDD were-

- It does not have any built-in optimization engine.
- There is no provision to handle structured data.

Thus, to overcome these limitations the picture of DataFrame came into existence. Some of the key features of DataFrame in Spark are:

i. DataFrame is a distributed collection of data organized in named column. It is equivalent to the table in RDBMS.

ii. It can deal with both structured and unstructured data formats. For Example Avro, CSV, elastic search, and Cassandra. It also deals with storage systems HDFS, HIVE tables, MySQL, etc.

iii. Catalyst supports optimization. It has general libraries to represent trees. DataFrame uses **Catalyst tree transformation** in four phases:

- Analyze logical plan to solve references
- Logical plan optimization
- Physical planning
- Code generation to compile part of a query to Java bytecode.

You can refer this guide to learn Spark SQL optimization phases in detail.

iv. The DataFrame API's are available in various programming languages. For example Java, Scala, Python, and R.

v. It provides Hive compatibility. We can run unmodified Hive queries on existing Hive warehouse.

vi. It can scale from kilobytes of data on the single laptop to petabytes of data on a large cluster.

vii. DataFrame provides easy integration with Big data tools and framework via **Spark core**.

5. Creating DataFrames in Apache Spark

To all the functionality of Spark, **SparkSession** class is the entry point. For the creation of basic SparkSession just use

SparkSession.builder()

Using Spark Session, an application can create DataFrame from an existing RDD, Hive table or from Spark data sources. **Spark SQL** can operate on the variety of data sources using DataFrame interface. Using Spark SQL DataFrame we can create a temporary view. In the temporary view of dataframe, we can run the SQL query on the data.

6. Limitations of DataFrame in Spark

- Spark SQL DataFrame API does not have provision for **compile time type safety**. So, if the structure is unknown, we cannot manipulate the data.
- Once the domain object is converted into dataframe, the regeneration of domain object is not possible.

7. Conclusion

Hence, DataFrame API in Spark SQL improves the performance and scalability of Spark. It avoids the garbage-collection cost of constructing individual objects for each row in the dataset.

The Spark DataFrame API is different from the RDD API because it is an API for building a relational query plan that Spark's Catalyst optimizer can then execute. This DataFrame API is good for developers who are familiar with building query plans. It is not good for the majority of developers.

To Play with DataFrame in spark, install Apache Spark in Standalone mode and Spark installation in the multi-node cluster.

33.SPARK DATASET TUTORIAL – INTRODUCTION TO APACHE SPARK DATASET

1. Objective

In This blog on **Apache Spark** dataset, you can read all about what is dataset in Spark. Why DataSet needed, what is encoder and what is their significance in the dataset? You will get the answer to all these questions in this blog. We will also cover the features of the dataset in Apache Spark and How to create a dataset in this Spark tutorial.

2. Introduction to Spark Dataset

Dataset is a data structure in **SparkSQL** which is strongly typed and is a map to a relational schema. It represents structured queries with encoders. It is an extension to dataframe API. Spark Dataset provides both type safety and object-oriented programming interface. We encounter the release of the dataset in Spark 1.6.

The **encoder** is primary concept in *serialization* and *deserialization* (**SerDe**) framework in Spark SQL. Encoders translate between JVM objects and Spark's internal binary format. Spark has built-in encoders which are very advanced. They generate bytecode to interact with **off-heap** data.

An encoder provides on-demand access to individual attributes without having to de-serialize an entire object. To make input output time and space efficient, Spark SQL uses SerDe framework. Since encoder knows the schema of record, it can achieve serialization and deserialization.

Spark Dataset is structured and lazy query expression that triggers on the **action**. Internally dataset represents logical plan. The **logical plan** tells the computational query that we need to produce the data. the logical plan is a base catalyst query plan for the logical operator to form a logical query plan. When we analyze this and resolve we can form a physical query plan.

Dataset clubs the features of **RDD** and **DataFrame.** It provides:

- The convenience of RDD.
- Performance optimization of DataFrame.
- Static type-safety of **Scala**.

Thus, Datasets provides a more functional programming interface to work with structured data.

3. Need of Dataset in Spark

To overcome the limitations of RDD and Dataframe, Dataset emerged. In DataFrame, there was no provision for **compile time type safety**. Data cannot be altered without knowing its structure. In RDD there was no automatic optimization. So for optimization, we do it manually when needed.

4. Features of Dataset in Spark

After having introduction to dataSet, let's now discuss various features of Spark Dataset-

a. Optimized Query

Dataset in Spark provides Optimized query using **Catalyst Query Optimizer** and **Tungsten**. **Catalyst Query Optimizer** is an execution-agnostic framework. It represents and manipulates a data-flow graph. Data flow graph is a tree of expressions and relational operators. By optimizing the Spark job Tungsten improves the execution. Tungsten emphasizes on the hardware architecture of the platform on which Apache Spark runs.

b. Analysis at compile time

Using Dataset we can check syntax and analysis at compile time. It is not possible using Dataframe, RDDs or regular SQL queries.

c. Persistent Storage

Spark Datasets are both serializable and Queryable. Thus, we can save it to persistent storage.

d. Inter-convertible

We can convert the Type-safe dataset to an "untyped" DataFrame. To do this task Datasetholder provide three methods for conversion
from *Seq[T]* or *RDD[T]* types to *Dataset[T]:*

- ***toDS(): Dataset[T]***
- ***toDF(): DataFrame***
- ***toDF(colNames: String*): DataFrame***

e. Faster Computation

The implementation of Dataset is much faster than the RDD implementation. Thus increases the performance of the system. For same performance using the RDD, the user manually considers how to express computation that parallelizes optimally.

f. Less Memory Consumption

While caching, it creates a more optimal layout. Since Spark knows the structure of data in the dataset.

g. Single API for Java and Scala

It provides a single interface for **Java** and **Scala**. This unification ensures we can use Scala interface, code examples from both languages. It also reduces the burden of libraries. As libraries have no longer to deal with two different type of inputs.

5. Creating Dataset

To create Dataset we need:

a. SparkSession

SparkSession is the entry point to the SparkSQL. It is a very first object that we create while developing Spark SQL applications using fully typed Dataset data abstractions. Using **SparkSession.**Builder, we can create an instance of SparkSession. And can stop SparkSession using stop method (***spark.stop***).

b. QueryExecution

We represent structured query execution pipeline of the dataset using*QueryExecution*. To access QueryExecution of a Dataset use QueryExecution attribute. By executing a logical plan in Spark Session we get QueryExecution.

executePlan(plan: LogicalPlan): QueryExecution

executePlan executes the input LogicalPlan to produce a QueryExecution in the current SparkSession.

c. Encoder

An *encoder* provides conversion between tabular representation and JVM objects. With the help of encoder, we serialize the object. Encoder serializes objects for processing or transmitting over the network encoders.

6. Conclusion

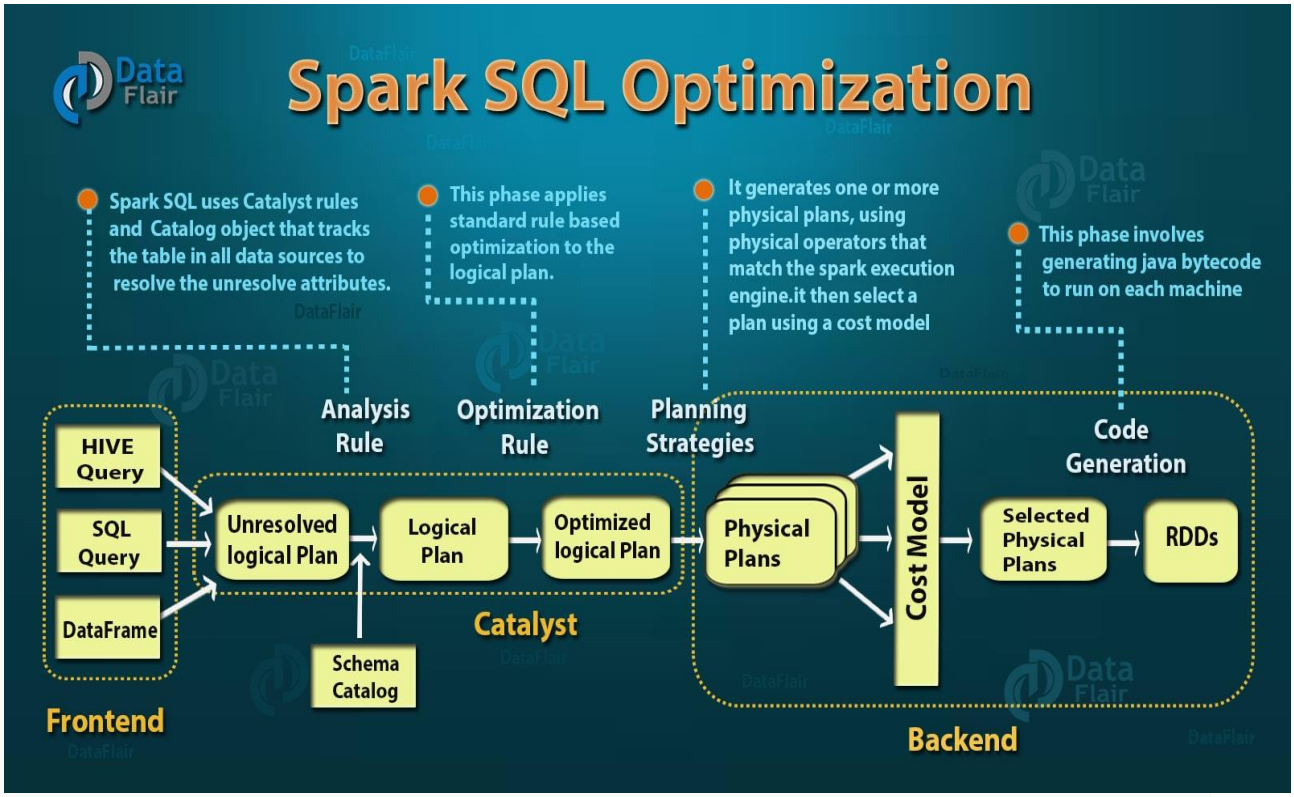
In conclusion to Dataset, we can say it is strongly typed data structure in Apache Spark. It represents structured queries. It fuses together the functionality of RDD and DataFrame. We can generate the optimized query using Dataset. Dataset lessens the memory consumption and provides a single API for both Java and Scala.

If you like this post and feel that I have missed any point, so, do let me

34.spark sql optimization – understanding the catalyst optimizer

1. Objective

The goal of this **Apache Spark** tutorial is to describe the **Spark SQL Optimization framework** and how it allows developers to express complex query transformations in very few lines of code. we will also describe How **Spark SQL** improves the execution time of queries by greatly improving its query optimization capabilities. We will also cover what is an optimization, why catalyst optimizer, what are its fundamental units of working and the phases of Spark execution flow in this tutorial.



2. Introduction to Apache Spark SQL Optimization

"The term **optimization** refers to a process in which a system is modified in such a way that it work more efficiently or it uses fewer resources."

Spark SQL is the most technically involved component of Apache Spark. Spark SQL deals with both SQL queries and DataFrame API. In the depth of Spark SQL there lies a **catalyst optimizer**. Catalyst optimization allows some advanced programming language features that allow you to build an extensible query optimizer.

A new extensible optimizer called Catalyst emerged to implement Spark SQL. This optimizer is based on functional programming construct in **Scala**.

Catalyst Optimizer supports both **rule-based** and **cost-based** optimization. In *rule-based optimization* the rule based optimizer use set of rule to determine how to execute the query. While the *cost based optimization* finds the most suitable way to carry out SQL statement. In cost-based optimization, multiple plans are generated using rules and then their cost is computed.

3. What is the need of Catalyst Optimizer?

There are two purposes behind Catalyst's extensible design:

- We want to add the easy solution to tackle various problems with **Bigdata** like a problem with semi-structured data and advanced data analytics.
- We want an easy way such that external developers can extend the optimizer.

4. Fundamentals of Catalyst Optimizer

Catalyst optimizer makes use of standard features of Scala programming like pattern matching. In the depth, Catalyst contains the **tree** and the set of **rules** to manipulate the tree. There are specific libraries to process relational queries. There are various rule sets which handle different phases of query execution like *analysis*, *query optimization, physical planning,* and *code generation* to compile parts of queries to Java bytecode. Let's discuss the tree and rules is detail-

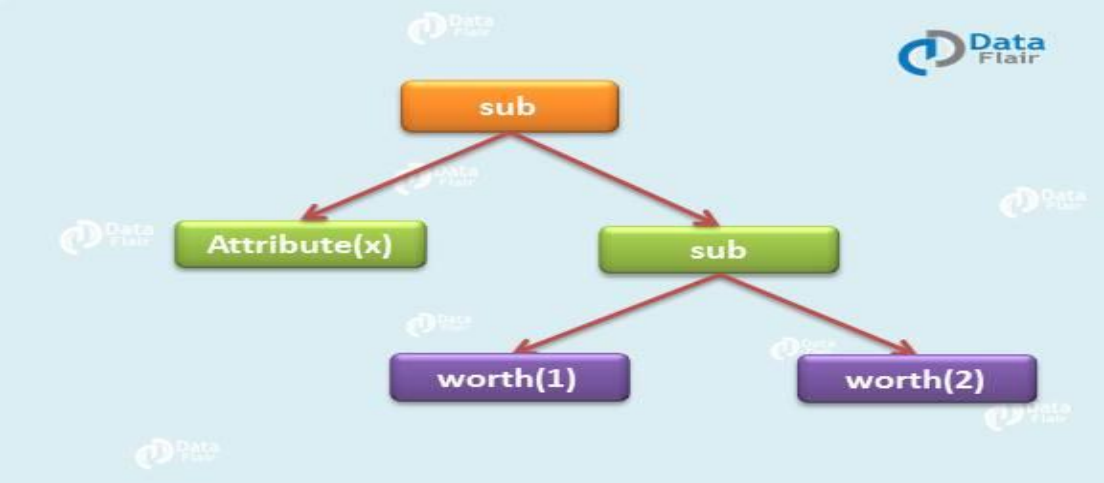
4.1. Trees

A tree is the main data type in the catalyst. A tree contains node object. For each node, there is a node. A node can have one or more children. New nodes are defined as subclasses of TreeNode class. These objects are immutable in nature. The objects can be manipulated using functional transformation. See RDD Transformations and Actions Guide for more details about Functional transformations.

For example, if we have three node classes: **worth**, **attribute**, and **sub** in which-

- worth(value: Int): a constant value
- attribute(name: String)
- sub (left: TreeNode, right: TreeNode): subtraction of two expressions.

Then a tree will look like-



4.2. Rules

We can manipulate tree using **rules**. We can define rules as a function from one tree to another tree. With rule we can run arbitrary code on input tree, the common approach to use a pattern matching function and replace subtree with a specific structure. In a tree with the help of **transform function**, we can recursively apply pattern matching on all the node of a tree. We get the pattern that matches each pattern to a result.

For example-

tree.transform {case Sub(worth(c1),worth(c2)) => worth(c1+c2) }

The expression that is passed during pattern matching to transform is a partial function. By partial function, it means it only needs to match to a subset of all possible input trees. Catalyst will see, to which part of a tree the given rule applies, and will automatically skip over the tree that does not match. With the same transform call, the rule can match multiple patterns.

For example-

tree.transform {

case Sub(worth(c1), worth(c2)) =>worth(c1-c2)

case Sub(left , worth(0)) => left

case Sub(worth(0), right) => right

}

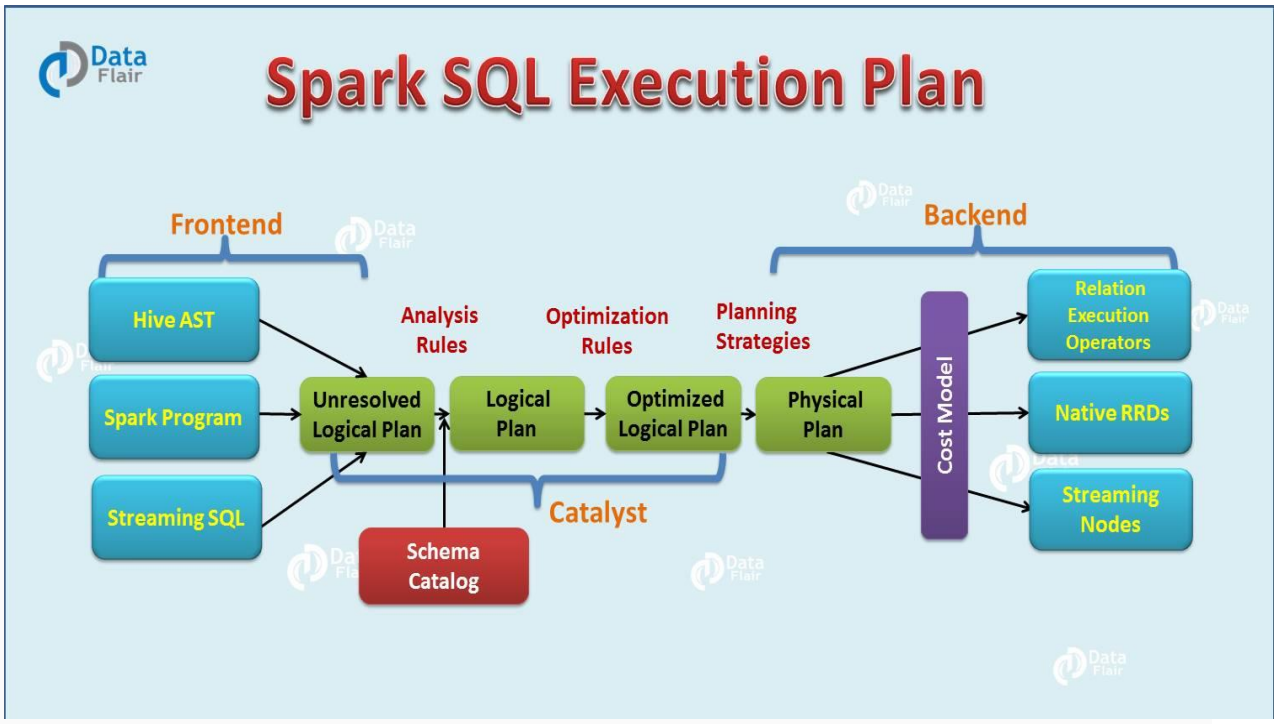
To fully transform a tree, rule may be needed to execute multiple time.

Catalyst work by grouping rules into batches and these batches are executed until a fixed point is achieved. Fixed point is a point after which tree stops changing even after applying rules.

5. Spark SQL Execution Plan

After the detailed introduction of Apache Spark SQL catalyst optimizer, now we will discuss the Spark SQL query execution phases. In four phases we use Catalyst's general tree transformation framework:

- Analysis
- Logical Optimization
- Physical planning
- Code generation



5.1. Analysis

Spark SQL Optimization starts from relation to be computed. It is computed either from **abstract syntax tree (AST)** returned by **SQL parser** or dataframe object created using API. Both may contain unresolved attribute references or relations. By unresolved attribute, it means we don't know its type or have not matched it to an input table. Spark SQL make use of Catalyst rules and a Catalog object that track data in all data sources to resolve these attributes. It starts by creating an unresolved logical plan, and then apply the following steps:

- Search relation BY NAME FROM CATALOG.
- Map the name attribute, for example, col, to the input provided given operator's children.
- Determine which attributes match to the same value to give them unique ID.

- Propagate and push type through expressions.

5.2. Logical Optimization

In this phase of Spark SQL optimization, the standard rule-based optimization is applied to the logical plan. It includes **constant folding**, **predicate pushdown**, **projection pruning** and other rules. It became extremely easy to add a rule for various situations.

5.3. Physical Planning

There are about 500 lines of code in the physical planning rules. In this phase, one or more physical plan is formed from the logical plan, using physical operator matches the Spark execution engine. And it selects the plan using the cost model. It uses Cost-based optimization only to select join algorithms. For small relation SQL uses broadcast join, the framework supports broader use of cost-based optimization. It can estimate the cost recursively for the whole tree using the rule.

Rule-based physical optimization, such as pipelining projections or filters into one Spark *map* Operation is also carried out by the physical planner. Apart from this, it can also push operations from the logical plan into data sources that support predicate or projection pushdown.

5.4. Code Generation

The final phase of Spark SQL optimization is code generation. It involves generating Java bytecode to run on each machine. Catalyst uses the special feature of Scala language, "**Quasiquotes**" to make code generation easier because it is very tough to build code generation engines. Quasiquotes lets the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. With the help of a catalyst, we can transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

6. Conclusion

Hence, Spark SQL optimization enhances the productivity of developers and the performance of the queries that they write. A good query optimizer automatically rewrites relational queries to execute more efficiently, using techniques such as filtering data early, utilizing available indexes, and even ensuring different data sources are joined in the most efficient order.

By performing these transformations, the optimizer improves the execution times of relational queries and frees the developer from focusing on the semantics of their application instead of its performance.

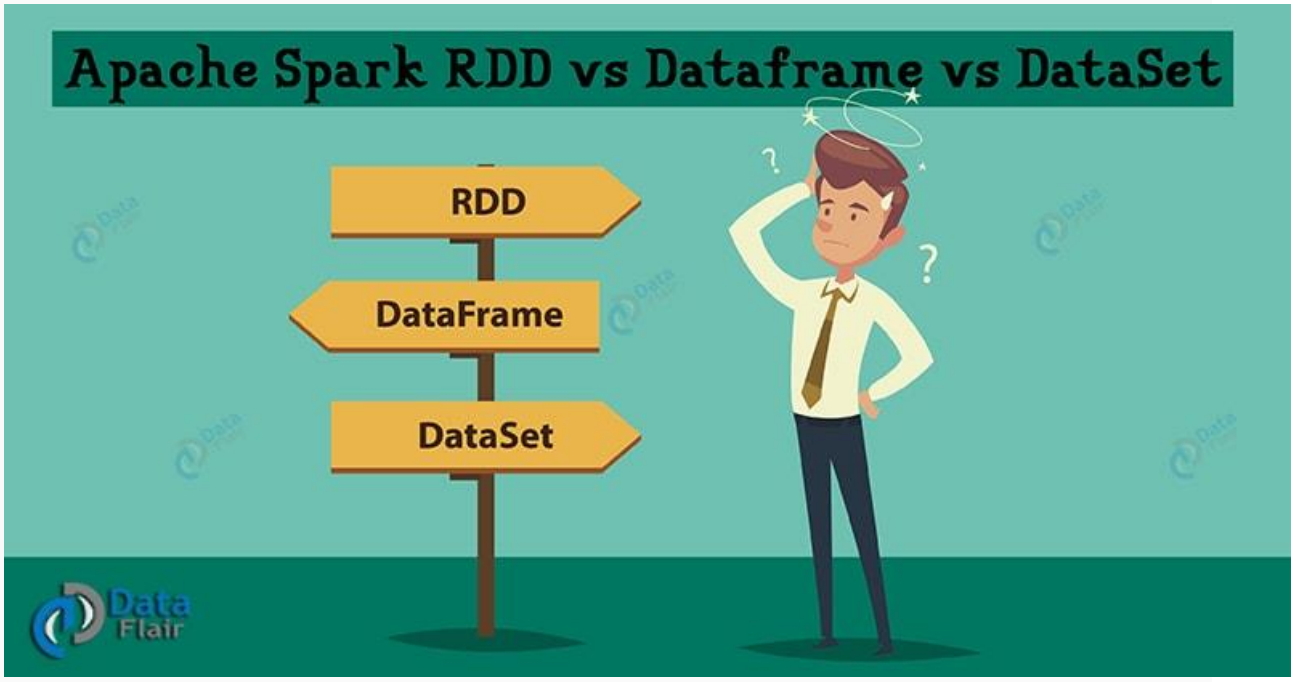
Catalyst makes use of Scala's powerful features such as pattern matching and runtime metaprogramming to allow developers to concisely specify complex relational optimizations.

35.Apache Spark RDD vs DataFrame vs DataSet

1. Objective

This Spark tutorial will provide you the detailed feature wise comparison between**Apache Spark** RDD vs DataFrame vs DataSet. We will cover the brief introduction of Spark APIs i.e. RDD, DataFrame and Dataset, Differences between these Spark API based on various features. For example, Data Representation, Immutability, and Interoperability etc. We will also illustrate, where to use RDD, DataFrame API, and Dataset API of Spark.

Learn easy steps to **Install Apache Spark on the single node** and on **Multi-node cluster**.



2. Apache Spark APIs – RDD, DataFrame, and DataSet

Before starting the comparison between Spark RDD vs DataFrame vs Dataset, let us see RDDs, DataFrame and Datasets in Spark:

- **Spark RDD APIs –** An RDD stands for Resilient Distributed Datasets. It is Read-only partition collection of records. RDD is the fundamental data structure of Spark. It allows a programmer to perform in-memory computations on large clusters in a **fault-tolerant** manner. Thus, speed up the task. Follow this link to**learn Spark RDD in great detail.**
- **Spark Dataframe APIs –** Unlike an RDD, data organized into named columns. For example a table in a relational database. It is an immutable distributed collection of data. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. Follow this link to **learn Spark DataFrame in detail.**
- **Spark Dataset APIs –** Datasets in Apache Spark are an extension of DataFrame API which provides type-safe, object-oriented programming interface. Dataset takes advantage of Spark's Catalyst optimizer by exposing expressions and data fields to a query planner. Follow this link to **learn Spark DataSet in detail.**

3. RDD vs Dataframe vs DataSet in Apache Spark

Let us now learn the feature wise difference between RDD vs DataFrame vs DataSet API in Spark:

3.1. Spark Release

- **RDD –** The **RDD** APIs have been on Spark since the 1.0 release.
- **DataFrames –** Spark introduced DataFrames in Spark 1.3 release.
- **DataSet –** Spark introduced Dataset in Spark 1.6 release.

3.2. Data Representation

- **RDD –** RDD is a distributed collection of data elements spread across many machines in the cluster. RDDs are a set of Java or **Scala** objects representing data.
- **DataFrame –** A DataFrame is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database.
- **DataSet –** It is an extension of DataFrame API that provides the functionality of – type-safe, object-oriented programming interface of the RDD API and performance benefits of the Catalyst query optimizer and off heap storage mechanism of a DataFrame API.

3.3. Data Formats

- **RDD –** It can easily and efficiently process data which is structured as well as unstructured. But like Dataframe and DataSets, RDD does not infer the schema of the ingested data and requires the user to specify it.
- **DataFrame –** It can process structured and unstructured data efficiently. It organizes the data in the named column. DataFrames allow the Spark to manage schema.
- **DataSet –** It also efficiently processes structured and unstructured data. It represents data in the form of JVM objects of row or a collection of row object. Which is represented in tabular forms through encoders.

3.4. Data Sources API

- **RDD –** Data source API allows that an RDD could come from any data source e.g. text file, a database via JDBC etc. and easily handle data with no predefined structure.
- **DataFrame –** Data source API allows Data processing in different formats (AVRO, CSV, JSON, and storage system **HDFS**, **HIVE** tables, MySQL). It can read and write from various data sources that are mentioned above.
- **DataSet –** Dataset API of spark also support data from different sources.

3.5. Immutability and Interoperability

- **RDD –** RDDs contains the collection of records which are partitioned. The basic unit of parallelism in an RDD is called partition. Each partition is one logical division of data which is immutable and created through some transformation on existing partitions. Immutability helps to achieve consistency in computations. We can move from RDD to DataFrame (If RDD is in tabular format) by **toDF()**method or we can do the reverse by the **.rdd** method. Learn various **RDD Transformations and Actions APIs with examples**.
- **DataFrame –** After transforming into DataFrame one cannot regenerate a domain object. For example, if you generate testDF from testRDD, then you won't be able to recover the original RDD of the test class.
- **DataSet –** It overcomes the limitation of DataFrame to regenerate the RDD from Dataframe.
  Datasets allow you to convert your existing RDD and DataFrames into Datasets.

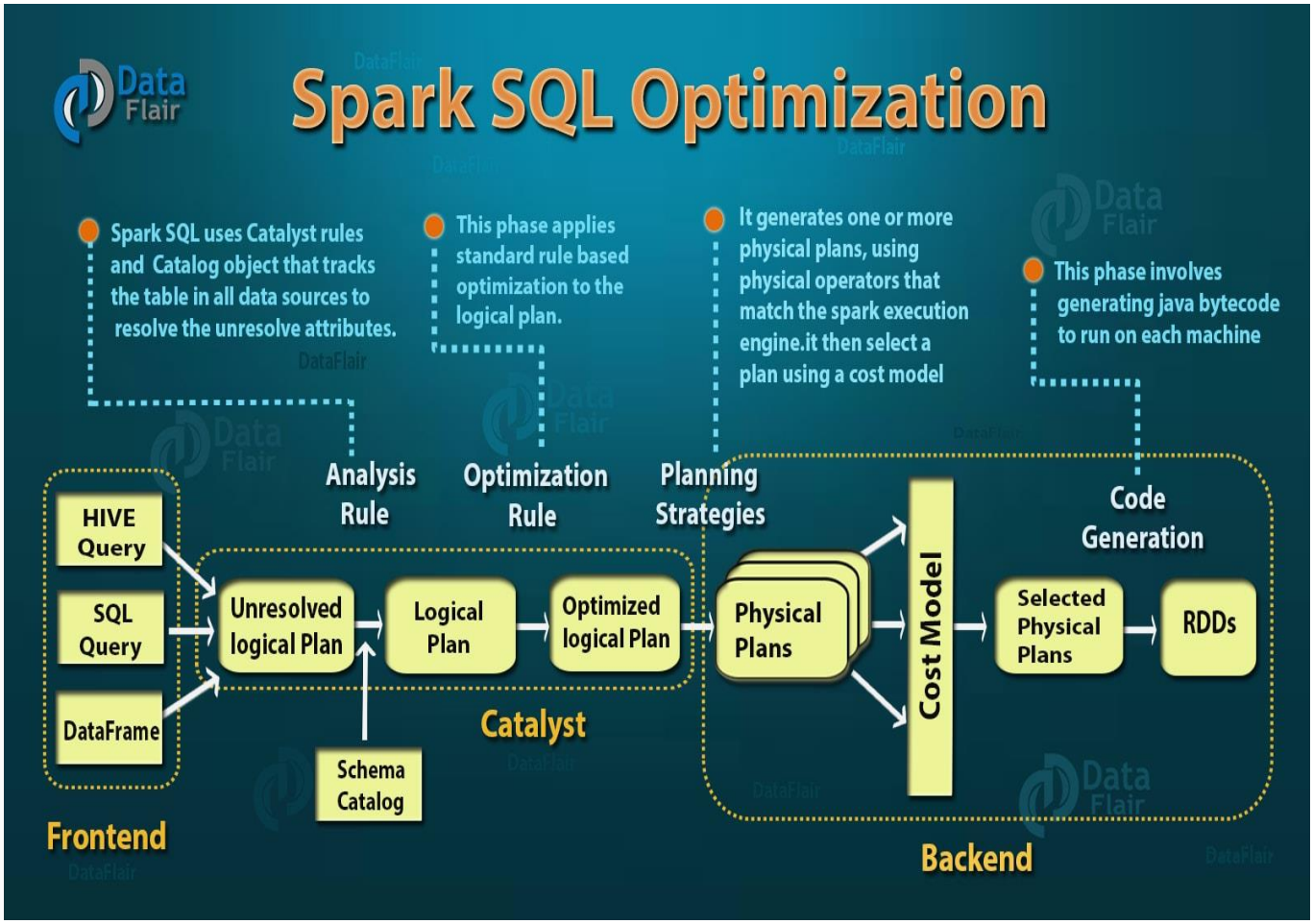
3.6. Compile-time type safety

- **RDD –** RDD provides a familiar object-oriented programming style with compile-time type safety.
- **DataFrame –** If you are trying to access the column which does not exist in the table in such case Dataframe APIs does not support compile-time error. It detects attribute error only at runtime.
- **DataSet –** It provides compile-time type safety.

Learn: Apache Spark vs. Hadoop MapReduce

3.7. Optimization

- **RDD –** No inbuilt optimization engine is available in RDD. When working with structured data, RDDs cannot take advantages of sparks advance optimizers. For example, catalyst optimizer and Tungsten execution engine. Developers optimise each RDD on the basis of its attributes.
- **DataFrame –** Optimization takes place using catalyst optimizer. Dataframes use catalyst tree transformation framework in four phases: a) Analyzing a logical plan to resolve references. b) Logical plan optimization. c) Physical planning. d) Code generation to compile parts of the query to Java bytecode. The brief overview of optimization phase is also given in the below figure:



- **Dataset –** It includes the concept of Dataframe Catalyst optimizer for optimizing query plan.

3.8. Serialization

- **RDD –** Whenever **Spark** needs to distribute the data within the cluster or write the data to disk, it does so use Java serialization. The overhead of serializing individual Java and Scala objects is expensive and requires sending both data and structure between nodes.
- **DataFrame –** Spark DataFrame Can serialize the data into off-heap storage (**in memory**) in binary format and then perform many transformations directly on this off heap memory because spark understands the schema. There is no need to use java serialization to encode the data. It provides a Tungsten physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.
- **DataSet –** When it comes to serializing data, the Dataset API in Spark has the concept of an encoder which handles conversion between JVM objects to tabular representation. It stores tabular representation using

spark internal Tungsten binary format. Dataset allows performing the operation on serialized data and improving memory use. It allows on-demand access to individual attribute without desterilizing the entire object.

3.9. Garbage Collection

- **RDD –** There is overhead for garbage collection that results from creating and destroying individual objects.
- **DataFrame –** Avoids the garbage collection costs in constructing individual objects for each row in the dataset.
- **DataSet –** There is also no need for the garbage collector to destroy object because serialization
  takes place through Tungsten. That uses off heap data serialization.

Learn: Apache Spark Terminologies and Concepts You Must Know

3.10. Efficiency/Memory use

- **RDD –** Efficiency is decreased when serialization is performed individually on a java and scala object which takes lots of time.
- **DataFrame –** Use of off heap memory for serialization reduces the overhead. It generates byte code dynamically so that many operations can be performed on that serialized data. No need for deserialization for small operations.
- **DataSet –** It allows performing an operation on serialized data and improving memory use. Thus it allows on-demand access to individual attribute without deserializing the entire object.

3.11. Lazy Evolution

- **RDD –** Spark evaluates RDDs lazily. They do not compute their result right away. Instead, they just remember the transformation applied to some base data set. Spark compute Transformations only when an action needs a result to sent to the driver program. Refer this guide if you are new to the **Lazy Evaluation feature of Spark.**

- **DataFrame –** Spark evaluates DataFrame lazily, that means computation happens only when action appears (like display result, save output).
- **DataSet –** It also evaluates lazily as RDD and Dataset.

3.12. Programming Language Support

- **RDD –** RDD APIs are available in **Java, Scala, Python,** and **R** languages. Hence, this feature provides flexibility to the developers.
- **DataFrame –** It also has APIs in the different languages like Java, Python, Scala, and R.
- **DataSet –** Dataset APIs is currently only available in Scala and Java. Spark version 2.1.1 does not support Python and R.

Get the **Best Books of Scala** and **R** to become a master.

3.13. Schema Projection

- **RDD –** In RDD APIs use schema projection is used explicitly. Hence, we need to define the schema (manually).
- **DataFrame –** Auto-discovering the schema from the files and exposing them as tables through the **Hive Meta store**. We did this to connect standard SQL clients to our engine. And explore our dataset without defining the schema of our files.
- **DataSet –** Auto discover the schema of the files because of using **Spark SQL**engine.

3.14. Aggregation

- **RDD –** RDD API is slower to perform simple grouping and aggregation operations.
- **DataFrame –** DataFrame API is very easy to use. It is faster for exploratory analysis, creating aggregated statistics on large data sets.
- **DataSet –** In Dataset it is faster to perform aggregation operation on plenty of data sets.

Learn: Spark Shell Commands to Interact with Spark-Scala

3.15. Usage Area

RDD-

- You can use RDDs When you want low-level transformation and actions on your data set.
- Use RDDs When you need high-level abstractions.

DataFrame and DataSet-

- One can use both DataFrame and dataset API when we need a high level of abstraction.
- For unstructured data, such as media streams or streams of text.
- You can use both Data Frames or Dataset when you need domain specific APIs.
- When you want to manipulate your data with functional programming constructs than domain specific expression.
- We can use either datasets or DataFrame in the high-level expression. For example, filter, maps, aggregation, sum, **SQL** queries, and columnar access.
- When you do not care about imposing a schema, such as columnar format while processing or accessing data attributes by name or column.
- in addition, If we want a higher degree of type safety at compile time.

4. Conclusion

Hence, from the comparison between RDD vs DataFrame vs Dataset, it is clear when to use RDD or DataFrame and/or Dataset.

As a result, RDD offers low-level functionality and control. The DataFrame and Dataset allow custom view and structure. It offers high-level domain-specific operations, saves space, and executes at high speed. Select one out

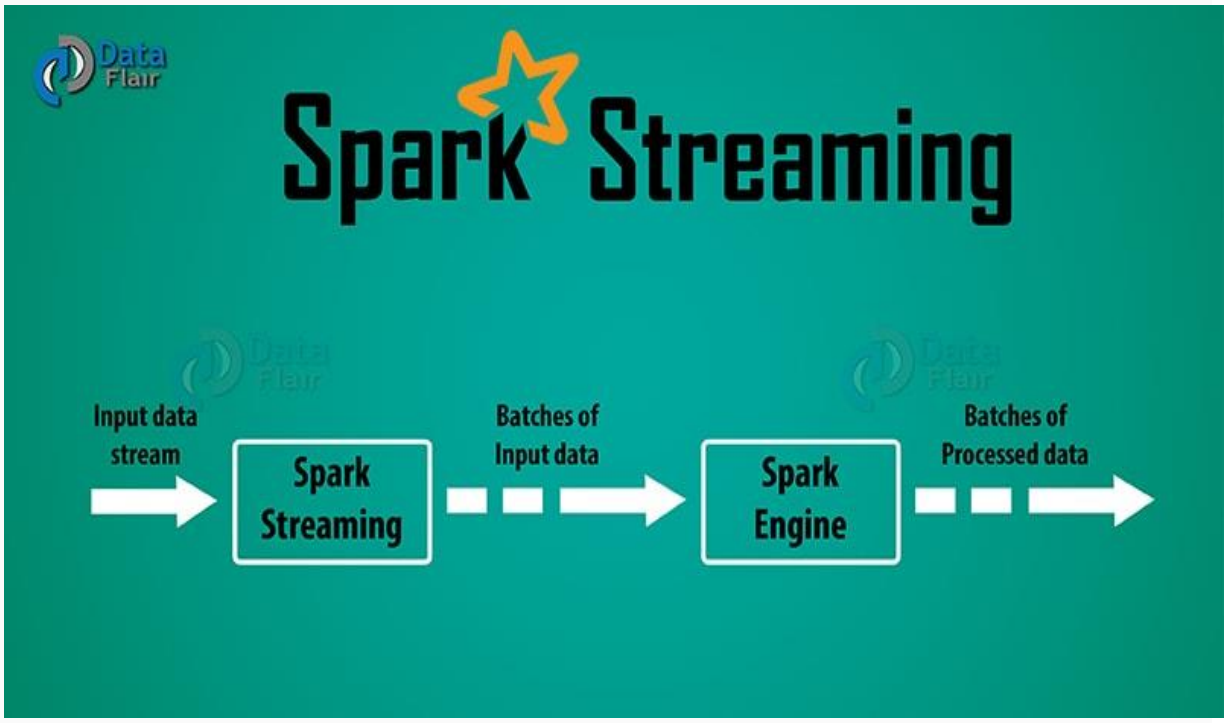
of DataFrames and/or Dataset or RDDs APIs, that meets your needs and play with Spark.

If you like this post about RDD vs Dataframe vs DataSet so do let me know by leaving a comment.

35.spark streaming tutorial for beginners

1. Objective

Through this Spark Streaming tutorial, you will learn basics of Apache Spark Streaming, what is the need of streaming in **Apache Spark**, Streaming in Spark architecture, how streaming works in Spark. You will also understand what are the Spark streaming sources and various Streaming Operations in Spark, Advantages of Apache Spark Streaming over **Big Data** Hadoop and Storm.



2. What is Apache Spark Streaming?

A data stream is an unbounded sequence of data arriving continuously. Streaming divides continuously flowing input data into discrete units for further processing. Stream processing is low latency processing and analyzing of streaming data.

Spark Streaming was added to Apache Spark in 2013, an extension of the core Spark API that provides scalable, high-throughput and fault-tolerant

stream processing of live data streams. Data ingestion can be done from many sources like Kafka, **Apache Flume**, Amazon Kinesis or TCP sockets and processing can be done using complex algorithms that are expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases and live dashboards.

Its internal working is as follows. Live input data streams is received and divided into batches by Spark streaming, these batches are then processed by the Spark engine to generate the final stream of results in batches.

Its key abstraction is Apache Spark **Discretized Stream** or, in short, a **Spark DStream**, which represents a stream of data divided into small batches. DStreams are built on Spark **RDDs**, Spark's core data abstraction. This allows Streaming in Spark to seamlessly integrate with any other Apache Spark components like Spark MLlib and **Spark SQL**.

3. Need for Streaming in Apache Spark

To process the data, most traditional stream processing systems are designed with a continuous operator model, which works as follows:

- Streaming data is received from data sources (e.g. live logs, system telemetry data, IoT device data, etc.) into some data ingestion system like Apache Kafka, Amazon Kinesis, etc.
- The data is then processed in parallel on a cluster.
- Results are given to downstream systems like **HBase**, Cassandra, Kafka, etc.

There is a set of worker nodes, each of which runs one or more continuous operators. Each continuous operator processes the streaming data one record at a time and forwards the records to other operators in the pipeline.

Data is received from ingestion systems via Source operators and given as output to downstream systems via sink operators.

Continuous operators are a simple and natural model. However, this traditional architecture has also met some challenges with today's trend towards larger scale and more complex real-time analytics:-

a) Fast Failure and Straggler Recovery

In real time, the system must be able to fastly and automatically recover from failures and stragglers to provide results which is challenging in traditional systems due to the static allocation of continuous operators to worker nodes.

b) Load Balancing

In a continuous operator system, uneven allocation of the processing load between the workers can cause bottlenecks. The system needs to be able to dynamically adapt the resource allocation based on the workload.

c) Unification of Streaming, Batch and Interactive Workloads

In many use cases, it is also attractive to query the streaming data interactively, or to combine it with static datasets (e.g. pre-computed models). This is hard in continuous operator systems which are not designed to new operators for ad-hoc queries. This requires a single engine that can combine batch, streaming and interactive queries.

d) Advanced Analytics with Machine learning and SQL Queries

Complex workloads require continuously learning and updating data models, or even querying the streaming data with SQL queries. Having a common abstraction across these analytic tasks makes the developer's job much easier.

4. Why Streaming in Spark?

Batch processing systems like **Apache Hadoop** have high latency that is not suitable for near real time processing requirements. Processing of a record is guaranteed by Storm if it hasn't been processed, but this can lead to inconsistency as repetition of record processing might be there. The state is lost if a node running Storm goes down. In most environments, Hadoop is used for batch processing while Storm is used for stream processing that causes an increase in code size, number of bugs to fix, development effort, introduces a learning curve, and causes other issues. This creates the difference between Big data Hadoop and Apache Spark.

Spark Streaming helps in fixing these issues and provides a scalable, efficient, resilient, and integrated (with batch processing) system. Spark has provided a unified engine that natively supports both batch and streaming workloads. Spark's single execution engine and unified Spark programming model for batch and streaming lead to some unique benefits over other traditional streaming systems.

Key reason behind Spark Streaming's rapid adoption is the unification of disparate data processing capabilities. This makes it very easy for developers to use a single framework to satisfy all the processing needs. Furthermore, data from streaming sources can be combined with a very large range of static data sources available through Apache Spark SQL.

To address the problems of traditional stream processing engine, Spark Streaming uses a new architecture called **Discretized Streams** that directly leverages the rich libraries and fault tolerance of the Spark engine.

5. Spark Streaming Architecture and Advantages

Instead of processing the streaming data one record at a time, Spark Streaming discretizes the data into tiny, sub-second micro-batches. In other words, Spark Streaming receivers accept data in parallel and buffer it in the memory of Spark's workers nodes. Then the latency-optimized Spark engine runs short tasks to process the batches and output the results to other systems.

Unlike the traditional continuous operator model, where the computation is statically allocated to a node, Spark tasks are assigned to the workers dynamically on the basis of data locality and available resources. This enables better load balancing and faster fault recovery.

Each batch of data is a **Resilient Distributed Dataset (RDD)** in Spark, which is the basic abstraction of a fault-tolerant dataset in Spark. This allows the streaming data to be processed using any Spark code or library.

5.1. Goals of Spark Streaming

This architecture allows Spark Streaming to achieve the following goals:

a) Dynamic load balancing

Dividing the data into small micro-batches allows for fine-grained allocation of computations to resources. Let us consider a simple workload where partitioning of input data stream needs to be done by a key and processed. In the traditional record-at-a-time approach, if one of the partitions is more computationally intensive than others, the node to which that partition is assigned will become a bottleneck and slow down the pipeline. The job's tasks will be naturally load balanced across the workers where some workers will process a few longer tasks while others will process more of the shorter tasks in Spark Streaming.

b) Fast failure and straggler recovery

Traditional systems have to restart the failed operator on another node to recompute the lost information in case of node failure. Only one node is handling the recomputation due to which the pipeline cannot proceed until the new node has caught up after the replay. In Spark, the computation is discretized into small tasks that can run anywhere without affecting correctness. So failed tasks can be distributed evenly on all the other nodes in the cluster to perform the recomputations and recover from the failure faster than the traditional approach.

c) Unification of batch, streaming and interactive analytics

A DStream in Spark is just a series of RDDs in Spark that allows batch and streaming workloads to interoperate seamlessly. Arbitrary Apache Spark functions can be applied on each batch of streaming data. Since the

batches of streaming data are stored in the Spark's worker memory, it can be interactively queried on demand.

d) Advanced analytics like machine learning and interactive SQL

Spark interoperability extends to rich libraries like MLlib (machine learning), SQL, DataFrames, and GraphX. RDDs generated by DStreams can be converted to DataFrames and queried with SQL.

Machine learning models generated offline with MLlib can applied on streaming data.

e) Performance

Spark Streaming's ability to batch data and leverage the Spark engine leads to almost higher throughput to other streaming systems. Spark Streaming can achieve latencies as low as a few hundred milliseconds.

6. How does Spark Streaming works?

In Spark Streaming data stream is divided into batches called **DStreams**, which internally is a sequence of RDDs. The RDDs are then processed using Spark APIs, and the results are returned in batches.

Spark Streaming provides an API in **Scala**, Java, and Python. The Python API was recently introduced in Spark 1.2 and still lacks many features.

Spark Streaming maintains a state based on data coming in a stream and this is called as stateful computations. It also allows window operations (i.e., allows the developer to specify a time frame to perform operations on the data that flows in that time window). There is sliding interval in the window, which is the time interval of updating the window.

7. Spark Streaming Sources

Every input DStream (except file stream) is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing.

There are two categories of built-in streaming sources:

- **Basic sources –** These are the sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- **Advanced sources –** Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies.

There are two types of receivers base on their reliability:

- **Reliable Receiver –** A reliable receiver is the one that correctly sends acknowledgment to a source when the data has been received and stored in Spark with replication.
- **Unreliable Receiver –** An unreliable receiver does not send acknowledgment to a source. This can be used for sources when one does not want or need to go into the complexity of acknowledgment.

8. Spark Streaming Operations

Spark streaming support two types of operations:

8.1. Transformation Operations in Spark

Similar to Spark RDDs, Spark transformations allow modification of the data from the input DStream. DStreams support many transformations that are available on normal Spark RDD's. Some of the common ones are as follows.

map(), flatMap(), filter(), repartition(numPartitions), union(otherStream), count(), reduce(), countByValue(), reduceByKey(func, [numTasks]), join(otherStream, [numTasks]), cogroup(otherStream, [numTasks]), transform(), updateStateByKey(), Window()

8.2. Output Operations in Apache Spark

DStream's data is pushed out to external systems like a database or file systems using Output Oprations. Since external systems consume the transformed data as allowed by the output operations, they trigger the actual execution of all the DStream transformations. Currently, the following output operations are defined:

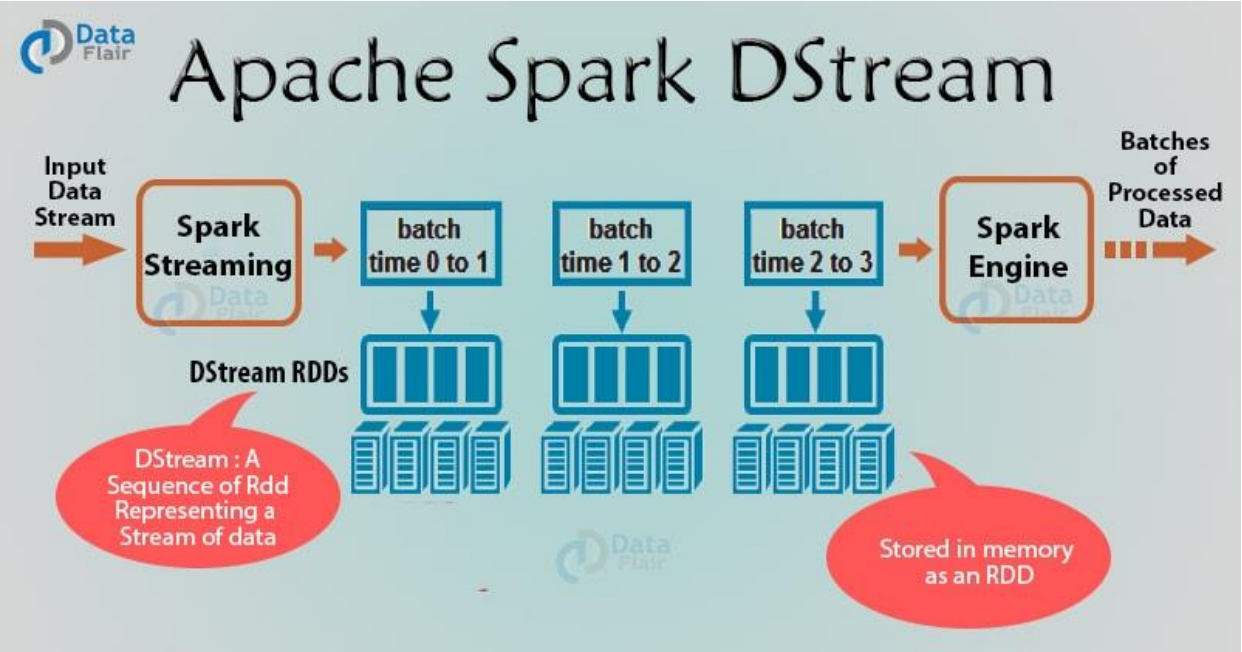
print(), saveAsTextFiles(prefix, [suffix])"prefix-TIME_IN_MS[.suffix]", saveAsObjectFiles(prefix, [suffix]), saveAsHadoopFiles(prefix, [suffix]), foreachRDD(func)

Hence, DStreams like RDDs are executed lazily by the output operations. Specifically, received data is processed forcefully by **RDD actions** inside the DStream output operations. By default, output operations are executed one-at-a-time. And they are executed in the order they are defined in the Spark applications.

36.Apache spark dstream (discretized streams)

1. Objective

This Spark tutorial, walk you through the **Apache Spark DStream**. First of all, we will see what is Spark Streaming, then, what is DStream in Apache Spark. Discretized Stream Operations i.e Stateless and Stateful Transformations, Output operation, Input DStream, and Receivers are also discussed in this Apache Spark blog.



2. Introduction to DStream in Apache Spark

In this section, we will learn about DStream. What are its role, and responsibility in Spark Streaming? It includes what all methods are inculcated to deal with live streaming of data.

As an extension to Apache Spark API, **Spark Streaming** is fault tolerant, high throughput system. It processes the live stream of data. Spark Streaming takes input from various reliable inputs sources like **Flume**, **HDFS**, and **Kafka** etc. and then sends the processed data to filesystems, database or live dashboards. The input data stream is divided into the batches of data and then generates the final stream of the result in batches.

Spark **DStream (Discretized Stream)** is the basic abstraction of **Spark Streaming**. DStream is a continuous stream of data. It receives input from various sources like *Kafka, Flume, Kinesis, or TCP sockets.* It can also be a data stream generated by transforming the input stream. At its core, ***DStream is a continuous stream of RDD (Spark abstraction)***. Every RDD in DStream contains data from the certain interval.

Any operation on a DStream applies to all the underlying RDDs. DStream covers all the details. It provides the developer a high-level API for convenience. As a result, Spark DStream facilitates working with streaming data.

Spark Streaming offers fault-tolerance properties for DStreams as that for RDDs. as long as a copy of the input data is available, it can recompute any state from it using the lineage of the RDDs. By default, Spark replicates data on two nodes. As a result, Spark Streaming can bear single worker failures.

3. Apache Spark DStream Operations

Like RDD, Spark DStream also support two types of Operations: Transformations and output Operations-

3.1. Transformation

There are two types of transformation in DStream:

- Stateless Transformations
- Stateful Transformations

3.1.1. Stateless Transformations

The processing of each batch has no dependency on the data of previous batches. *Stateless transformations* are simple RDD transformations. It applies on every batch meaning every RDD in a DStream. It includes common **RDD transformations** like *map(), filter(), reduceByKey()* etc.

Although these functions seem like applying to the whole stream, ***each DStream is a collection of many RDDs (batches)***. As a result, each stateless transformation applies to each RDD.

Stateless transformations are capable of combining data from many DStreams within each time step. For example, key/value DStreams have the same join-related transformations as RDDs— *cogroup(), join(), leftOuterJoin()* etc.

We can use these operations on DStreams to perform underlying RDD operations on each batch.

If *stateless transformations* are insufficient, DStreams comes with an advanced operator called transform(). **transform()** allow operating on the

RDDs inside them. The *transform()* allows any arbitrary RDD-to-RDD function to act on the DStream. This function gets called on each batch of data in the stream to produce a new stream.

3.1.2. Stateful Transformations

It uses data or intermediate results from previous batches and computes the result of the current batch. *Stateful transformations* are operations on DStreams that track data across time. Thus it makes use of some data from previous batches to generate the results for a new batch. The two main types are **windowed operations**, which act over a sliding window of time periods, and **updateStateByKey()**, which is used to track state across events for each key (e.g., to build up an object representing each user session).

Follow this link to Read DStream Transformations in detail with the examples.

3.2. Output Operation

Once we get the data after transformation, on that data output operation are performed in Spark Streaming. After the debugging of our program, using output operation we can only save our output. Some of the output operations are print(), save() etc.. The save operation takes directory to save file into and an optional suffix. The print() takes in the first 10 elements from each batch of the DStream and prints the result.

4. Input DStreams and Receivers

Input DStream is a DStream representing the stream of input data from streaming source. Receiver (Scala doc, Java doc) object associated with every input DStream object. It receives the data from a source and stores it in Spark's memory for processing.

Spark Streaming provides two categories of built-in streaming sources:

- ***Basic sources –*** These are Source which is directly available in the*StreamingContext API.* Examples: file systems, and socket connections.
- ***Advanced Sources –*** These sources are available by extra utility classes like *Kafka, Flume, Kinesis.* Thus, requires linking against extra dependencies.

For example:

- ***Kafka:*** the artifact required for Kafka is *spark-streaming-kafka-0-8_2.11.*
- ***Flume:*** the artifact requires for Flume is *dspark-streaming-flume_2.11.*
- ***Kinesis:*** the artifact required for Kinesis is *spark-streaming-kinesis-asl_2.11.*

It creates many inputs DStream to receive multiple streams of data in parallel. It creates multiple receivers that receive many data stream. Spark worker/executor is a long-running task. Thus, occupies one of the cores which associate to Spark Streaming application. So, it is necessary that, Spark Streaming application has enough cores to process received data.

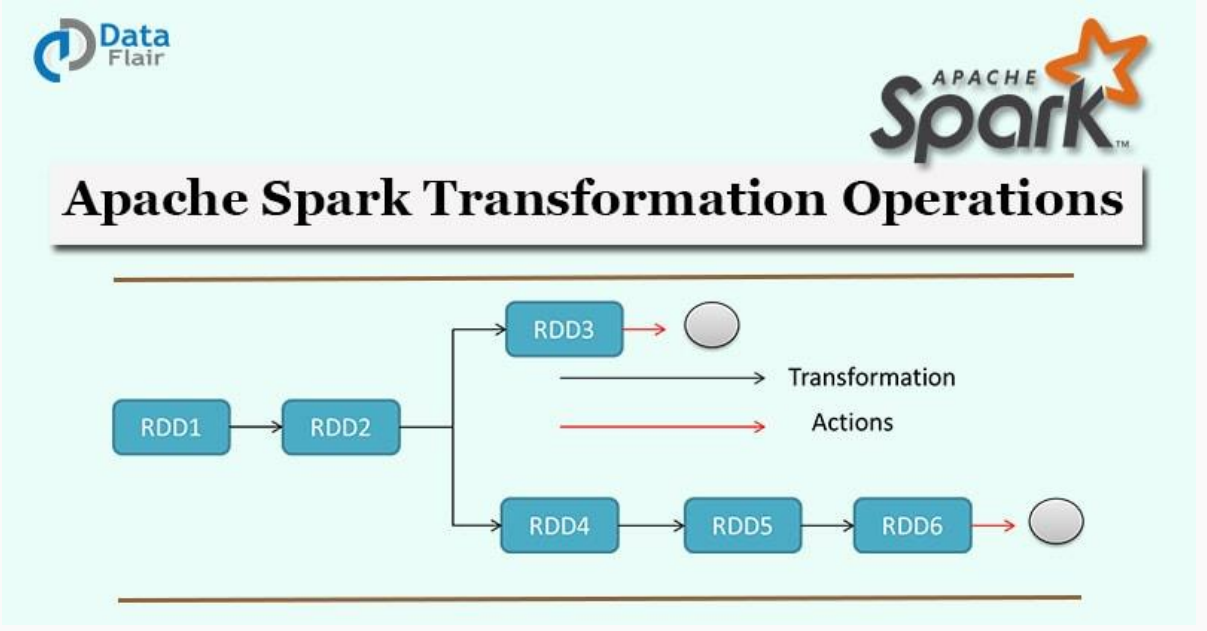
4. Conclusion

In conclusion, just like RDD in Spark, Spark Streaming provides a high-level abstraction known as DStream. DStream represents a continuous stream of data. Internally, DStream is portrait as a sequence of RDDs. Thus, like RDD, we can obtain DStream from input DStream like Kafka, Flume etc. Also, the transformation could be applied on the existing DStream to get a new DStream.

38.Apache spark streaming transformation operations

1. Objective

Through this Apache Spark Transformation Operations tutorial, you will learn about various Apache Spark streaming transformation operations with example being used by Spark professionals for playing with Apache Spark Streaming concepts. You will learn the Streaming operations like Spark Map operation, flatmap operation, Spark filter operation, count operation, Spark ReduceByKey operation, Spark CountByValue operation with example and Spark UpdateStateByKey operation with example that will help you in your Spark jobs.



2. Introduction to Apache Spark Streaming Transformation Operations

Before we start learning the various Streaming operations in Spark, let us revise Spark Streaming concepts.

Below are the various most common Streaming transformation functions being used in Spark industry:

a. map()

Map function in Spark passes each element of the source DStream through a function and returns a new DStream.

Spark map() example

```
1. val conf = new SparkConf().setMaster("local[2]") .setAppName("MapOpTest")
2. val ssc = new StreamingContext(conf , Seconds(1))
3. val words = ssc.socketTextStream("localhost", 9999)
4. val ans = words.map { word => ("hello" ,word ) } // map hello with each line
5. ans.print()
6. ssc.start() // Start the computation
7. ssc.awaitTermination() // Wait for termination
8. }
```

b. flatMap()

FlatMap function in Spark is similar to Spark map function, but in flatmap, input item can be mapped to 0 or more output items. This creates difference between map and flatmap operations in spark.

Spark FlatMap Example

```
1. val lines = ssc.socketTextStream("localhost", 9999)
2. val words = lines.flatMap(_.split(" ")) // for each line it split the words by space
3. val pairs = words.map(word => (word, 1))
4. val wordCounts = pairs.reduceByKey(_ + _)
5. wordCounts.print()
```

c. filter()

Filter function in Apache Spark returns selects only those records of the source DStream on which func returns true and returns a new DStream of those records.

Spark Filter function example

```
1. val lines = ssc.socketTextStream("localhost", 9999)
```

```
val words = lines.flatMap(_.split(" "))
val output = words.filter { word => word.startsWith("s") } // filter the words starts with letter“s”
output.print()
```

d. reduceByKey(func, [numTasks])

When called on a DStream of (K, V) pairs, ReduceByKey function in Spark returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

Spark reduceByKey example

```
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```

e. countByValue()

CountByValue function in Spark is called on a DStream of elements of type K and it returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each Spark RDD of the source DStream.

Spark CountByValue function example

```
val line = ssc.socketTextStream("localhost", 9999)
val words = line.flatMap(_.split(" "))
words.countByValue().print()
```

f. UpdateStateByKey()

The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps.

Define the state – The state can be an arbitrary data type.

Define the state update function – Specify with a function how to update the state using the previous state and the new values from an input stream.

In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not. If the update function returns None then the key-value pair will be eliminated.

Spark UpdateByKey Example

```
def updateFunc(values: Seq[Int], state: Option[Int]): Option[Int] = {
val currentCount = values.sum
val previousCount = state.getOrElse(0)
Some(currentCount + previousCount)
}
val ssc = new StreamingContext(conf , Seconds(10))
```

```
7.  val line = ssc.socketTextStream("localhost", 9999)
8.  ssc.checkpoint("/home/asus/checkpoints/") // Here ./checkpoints/ are the directory where all checkpoints
    are stored.
9.  val words = line.flatMap(_.split(" "))
10. val pairs = words.map(word => (word, 1))
11. val globalCountStream = pairs.updateStateByKey(updateFunc)
12. globalCountStream.print()
13. ssc.start() // Start the computation
14. ssc.awaitTermination()
```

For more Spark operations and examples, you can refer Best books to master Apache Spark.

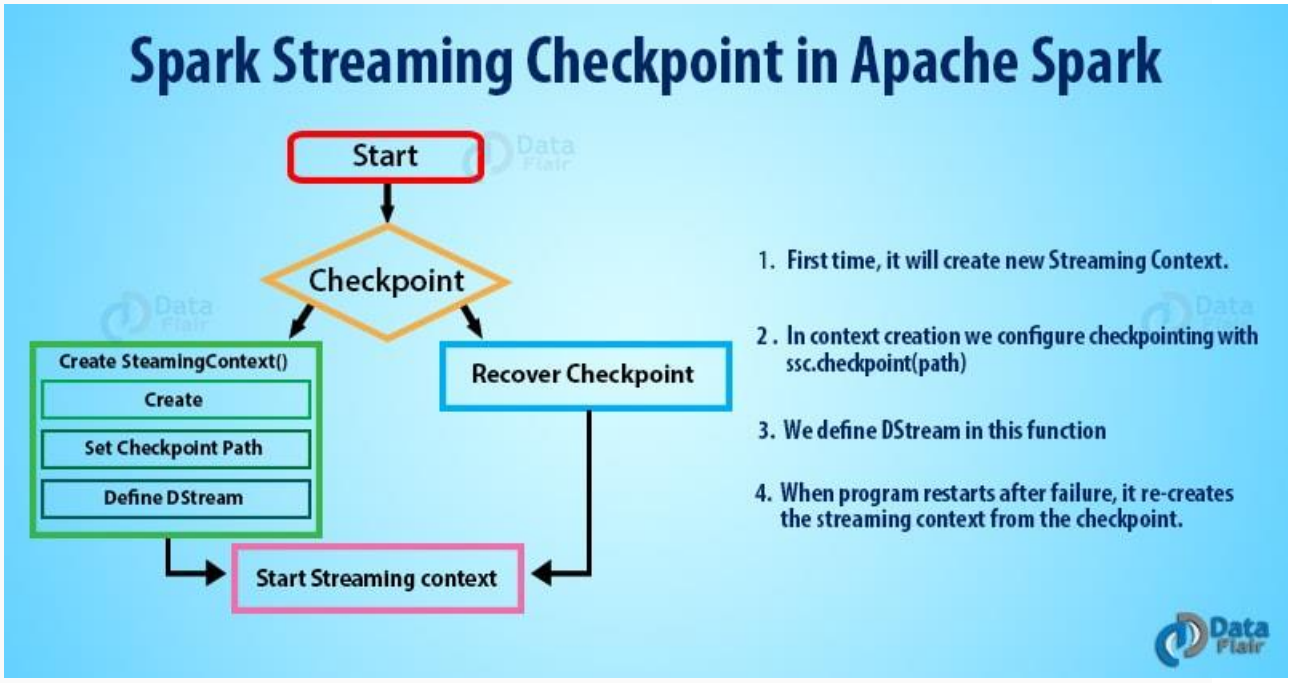
Source:

http://spark.apache.org/docs/latest/programming-guide.html

39.Spark Streaming Checkpoint in Apache Spark

1. Objective

In this blog of **Apache Spark** Streaming Checkpoint, you will read all about Spark Checkpoint. First of all, we will discuss What is Checkpointing in Spark, then, How Checkpointing helps to achieve Fault Tolerance in Apache Spark. Types of Apache Spark Checkpoint i.e Metadata Checkpointing, Data Checkpointing, the comparison between Checkpointing vs Persist() in Spark are also covered in this Spark tutorial.



If you want to learn Apache Spark from basics then our previous post on **Apache Spark Introduction** will help you. It has a Spark video which helps you to learn Apache Spark simply way.

2. Introduction to Spark Streaming Checkpoint

The need with **Spark Streaming** application is that it should be operational 24/7. Thus, the system should also be fault tolerant. If any data is lost, the recovery should be speedy. Spark streaming accomplishes this using **checkpointing.**

So, *Checkpointing* is a process to truncate RDD lineage graph. It saves the application state timely to reliable storage (**HDFS**). As the driver restarts the recovery takes place.

There are two types of data that we checkpoint in Spark:

1. **Metadata Checkpointing –** *Metadata* means the data about data. It refers to saving the metadata to fault tolerant storage like HDFS. Metadata includes configurations, **DStream** operations, and incomplete batches. Configuration refers to the configuration used to create streaming **DStream operations** are operations which define the steaming application. Incomplete batches are batches which are in the queue but are not complete.
2. **Data Checkpointing –**: It refers to save the **RDD** to reliable storage because its need arises in some of the *stateful transformations*. It is in the case when the upcoming RDD depends on the RDDs of previous batches. Because of this, the dependency keeps on increasing with time. Thus, to avoid such increase in recovery time the intermediate RDDs are periodically checkpointed to some reliable storage. As a result, it cuts down the dependency chain.

To set the Spark checkpoint directory
call: *SparkContext.setCheckpointDir(directory: String)*

3. Types of Checkpointing in Apache Spark

There are two types of Apache Spark checkpointing:

1. **Reliable Checkpointing –** It refers to that checkpointing in which the actual RDD is saved in reliable distributed file system, e.g. HDFS. To set

the checkpoint directory call: *SparkContext.setCheckpointDir(directory: String)*. When running on the cluster the directory must be an HDFS path since the driver tries to recover the checkpointed RDD from a local file. While the checkpoint files are actually on the executor's machines.

2. **Local Checkpointing:** In this checkpointing, in *Spark Streaming* or *GraphX* we truncate the RDD lineage graph in Spark. In this, the RDD is persisted to local storage in the executor.

4. Difference between Spark Checkpointing and Persist

There are various differences between Spark Checkpoint vs Persist. Let's discuss it one by one-

Persist

- When we **persist RDD** with **DISK_ONLY** storage level the RDD gets stored in a location where the subsequent use of that RDD will not reach that points in recomputing the lineage.
- After persist() is called, Spark remembers the lineage of the RDD even though it doesn't call it.
- Secondly, after the job run is complete, the cache is cleared and the files are destroyed.

Checkpointing

- Checkpointing stores the RDD in HDFS. It deletes the lineage which created it.
- On completing the job run unlike cache the checkpoint file is not deleted.
- When we checkpointing an RDD it results in double computation. The operation will first call a cache before accomplishing the actual job of computing. Secondly, it is written to checkpointing directory.

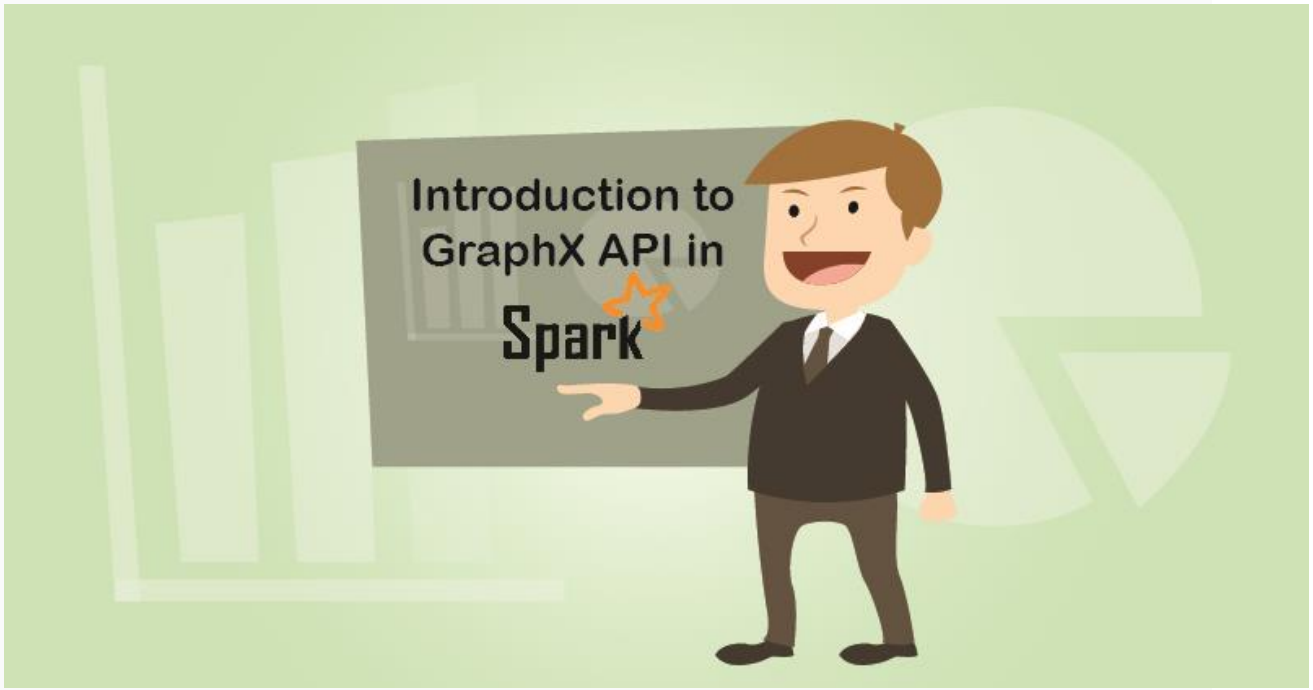
5. Conclusion

In the conclusion, using the method of checkpointing one can achieve **fault tolerance**. It provides fault tolerance to the streaming data when it needs. Also when the read operation is complete the files are not removed like in persist method. Thus, the RDD needs to be checkpointed if the computation takes a long time or the computing chain is too long or if it depends on too many RDDs.

40.Graphx API in Apache Spark: An Introductory Guide

1. Objective

For graphs and graph-parallel computation, **Apache Spark** has an additional API, GraphX. In this blog, we will learn the whole concept of GraphX API in Spark. We will also learn how to import Spark and GraphX into the project. Moreover, we will understand the concept of Property Graph. Also, we will cover graph operators and Pregel API in detail. In addition, we will also learn features of GraphX. Furthermore, we will also see Use cases of GraphX API.



2. Introduction to Spark GraphX API

For graphs and graph-parallel computation, Apache Spark has an additional API, GraphX. To simplify graph analytics tasks it includes the collection of graph algorithms & builders.

In addition, it extends the **Spark RDD** with a Resilient Distributed Property Graph. Basically, the property graph is a directed multigraph. It has multiple edges in parallel. Here, every vertex and edge have user-defined properties associated with it. Moreover, parallel edges allow multiple relationships between the same vertices

3. Getting Started with GraphX

You first need to import Spark and GraphX into your project, To get started, code as Follows:

import org.apache.spark._

import org.apache.spark.graphx._

// To make some of the examples work we will also need RDD

import org.apache.spark.rdd.RDD

Note: you will also need a SparkContext if you are not using the Spark shell.

4. The Property Graph

A directed multigraph with user-defined objects attached to each vertex and edge is a property graph. It is a graph with potentially multiple parallel edges. They are also sharing the same source and destination vertex. Where there can be multiple relationships between the same vertices, they are able to support parallel edges. It also simplifies modeling scenarios. By a unique 64-bit long identifier (VertexId) each vertex is keyed. It does not impose any ordering constraints on the vertex identifiers. Hence, edges have corresponding source and destination vertex identifiers.

Basically, it is parameterized over the vertex (VD) and edge (ED) types. Ultimately, these are the types of the objects those are associated with each vertex and edge respectively.

Some more Insights

In addition, it optimizes the representation of vertex and edge types. While they are primitive data types it reduces the in memory footprint. Even by storing them in specialized arrays.

As same as RDDs, property graphs are also immutable, distributed, and fault-tolerant. we can produce a new graph with the desired changes by changing the values or structure of the graph. We can reuse substantial parts of the original graph in the new graph. It also reduces the cost of this inherently functional data structure. Using a range of vertex partitioning heuristics, a graph is partitioned across the executors. In the event of a failure, each partition of the graph can be recreated on a different machine.

5. Spark Graph Operators

As same as RDDs basic operations like map, filter, property graphs also have a collection of basic operators. Those operators take UDFs (user defined functions) and produce new graphs. Moreover, these are produced with transformed properties and structure. In GraphX, there are some core operators defined that have optimized implementations. While in GraphOps there are some convenient operators that are expressed as compositions of

the core operators. Although, Scala operators in GraphOps are automatically available as members of Graph.

For Example

To compute in-degree of each vertex (defined in GraphOps):

```
val graph1: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees1: VertexRDD[Int] = graph.inDegrees1
```

The only difference between core graph operations and GraphOps is different graph representations. In addition, it is must that each graphical representation provides an implementation of the core operations. We can reuse many of the useful GraphOps operations.

There is a set of fundamental operators. For example. Subgraph, joinVertices, and aggregateMessages. Let's discuss all in detail:

a. Structural Operators

It supports a simple set of commonly used structural operators. The following is a list of the basic structural operators in Spark.

```
class Graph[VD, ED] {
def reverse: Graph[VD, ED]
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
        vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

i. The reverse operator

With all the edge directions reversed, it returns a new graph. when trying to compute the inverse PageRank, this can be very useful.

ii. The subgraph operator

It returns the graph containing only the vertices, by taking vertex and edge predicates. We can use subgraph operator to restrict the graph to the vertices and edges of interest. Also to eliminate broken links.

iii. The mask operator

It constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph. We can use it in conjunction with the subgraph operator to restrict a graph based on the properties in another related graph.

iv. The groupEdges operator

It merges parallel edges in the multigraph.

b. Join Operators

Sometimes it is the major requirement to join data from external collections (RDDs) with graphs. Like it is possible that we have extra user properties that we want to merge with an existing graph. Also, we might want to pull vertex properties from one graph into another. Hence, we can accomplish it by using the *join* operators. In addition, there are two Join operators, joinvertices, and Outerjoinvertices.

```
class Graph[VD, ED] {

def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD,
U) => VD)

  : Graph[VD, ED]

def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId,
VD, Option[U]) => VD2)

  : Graph[VD2, ED]

}
```

c. Aggregate Messages (aggregateMessages)

In GraphX, the core aggregation operation is aggregateMessages. It applies a user-defined sendMsg function to each *edge triplet* in the graph. Afterwards, it uses the mergeMsg function to aggregate those messages at their destination vertex.

```
class Graph[VD, ED] {

def aggregateMessages[Msg: ClassTag](

    sendMsg: EdgeContext[VD, ED, Msg] => Unit,

    mergeMsg: (Msg, Msg) => Msg,

    tripletFields: TripletFields = TripletFields.All)

  : VertexRDD[Msg]

}
```

6. Pregel API

Since properties of vertices depend on properties of their neighbors, Graphs are inherently recursive data structures. That, in turn, depend on properties of *their*neighbors. As a matter of fact, many important graph algorithms iteratively recompute the properties of each vertex. It recomputes until it

reaches a fixed-point condition. Basically, To express these iterative algorithms it proposes a graph-parallel abstraction. Hence, there is a variant of the Pregel API, exposed by GraphX.

7. Spark GraphX Features

The features of Spark GraphX are as follows:

a. Flexibility

We can work with both graphs and computations with Spark GraphX. It includes exploratory analysis, ETL (Extract, Transform & Load), the iterative graph in 1 system. It is possible to view the same data as both graphs, collections, transform and join graphs with RDDs. Also using the Pregel API it is possible to write custom iterative graph algorithms.

b. Speed

It offers comparable performance to the fastest specialized graph processing systems. Also comparable with the fastest graph systems. Even while retaining Spark's flexibility, fault tolerance and ease of use.

c. Growing Algorithm Library

Spark GraphX offers growing library of graph algorithms. Basically, We can choose from it. For example pagerank, connected components, SVD++, strongly connected components and triangle count.

8. Conclusion

As a result, we have learned the whole concept of GraphX API. Moreover, we have covered the brief introduction of Operators and Variant, Pregel API. Also, we have seen how GraphX API in Apache Spark: An Introductory Guide simplifies graph analytics tasks in Spark. Hence, GraphX API is very useful for graphs and graph-parallel computation in Spark.

41.Apache Storm vs Spark Streaming – Feature wise Comparison

1. Objective

This tutorial will cover the comparison between Apache Storm vs Spark streaming. **Apache Storm** is the stream processing engine for processing real-time streaming data. While **Apache Spark** is general purpose

computing engine. It provides **Spark Streaming** to handle streaming data. It process data in near real-time. Let's understand which is better in the battle of Spark vs storm.

2. Apache Storm vs Spark Streaming Comparison

The following description shows the detailed feature wise difference between Apache Storm vs Spark Streaming. These differences will help you know which is better to use between Apache Storm and Spark. Let's have a look on each feature one by one-

2.1. Processing Model

- **Storm:** It supports true stream processing model through core storm layer.
- **Spark Streaming:** **Apache Spark Streaming** is a wrapper over Spark batch processing.

2.2. Primitives

- **Storm:** It provides a very rich set of primitives to perform tuple level process at intervals of a stream (filters, functions). Aggregations over messages in a stream are possible through group by semantics. It supports left join, right join, inner join (default) across the stream.
- **Spark Streaming:** It provides 2 wide varieties of operators. First is **Stream transformation operators** that transform one *DStream* into another *DStream*. Second is **output operators** that write information to external systems. The previous includes stateless operators (filter, map, mapPartitions, union, distinct than on) still as stateful window operators (countByWindow, reduceByWindow then on).

2.3. State Management

- **Storm:** Core Storm by default doesn't offer any framework level support to store any intermediate bolt output (the result of user operation) as a

state. Hence, any application has to create/update its own state as and once required.

- **Spark Streaming:** The underlying Spark by default treats the output of every **RDD** operation(Transformations and Actions) as an intermediate state. It stores it as RDD. Spark Streaming permits maintaining and changing state via *updateStateByKey* API. A pluggable method couldn't be found to implement state within the external system.

2.4. Message Delivery Guarantees (Handling message level failures)

- **Storm:** It supports 3 message processing guarantees: *at least once*, *at-most-once* and *exactly once*. Storm's reliability mechanisms are distributed, scalable, and fault-tolerant.
- **Spark Streaming:** Apache Spark Streaming defines its fault tolerance semantics, the guarantees provided by the recipient and output operators. As per the **Apache Spark architecture**, the incoming data is read and replicated in different Spark executor's nodes. This generates failure scenarios data received but may not be reflected. It handles fault tolerance differently in the case of worker failure and driver failure.

2.5. Fault Tolerance (Handling process/node level failures)

- **Storm:** Storm is intended with fault-tolerance at its core. Storm daemons (*Nimbus and Supervisor*) are made to be fail-fast (that means that method self-destructs whenever any sudden scenario is encountered) and stateless (all state is unbroken in Zookeeper or on disk).
- **Spark Streaming:** The Driver Node (an equivalent of JT) is *SPOF*. If driver node fails, then all executors will be lost with their received and replicated in-memory information. Hence, Spark Streaming uses *data checkpointing* to get over from driver failure.

2.6. Debuggability and Monitoring

- **Storm:** Apache Storm UI support image of every topology; with the entire break-up of internal spouts and bolts. UI additionally contributes information having any errors coming in tasks and fine-grained stats on the throughput and latency of every part of the running topology. It helps in debugging problems at a high level. ***Metric based***

monitoring: Storm's inbuilt metrics feature supports framework level for applications to emit any metrics, which can then be simply integrated with external metrics/monitoring systems.

- **Spark Streaming:** Spark web UI displays an extra Streaming tab that shows statistics of running receivers (whether receivers are active, the variety of records received, receiver error, and so on.) and completed batches (batch process times, queuing delays, and so on). It is useful to observe the execution of the application. The following 2 info in Spark web UI are significantly necessary for standardization of batch size:

1. *Processing Time* – The time to process every batch of data.
2. *Scheduling Delay* – The time a batch stays in a queue for the process previous batches to complete.

2.7. Auto Scaling

- **Storm:** It provides configuring initial parallelism at various levels per topology – variety of worker processes, executors, tasks. Additionally, it supports dynamic rebalancing, that permits to increase or reduces the number of worker processes and executors w/o being needed to restart the cluster or the topology. But, many initial tasks designed stay constant throughout the life of topology.
  Once all supervisor nodes are fully saturated with worker processes, and there's a need to scale out, one merely has to begin a replacement supervisor node and inform it to cluster wide Zookeeper.
  It is possible to transform the logic of monitor the present resource consumption on every node in a very Storm cluster, and dynamically add a lot of resources. STORM-594 describes such auto-scaling mechanism employing a feedback system.
- **Spark Streaming:** The community is currently developing on dynamic scaling to streaming applications. At the instant, elastic scaling of Spark streaming applications isn't supported.
  Essentially, dynamic allocation isn't meant to be used in Spark streaming at the instant (1.4 or earlier). the reason is that presently the receiving topology is static. the number of receivers is fixed. One receiver is allotted with every DStream instantiated and it'll use one core within the cluster. Once the StreamingContext is started, this topology cannot be modified. Killing receivers leads to stopping the topology.

2.8. Yarn Integration

- **Storm:** The Storm integration alongside **YARN** is recommended through ***Apache Slider***. A slider is a YARN application that deploys non-YARN distributed applications over a YARN cluster. It interacts with **YARN RM** to spawn *containers*for distributed application then manages the lifecycle of these containers. Slider provides out-of-the-box application packages for Storm.
- **Spark Streaming:** Spark framework provides native integration along with YARN. Spark streaming as a layer above Spark merely leverages the integration. Every Spark streaming application gets reproduced as an individual Yarn application. The *ApplicationMaster container* runs the Spark driver and initializes the **SparkContext**.
  Every *executor* and *receiver* run in containers managed by *ApplicationMaster*. The ApplicationMaster then periodically submits one job per micro-batch on the YARN containers.

2.9. Isolation

- **Storm:** Each employee process runs executors for a particular topology. That's mixing of various topology tasks isn't allowed at worker process level which supports topology level runtime isolation. Further, every executor thread runs one or more tasks of an identical element (spout or bolt), that's no admixture of tasks across elements.
- **Spark Streaming:** Spark application is a different application run on YARN cluster, wherever every executor runs in a different YARN container. Thus, JVM level isolation is provided by Yarn since 2 totally different topologies can't execute in same JVM. Besides, YARN provides resource level isolation so that container level resource constraints (CPU, memory limits) can be organized.

2.11. Open Source Apache Community

- **Storm:** Apache Storm powered-by page healthy list of corporations that are running Storm in production for many use-cases. Many of them are large-scale web deployments that are pushing the boundaries for performance and scale. For instance, Yahoo reading consists of two, 300 nodes running Storm for near-real-time event process, with the largest topology spanning across four hundred nodes.
- **Spark Streaming:** Apache Spark streaming remains rising and has restricted expertise in production clusters. But, the general umbrella

Apache Spark community is well one in all the biggest and thus the most active open supply communities out there nowadays. The general charter is space evolving given the massive developer base. this could cause maturity of Spark Streaming within the close to future.

2.12. Ease of development

- **Storm:** It provides extremely easy, rich and intuitive APIs that simply describe the **DAG** nature of process flow (topology). The Storm tuples, which give the abstraction of data flowing between nodes within the DAG, are dynamically written. The motivation there's to change the APIs for simple use. Any new custom tuple can be plugged in once registering its **Kryo serializer**. Developers will begin with writing topologies and run them in native cluster mode. In local mode, threads are used to simulate worker nodes, permitting the developer to set breakpoints, halt the execution, examine variables, and profile before deploying it to a distributed cluster wherever all this is often way tougher.
- **Spark Streaming:** It offers **Scala** and **Java** APIs that have a lot of a practical programming (transformation of data). As a result, the topology code is way a lot of elliptic. There's an upscale set of API documentation and illustrative samples on the market for the developer.

2.13. Ease of Operability

- **Storm:** It is little tricky to deploy/install Storm through many tools (puppets, and then on) and deploys the cluster. Apache Storm contains a dependency on *Zookeeper cluster.* So that it can meet coordination over clusters, store state and statistics. It implements CLI support to install actions like submit, activate, deactivate, list, kill topology. a powerful fault tolerance suggests that any daemon period of time doesn't impact executing topology.
  In *standalone mode*, Storm daemons are compel to run in supervised mode. In*YARN cluster mode*, Storm daemons emerged as containers and driven by Application Master (Slider).
- **Spark Streaming:** It uses Spark as the fundamental execution framework. It should be easy to feed up **Spark cluster** on YARN. There are many deployment requirements. Usually we enable checkpointing for fault tolerance of application driver. This could bring a dependency on fault-tolerant storage (**HDFS**).

2.14. Language Options

- **Storm:** We can create Storm applications in *Java, Clojure, and Scala.*
- **Spark Streaming:** We can create Spark applications in *Java, Scala, Python, and **R.***

3. Conclusion

Hence, the difference between Apache Storm vs Spark Streaming shows that Apache Storm is a solution for real-time stream processing. But Storm is very complex for developers to develop applications. There are very limited resources available in the market for it.

Storm can solve only one type of problem i.e Stream processing. But the industry needs a generalized solution which can solve all the types of problems. For example Batch processing, stream processing interactive processing as well as iterative processing. Here **Apache Spark** comes into limelight which is a general purpose computation engine. It can handle any type of problem. Apart from this Apache Spark is much too easy for developers and can integrate very well with **Hadoop.**

If you feel like something is missing in above article of Apache Storm vs Spark. So, please drop a comment.

42.Apache spark vs. Hadoop mapreduce

1. Objective

Apache Spark is an open-source, lightning fast big data framework which is designed to enhance the computational speed. **Hadoop MapReduce,** read and write from the disk as a result it slows down the computation. While Spark can run on top of Hadoop and provides a better computational speed solution. This tutorial gives a thorough comparison between Apache Spark vs. Hadoop **MapReduce**.

In this guide, we will cover what is the difference between Spark and Hadoop MapReduce, how Spark is 100x faster than MapReduce. This comprehensive guide will provide feature wise comparison between Apache Spark and Hadoop MapReduce.

2. Comparison between Apache Spark vs. Hadoop MapReduce

2.1. Introduction

- **Apache Spark –** It is an open source big data framework. It provides faster and more general purpose data processing engine. Spark is basically designed for fast computation. It also covers wide range of workloads for example batch, interactive, iterative and streaming.
- **Hadoop MapReduce –** It is also an open source framework for writing applications. It also processes structured and unstructured data that are stored in **HDFS**. Hadoop MapReduce is designed in a way to process a large volume of data on a cluster of commodity hardware. MapReduce can process data in batch mode.

2.2. Speed

- **Apache Spark –** Spark is lightning fast **cluster** computing tool. Apache Spark runs applications up to 100x faster in memory and 10x faster on disk than Hadoop. Because of reducing the number of read/write cycle to disk and storing intermediate data in-memory Spark makes it possible.
- **Hadoop MapReduce –** MapReduce reads and writes from disk, as a result, it slows down the processing speed.

2.3. Difficulty

- **Apache Spark –** Spark is easy to program as it has tons of high-level operators with **RDD – Resilient Distributed Dataset.**
- **Hadoop MapReduce –** In MapReduce, developers need to hand code each and every operation which makes it very difficult to work.

2.4. Easy to Manage

- **Apache Spark –** Spark is capable of performing batch, interactive and Machine Learning and Streaming all in the same cluster. As a result makes it a complete**data analytics** engine. Thus, no need to manage different component for each need. **Installing Spark on a cluster** will be enough to handle all the requirements.
- **Hadoop MapReduce –** As MapReduce only provides the batch engine. Hence, we are dependent on different engines. For example- Storm, Giraph, Impala, etc. for other requirements. So, it is very difficult to manage many components.

2.5. Real-time analysis

- **Apache Spark –** It can process real time data i.e. data coming from the real-time event streams at the rate of millions of events per second, e.g.

Twitter data for instance or Facebook sharing/posting. Spark's strength is the ability to process live streams efficiently.

- **Hadoop MapReduce –** MapReduce fails when it comes to real-time data processing as it was designed to perform batch processing on voluminous amounts of data.

Learn: Apache Hive vs Spark SQL: Feature wise comparison

2.6. latency

- **Apache Spark –** Spark provides low-latency computing.
- **Hadoop MapReduce –** MapReduce is a high latency computing framework.

2.7. Interactive mode

- **Apache Spark –** Spark can process data interactively.
- **Hadoop MapReduce –** MapReduce doesn't have an interactive mode.

2.8. Streaming

- **Apache Spark –** Spark can process real time data through **Spark Streaming**.
- **Hadoop MapReduce –** With MapReduce, you can only process data in batch mode.

2.9. Ease of use

- **Apache Spark –** Spark is easier to use. Since, its abstraction (**RDD**) enables a user to process data using **high-level operators**. It also provides rich APIs in Java, **Scala**, Python, and **R**.
- **Hadoop MapReduce –** MapReduce is complex. As a result, we need to handle low-level APIs to process the data, which requires lots of hand coding.

2.10. Recovery

- **Apache Spark –** RDDs allows recovery of partitions on failed nodes by re-computation of the **DAG** while also supporting a more similar recovery style to Hadoop by way of **checkpointing**, to reduce the dependencies of an RDDs.
- **Hadoop MapReduce –** MapReduce is naturally resilient to system faults or failures. So, it is a highly fault-tolerant system.

2.11. Scheduler

- **Apache Spark –** Due to **in-memory computation** spark acts its own flow scheduler.
- **Hadoop MapReduce –** MapReduce needs an external job scheduler for example, **Oozie** to schedule complex flows.

2.12. Fault tolerance

- **Apache Spark – Spark is fault-tolerant**. As a result, there is no need to restart the application from scratch in case of any failure.
- **Hadoop MapReduce –** Like Apache Spark, MapReduce is also fault-tolerant, so there is no need to restart the application from scratch in case of any failure.

2.13. Security

- **Apache Spark –** Spark is little less secure in comparison to MapReduce because it supports the only authentication through shared secret password authentication.
- **Hadoop MapReduce –** Apache Hadoop MapReduce is more secure because of Kerberos and it also supports **Access Control Lists (ACLs)** which are a traditional file permission model.

Learn: Important Terminologies and Concepts in Apache Spark

2.14. Cost

- **Apache Spark –** As spark requires a lot of RAM to run in-memory. Thus, increases the cluster, and also its cost.
- **Hadoop MapReduce –** MapReduce is a cheaper option available while comparing it in terms of cost.

2.15. Language Developed

- **Apache Spark –** Spark is developed in **Scala**.
- **Hadoop MapReduce –** Hadoop MapReduce is developed in **Java.**

2.16. Category

- **Apache Spark –** It is **data analytics** engine. Hence, it is a choice for **Data Scientist.**
- **Hadoop MapReduce –** It is basic data processing engine.

2.17. License

- **Apache Spark –** Apache License 2

- **Hadoop MapReduce –** Apache License 2

2.18. OS support

- **Apache Spark –** Spark supports cross-platform.
- **Hadoop MapReduce –** Hadoop MapReduce also supports cross-platform.

Learn: Apache Spark Ecosystem – Complete Spark Components Guide

2.19. Programming Language support

- **Apache Spark –** Scala, Java, Python, R, SQL.
- **Hadoop MapReduce –** Primarily Java, other languages like C, C++, Ruby, Groovy, Perl, Python are also supported using Hadoop streaming.

2.20. SQL support

- **Apache Spark –** It enables the user to run SQL queries using **Spark SQL**.
- **Hadoop MapReduce –** It enables users to run SQL queries using Apache **Hive**.

2.21. Scalability

- **Apache Spark –** Spark is highly scalable. Thus, we can add n number of nodes in the cluster. Also a largest known **Spark Cluster** is of 8000 nodes.
- **Hadoop MapReduce –** MapReduce is also highly scalable we can keep adding n number of nodes in the cluster. Also, a largest known **Hadoop cluster** is of 14000 nodes.

2.22. The line of code

- **Apache Spark –** Apache Spark is developed in merely 20000 line of codes.
- **Hadoop MapReduce –** Hadoop 2.0 has 1,20,000 line of codes

2.23. Machine Learning

- **Apache Spark –** Spark has its own set of machine learning ie **MLlib.**
- **Hadoop MapReduce –** Hadoop requires machine learning tool for example **Apache Mahout**.

Learn: Apache Storm vs Spark Streaming – Feature wise Comparison

2.24. Caching

- **Apache Spark –** Spark can cache data in memory for further iterations. As a result it enhances the system performance.
- **Hadoop MapReduce –** MapReduce cannot cache the data in memory for future requirements. So, the processing speed is not that high as that of Spark.

2.25. Hardware Requirements

- **Apache Spark – Spark** needs mid to high-level hardware.
- **Hadoop MapReduce –** MapReduce runs very well on commodity hardware.

2.26. Community

- **Apache Spark –** Spark is one of the most active project at Apache. Since, it has a very strong community.
- **Hadoop MapReduce –** MapReduce community has been shifted to Spark.

Learn: Apache Storm vs Spark Streaming – Feature wise Comparison

3. Conclusion

Hence, the differences between Apache Spark vs. Hadoop MapReduce shows that Apache Spark is much-advance cluster computing engine than MapReduce. Spark can handle any type of requirements (batch, interactive, iterative, streaming, graph) while MapReduce limits to Batch processing. Spark is one of the favorite choices of data scientist. Thus Apache Spark is growing very quickly and replacing MapReduce. The framework Apache Flink, surpasses Apache Spark.To know the difference, please read comparison on **Hadoop vs Spark vs Flink.**

43.Apache Hive vs Spark SQL: Feature wise comparison

1. Objective

While **Apache Hive** and **Spark SQL** perform the same action, retrieving data, each does the task in a different way. However, Hive is planned as an interface or convenience for querying data stored in **HDFS**. Though, MySQL is planned for online operations requiring many reads and writes. So we will discuss Apache Hive vs Spark SQL on the basis of their feature.

This blog totally aims at differences between Spark SQL vs Hive in **Apache Spark**. We will also cover the features of both individually. To understand more, we will also focus on the usage area of both. Also, there are several **limitations with Hive** as well as SQL. We will discuss all in detail to understand the difference between Hive and SparkSQL.

2. Comparison between Apache Hive vs Spark SQL

At first, we will put light on a brief introduction of each. Afterwards, we will compare both on the basis of various features.

2.1. Introduction

Apache Hive:

Apache Hive is built on top of Hadoop. Moreover, It is an open source data warehouse system. Also, helps for analyzing and querying large datasets stored in **Hadoop** files. At First, we have to write complex **Map-Reduce** jobs. But, using Hive, we just need to submit merely SQL queries. Users who are comfortable with SQL, Hive is mainly targeted towards them.

Spark SQL:

In Spark, we use Spark SQL for structured data processing. Moreover, We get more information of the structure of data by using SQL. Also, gives information on computations performed. One can achieve extra optimization in Apache Spark, with this extra information. Although, Interaction with Spark SQL is possible in several ways. Such as **DataFrame** and the **Dataset** API.

2.2. Initial release

Apache Hive:

Apache Hive was first released in 2012.

Spark SQL:

While Apache Spark SQL was first released in 2014.

2.3. Current release

Apache Hive:

Currently released on 24 October 2017: version 2.3.1

Spark SQL:

Currently released on 09 October 2017: version 2.1.2

2.4. License

Apache Hive:

It is open sourced, from Apache Version 2.

Spark SQL:

It is open sourced, through Apache Version 2.

2.5. Implementation language

Apache Hive:

Basically, we can implement Apache Hive on Java language.

Spark SQL:

We can implement Spark SQL on Scala, Java, Python as well as R language.

2.6. Primary database model

Apache Hive:

Primarily, its database model is Relational DBMS.

Spark SQL:

Primarily, its database model is also Relational DBMS

2.7. Additional database models

Apache Hive:

It supports an additional database model, i.e. Key-value store

Spark SQL:

As similar as Hive, it also supports Key-value store as additional database model.

2.8. Developer

Apache Hive:

Hive is originally developed by Facebook. But later donated to the Apache Software Foundation, which has maintained it since.

Spark SQL:

It is originally developed by Apache Software Foundation.

2.9. Server operating systems

Apache Hive:

Basically, it supports all Operating Systems with a Java VM.

Spark SQL:

It supports several operating systems. For example Linux OS, X, and Windows.

2.10. Data Types

Apache Hive:

It has predefined data types. For example, float or date.

Spark SQL:

As similar to Spark SQL, it also has predefined data types. For Example, float or date.

2.11. Support of SQL

Apache Hive:

It possesses SQL-like DML and DDL statements.

Spark SQL:

Like Apache Hive, it also possesses SQL-like DML and DDL statements.

2.12. APIs and other access methods

Apache Hive:

Apache Hive supports JDBC, ODBC, and Thrift.

Spark SQL:

Spark SQL supports only JDBC and ODBC.

2.13. Programming languages

Apache Hive:

We can use several programming languages in Hive. For example C++, Java, PHP, and Python.

Spark SQL:

We can use several programming languages in Spark SQL. For example Java, **Python**, **R**, and Scala. This creates difference between SparkSQL and Hive.

2.14. Partitioning methods

Apache Hive:

It uses data sharding method for storing data on different nodes.

Spark SQL:

It uses spark core for storing data on different nodes.

2.15. Replication methods

Apache Hive:

There is a selectable replication factor for redundantly storing data on multiple nodes.

Spark SQL:

Basically, for redundantly storing data on multiple nodes, there is a no replication factor in Spark SQL.

2.16. Concurrency

Apache Hive:

Basically, hive supports concurrent manipulation of data.

Spark SQL:

Whereas, spark SQL also supports concurrent manipulation of data.

Let’s see few more difference between Apache Hive vs Spark SQL.

2.17. Durability

Apache Hive:

Basically, it supports for making data persistent.

Spark SQL:

As same as Hive, Spark SQL also support for making data persistent.

2.18. User concepts

Apache Hive:

There are access rights for users, groups as well as roles.

Spark SQL:

There are no access rights for users.

2.19. Usage

Apache Hive:

- Schema flexibility and evolution.
- Also, can portion and bucket, tables in Apache Hive.
- As JDBC/ODBC drivers are available in Hive, we can use it.

Spark SQL:

- Basically, it performs SQL queries.
- Through Spark SQL, it is possible to read data from existing Hive installation.
- We get the result as Dataset/DataFrame if we run Spark SQL with another programming language.

2.20. Limitations

Apache Hive:

- It does not offer real-time queries and row level updates.
- Also provides acceptable latency for interactive data browsing.
- Hive does not support online transaction processing.
- In Apache Hive, latency for queries is generally very high.

Spark SQL:

- It does not support union type
- Although, no provision of error for oversize of varchar type
- It does not support transactional table
- However, no support for Char type
- It does not support time-stamp in Avro table.

3. Conclusion

Hence, we can not say SparkSQL is not a replacement for Hive neither is the other way. As a result, we have seen that SparkSQL is more spark API and developer friendly. Also, SQL makes programming in spark easier. While, Hive's ability to switch execution engines, is efficient to query huge data sets. Although, we can just say it's usage is totally depends on our goals. Apart from it, we have discussed we have discussed Usage as well as limitations above. Also discussed complete discussion of Apache Hive vs Spark SQL. So, hopefully, this blog may answer all the questions occurred in mind regarding Apache Hive vs Spark SQL.