This PySpark material prepared by Nireekshan under Arjun, Prasad & Ramesh sir guidelines @DVS

Feel free to contact if any queries:

Primary : dvs.training@gmail.com
Secondary : nirekshan@gmail.com

Spark Core and SQL modules

Spark Core module index

Part – 1: Fundamentals

- ✓ Couple of minutes discussion
- ✓ Big Data
  - Too Much Data discussion
- ✓ Challenges with BigData
- ✓ Initially who solved Big Data challenges
- ✓ Hadoop creator profile
- ✓ What can do by Hadoop?
  - Advantages
  - Limitations of Hadoop
  - Spark overcome those limitations
- ✓ Spark creator profile

Part – 2: Introduction

- ✓ What is Apache Spark?
- ✓ Purpose of Spark
- ✓ Spark is written in
- ✓ Can Spark integrate with Hadoop?
- ✓ What kind of files spark support?
- ✓ Is Spark depending on Hadoop?
- ✓ Can I install spark on windows?
- ✓ History of Spark
- ✓ Spark shines on where?
- ✓ Spark is Fast
- ✓ Spark features
- ✓ Data Processing terminology
- ✓ Spark - Before and After
- ✓ Spark is called as unified stack

Part – 3: Spark modules and terminology

- ✓ Apache Spark Components or modules
  - Core
  - SQL
  - Streaming
  - MLib
  - GraphX
  - SparkR
- ✓ Cluster Managers
- ✓ Storage Layers for Spark
- ✓ Spark Execution Model
- ✓ Spark Terminology table
- ✓ Spark follows…
- ✓ Driver program
- ✓ Executors
- ✓ SparkContext

- ✓ How many SparkContext objects can create for one application?
- ✓ Stopping SparkContext object
- ✓ SparkContext responsibilities
- ✓ Spark 1.x version
- ✓ Solution in Spark 2.x
- ✓ Understanding Spark Cluster *Architecture*
- ✓ Anatomy of Spark Application
- ✓ Components
- ✓ **Py4J**
- ✓ Spark clusters
  - Spark clusters: Standalone cluster
  - Spark on YARN
    - YARN - client mode
    - YARN - cluster mode

Part – 4: RDD

- ✓ Importance of RDD
- ✓ Partitions in RDD
- ✓ Creating RDD
- ✓ Caching
- ✓ Persistent
- ✓ Fault-Recovery Mechanism
- ✓ If RAM is inefficient to store RDD then where it stores?
- ✓ RDD features
- ✓ Spark RDD Operations
- ✓ Transformations
- ✓ Types of Transformations
  - Narrow Transformations
  - Wide Transformations
- ✓ Actions
- ✓ Limitation of RDD
- ✓ RDD Operations
  - Transformations & Actions

- ✓ Programs
- ✓ Coalesce and Repartition
- ✓ Internals of Job execution in Spark
- ✓ PySpark URL to find more examples from official website

Spark SQL module index

- ✓ Spark SQL introduction
- ✓ How can we write Spark SQL programs?
- ✓ Spark SQL features
  - Integrated
  - Unified data access
  - Performance optimization

- ✓ DataFrame
- ✓ Introduction about DataFrame
- ✓ Creating DataFrame by loading csv file
- ✓ DataFrame can support what kind of file formats?
- ✓ DataFrame characteristics
- ✓ Programming languages to created DataFrame
- ✓ Why DataFrame
- ✓ Custom management
- ✓ Optimized execution plan
- ✓ Spark SQL execution plan
- ✓ Spark SQL terminology
- ✓ Spark SQL programs
- ✓ PySpark URL to find more examples from official website

Couple of minutes discussion

Discussion

- ✓ One fine day good discussion happened between me and my friend Peter while having cup of Coffee regarding Spark.
- ✓ I'm sure you will enjoy the discussion.

Are you guys ready?

- ✓ Then let us start

Discussion starts

Peter : Hey Nireekshan, what is Spark?

Nireekshan :

Spark is a cluster computing platform or engine designed to be fast and general purpose.

Peter : Okay.

Peter : In how many languages we can write spark programs?

Nireekshan : Yes Peter, let me explain in simple way

Nireekshan : Take general examples.

To write Python program we need to use Python programming language.
To write C program we need to use C programming language.
To write Java program we need to use Java programming language.
To write C++ program we need to use C++ programming language.

Nireekshan : So, we can write Spark programs by using **FOUR** programming languages

Those are,

1. **Python** programming language
2. Java programming language
3. Scala programming language
4. R programming language

Peter : Yeah Nireekshan, I understand.

Peter : Above explanation is for *what is Spark*, right?

Nireekshan : Yes

Peter : Then what is PySpark?

Nireekshan : Good question.

Nireekshan :

PySpark = Python + Spark

So, if you are writing spark program by using Python then that programs are called as PySpark programs

Peter : Wowww I got it about pyspark.

Peter : Sorry Nireekshan, one more final question.

Nireekshan : Its okay Cool, you can ask any question I mean technical related question.

Peter : So, What about JaSpark and ScaSpark?

Nireekshan : Whaaaaaat???
Come again…

Peter :

I mean, so if I'm writing spark program by using Java then that programs are called as JaSpark programs naa

JaSpark = Java + Spark

I mean, so if I'm writing spark program by using Scala then that programs are called as ScaSpark programs naa

ScaSpark = Scala + Spark

Nireekshan : Got shocked and drunk water then replied.

Excellent point BOSS, I'm sorry for saying there is no such kind of terminology like JaSpark and ScaSpark.

Nireekshan :

If you are choosing either Java or Scala to write Spark programs, then those programs are by default called as Spark programs.

Peter : Ohhh okay, thank you Nireekshan to clarify my queries.

Nireekshan : It's my pleasure Dude.

Big Data

- ✓ Please observe the data measurement table.
- ✓ So, measurement starts with bit, byte, kilobyte, megabyte… etc.
- ✓ If you are discussing about terabytes, petabytes, exabytes…etc then your discussion is related to big data and that data is called as BigData.
- ✓ Big data can be structured, semi structured and un-structured.

Unit	Value	Size
bit (b)	0 or 1	1/8 of a byte
byte (B)	8 bits	1 byte
kilobyte (KB)	1000^1 bytes	1,000 bytes
megabyte (MB)	1000^2 bytes	1,000,000 bytes
gigabyte (GB)	1000^3 bytes	1,000,000,000 bytes
terabyte (TB)	1000^4 bytes	1,000,000,000,000 bytes
petabyte (PB)	1000^5 bytes	1,000,000,000,000,000 bytes
exabyte (EB)	1000^6 bytes	1,000,000,000,000,000,000 bytes
zettabyte (ZB)	1000^7 bytes	1,000,000,000,000,000,000,000 bytes
yottabyte (YB)	1000^8 bytes	1,000,000,000,000,000,000,000,000 bytes

Fact

- ✓ So, today every company is holding and generating big data regularly.

Discussion between Prasad and Nireekshan

- ✓ Prasad : Hey Nireekshan, can I say big data means ***Too Much Data*** ☺
- ✓ Nireekshan : Yes Prasad, we can say.
- ✓ Prasad : So, what kind of challenges will come from too much data?
- ✓ Nireekshan : So, instead of replying orally I would like to show one best image.
- ✓ Nireekshan : Prasad, requesting you please observe below image.

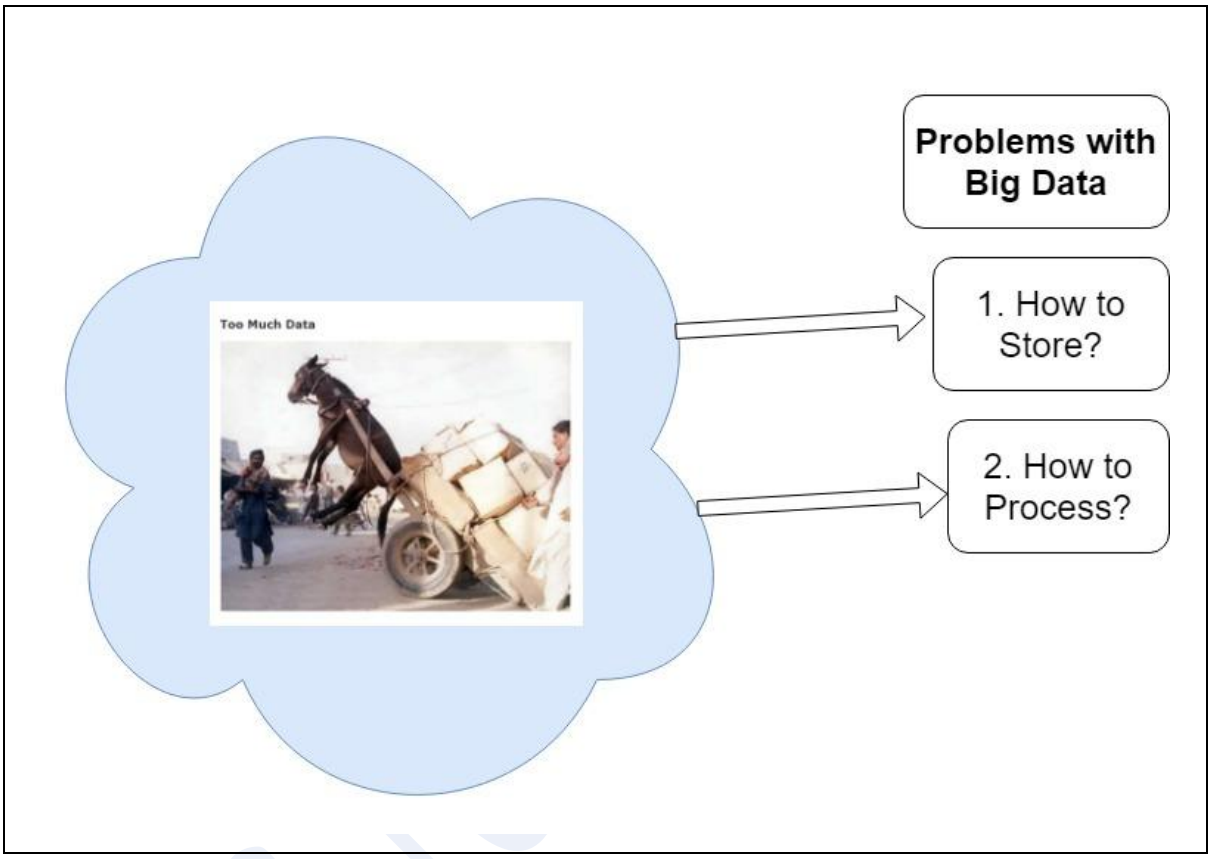
Too Much Data

- ✓ Nireekshan : I hope you understand clearly.
- ✓ Prasad : OMG…!!!
- ✓ Prasad : ☹
  I'm feeling pity on that Animal.
  Requesting, can you please remove that overload from that truck.
- ✓ Nireekshan :

  Hey Prasad, appreciate your merciful on that Animal.
  That burden is removed from animal and that animal is in safe position now.
- ✓ Prasad : Thanks for awesome explanation about BigData definition.

Challenges with BigData

- ✓ So, when it comes to Big data, we need to take care about two challenges.
- ✓ If we don't know how to face those challenges, then it turns to problems.



Discussion between Mohan and Nireekshan

- ✓ Mohan : Nireekshan a *smart* question for you because you are smart

- ✓ Nireekshan : Thanks for your Bisket Mohan ☺, yes what is your question?

- ✓ Mohan : Yeah, thanks too you for understanding Bisket ☺

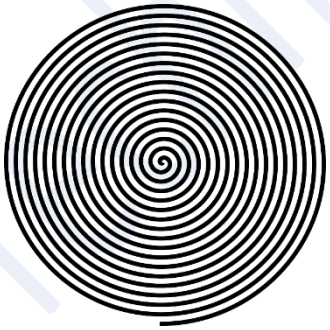
  So, my question is,

  Now a day's companies came to know how to store and process big data.

  Initially who solved these two problems, means storing and processing problems?

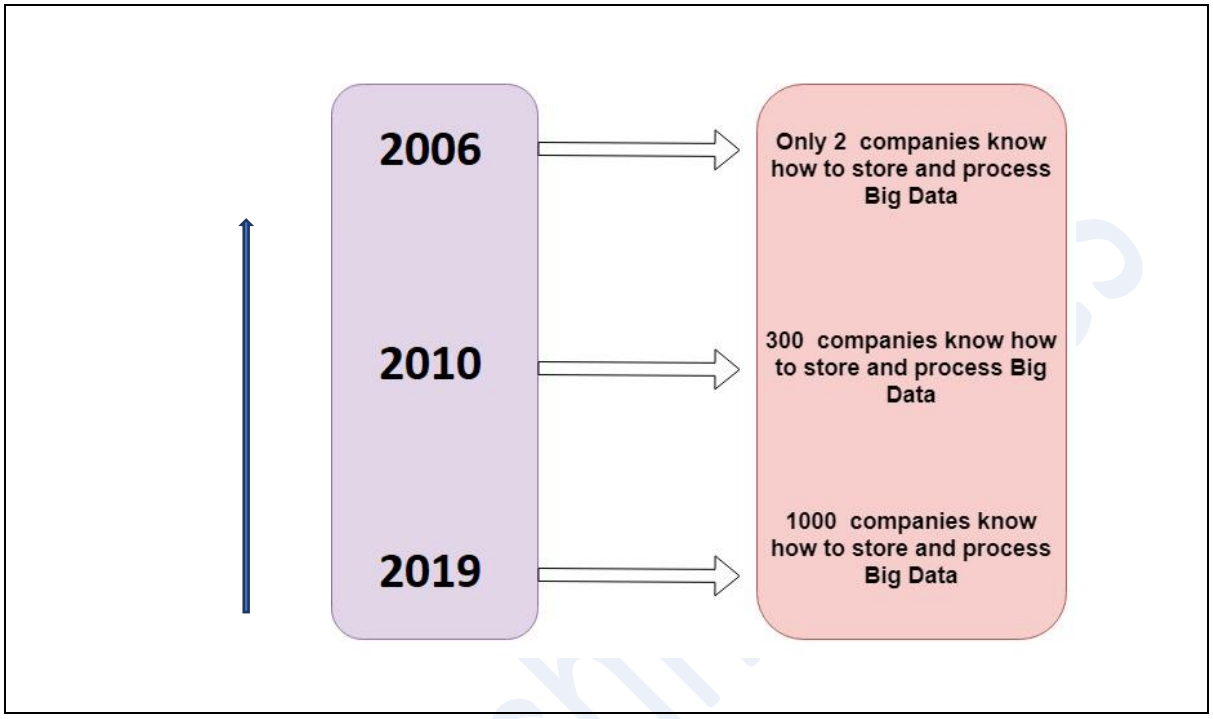
- ✓ Nireekshan : Good question Mohan.

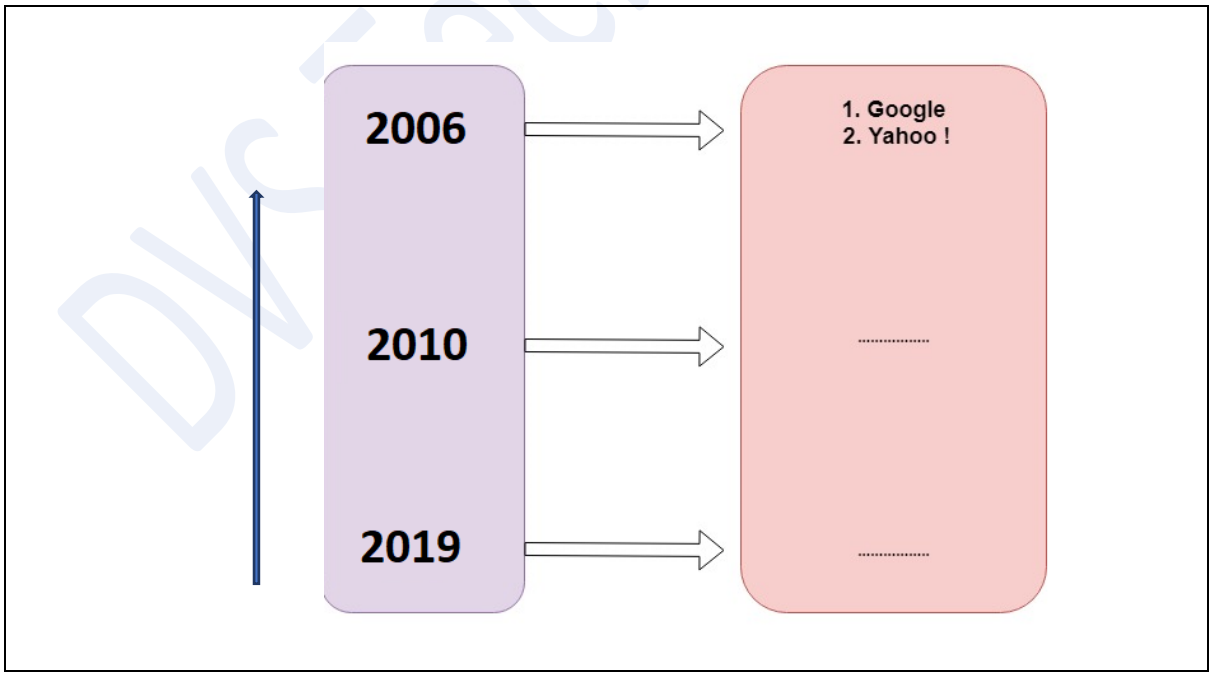
  close your eyes and imagine …

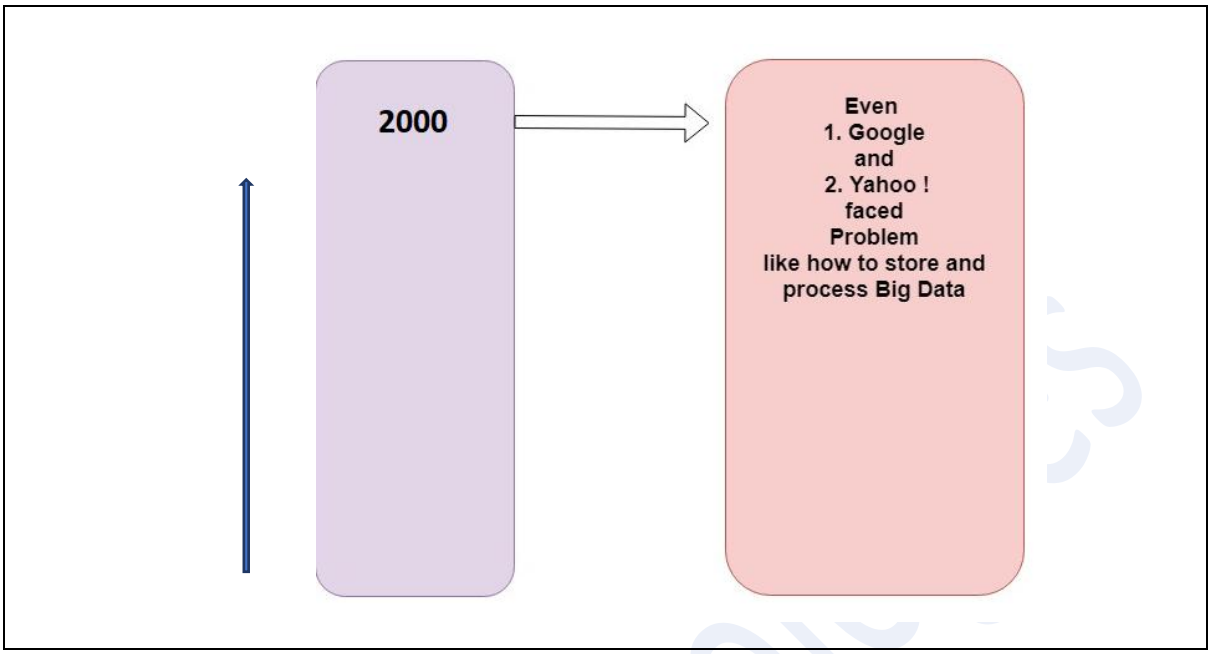
Imagination  slides

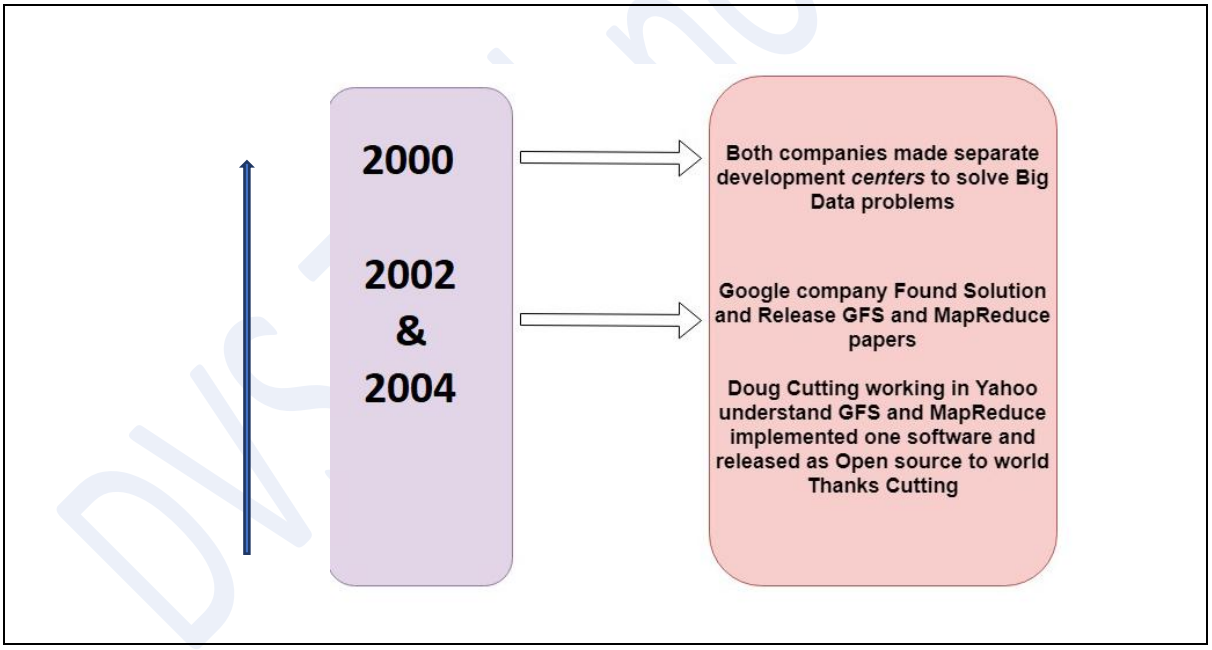
Small real story

- ✓ Once upon a time few year back ……..






2000
Even
1. Google
and
2. Yahoo !
faced
Problem
like how to store and
process Big Data

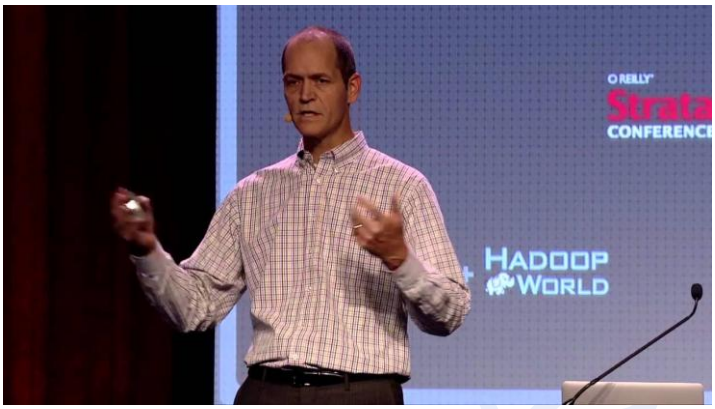

2000
2002
&
2004
Both companies made separate development centers to solve Big Data problems
Google company Found Solution and Release GFS and MapReduce papers
Doug Cutting working in Yahoo understand GFS and MapReduce implemented one software and released as Open source to world Thanks Cutting

* Alert *

- ✓ We started discussed like imagine right…
- ✓ So, now imagination is done.
- ✓ Requesting please wake up those who are in sleeping mode.

Hadoop creator

- ✓ THANKS A LOT, ☺ to *Doug cutting* for creating Hadoop and made it as open source.



Discussion between Hema and Nireekshan

- ✓ Hema : Hey Nireekshan, what we can do by using **Hadoop**?
- ✓ Nireekshan : Okay, let me explain about Hadoop.
- ✓ Nireekshan :

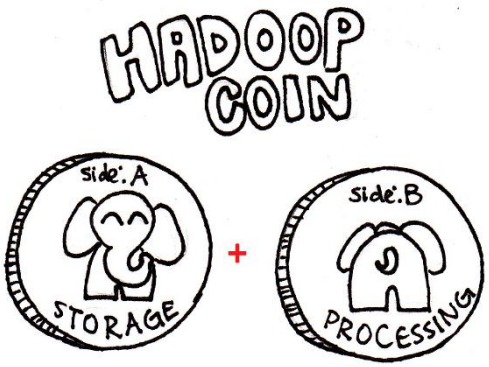
  A general example,
  Take a coin, so a coin contains how many parts?
- ✓ Hema : Two parts.
- ✓ Nireekshan :

  Good then hope this image is valid.
  Just follow below image.

Coin = 

✓ Nireekshan : So, I can define Hadoop in simple way

Hadoop =



Hadoop = Store **+** Process

Hadoop 1.0 = HDFS **+** MapReduce

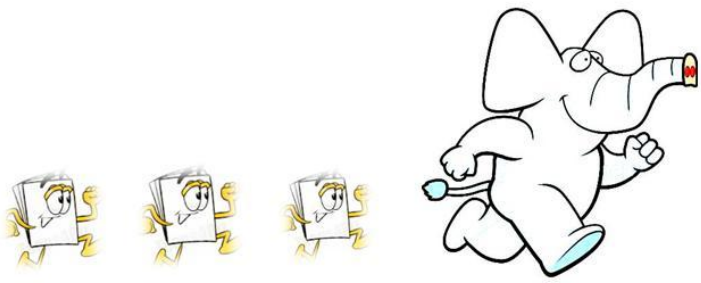
Hadoop 2.0 = HDFS **+** YARN

HDFS = **H**adoop **D**istributed **F**ile **S**ystem

YARN = **Y**et **A**nother **R**esource **N**egotiator

Resource allocation + processing

All Big data companies
started following Hadoop
☺



Advantages

- ✓ By using Hadoop, we can store and process Big Data.
- ✓ HDFS is the BEST storage file system to store Big data
- ✓ But… coming to processing ☹

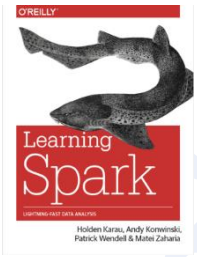
Hadoop have few limitations ☹

- ✓ I'm sorry for saying, when it comes to Hadoop processing area (MapReduce), Hadoop itself having few limitations.
- ✓ While processing big data, more read and write disk operations are required in Hadoop, so processing is very slow.

Discussion between Abhishek and Nireekshan

- ✓ Abhishek : Sorry to hear about that, Hadoop have limitations while processing.
- ✓ Abhishek : So Nireekshan, how to overcome Hadoop limitation?
- ✓ Nireekshan : Cool Dude, don't worry, Spark overcomes Hadoop limitation.
- ✓ Abhishek : Awesome Boss…☺ Jaiii Spark.
- ✓ Abhishek : So, can you please explain about Spark.
- ✓ Nireekshan : Yes, off course, Let us start

Spark creator profile



Name:

- ✓ Matei Zaharia

Matei Zaharia

- ✓ Worked with Hadoop team.
- ✓ He designed and implemented the core scheduling algorithms which are used in Apache Hadoop.

While doing PhD studies

- ✓ He created the Apache Spark.

Profession:

- ✓ Computer scientist and Assistant ***@Stanford University***.

World Record in 2014

- ✓ Zaharia holds award in the 2014 world record for BigData sorting by using Apache Spark 100 TB in **1,406** seconds

Written Spark book

- ✓ Learning Spark (O'reilly publications)

What is Apache spark?

Definition 1

- ✓ Spark is a unified analytics engine for large-scale data processing.
- ✓ Analytics: The systematic processing of data.

Definition 2

- ✓ Spark is a cluster computing platform designed to be fast and general purpose.
- ✓ Also called as, lightning fast and general-purpose cluster computing system.

Definition 3

- ✓ Spark is a framework, that allows you to solve different type of complex data problems while processing.

Purpose of Spark

- ✓ Spark is only for big data processing.

Spark is written in…

- ✓ Spark was written in Scala programming language.
- ✓ It provides high-level API, for example,
  - *Python*
  - Java
  - Scala
  - R

Can Spark integrate with Hadoop?

- ✓ Yes, spark easily integrated with Hadoop to access and store data.
- ✓ Spark can integrate with other storage systems also to process data.
  - Local file system
  - Oracle
  - Hive
  - Mysql
  - Cassandra
  - Hbase etc

What kind of files spark support?

- Text files
- SequenceFiles
- Avro
- Parquet
- Any other Hadoop InputFormat.

Is Spark depending on Hadoop?

- ✓ Spark does not require Hadoop.
- ✓ By default, spark search and store files in HDFS.

Can I install spark on windows?

- ✓ You can install spark on any operating systems like,
  - Windows
  - Linux
  - Mac etc

History of Spark

- ✓ Spark started in the year of 2009 as a research project.
- ✓ It was open sourced in 2010.
- ✓ In 2013 spark was donated to Apache Software Foundation where it became top-level Apache project in 2014.

Spark shines on where?

- ✓ In Hadoop MapReduce is a bit slow while processing large datasets because it requires more read and write operations.
- ✓ Spark offers speed while processing large datasets because it's not requires more read and write operations.

Chaithanya : Spark is Fast Fast Fast ….. in processing right, so how much Fast?

100 times fast in-memory

- ✓ Spark is up to 100 times fast than MapReduce in terms of in-memory processing

10 times fast on disk

- ✓ Spark is up to 10 times fast than MapReduce in terms of disk processing

Spark features

- ✓ Speed
- ✓ In memory computing
- ✓ Fault tolerance
- ✓ Lazy evaluation etc

1. Speed

- ✓ Spark is speed about 100x faster in memory and 10x faster on the disk.

2. In memory computing

- ✓ With in-memory processing, we can increase the processing speed.
- ✓ Here the data is being cached so we need not fetch data from the disk every time thus the time is saved.

3. Fault tolerant

- ✓ Fault refers to failure, thus fault tolerance in Apache Spark is the capability to operate and to recover loss after a failure occurs.

4. Lazy evaluation

- ✓ All the transformations are Lazy in nature, that is it does not give the result immediately.
- ✓ Once the action trigger then it's starts processing.

Data Processing terminology

	Batch	Stream	Interactive	Graph
Definition	To processing historical data.	As soon as data generated need to process. (Real time data)	Takes input from the users to run quickly.	It uses graphical representation.
Best use case	If working with very large data	If you want the results in real time mode	frequent user interaction	social network analysis

Spark - Before and After

Before

- ✓ Hadoop MapReduce can only perform batch processing.
- ✓ Apache Storm / S4 can only perform stream processing.
- ✓ Apache Impala / Apache Tez can only perform interactive processing.
- ✓ Neo4j / Apache Giraph can only perform to graph processing.

After

- ✓ Apache Spark can perform,
    - Batch processing.
    - Stream processing.
    - Interactive processing
    - Graph processing

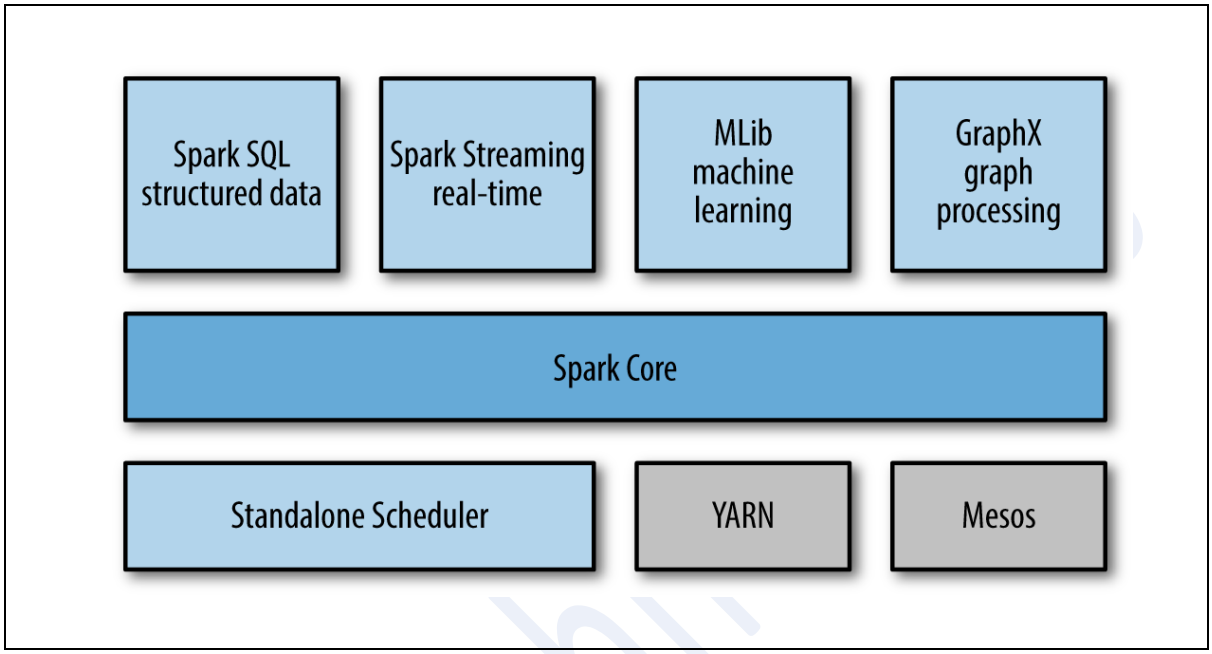
Spark is called as unified stack

- ✓ Any organization prefers to run only one software which having all stack of processing, instead of running 5–10 independent software systems.
- ✓ These costs include deployment, maintenance, testing, support etc.
- ✓ Spark is a unified stack integrated with different processing.

Make a note

- ✓ So, we can say as, Spark is a Single Point of Contact for all types of processing.

Apache Spark Components or modules



Spark Core

- ✓ Spark core module provides built in features and functionalities.
- ✓ It delivers speed by providing in-memory computation capability.
- ✓ It is the foundation of parallel and distributed processing of huge dataset.

- ✓ key features,
  - It is in-charge of essential I/O functionalities.
  - Task scheduling
  - Memory management
  - Fault recovery
  - Interacting with storage systems etc.

- ✓ Spark Core provides different types of API to define an RDD (Resilient Distributed Datasets).

- ✓ Note:
  - We have separate discussion regarding RDD, please wait few more minutes to start.

Spark SQL

- ✓ Spark SQL module is used to process structured data by using SQL and HQL.
- ✓ It supports many sources of data
    - Hive tables
    - Parquet
    - JSON
    - etc…

Spark Streaming

- ✓ Spark Streaming module enables processing of live streams of data.
- ✓ In spark, live streams are converted into micro-batches which are executed on top of spark core.

MLlib

- ✓ Spark comes with a library containing common machine learning (ML) functionality, called MLlib.
- ✓ MLlib provides multiple types of machine learning algorithms, like
    - classification
    - regression
    - clustering etc

GraphX

- ✓ GraphX is a library for manipulating graphs and performing graph-parallel computations. (e.g., a social network's friend graph)
- ✓ Spark Streaming internally uses Spark RDD concepts only.

SparkR

- ✓ It is R package that gives lightweight frontend to use Apache Spark from R.
- ✓ It allows data scientists to analyse large datasets and interactively run jobs.

Cluster Managers

- ✓ To run applications in cluster, Spark requires a cluster manager.
- ✓ Spark can run over a different type cluster manager, including,
    - Hadoop YARN
        - When we run on YARN, Spark application processes are managed by the YARN ResourceManager and NodeManager roles.
    - Standalone
        - When we run on Spark Standalone, Spark application processes are managed by Spark Master and Worker roles.

Storage Layers for Spark

- ✓ Spark is only for data processing purpose.
- ✓ There is no own or separate storage system for spark.
- ✓ So, after data processed then spark can store the data based on provided systems, it can be
    - HDFS
    - Local File System
    - Amazon S3
    - Cassandra
    - Hive
    - HBase etc.

Spark Execution Model

- ✓ Spark application execution involves runtime concepts such
    - Driver
    - Executor
    - Task
    - Job
    - Stage.
- ✓ We need to understand these concepts.

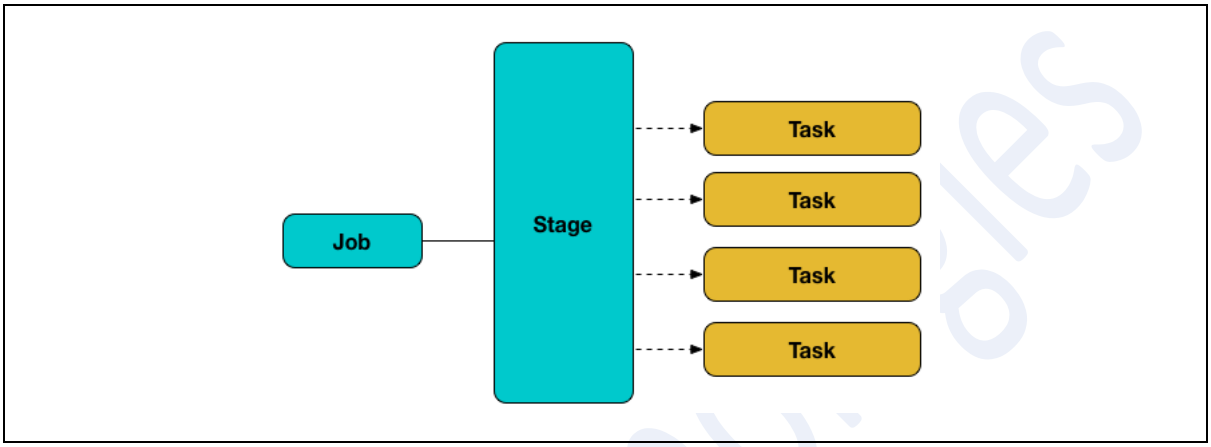
Spark Terminology

Term	Meaning
1. Application	✓ It's a program consists of a *driver program* and *executors* on the cluster.
3. Driver program	✓ The process running the main functionality of the application and creating the SparkContext.
4. Cluster manager	✓ An external service for acquiring resources on the cluster (e.g. standalone manager, YARN and Mesos).
5. Deploy mode	✓ Speaks about where the driver process runs. ✓ In "cluster" mode, the framework launches the driver inside of the cluster. ✓ In "client" mode, the submitter launches the driver outside of the cluster.
6. Worker node	✓ A computer which is functioning properly. ✓ Any computer that can run application code in the cluster.
7. Executor	✓ A process launched on worker node for processing. ✓ Executor runs tasks. ✓ Stores data in-memory or disk.
8. Task	✓ A unit of work that will be sent to one executor.
9. Job	✓ Collection of tasks.
10. Stage	✓ Each job gets divided into smaller sets of tasks called *stages*
11. DAG	✓ *DAG* full form is, Directed Acyclic Graph.

Job – Stage – Task

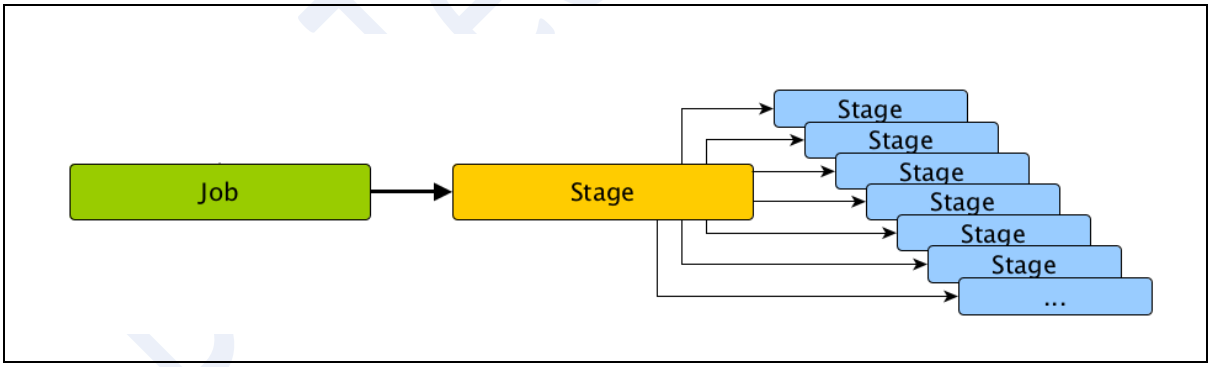
Job

- ✓ A stage is a collection of tasks



Stage

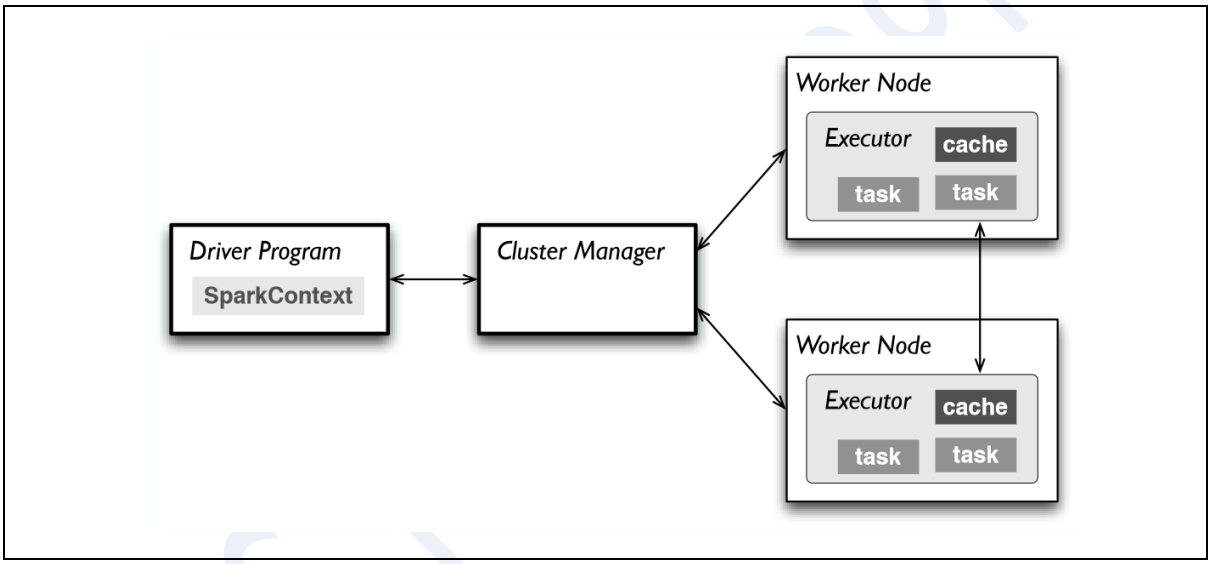
- ✓ A stage can contains several stages



Spark follows…

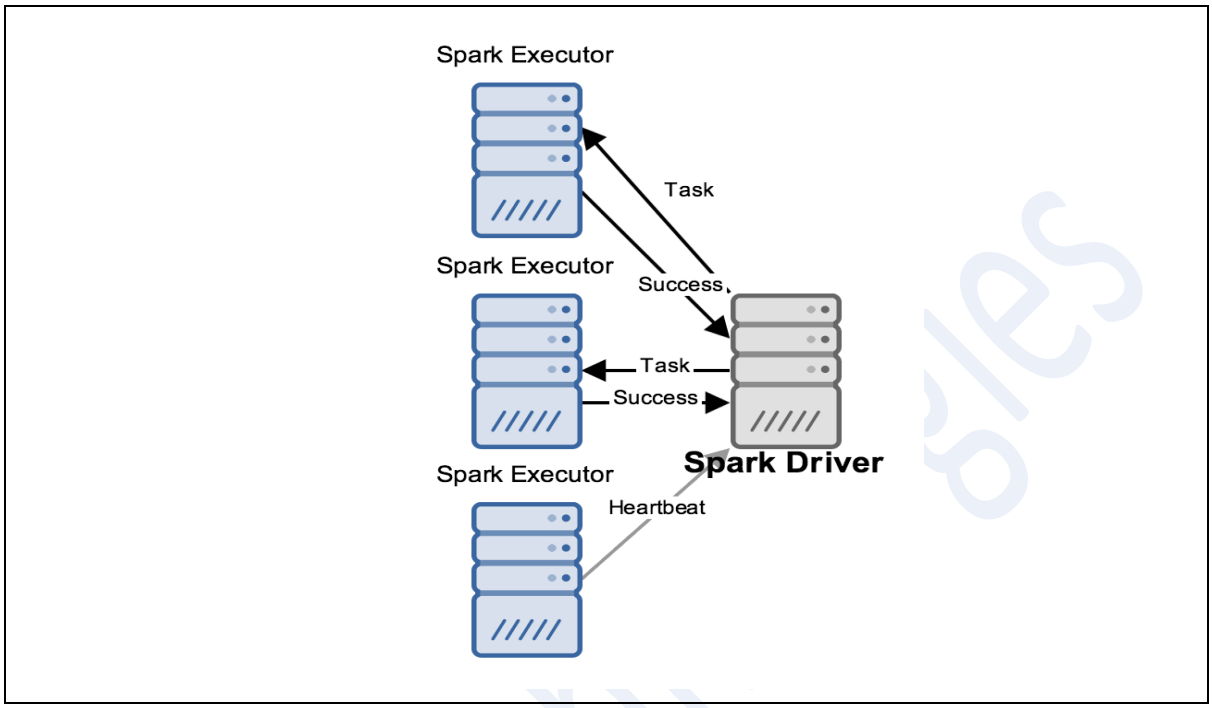
- ✓ Spark follows a master/worker(slave) architecture.
- ✓ There is a driver that talks to a single coordinator called master that manages workers in which executors run.
- ✓ Master Daemon also called as master process.
- ✓ Worker Daemon also called as Slave process.
- ✓ Spark cluster has a single Master and any number of Slaves/Workers.
- ✓ We can run them all on the same (horizontal cluster) or separate machines (vertical cluster)

Driver program



- ✓ At runtime, a Spark application maps to a single driver process and a set of executor processes distributed in cluster.
- ✓ Every Spark application contains Driver program.
    - The main purpose of driver program is, it launches various parallel operations on a cluster.
- ✓ Driver program contains our application main functionality,
    - It manages the job flow and schedules tasks.
    - It defines distributed dataset on the cluster.
    - It runs operations on distributed dataset.
- ✓ To run operations, driver programs manage number of nodes called executors.
- ✓ Driver programs communicate to cluster by using SparkContext.

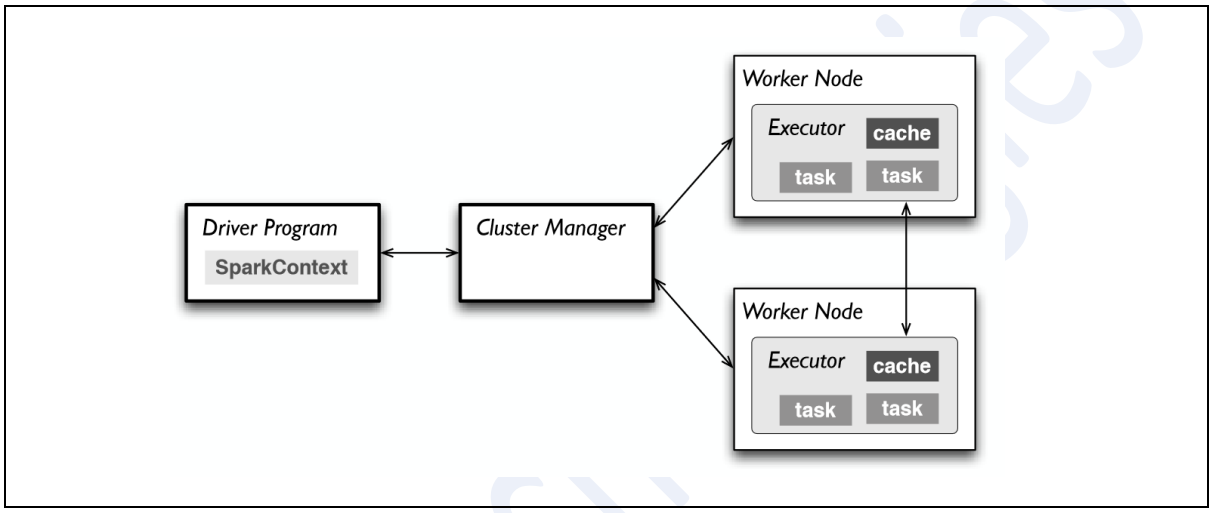
Executors

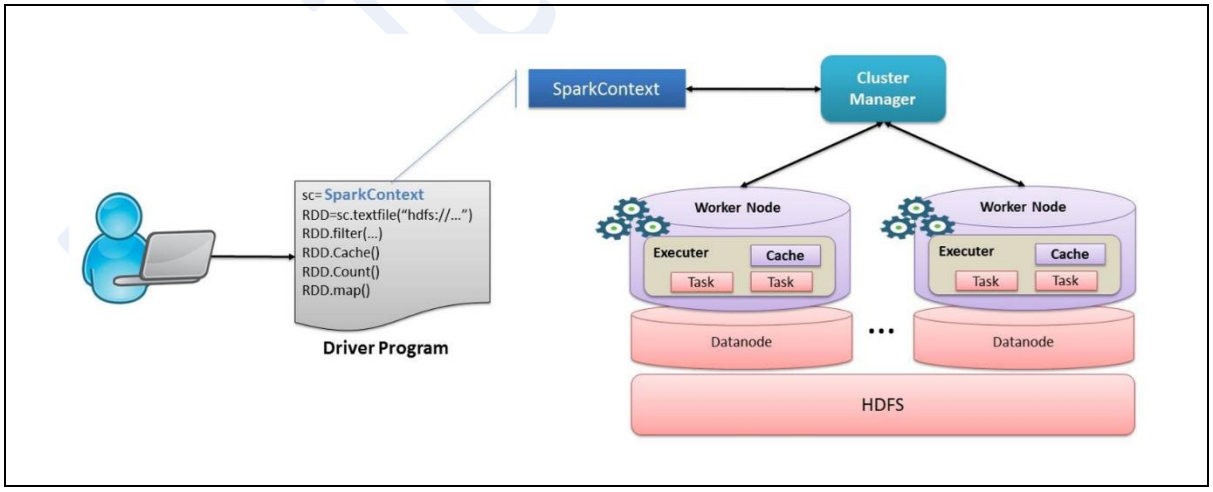


- ✓ Executors are responsible for,
  - Performing work in the form of tasks.
  - Status update.
  - Storing data in cache or disk.
- ✓ An executor has several slots for running tasks.

SparkContext

- ✓ SparkContext is the entry point of Spark functionality.
- ✓ If there is no SparkContext then there are no computations.
- ✓ The most important step of any Spark driver application is to create SparkContext object.
- ✓ SparkContext uses cluster manager to connect the cluster.
- ✓ The cluster manager can be one of these,
    - YARN
    - Spark Standalone





How many SparkContext objects can create for one application?

- ✓ Only one SparkContext object is available for whole over Spark application.
- ✓ We should not create more than one SparkContext objects.

ValueError

- ✓ If we are trying to create more than one SparkContext objects, then we will get ValueError

  - ValueError: Cannot run multiple SparkContexts at once;

Stopping SparkContext object

- ✓ We can stop currently running SparkContext object

Program Name: Creating one SparkContext object
demo1.py

```
from pyspark import SparkContext

print("Successfully imported pyspark module ")

sc = SparkContext('local[2]', 'Basic example')

print("SparkContext object is created successfully ")
```

Output

```
Successfully imported pyspark module
SparkContext object is created successfully
```

local[2] means

- ✓ Spark runs program in local and launch 2 cores locally to run spark standalone cluster.

Program Name: Creating two SparkContext objects which throws ValueError
demo2.py

```
from pyspark import SparkContext

print("Successfully imported pyspark module ")

sc1 = SparkContext('local[2]', 'Basic example')
sc2 = SparkContext('local[2]', 'Basic example')

print("SparkContext object is created successfully ")
```

ValueError

Cannot run multiple SparkContexts at once;

Stopping SparkContext

- ✓ We can stop SparkContext object by using stop () method.
- ✓ So, if we want to create more than one SparkContext object then we need to stop() existing SparkContext object.

Syntax

```
sparkContext.stop()
```

Program Name: Stopping SparkContext object and creating again SparkContext
demo3.py

```
from pyspark import SparkContext

print("Successfully imported pyspark module ")

sc1 = SparkContext('local[2]', 'Basic example')

sc1.stop()

sc2 = SparkContext('local[2]', 'Basic example')

print("SparkContext object is created successfully ")
```

Output

```
Successfully imported pyspark module
SparkContext object is created successfully
```

SparkContext responsibilities

- ✓ To get the current status of Spark Application
- ✓ To set the configuration
- ✓ To Access various services
- ✓ To Cancel a job
- ✓ To Cancel a stage
- ✓ To create RDD etc

Spark 1.x version

- ✓ SparkContext is entry point for Spark core module.
- ✓ SQLContext is entry point for Spark sql module.
- ✓ HiveContext is entry point for Spark sql module.
- ✓ StreamingContext is entry point of Spark streaming module

Conclusion of the story

- ✓ So, based on module we need to create separate Context object to work with corresponding module.

Solution in Spark 2.x

- ✓ In Spark 2.x SparkSession was introduced.
- ✓ SparkSession is single point of contact to work with all modules.
- ✓ SparkSession is the encapsulation to,
  - SparkContext,
  - SQLContext,
  - HiveContext
  - StreamingContext.

Program Name: Spark 1.x Creating SparkContext object
demo4.py

```
from pyspark import SparkContext

print("Successfully imported pyspark module ")

sc = SparkContext('local[2]', 'Basic example')

print("SparkContext object is created successfully ")
```

Output

```
Successfully imported pyspark module
SparkContext object is created successfully
```

Program Name	Spark 2.x Creating SparkSession object demo5.py
	from pyspark.sql import SparkSession sparkSession = SparkSession.builder.appName("example").getOrCreate() print("SparkSession object is created successfully")
Output	
	Successfully imported pyspark module SparkSession object is created successfully

Understanding Spark Cluster *Architecture*

- ✓ The beauty of Apache Spark is cluster Architecture.
- ✓ Before you begin your journey as a Spark programmer,
  - ○ We should have a solid understanding of the Spark internal architecture.

Anatomy of Spark Application

- ✓ A Spark application contains several components.
- ✓ All components exist whether you're running your Spark application on either,
  - ○ A single machine or
  - ○ Across a cluster of hundreds or thousands of nodes.
- ✓ Each component has a specific role in spark.

Components

- ✓ The main components of a Spark application are,
  - Driver.
  - Master.
  - Cluster Manager.
  - Executor(s).
- ✓ In upcoming chapter, we will discuss about each component

Spark clusters

- ✓ Standalone cluster
- ✓ YARN cluster

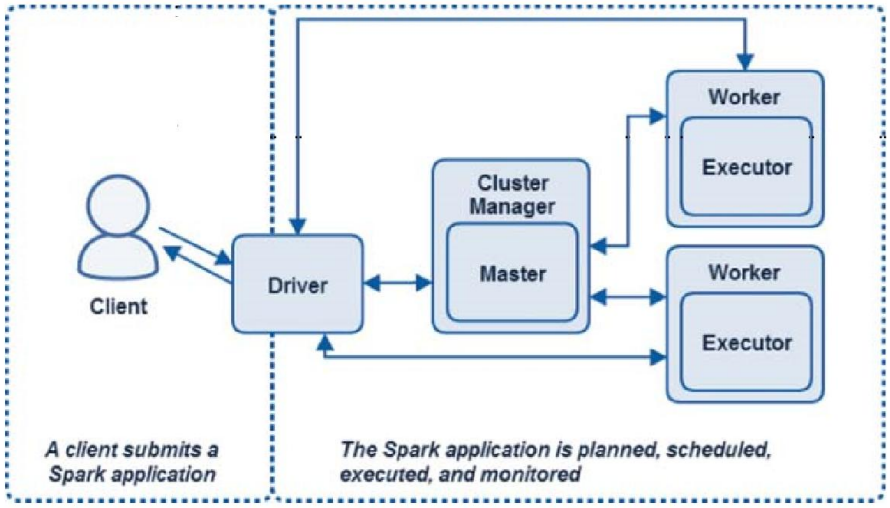
Question

Shahid : Suddenly why YARN came into the picture?

Nireekshan :

Hadoop is a very popular and common deployment platform for Spark. So, YARN is a part of Hadoop.

Spark clusters: Standalone cluster



Spark Standalone cluster application components.

- ✓ It is local mode as well
- ✓ This is useful development, testing, and debugging.
- ✓ All Spark components likes the Driver, Master, and Executor processes, run in Java virtual Machines (JVM)

What is JVM?

- ✓ JVM is a program takes responsible to convert byte code instructions (compiled file) into machine understandable format.
- ✓ When running Spark in Local mode, the application UI is available at, http://localhost:4040.

Make a note

- ✓ PySpark is built on top of Spark's Java API.

Py4J

- ✓ Py4J is a bridge between Python and Java
- ✓ Py4J enables Python programs running in a Python interpreter and connect to Java Virtual Machine.

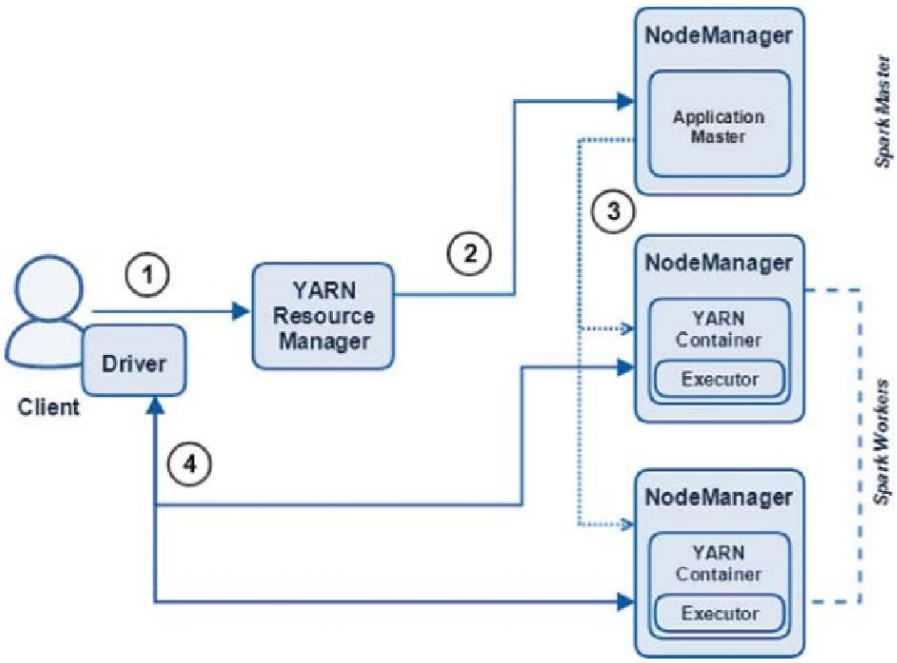
- ✓ Reference url:
  - o https://github.com/apache/spark/blob/master/python/pyspark/java_gateway.py

Spark on YARN

- ✓ There are two deployment modes to submit spark application on YARN.
  - o YARN - client mode
  - o YARN - cluster mode

YARN-Client Mode

- ✓ In YARN-client mode Driver runs in the client side.



Spark application running in YARN Client mode.

YARN-Client mode flow

1. The client submits a Spark application to the Cluster Manager (the YARN ResourceManager). The Driver process, SparkSession are created and run on the client.

2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application.

3. The ApplicationMaster requests containers to be used for Executors from the ResourceManager. With the containers assigned, the Executors spawn.

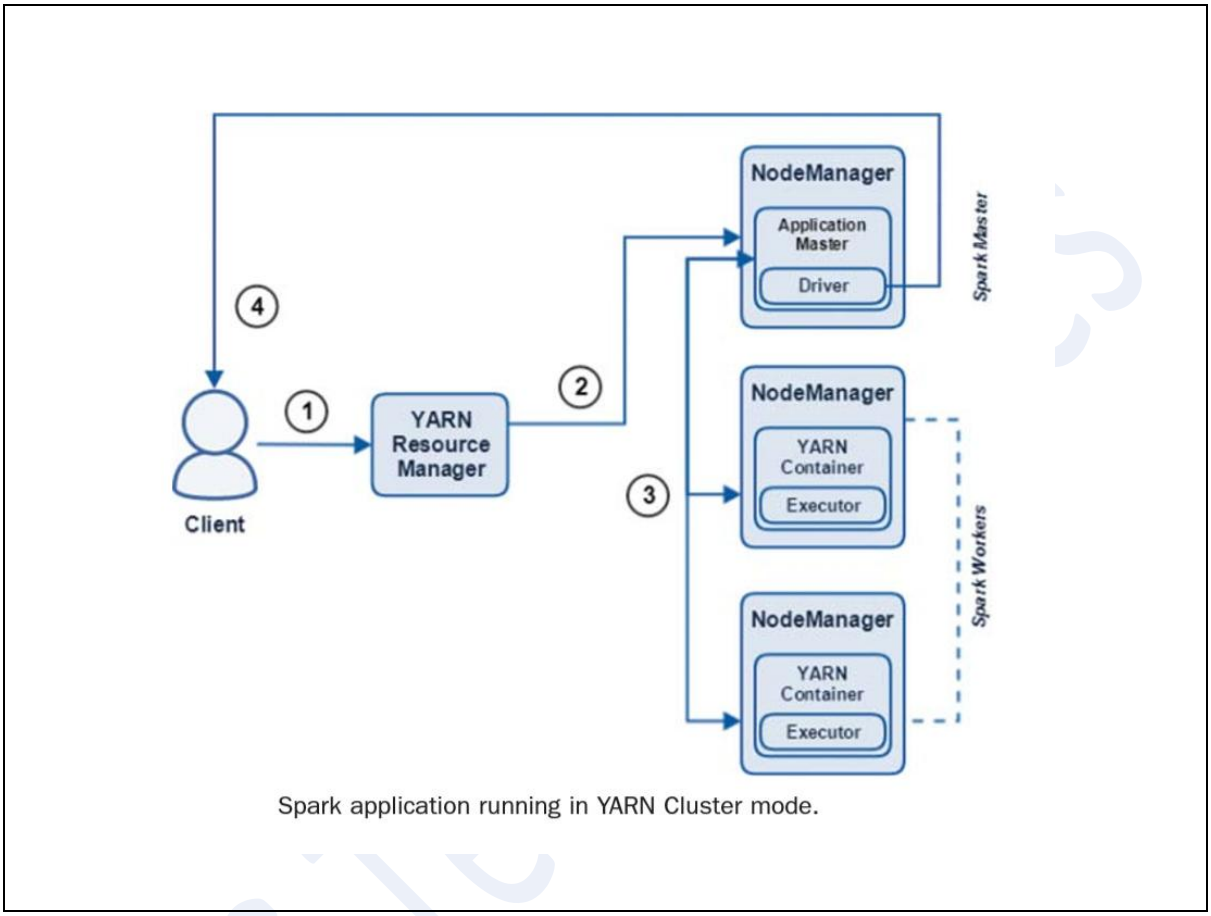
4. The Driver, located on the client, then communicates with the Executors for processing of tasks and stages of the Spark program. The Driver returns the progress, results, and status to the client.

Note

- ✓ The Client deployment mode is the simplest mode to use.

YARN-Cluster Mode

- ✓ In Yarn-cluster mode, the Spark Driver runs inside an Application Master process
- ✓ Yarn-cluster mode is used for production purpose.



Spark application running in YARN Cluster mode.

YARN-cluster mode flow

1. The client, a user process that calls spark-submit, submits a Spark application to the Cluster Manager (the YARN ResourceManager).

2. The ResourceManager assigns an ApplicationMaster (the Spark Master) for the application. The Driver process is created on the same cluster node.

3. The ApplicationMaster requests containers for Executors from the ResourceManager. Executors are spawned within the containers allocated to the ApplicationMaster by the ResourceManager. The Driver then communicates with the Executors for processing of tasks and stages of the Spark program.

4. The Driver, running on a node in the cluster, returns progress, results, and status to the client.

Importance of RDD

What is RDD?

Importance of RDD

- ✓ In Spark all work is expressed as,
  - Either creating new RDDs.
  - Transforming existing RDDs.
  - Calling operations on RDDs to compute a result.

1. What is RDD?

- ✓ The full form of RDD is "Resilient Distributed Dataset".
- ✓ It is the fundamental data structure in Spark.

Definition 1

- ✓ A group of items distributed in cluster.

Definition 2

- ✓ A collection of partitions in cluster.

Definition 3

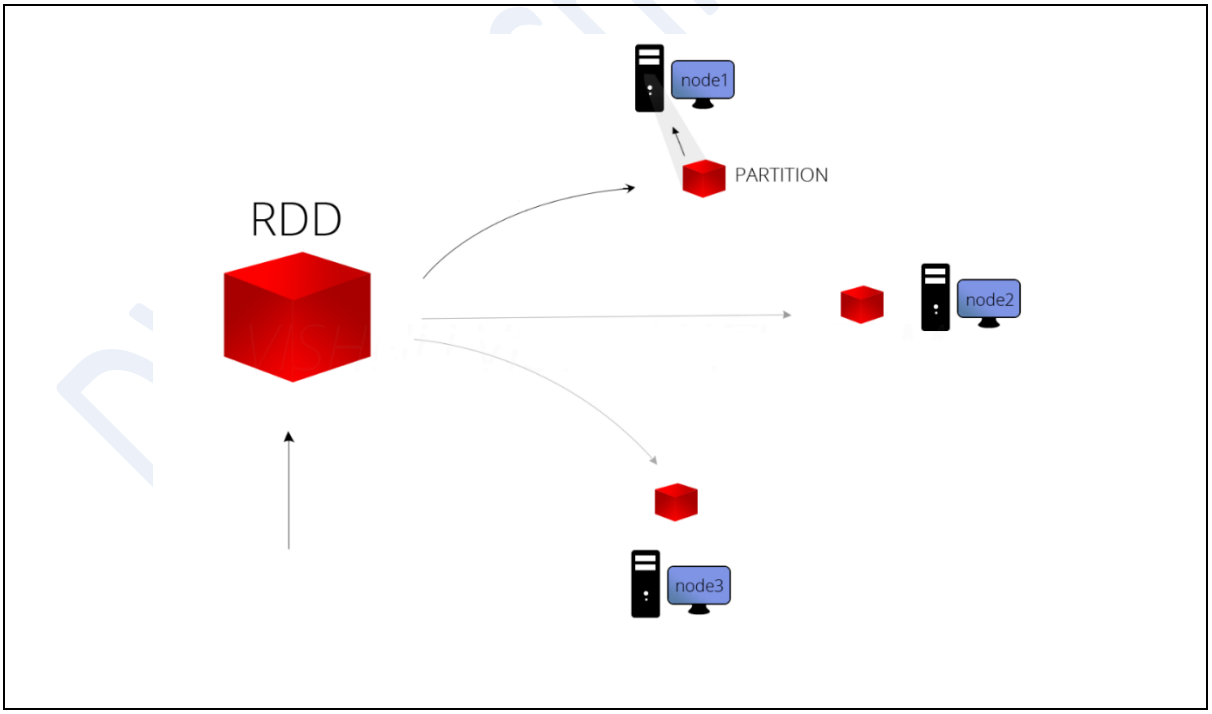
- ✓ RDD is immutable means once created then cannot modified.
- ✓ RDD is a collection of objects distributed in many nodes that can be processed in parallel.

RDD full form:

Resilient	Distributed	Dataset
Fault-tolerant with the help of RDD lineage graph (DAG), able to re-compute missing or damaged partitions due to node failures.	Data distributed on multiple nodes.	Load the data-set externally which can be either JSON file, CSV file, text file or database.

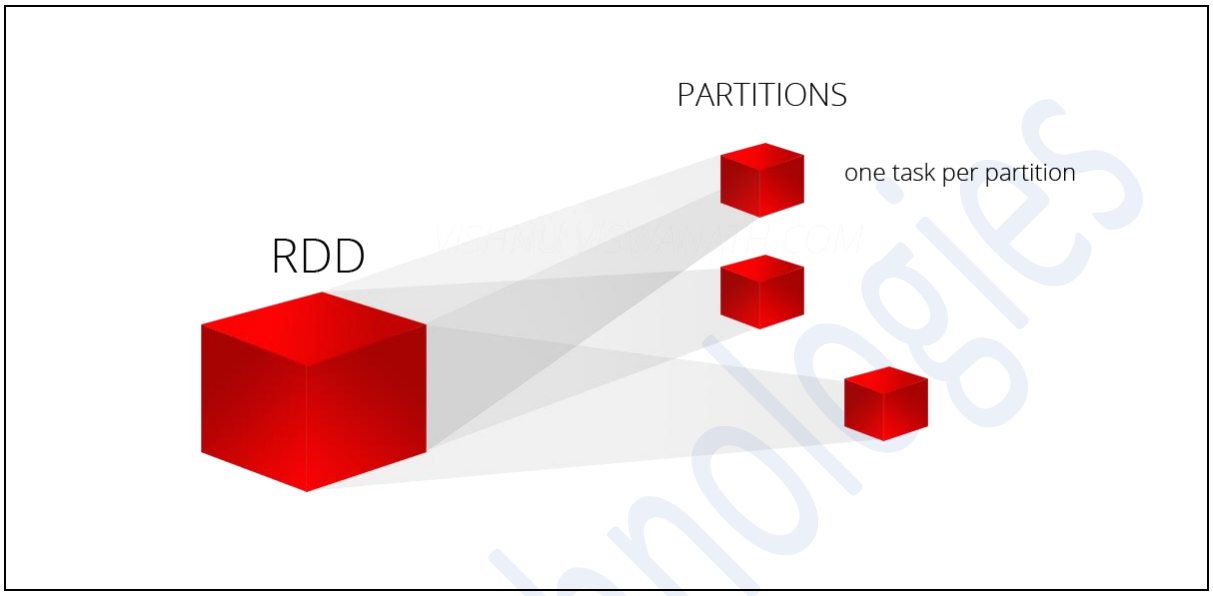
RDD computation

- ✓ RDD is a collection of records spread over one or many partitions.
- ✓ So, those partitions are used to compute parallelly on different nodes of the cluster.



Partitions in RDD

- ✓ RDD divided into smaller chunks called as Partitions.
- ✓ Partitions that helps parallelize distributed data processing.
- ✓ RDDs get partitioned automatically without programmer intervention



- ✓ When you execute some action on RDD then a task is launched per partition.
- ✓ So, it means more partitions brings more parallelism.
- ✓ Spark achieves parallelism by using partitions.
- ✓ Spark automatically decides the number of partitions to RDD.
- ✓ We can specify the number of partitions to RDD while creating if want customization.

Creating RDD

- ✓ There are two ways to create RDDs in Spark such as,
    - Parallelized Collections.
    - From External Datasets.

Parallelized Collections.

- ✓ By using parallelize() method we can create RDD.
- ✓ parallelize() is predefined method exists in SparkContext class.
- ✓ So, to call parallelize() method we need to create object SparkContext

Program Name: Creating rdd and applying transformations and actions
demo6.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

l = [1, 2, 3, 4, 5]

listRdd = sc.parallelize(l)

print(listRdd)
print(listRdd.collect())
```

Output

```
ParallelCollectionRDD[0] at parallelize at PythonRDD

[1, 2, 3, 4, 5]
```

Make a note

- ✓ In the above program collect() is action.
- ✓ In upcoming slides we will see transformations and actions with examples

Caching

- ✓ Spark RDD can be cached and partitioned.
- ✓ If we want to reuse existing RDD then good approach to cache the RDD.

Persistent

- ✓ Programmers can use persist method to store RDD for future operations.
- ✓ By default, spark uses in-memory to store RDD, but it can spill them to disk if not enough memory in RAM.

Fault-Recovery Mechanism

- ✓ The lost data can be easily recovered by using Lineage Graph.
- ✓ Each transformation creates new RDD.
- ✓ RDD is immutable so it's easy to recover.

If RAM is inefficient to store RDD then where it stores?

- ✓ If there is no space to store RDD in RAM, then RDD will be stored in disk.
- ✓ Spark will user RAM and Hard disk to store RDD.

Tip

- ✓ Don't be in assumption like it RDDs will be store only in RAM or DISK.
- ✓ Hope you are clear about these points.

RDD features

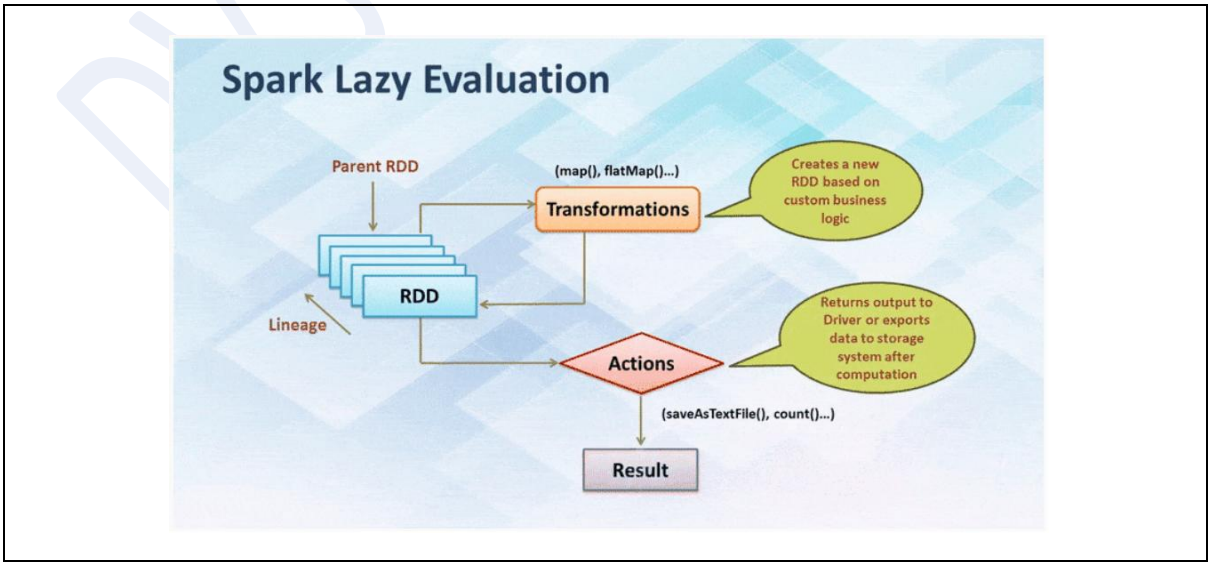
1. In-memory Computation
2. Lazy Evaluations
3. Fault Tolerance
4. Immutability
5. Partition and etc…

1. In-memory Computation

- ✓ RDD stores intermediate results in RAM to compute instead of disk.

2. Lazy Evaluations

- ✓ All transformations are lazy, means they do not compute results immediately.
- ✓ When action got triggered then only transformations compute results.



3. Fault Tolerance

- ✓ RDDs are automatically rebuilt on lost data by using lineage graph.

4. Immutability

- ✓ Once the RDD created then it is constant, means cannot modify.
- ✓ Immutability means, if we are trying to apply operations on existing RDD then with those changes new RDD will be created.

5. Partition

- ✓ RDD is logically partitioned in cluster.

Spark RDD Operations

- ✓ RDD supports two types of operations:
    - Transformations.
    - Actions.

6.1. Transformations

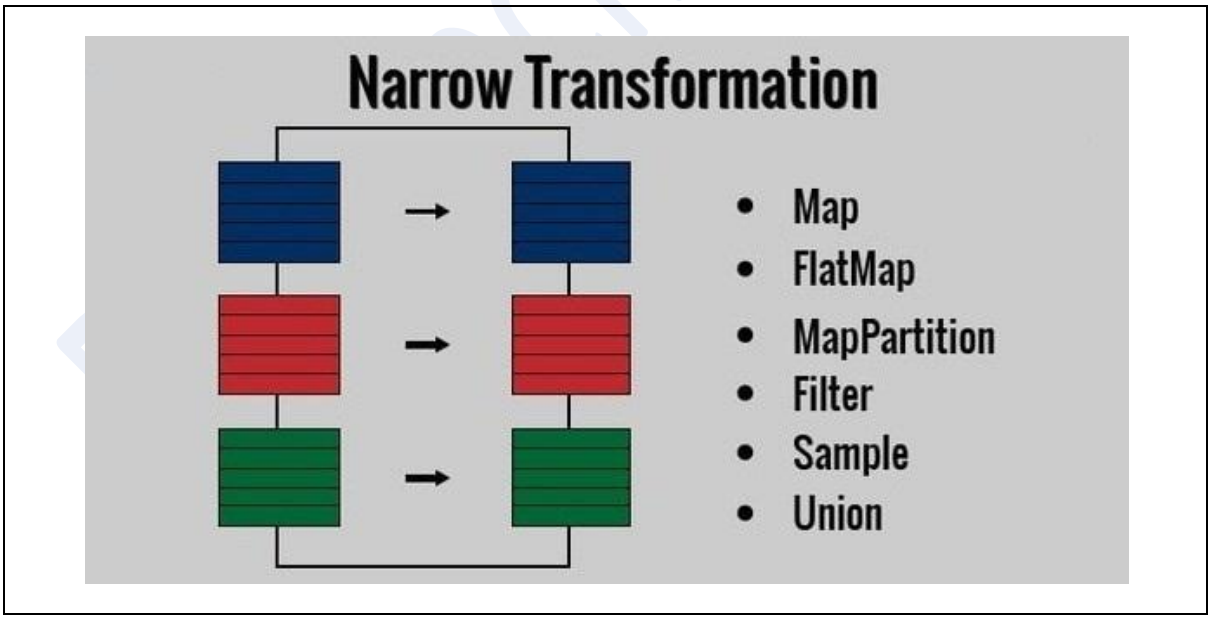
- ✓ Transformations are functions that take an RDD as the input and creates one or more new RDDs as the output.
- ✓ Transformations are executed when an action got triggered.
- ✓ Transformations are lazy operations.
- ✓ Transformations do not change the input RDD because RDDs are immutable.

Types of Transformations

- ✓ There are two kinds of transformations,
    - Narrow transformations.
    - Wide transformations.

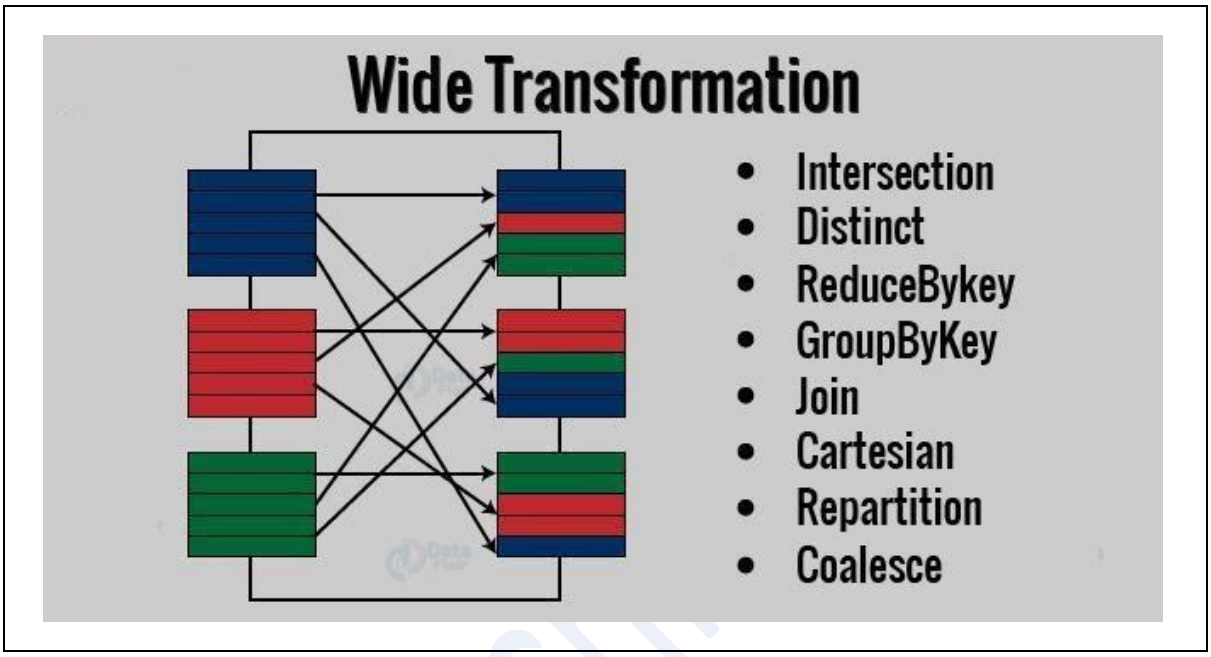
Narrow Transformations

- ✓ It is the result of map, flatMap, MapPartition and filter methods.
- ✓ In Narrow transformations, data is from a single partition only.
- ✓ Spark groups narrow transformations as a stage known as pipelining.



Wide Transformations

- ✓ It is the result of groupByKey() and reduceByKey() functions.
- ✓ The output of RDD partition may come from many parent partitions.
- ✓ Wide transformations are also known as shuffle transformations.



Actions

- ✓ An action is a result of RDD computations.
- ✓ When action got triggered then,
  - It loads data into original RDD by using lineage graph.
  - Executes all intermediate transformations and return final results to Driver program or write it out to file system.
- ✓ An Action is one of the ways to send result from executors to the driver.
- ✓ Few examples of actions
  - first (),
  - take (),
  - reduce (),
  - collect (),
  - count () etc.

Limitation of RDD

No inbuilt optimization engine

- ✓ RDDs are unfit when working with structured data.
- ✓ RDDs cannot take advantages of catalyst optimizer and Tungsten execution engine.
- ✓ Developers need to optimize each RDD based on its attributes.

RDDs are unable to handle structured data

- ✓ RDDs are unable to infer schema for data.
- ✓ While working with RDD, developers need to write the schema explicitly.

Performance limitation

- ✓ RDDs involve the overhead of Garbage Collection

RDD Operations
Transformations & Actions

RDD transformations on single RDD

Transformation name	Purpose
map()	✓ Apply on each element in RDD and return an RDD of the result.
filter()	✓ Return an RDD consisting of only elements that pass the condition.
distinct()	✓ Remove duplicates.

Program Name: map() transformation in pyspark
demo7.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

items_cost = [1000, 2000, 3000, 3000]
items_cost_rdd = sc.parallelize(items_cost)
items_cost_gst = items_cost_rdd.map(lambda x: x + 5).collect()

print("Before GST each item cost: ", items_cost)
print("After GST each item cost: ", items_cost_gst)
```

Output

```
Before GST each item cost: [1000, 2000, 3000, 3000]
After GST each item cost: [1005, 2005, 3005, 3005]
```

Program Name: map() function in python
demo8.py

```
items_costs = [1000, 2000, 3000, 3000]
print("Before GST each item cost: ", items_costs)
items_costs_gst = map(lambda x: x+5, items_costs)
print("After GST each item cost: ", list(items_costs_gst))
```

Output

```
Before GST each item cost: [1000, 2000, 3000, 3000]
After GST each item cost: [1005, 2005, 3005, 3005]
```

What is the difference between Python map() function and PySpark map() method

Python map() function

- ✓ This function works in non-cluster mode means with single node

PySpark map() method

- ✓ This method works with non-cluster mode and cluster(multiple nodes) as well.

Program Name: filter() transformation
demo9.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

items_cost = [1000, 2000, 3000, 3000, 4000, 5000]
items_cost_rdd = sc.parallelize(items_cost)
items_cost_lt2000 = items_cost_rdd.filter(lambda x: x > 2000).collect()

print("Each transaction cost is: ", items_cost)
print("filter less than 2000 txns : ", items_cost_lt2000)
```

Output

```
Each transaction cost is: [1000, 2000, 3000, 3000, 4000, 5000]
Filter greater than 2000 txns :  [3000, 3000, 4000, 5000]
```

Program Name: distinct() transformation
demo10.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

cust_ids = ['nirekshan@gmail.com', 'hema@gmail.com', 'mohan@gmail.com',
'nirekshan@gmail.com', 'mohan@gmail.com']

cust_ids_rdd = sc.parallelize(cust_ids)
cust_ids_dist = cust_ids_rdd.distinct().collect()

print("All customers ids: ", cust_ids)
print("Distinct customer ids : ", cust_ids_dist)
```

Output

```
All customers ids:
['nirekshan@gmail.com', 'hema@gmail.com', 'mohan@gmail.com',
'nirekshan@gmail.com', 'mohan@gmail.com']

Distinct customer ids:
['nirekshan@gmail.com', 'hema@gmail.com', 'mohan@gmail.com']
```

Program Name: distinct() transformation
demo11.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

cust_ids = [1, 2, 3, 3, 4, 5, 5]
cust_ids_rdd = sc.parallelize(cust_ids)
cust_ids_dist = cust_ids_rdd.distinct().collect()

print("All customers ids: ", cust_ids)
print("Removing duplicates in customer ids : ", cust_ids_dist)
```

Output

```
All customers ids: [1, 2, 3, 3, 4, 5, 5]
Removing duplicates in customer ids: [2, 4, 1, 3, 5]
```

Two RDD transformations

Transformation name	Purpose
union()	✓ It combines both RDDs and return single RDD
intersection()	✓ RDD containing only elements found in both RDDs
subtract()	✓ Return each value in self that is not contained in other.
cartesian()	✓ Cartesian product with the other RDD

Program Name: union() transformation
demo12.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

jan_txns = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000]
feb_txns = [1000, 2000, 13000, 14000, 15000, 6000, 7000, 8000]

jan_txns_rdd=sc.parallelize(jan_txns)
feb_txns_rdd=sc.parallelize(feb_txns)

total_txns_rdd= jan_txns_rdd.union(feb_txns_rdd)

print(total_txns_rdd.collect())
```

Output

```
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 1000, 2000, 13000, 14000,
15000, 6000, 7000, 8000]
```

Program Name: union() transformation
demo13.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

jan_txns = [("cust1", 10000), ("cust2", 20000), ("cust3", 30000), ("cust4",
40000)]

feb_txns = [("cust1", 11000), ("cust2", 22000), ("cust3", 33000), ("cust4",
44000), ("cust5", 50000), ("cust6", 60000)]

jan_txns_rdd=sc.parallelize(jan_txns)
feb_txns_rdd=sc.parallelize(feb_txns)

union_rdd = jan_txns_rdd.union(feb_txns_rdd)

print(union_rdd.collect())
```

Output

```
[('cust1', 10000),
('cust2', 20000),
('cust3', 30000),
('cust4', 40000),
('cust1', 11000),
('cust2', 22000),
('cust3', 33000),
```

```
('cust4', 44000),
('cust5', 50000),
('cust6', 60000)]
```

Program Name: union() transformation
demo14.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

jan_txns = [("cust1", 10000), ("cust2", 20000), ("cust3", 30000), ("cust4",
40000)]

feb_txns = [("cust1", 11000), ("cust2", 22000), ("cust3", 33000), ("cust4",
44000), ("cust5", 50000), ("cust6", 60000)]

jan_txns_rdd=sc.parallelize(jan_txns)
feb_txns_rdd=sc.parallelize(feb_txns)

union_rdd = jan_txns_rdd.union(feb_txns_rdd)

print(union_rdd.collect())

total_txns_by_customers = union_rdd.reduceByKey(lambda x, y: x+y)

print(total_txns_by_customers.collect())
```

Output

```
[('cust1', 10000),
('cust2', 20000),
('cust3', 30000),
('cust4', 40000),
('cust1', 11000),
('cust2', 22000),
('cust3', 33000),
('cust4', 44000),
('cust5', 50000),
('cust6', 60000)]


[('cust4', 84000),
('cust1', 21000),
('cust3', 63000),
('cust5', 50000),
('cust6', 60000),
('cust2', 42000)]
```

Program Name: intersection() transformation
demo15.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

jan_cust_txns = ["cust1", "cust2", "cust3","cust4"]
feb_cust_txns = ["cust1", "cust2", "cust3","cust4", "cust5", "cust6", "cust7"]

jan_cust_txns_rdd=sc.parallelize(jan_cust_txns)
feb_cust_txns_rdd=sc.parallelize(feb_cust_txns)

intersection_rdd = jan_cust_txns_rdd.intersection(feb_cust_txns_rdd)

print(intersection_rdd.collect())
```

Output

```
['cust4', 'cust1', 'cust3', 'cust2']
```

Program Name: subtract() transformation
demo16.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

jan_cust_txns = ["cust1", "cust2", "cust3","cust4", "cust5", "cust6", "cust7"]
feb_cust_txns = ["cust1", "cust2", "cust3", "cust4"]

jan_cust_txns_rdd=sc.parallelize(jan_cust_txns)
feb_cust_txns_rdd=sc.parallelize(feb_cust_txns)

print(jan_cust_txns_rdd)
print(feb_cust_txns_rdd)

subtract_rdd = jeb_cust_txns_rdd.subtract(feb_cust_txns_rdd)

print(subtract_rdd.collect())
```

Output

```
['cust7', 'cust5', 'cust6']
```

Action on RDD

Action name	Purpose
collect()	✓ Return all elements from the RDD
count()	✓ Number of elements in the RDD.
countByValue()	✓ Number of times each element occurs in the RDD.
reduce()	✓ Combine the elements of the RDD together in parallel (e.g., sum)
foreach()	✓ Apply the provided function to each element of the RDD.

Program Name: collect() action
demo17.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

emp_names = ['Senthil', 'Jeeva', 'Vignesh', 'Amritha', 'Aveena', 'Chaithanya']
emp_names_rdd = sc.parallelize(emp_names)

print(emp_names_rdd)

print(emp_names_rdd.collect())
```

Output

```
ParallelCollectionRDD[0] at parallelize at PythonRDD

['Senthil', 'Jeeva', 'Vignesh', 'Amritha', 'Aveena', 'Chaithanya']
```

Program Name: count() action
demo18.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

txns = [1000, 2000, 3000, 4000, 5000, 6000]
txns_rdd = sc.parallelize(txns)

print(txns_rdd.count())
```

Output

```
6
```

Program Name: countByValue() action
demo19.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

txns = [100, 100, 300, 400, 500, 500]
txns_rdd = sc.parallelize(txns)

print(txns_rdd.countByValue().items())
```

Output

```
[(100, 2), (300, 1), (400, 1), (500, 2)]
```

Program Name: reduce() action
demo20.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

txns = [1000, 2000, 3000, 4000, 5000, 6000]
txns_rdd = sc.parallelize(txns)
total_txns_amount=txns_rdd.reduce(lambda x, y : x+y)

print(total_txns_amount)
```

Output

```
21000
```

Transformation on one pair RDD

Transformation name	Purpose
reduceByKey()	✓ Combine values with the same key
groupByKey()	✓ Group values with the same key.
mapValues()	✓ Apply a function to each value of a pair RDD without changing the key
keys()	✓ Return an RDD of just the keys
values()	✓ Return an RDD of just the values.
sortByKey()	✓ Return an RDD sorted by the key

Program Name: reduceByKey() transformation
demo21.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

custmers_txns = [("cust1", 10000), ("cust2", 20000), ("cust3", 30000),
("cust1", 40000)]

custmers_txns_rdd = sc.parallelize(custmers_txns)

each_cust_total_txns_amt=custmers_txns_rdd.reduceByKey(lambda x, y : x+y)

print(each_cust_total_txns_amt.collect())
```

Output

```
[('cust1', 50000), ('cust3', 30000), ('cust2', 20000)]
```

Program Name: groupByKey() transformation
demo22.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

custmers_txns = [("cust1", 10000), ("cust2", 20000), ("cust3", 30000),
("cust1", 40000)]

custmers_txns_rdd = sc.parallelize(custmers_txns)

each_cust_txns_amt=custmers_txns_rdd.groupByKey().mapValues(list)

print(each_cust_txns_amt.collect())
```

Output

```
[('cust1', [10000, 40000]), ('cust3', [30000]), ('cust2', [20000])]
```

Program Name: keys() transformation
demo23.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

each_item_cost = [("item1", 10000), ("item2", 20000), ("item3", 30000)]
each_item_cost_rdd = sc.parallelize(each_item_cost)
each_item_key = each_item_cost_rdd.keys()

print(each_item_key.collect())
```

Output

```
['item1', 'item2', 'item3']
```

Program Name: values() transformation
demo24.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

each_item_cost = [("item1", 10000), ("item2", 20000), ("item3", 30000)]
each_item_cost_rdd = sc.parallelize(each_item_cost)
each_item_key = each_item_cost_rdd.values()

print(each_item_key.collect())
```

Output

```
[10000, 20000, 30000]
```

Program Name: sortByKey transformation
demo25.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

each_item_cost = [("z", 10000), ("x", 30000), ("y", 20000)]
each_item_cost_rdd = sc.parallelize(each_item_cost)
result = each_item_cost_rdd.sortByKey()

print(result.collect())
```

Output

```
[('x', 30000), ('y', 20000), ('z', 10000)]
```

Program Name: min()
demo26.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

each_item_cost = [1000, 2000, 3000, 4000]
each_item_cost_rdd = sc.parallelize(each_item_cost)
print(each_item_cost_rdd.min())
```

Output

```
1000
```

Program Name: max()
demo27.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

each_item_cost = [1000, 2000, 3000, 4000]
each_item_cost_rdd = sc.parallelize(each_item_cost)
print(each_item_cost_rdd.max())
```

Output

```
4000
```

Coalesce and Repartition

- ✓ Spark splits data into partitions and executes computations on the partitions in parallel.

Coalesce

- ✓ The coalesce method reduces the number of partitions

Increasing partitions

- ✓ If we increase the number of partitions with coalesce, but it won't work!

Repartition

- ✓ The repartition method can be used to either increase or decrease the number of partitions.

Program Name: coalesce() reducing partitions
demo28.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

rdd=sc.parallelize([1, 2, 3, 4, 5], 3).glom()
print(rdd.collect())

colrdd=sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom()
print(colrdd.collect())

print("Before coalesce num of partitions: ",len(rdd.collect()))
print("After coalesce num of partitions: ",len(colrdd.collect()))
```

Output

```
[[1], [2, 3], [4, 5]]

[[1, 2, 3, 4, 5]]

Before coalesce num of partitions:  3
After coalesce num of partitions:  1
```

Program Name: coalesce() increasing partitions but not possible by coalesce
demo29.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

rdd=sc.parallelize([1, 2, 3, 4, 5], 3).glom()
print(rdd.collect())

colrdd=sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(5).glom()
print(colrdd.collect())

print("Before coalesce num of partitions: ",len(rdd.collect()))
print("After coalesce num of partitions: ",len(colrdd.collect()))
```

Output

```
[[1], [2, 3], [4, 5]]

[[1], [2, 3], [4, 5]]

Before coalesce num of partitions:  3
After coalesce num of partitions:  3
```

Program Name: repartition(), increasing number of partitions
demo30.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'pyspark tutorial')

rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
print(rdd.glom().collect())

rep2rdd=rdd.repartition(2).glom().collect()
rep10rdd=rdd.repartition(10).glom().collect()

print("reduced partitions: ",len(rep2rdd))
print("increased partitions: ",len(rep10rdd))
```

Output

```
[[1], [2, 3], [4, 5], [6, 7]]

reduced partitions: 2
increased partitions: 10
```

Program Name: creating RDD and checking number of partitions
demo31.py

```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

l = [1, 2, 3, 4, 5, 6, 7, 8, 9]

rdd= sc.parallelize(l)
print(rdd.collect())
print(rdd.getNumPartitions())
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
2
```

Program Name: glom(): creating RDD and checking number each partition elements
demo32.py

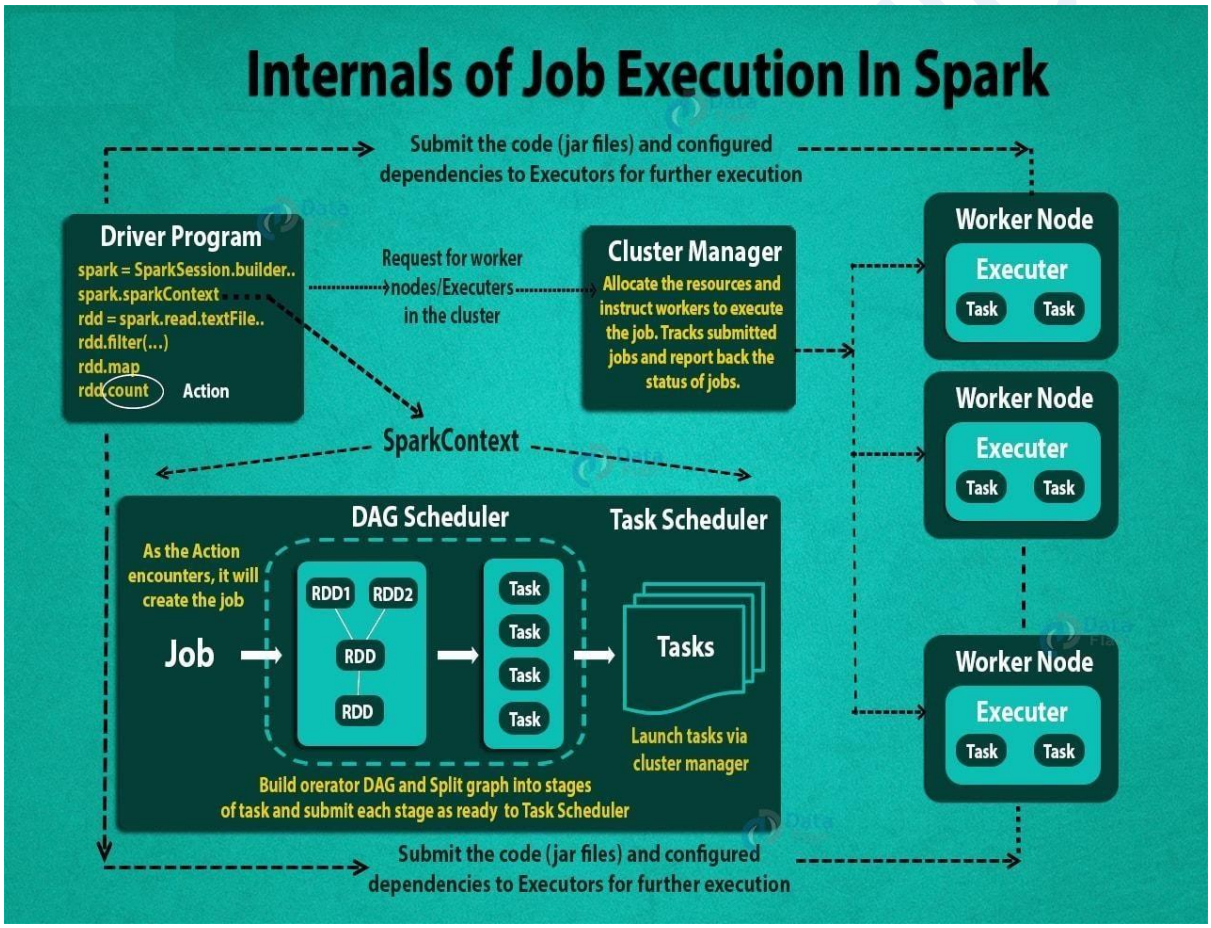
```
from pyspark import SparkContext
sc = SparkContext('local[2]', 'map example')

l = [1, 2, 3, 4, 5, 6, 7, 8, 9]

rdd= sc.parallelize(l)
print(rdd.collect())
print(rdd.glom().collect())
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[[1, 2, 3, 4], [5, 6, 7, 8, 9]]
```

Internals of Job execution in Spark

- ✓ By using spark-submit, the user submits the application.
- ✓ In spark-submit, the main running file specified by the user will be called.
- ✓ 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.



Reference urls:

PySpark examples:

http://spark.apache.org/docs/latest/api/python/pyspark.html#subpackages

https://github.com/tirthajyoti/Spark-with-Python/blob/master/SparkContext%20and%20RDD%20Basics.ipynb

https://github.com/awantik/pyspark-tutorial/wiki/Transformations

Spark SQL

Introduction

- ✓ Spark SQL is a separate module.
- ✓ This module runs on top of Spark Core module.
- ✓ The purpose spark sql module is to process structured data.
- ✓ We can write SQL queries inside Spark programs.

Spark SQL programs

- ✓ Spark SQL programs we can write by using
    - DataFrame API
    - SQL etc

Spark SQL Features

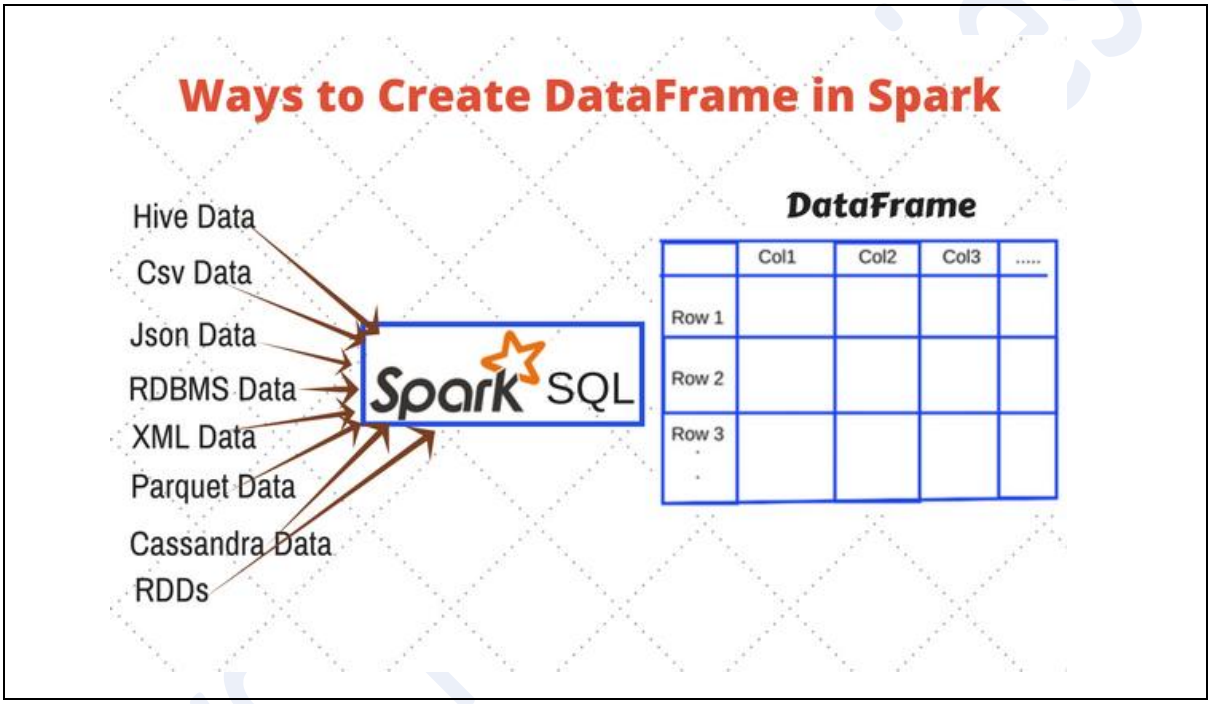
- ✓ Integrated
    - Spark SQL modules uses,
        - SQL queries to process data, these queries we need to write inside spark programs.
        - DataFrames and DataSet APIs to process data.
- ✓ Unified data access
    - Spark SQL can access variety of data sources.
    - Variety of data sources: Hive, Avro, Parquet, ORC, JSON, as well as JDBC and etc
- ✓ Performance Optimization
    - Performance wise Spark SQL is good.

DataFrame

When DataFrame was introduced?

- ✓ DataFrame introduced in Spark 1.3.0 version.

Creating DataFrame



What is DataFrame?

- ✓ DataFrame is a distributed collection of data organized in named columns.

Name: sales.csv

Customer_id	mail_id	Txn_Id	Product_Name	Product_Cost
1	nirekshan@gmail.com	txn01	Samsung	10000
2	mohan@gmail.com	txn02	iPhone	20000
3	nokiafan@gmail.com	txn03	Nokia	1000
4	prasad@gmail.com	txn04	hTC	5000
1	nirekshan@gmail.com	txn05	Samsung	10000
5	praveen@gmail.com	txn06	iPhone	30000
4	prasad@gmail.com	txn07	hTC	5000

Create a DataFrame from reading a CSV file

- ✓ To create DataFrame we need to create object for SparkSession we need to call csv()
- ✓ Below program if sales.csv file is available in current directory then it works.

Program Name: Creating DataFrame by loading csv file
demo1.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+
```

- ✓ Below program we have provided file path, if file exists in different location

Program Name: Creating DataFrame by loading csv file
demo2.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('file:///D:/Nireekshan/Docs/sales.csv',
mode="DROPMALFORMED", inferSchema=True, header = True)

df.show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+
```

Discussion between Praveen and Nireekshan

- ✓ Praveen :

  Hey Nireekshan, It's just only few rows data man.
  Can you please load more than two 100 rows data?

- ✓ Nireekshan : Sure, then let me take **1000000+** lakhs records.
- ✓ Praveen : Arey Nireekshan… Are you **joking**?
- ✓ Nireekshan : No Boss, I'm not joking, just let us try with 10L records.
- ✓ Praveen : okay good then you try, meanwhile I 'll drink water.
- ✓ Nireekshan : Sure.

Create a DataFrame from reading a CSV file

- ✓ Creating DataFrame by reading 1000000+ records.

Program Name: Creating DataFrame by loading csv file
demo3.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame example").getOrCreate()

df=sparkSession.read.csv('1048575_sales_record.csv', mode="DROPMALFORMED", inferSchema=True, header = True)

df.show()
```

Output

	Region	Country	Item Type	Sales Char	Order Pric	Order Dat	Order ID	Ship Date	Units Sold	Unit Price	Unit Cost	Total Reve	Total Cost	Total Profit
2	Sub-Sahara	South Afric	Fruits	Offline	M	7/27/2012	4.43E+08	7/28/2012	1593	9.33	6.92	14862.69	11023.56	3839.13
3	Middle East	Morocco	Clothes	Online	M	9/14/2013	6.68E+08	10/19/201	4611	109.28	35.84	503890.1	165258.2	338631.84
4	Australia ar	Papua New	Meat	Offline	M	5/15/2015	9.41E+08	########	360	421.89	364.69	151880.4	131288.4	20592
5	Sub-Sahara	Djibouti	Clothes	Offline	H	5/17/2017	8.81E+08	########	562	109.28	35.84	61415.36	20142.08	41273.28
6	Europe	Slovakia	Beverages	Offline	L	10/26/201	1.75E+08	########	3973	47.45	31.79	188518.9	126301.7	62217.18
7	Asia	Sri Lanka	Fruits	Online	L	########	8.3E+08	12/18/201	1379	9.33	6.92	12866.07	9542.68	3323.39
8	Sub-Sahara	Seychelles	Beverages	Online	M	1/18/2013	4.26E+08	2/16/2013	597	47.45	31.79	28327.65	18978.63	9349.02
9	Sub-Sahara	Tanzania	Beverages	Online	L	11/30/201	6.6E+08	1/16/2017	1476	47.45	31.79	70036.2	46922.04	23114.16
10	Sub-Sahara	Ghana	Office Supp	Online	L	3/23/2017	6.01E+08	4/15/2017	896	651.21	524.96	583484.2	470364.2	113120
11	Sub-Sahara	Tanzania	Cosmetics	Offline	L	5/23/2016	7.39E+08	5/24/2016	7768	437.2	263.33	3396170	2045547	1350622.2
12	Asia	Taiwan	Fruits	Offline	M	########	7.33E+08	2/23/2014	8034	9.33	6.92	74957.22	55595.28	19361.94
13	Middle East	Algeria	Cosmetics	Online	M	2/18/2011	7.62E+08	2/24/2011	9669	437.2	263.33	4227287	2546138	1681149
14	Asia	Singapore	Snacks	Online	C	1/28/2013	1.76E+08	########	7676	152.58	97.44	1171204	747949.4	423254.64
15	Australia ar	Papua New	Clothes	Offline	L	6/20/2011	6.47E+08	7/14/2011	9092	109.28	35.84	993573.8	325857.3	667716.48
16	Asia	Vietnam	Personal Ca	Online	M	########	3.15E+08	########	7984	81.73	56.67	652532.3	452453.3	200079.04
17	Sub-Sahara	Uganda	Personal Ca	Online	M	6/19/2014	5.39E+08	7/21/2014	451	81.73	56.67	36860.23	25558.17	11302.06
18	Sub-Sahara	Zimbabwe	Office Supp	Offline	C	3/28/2011	9.53E+08	########	9623	651.21	524.96	6266594	5051690	1214903.8
19	Sub-Sahara	Ethiopia	Cosmetics	Online	M	########	8.08E+08	7/25/2011	662	437.2	263.33	289426.4	174324.5	115101.94
20	Europe	France	Cosmetics	Online	M	########	3.25E+08	1/18/2016	5758	437.2	263.33	2517398	1516254	1001143.5

Siddhu : Hey Nireekshan, can I say DataFrame means a table

Nireekshan :

- Yes, we can say as, DataFrame is like a table in a relational database.
- DataFrame exists virtually in spark
- DataFrame contains rows with Schema.

DataFrame support file formats

- ✓ Supports different data formats,
  - Avro,
  - csv etc
  - Store systems like Cassandra, HDFS, HIVE tables, MySql, etc.

Advantage

- ✓ DataFrame brings good optimization technique.

DataFrame characteristics

- ✓ Immutability.
- ✓ In-Memory.
- ✓ Fault tolerant.
- ✓ Resilient distributed computing capability etc

Programming languages to create DataFrame

- ✓ Below programming languages, we can use to create DataFrame,
  - Python
  - Java
  - Scala
  - R

Why DataFrame?

- ✓ DataFrame is one step ahead of RDD.
- ✓ DataFrame is advanced to RDD.
- ✓ DataFrame provides memory management and optimized execution plan.

Custom Management

- ✓ This is also known as Project Tungsten.
- ✓ There is no Garbage Collection overhead.
- ✓ Expensive Java serialization is also avoided.

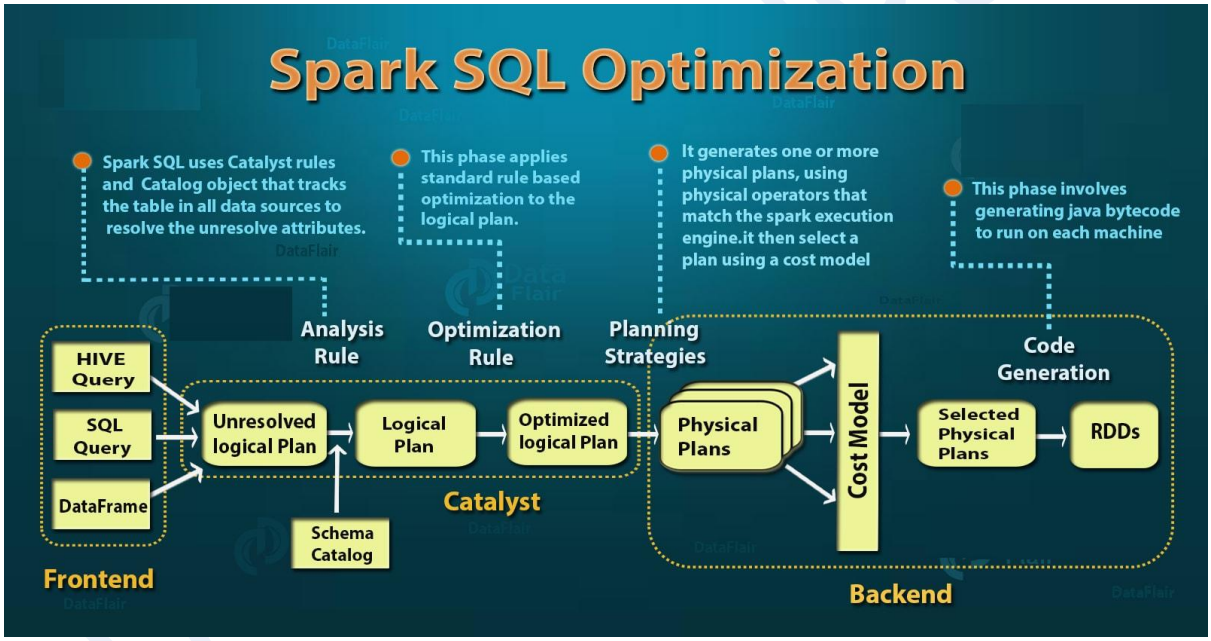
Optimized execution plan

- ✓ This is also known as the query optimizer.
- ✓ Using this, an optimized execution plan is created for the execution of a query.
- ✓ It makes queries run faster.

Spark SQL Execution Plan

Spark SQL query execution phases. In four phases we use Catalyst's general tree transformation framework:

- ✓ Analysis
- ✓ Logical Optimization
- ✓ Physical planning
- ✓ Code generation



Spark SQL terminology

Name	Purpose
DataFrame	✓ It's a structured data organized into named columns
SQLContext	✓ It's the entry point for working with structured data (rows and columns). ✓ It Allows the creation of DataFrame objects as well as the execution of SQL queries
HiveContext	✓ Working with Hive tables, a descendant of SQLContext. ✓ Hive Context is more battle-tested and provides a richer functionality than SQLContext.
DataSet	✓ Datasets extend the benefit of compile-time type safety
JDBC Datasource	✓ JDBC data source can read data from relational databases using JDBC API
Catalyst Optimizer	✓ It is the newest and most technical component of Spark SQL. A catalyst is a query plan optimizer

Spark – SQL – Programs

- ✓ Creating DataFrame by using list of tuples

Name Program: Creating DataFrame with list of tuples but not calling action
demo4.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

l = [('Praveen', 10), ('Nireekshan', 20), ('Ramesh', 30)]

spark.createDataFrame(l, ['name', 'age'])
```

Output

- ✓ Creating DataFrame by using list of tuples.
- ✓ After creating DataFrame we need to call action on top of DataFrame object
- ✓ Here action is show()

Name Program: Creating DataFrame with list of tuples
demo5.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

l = [('Praveen', 10), ('Nireekshan', 20), ('Ramesh', 30)]

spark.createDataFrame(l, ['name', 'age']).show()
```

Output

```
+----------+---+
|      name|age|
+----------+---+
|   Praveen| 10|
|Nireekshan| 20|
|    Ramesh| 30|
+----------+---+
```

✓ Creating DataFrame by using list of tuples

Name Program: Creating DataFrame by using list of tuples and demo6.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

l = [('Praveen', 1, 1000), ('nireekshan', 2, 2000), ('Ramesh', 3, 3000)]

spark.createDataFrame(l, ['name', 'age', 'salary']).show()
```

Output

```
+----------+---+------+
|      name|age|salary|
+----------+---+------+
|   Praveen|  1|  1000|
|Nireekshan|  2|  2000|
|    Ramesh|  3|  3000|
+----------+---+------+
```

✓ Creating DataFrame by loading csv file

Program Name: Creating DataFrame by loading csv file demo7.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+
```

- ✓ Creating DataFrame by loading csv file.

Name inferSchema = False and printSchema()

Program demo8.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv')

df.show()

df.printSchema()
```

Output

```
+-----------+------------------+------+------------+------------+
|        _c0|               _c1|   _c2|         _c3|         _c4|
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
```

printSchema() method

- ✓ printSchema() method prints schema of DataFrame

Name inferSchema = False and printSchema()

Program demo9.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED", header=True)

df.show()

df.printSchema()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

root
 |-- Customer_id: string (nullable = true)
 |-- mail_id: string (nullable = true)
 |-- Txn_Id: string (nullable = true)
 |-- Product_Name: string (nullable = true)
 |-- Product_Cost: string (nullable = true)
```

- ✓ If inferSchema=True and header=True, then spark provides schema and keeps first row as header part.

Name inferSchema=True and header = True

Program demo10.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.show()

df.printSchema()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

root
 |-- Customer_id: integer (nullable = true)
 |-- mail_id: string (nullable = true)
 |-- Txn_Id: string (nullable = true)
 |-- Product_Name: string (nullable = true)
 |-- Product_Cost: integer (nullable = true)
```

df.columns

- ✓ It prints total columns names of DataFrame

Name Displaying all columns names
Program demo11.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.show()

print(df.columns)
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

['Customer_id', 'mail_id', 'Txn_Id', 'Product_Name', 'Product_Cost']
```

filter() method

- ✓ We can apply conditions on DataFrame by using filter method

Name Applying filter
Program demo12.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.filter("Product_Cost< 20000").show(5)
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+
```

select() method

- ✓ We can get specific columns from DataFrame by using select() method.

Name select specific column from DataFrame
Program demo13.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.select('Product_Name').show()
```

Output

```
+------------+
|Product_Name|
+------------+
|     Samsung|
|      iPhone|
|       Nokia|
|         hTC|
|     Samsung|
|      iPhone|
|         hTC|
+------------+
```

Name select specific columns from DataFrame

Program demo14.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.select('Product_Name', 'Product_Cost').show()
```

Output

```
+------------+------------+
|Product_Name|Product_Cost|
+------------+------------+
|     Samsung|       10000|
|      iPhone|       20000|
|       Nokia|        1000|
|         hTC|        5000|
|     Samsung|       10000|
|      iPhone|       30000|
|         hTC|        5000|
+------------+------------+
```

filter() and select()

- ✓ We can apply condition by using filter() method and selecting specific columns

Name filter and select column from DataFrame

Program demo15.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.filter("Product_Cost < 20000").show()

df.filter("Product_Cost < 20000").select(['Product_Name',
'Product_Cost']).show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+


+------------+------------+
|Product_Name|Product_Cost|
+------------+------------+
|     Samsung|       10000|
|       Nokia|        1000|
|         hTC|        5000|
|     Samsung|       10000|
|         hTC|        5000|
+------------+------------+
```

- ✓ Applying group of conditions on DataFrame by using filter() method

Name Applying filter on DataFrame

Program demo16.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)


df.filter("Product_Cost < 10000 and Product_Cost > 1000").show(5)
```

Output

```
+-----------+----------------+------+------------+------------+
|Customer_id|         mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+----------------+------+------------+------------+
|          4|prasad@gmail.com| txn04|         hTC|        5000|
|          4|prasad@gmail.com| txn07|         hTC|        5000|
+-----------+----------------+------+------------+------------+
```

distinct() method

- ✓ Selecting unique or distinct values

Name: Displaying sold unique products

Program: demo17.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.select('Product_Name').distinct().show()
```

Output

```
+------------+
|Product_Name|
+------------+
|       Nokia|
|      iPhone|
|     Samsung|
|         hTC|
+------------+
```

count() method

- ✓ Counting unique values from DataFrame

Name Counting unique products

Program demo18.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.select('Product_Name').distinct().show()

print(df.select('Product_Name').distinct().count())
```

Output

```
+------------+
|Product_Name|
+------------+
|       Nokia|
|      iPhone|
|     Samsung|
|         hTC|
+------------+

4
```

like() method

- ✓ Selecting product names which are starts with S letter

Name Products starts with S letter by using filter

Program demo19.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.filter(df.Product_Name.like('S%')).show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
+-----------+------------------+------+------------+------------+
```

groupBy() method

- ✓ Grouping common counting the common products

Name groupBy
Program demo20.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.groupby('Product_Name').count().show()
```

Output

```
+------------+-----+
|Product_Name|count|
+------------+-----+
|       Nokia|    1|
|      iPhone|    2|
|     Samsung|    2|
|         hTC|    2|
+------------+-----+
```

groupBy() method

- ✓ Grouping, counting and applying condition by using filter() method

Name groupBy and filter
Program demo21.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.groupby('Product_Name').count().filter("count > 1").show(10)
```

Output

```
+------------+-----+
|Product_Name|count|
+------------+-----+
|      iPhone|    2|
|     Samsung|    2|
|         hTC|    2|
+------------+-----+
```

orderBy() method

- ✓ By using this method we can print the values in order

Name orderBy
Program demo22.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.orderBy("Customer_Id").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
+-----------+------------------+------+------------+------------+
```

Name orderBy
Program demo23.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.orderBy("Product_Cost").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
+-----------+------------------+------+------------+------------+
```

- ✓ We can provide our own schema to DataFrame by using,
  - StructField
  - IntegerType
  - StringType etc

Name Providing schema
Program demo24.py

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, IntegerType, StringType, StructType

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

data_schema = [StructField('Customer_Id', IntegerType(), True),
          StructField('mail_id', StringType(), True),
          StructField('Txn_id', StringType(), True),
          StructField('Product_Name', StringType(), True),
          StructField('Product_Cost', IntegerType(), True)]

final_struc = StructType(fields=data_schema)
df=sparkSession.read.csv('sales.csv', final_struc, header = True)

df.show()

df.printSchema()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_Id|           mail_id|Txn_id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

root
 |-- Customer_Id: integer (nullable = true)
 |-- mail_id: string (nullable = true)
 |-- Txn_id: string (nullable = true)
 |-- Product_Name: string (nullable = true)
 |-- Product_Cost: integer (nullable = true)
```

sql() method

- ✓ By using this we can execute sql queries

Make a note

- ✓ createOrReplaceTempView() creates a temporary table.

Name Sql select query
Program demo25.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

result = sparkSession.sql("select * from products")

result.show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+
```

catalog

- ✓ SparkSession stores all table in catalog

listTables() method

- ✓ listTables() method prints all tables in catalog

Name: Checking list of tables in catalog

Program: demo26.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

print(sparkSession.catalog.listTables())
```

Output

```
[Table(name='products', database=None, description=None, tableType='TEMPORARY',
isTemporary=True)]
```

- ✓ Creating two tables

Name Checking list of tables in catalog

Program demo26a.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products1')

df.createOrReplaceTempView('products2')

print(sparkSession.catalog.listTables())
```

Output

```
[Table(name='products1', database=None, description=None, tableType='TEMPORARY',
isTemporary=True), Table(name='products2', database=None, description=None, table
Type='TEMPORARY', isTemporary=True)]
```

sql("show tables") method

- ✓ By using sql() method we can execute query

Name Sql query to show total tables

Program demo27.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("show tables").show()
```

Output

```
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
|        | products|       true|
+--------+---------+-----------+
```

Name Sql query to show total tables

Program demo28.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products1')

df.createOrReplaceTempView('products2')

sparkSession.sql("show tables").show()
```

Output

```
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
|        |products1|       true|
|        |products2|       true|
+--------+---------+-----------+
```

Name Sql select query

Program demo29.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select Product_Name, Product_Cost from products limit
10").show()
```

Output

```
+------------+------------+
|Product_Name|Product_Cost|
+------------+------------+
|     Samsung|       10000|
|      iPhone|       20000|
|       Nokia|        1000|
|         hTC|        5000|
|     Samsung|       10000|
|      iPhone|       30000|
|         hTC|        5000|
+------------+------------+
```

Name Sql select and where query

Program demo30.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select * from products where Product_Name =
'Samsung'").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
+-----------+------------------+------+------------+------------+
```

Name Sql select and where query

Program demo31.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select * from products where Product_Name =
'Nokia'").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          3|nokiafan@gmail.com| txn03|       Nokia|        1000|
+-----------+------------------+------+------------+------------+
```

Name Sql select and count

Program demo32.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select count(*) as Product_Name from products where
Product_Name='Samsung'").show()
```

Output

```
+------------+
|Product_Name|
+------------+
|           2|
+------------+
```

Name Sql select and like

Program demo33.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select * from products where Product_Name like
'S%'").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
+-----------+------------------+------+------------+------------+
```

Name Sql select and where and like

Program demo34.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select * from products where Product_Name like 'S%' and
Customer_id = 4 or Product_Cost = 5000").show()
```

Output

```
+-----------+---------------+------+------------+------------+
|Customer_id|         mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+---------------+------+------------+------------+
|          4|prasad@gmail.com| txn04|         hTC|        5000|
|          4|prasad@gmail.com| txn07|         hTC|        5000|
+-----------+---------------+------+------------+------------+
```

Name Sql select and where

Program demo35.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select * from products where Product_Cost in (10000,
20000)").show()
```

Output

```
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
+-----------+------------------+------+------------+------------+
```

Name Sql select, count and group by

Program demo36.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select Product_Name, count(*) as count from products group
by Product_Name").show()
```

Output

```
+------------+-----+
|Product_Name|count|
+------------+-----+
|       Nokia|    1|
|      iPhone|    2|
|     Samsung|    2|
|         hTC|    2|
+------------+-----+
```

Name Sql select and group by

Program demo37.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select Product_Name, count(*) as count from products group
by Product_Name having count > 1").show()
```

Output

```
+------------+-----+
|Product_Name|count|
+------------+-----+
|      iPhone|    2|
|     Samsung|    2|
|         hTC|    2|
+------------+-----+
```

Name select and group by and order by

Program demo38.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.createOrReplaceTempView('products')

sparkSession.sql("select Customer_id, count(*) as count from products group by
Customer_id having count > 1 order by Customer_id").show()
```

Output

```
+-----------+-----+
|Customer_Id|count|
+-----------+-----+
|          1|    2|
|          4|    2|
+-----------+-----+
```

Name min
Program demo39.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

m = df.agg({"Product_Cost": "max"}).collect()[0][0]

print("Maximum product cost is: ", m)
```

Output

```
Maximum product cost is: 1000
```

Name min
Program demo40.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

m = df.agg({"Product_Cost": "min"}).collect()[0][0]

print("Minimum product cost is: ", m)
```

Output

```
Minimum product cost is: 1000
```

Name sum

Program demo41.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

s = df.agg({"Product_Cost": "sum"}).collect()[0][0]

print('Sum of all products cost is:', s)
```

Output

```
Sum of all products cost is: 81000
```

Name describe()
Program demo42.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df1=df.select("Product_Cost")

df1.show()

df1.describe().show()
```

Output

```
+------------+
|Product_Cost|
+------------+
|       10000|
|       20000|
|        1000|
|        5000|
|       10000|
|       30000|
|        5000|
+------------+

+-------+-----------------+
|summary|     Product_Cost|
+-------+-----------------+
|  count|                7|
|   mean|11571.42857142857|
| stddev|10113.64001167306|
|    min|             1000|
|    max|            30000|
+-------+-----------------+
```

Name Applying operation on specific column

Program demo43.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

df.withColumn('double_cost', df['Product_Cost']*2).show()
```

Output

```
+-----------+------------------+------+------------+------------+-----------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|double_cost|
+-----------+------------------+------+------------+------------+-----------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|      20000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|      40000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|       2000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|      10000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|      20000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|      60000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|      10000|
+-----------+------------------+------+------------+------------+-----------+
```

Name Column renaming
Program demo44.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

print("Before renaming column:")

df.show()

print("After renaming column:")

df.withColumnRenamed("Product_Cost", "item_cost").show()
```

Output

```
Before renaming column:
+-----------+------------------+------+------------+------------+
|Customer_id|           mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+------------------+------+------------+------------+

After renaming column:
+-----------+------------------+------+------------+---------+
|Customer_id|           mail_id|Txn_Id|Product_Name|item_cost|
+-----------+------------------+------+------------+---------+
|          1|nirekshan@gmail.com| txn01|     Samsung|    10000|
|          2|    mohan@gmail.com| txn02|      iPhone|    20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|     1000|
|          4|   prasad@gmail.com| txn04|         hTC|     5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|    10000|
|          5|  praveen@gmail.com| txn06|      iPhone|    30000|
|          4|   prasad@gmail.com| txn07|         hTC|     5000|
+-----------+------------------+------+------------+---------+
```

Name dropping column

Program demo45.py

```
from pyspark.sql import SparkSession

sparkSession = SparkSession.builder.appName("Data Frame
example").getOrCreate()

df=sparkSession.read.csv('sales.csv', mode="DROPMALFORMED",
inferSchema=True, header = True)

print("Before dropping")

df.show()

print("After dropping")

df.drop("Product_Cost").show()
```

Output

```
Before dropping
+-----------+-------------------+------+------------+------------+
|Customer_id|            mail_id|Txn_Id|Product_Name|Product_Cost|
+-----------+-------------------+------+------------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|       10000|
|          2|    mohan@gmail.com| txn02|      iPhone|       20000|
|          3| nokiafan@gmail.com| txn03|       Nokia|        1000|
|          4|   prasad@gmail.com| txn04|         hTC|        5000|
|          1|nirekshan@gmail.com| txn05|     Samsung|       10000|
|          5|  praveen@gmail.com| txn06|      iPhone|       30000|
|          4|   prasad@gmail.com| txn07|         hTC|        5000|
+-----------+-------------------+------+------------+------------+

After dropping
+-----------+-------------------+------+------------+
|Customer_id|            mail_id|Txn_Id|Product_Name|
+-----------+-------------------+------+------------+
|          1|nirekshan@gmail.com| txn01|     Samsung|
|          2|    mohan@gmail.com| txn02|      iPhone|
|          3| nokiafan@gmail.com| txn03|       Nokia|
|          4|   prasad@gmail.com| txn04|         hTC|
|          1|nirekshan@gmail.com| txn05|     Samsung|
|          5|  praveen@gmail.com| txn06|      iPhone|
|          4|   prasad@gmail.com| txn07|         hTC|
+-----------+-------------------+------+------------+
```

Joins

Name Creating two DataFrames

Program demo46.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

valuesA = [('P',1), ('M',2), ('N',3), ('S',4)]
TableA = spark.createDataFrame(valuesA, ['name', 'id'])

valuesB = [('R',1), ('P',2), ('N',3), ('D',4)]
TableB = spark.createDataFrame(valuesB, ['name', 'id'])

TableA.show()
TableB.show()
```

Output

```
+----+---+
|name| id|
+----+---+
|   P|  1|
|   M|  2|
|   N|  3|
|   S|  4|
+----+---+

+----+---+
|name| id|
+----+---+
|   R|  1|
|   P|  2|
|   N|  3|
|   D|  4|
+----+---+
```

Name inner join

Program demo47.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

valuesA = [('P',1),('M',2),('N',3),('S',4)]
TableA = spark.createDataFrame(valuesA,['name','id'])

valuesB = [('R',1),('P',2),('N',3),('D',4)]
TableB = spark.createDataFrame(valuesB,['name','id'])

ta = TableA.alias('ta')
tb = TableB.alias('tb')

inner_join = ta.join(tb, ta.name == tb.name)
inner_join.show()
```

Output

```
+----+---+----+---+
|name| id|name| id|
+----+---+----+---+
|   N|  3|   N|  3|
|   P|  1|   P|  2|
+----+---+----+---+
```

Name left join

Program demo48.py

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Data Frame example").getOrCreate()

valuesA = [('P',1),('M',2),('N',3),('S',4)]
TableA = spark.createDataFrame(valuesA,['name','id'])

valuesB = [('R',1),('P',2),('N',3),('D',4)]
TableB = spark.createDataFrame(valuesB,['name','id'])

ta = TableA.alias('ta')
tb = TableB.alias('tb')

left_join = ta.join(tb, ta.name == tb.name, how='left')
left_join.show()
```

Output

```
+----+---+----+----+
|name| id|name|  id|
+----+---+----+----+
|   M|  2|null|null|
|   N|  3|   N|   3|
|   S|  4|null|null|
|   P|  1|   P|   2|
+----+---+----+----+
```

Spark-SQL – Examples

-

https://spark.apache.org/docs/2.3.2/api/python/pyspark.sql.html