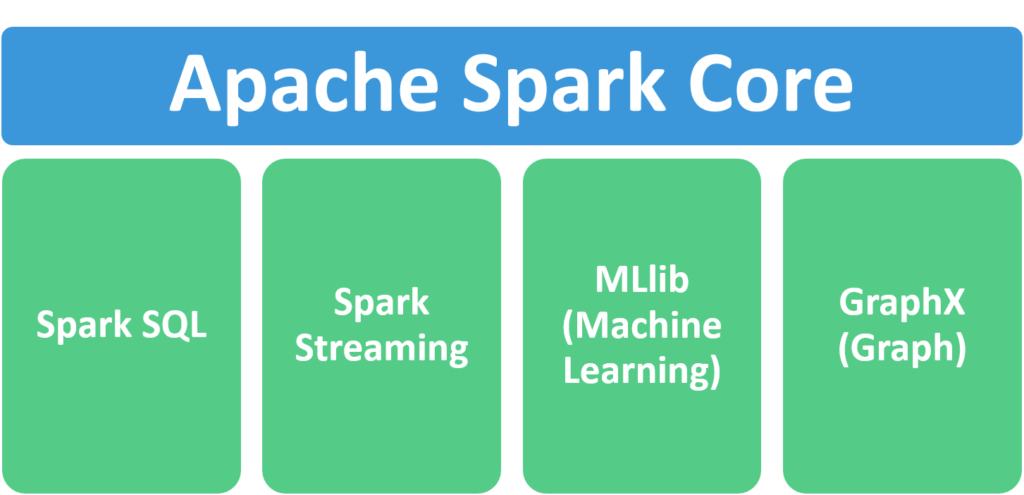
**What is Spark?**

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast queries against data of any size. Simply put, Spark is a **fast and general engine for large-scale data processing**.

The **fast** part means that it’s faster than previous approaches to work with Big Data like classical [MapReduce](https://www.ibm.com/analytics/hadoop/mapreduce). The secret for being faster is that Spark runs on memory (RAM), and that makes the processing much faster than on disk drives.

The **general** part means that it can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and much more.

**[Components](https://chartio.com/learn/data-analytics/what-is-spark/" \l "components)**



1. **Apache Spark Core** – Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built upon. It provides in-memory computing and referencing datasets in external storage systems.
2. **Spark SQL** – Spark SQL is Apache Spark’s module for working with structured data. The interfaces offered by Spark SQL provides Spark with more information about the structure of both the data and the computation being performed.
3. **Spark Streaming** – This component allows Spark to process real-time streaming data. Data can be ingested from many sources like Kafka, Flume, and HDFS (Hadoop Distributed File System). Then the data can be processed using complex algorithms and pushed out to file systems, databases, and live dashboards.
4. **MLlib (Machine Learning Library)** – Apache Spark is equipped with a rich library known as MLlib. This library contains a wide array of machine learning algorithms- classification, regression, clustering, and collaborative filtering. It also includes other tools for constructing, evaluating, and tuning ML Pipelines. All these functionalities help Spark scale out across a cluster.
5. **GraphX** – Spark also comes with a library to manipulate graph databases and perform computations called GraphX. GraphX unifies ETL (Extract, Transform, and Load) process, exploratory analysis, and iterative graph computation within a single system.

**[Features](https://chartio.com/learn/data-analytics/what-is-spark/" \l "features)**

1. **Fast processing** – The most important feature of Apache Spark that has made the big data world choose this technology over others is its speed. Big data is characterized by volume, variety, velocity, and veracity which needs to be processed at a higher speed. Spark contains [Resilient Distributed Dataset (RDD)](https://intellipaat.com/blog/tutorial/spark-tutorial/programming-with-rdds/) which saves time in reading and writing operations, allowing it to run almost **ten to one hundred times faster than Hadoop**.
2. **Flexibility** – Apache Spark supports multiple languages and allows the developers to write applications in Java, Scala, R, or Python.
3. **In-memory computing** – Spark stores the data in the RAM of servers which allows quick access and in turn accelerates the speed of analytics.
4. **Real-time processing** – Spark is able to process real-time streaming data. Unlike MapReduce which processes only stored data, Spark is able to process real-time data and is, therefore, able to produce instant outcomes.
5. **Better analytics** – In contrast to [MapReduce](https://www.ibm.com/analytics/hadoop/mapreduce) that includes Map and Reduce functions, Spark includes much more than that. Apache Spark consists of a rich set of SQL queries, machine learning algorithms, complex analytics, etc. With all these functionalities, analytics can be performed in a better fashion with the help of Spark.

**[Conclusion](https://chartio.com/learn/data-analytics/what-is-spark/" \l "conclusion)**

Apache Spark has seen immense growth over the past several years, becoming the most effective data processing and AI engine in enterprises today due to its speed, ease of use, and sophisticated analytics. However, the cost of Spark is high as it requires lots of RAM to run in-memory.

Spark unifies data and AI by simplifying data preparation at a massive scale across various sources. Moreover, it provides a consistent set of APIs for both data engineering and data science workloads, along with seamless integration of popular libraries such as TensorFlow, PyTorch, R and SciKit-Learn.

**[Resources](https://chartio.com/learn/data-analytics/what-is-spark/" \l "resources)**

1. [Apache Spark Documentation (latest)](https://spark.apache.org/docs/latest/)
2. Towards Data Science – [Deep Learning With Apache Spark](https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd)
3. Tutorials point – [Apache Spark - Introduction](https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm)
4. Cloudera – [Apache Spark](https://www.cloudera.com/products/open-source/apache-hadoop/apache-spark.html)

1.Word count :

import org.apache.spark.sql.SparkSession  
  
 object WordCount extends App {  
  
 val *spark* = SparkSession.*builder* .master("local[\*]")  
 .appName("Spark Word Count")  
 .getOrCreate()  
  
 val *lines* = *spark*.sparkContext.parallelize(  
 *Seq*("Spark Intellij Idea Scala test one",  
 "Spark Intellij Idea Scala test two",  
 "Spark Intellij Idea Scala test three"))  
  
 val *counts* = *lines* .flatMap(line => line.split(" "))  
 .map(word => (word, 1))  
 .reduceByKey(\_ + \_)  
  
 *counts*.foreach(*println*)  
 }

result :

(two,1)

(Spark,3)

(test,3)

(one,1)

(three,1)

(Scala,3)

(Intellij,3)

(Idea,3)

2. import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.functions.\_

object AddExample extends App {

val spark: SparkSession = SparkSession.builder()

.master("local[1]")

.appName("SparkByExamples.com")

.getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

import spark.implicits.\_

val simpleData = Seq(("James", "Sales", 3000),

("Michael", "Sales", 4600),

("Robert", "Sales", 4100),

("Maria", "Finance", 3000),

("James", "Sales", 3000),

("Scott", "Finance", 3300),

("Jen", "Finance", 3900),

("Jeff", "Marketing", 3000),

("Kumar", "Marketing", 2000),

("Saif", "Sales", 4100)

)

val df = simpleData.toDF("employee\_name", "department", "salary")

df.show()

//Distinct all columns

val distinctDF = df.distinct()

println("Distinct count: "+distinctDF.count())

distinctDF.show(false)

val df2 = df.dropDuplicates()

println("Distinct count: "+df2.count())

df2.show(false)

//Distinct using dropDuplicates

val dropDisDF = df.dropDuplicates("department","salary")

println("Distinct count of department & salary : "+dropDisDF.count())

dropDisDF.show(false)

}

Output :-

+-------------+----------+------+

|employee\_name|department|salary|

+-------------+----------+------+

| James| Sales| 3000|

| Michael| Sales| 4600|

| Robert| Sales| 4100|

| Maria| Finance| 3000|

| James| Sales| 3000|

| Scott| Finance| 3300|

| Jen| Finance| 3900|

| Jeff| Marketing| 3000|

| Kumar| Marketing| 2000|

| Saif| Sales| 4100|

+-------------+----------+------+

Distinct count: 9

+-------------+----------+------+

|employee\_name|department|salary|

+-------------+----------+------+

|James |Sales |3000 |

|Michael |Sales |4600 |

|Maria |Finance |3000 |

|Robert |Sales |4100 |

|Saif |Sales |4100 |

|Scott |Finance |3300 |

|Jeff |Marketing |3000 |

|Jen |Finance |3900 |

|Kumar |Marketing |2000 |

+-------------+----------+------+

Distinct count: 9

+-------------+----------+------+

|employee\_name|department|salary|

+-------------+----------+------+

|James |Sales |3000 |

|Michael |Sales |4600 |

|Maria |Finance |3000 |

|Robert |Sales |4100 |

|Saif |Sales |4100 |

|Scott |Finance |3300 |

|Jeff |Marketing |3000 |

|Jen |Finance |3900 |

|Kumar |Marketing |2000 |

+-------------+----------+------+

Distinct count of department & salary : 8

+-------------+----------+------+

|employee\_name|department|salary|

+-------------+----------+------+

|Maria |Finance |3000 |

|Scott |Finance |3300 |

|Jen |Finance |3900 |

|Kumar |Marketing |2000 |

|Jeff |Marketing |3000 |

|James |Sales |3000 |

|Robert |Sales |4100 |

|Michael |Sales |4600 |

+-------------+----------+------+

Process finished with exit code 0

3. import org.apache.spark.rdd.RDD

import org.apache.spark.sql.SparkSession

object RDDParallelize {

def main(args: Array[String]): Unit = {

val spark:SparkSession = SparkSession.builder().master("local[1]")

.appName("SparkByExamples.com")

.getOrCreate()

val rdd:RDD[Int] = spark.sparkContext.parallelize(List(1,2,3,4,5))

val rddCollect:Array[Int] = rdd.collect()

println("Number of Partitions: "+rdd.getNumPartitions)

println("Action: First element: "+rdd.first())

println("Action: RDD converted to Array[Int] : ")

rddCollect.foreach(println)

}

}

Output :-

Action: First element: 1

Action: RDD converted to Array[Int] :

1

2

3

4

5

**Pyspark programs using jupyter note book.**

1.from pyspark.sql import SparkSession

spark=SparkSession.builder.master("local[1]").appName("SparkExamples.com").getOrCreate()

print(spark)

output :-

<pyspark.sql.session.SparkSession object at 0x0000026AECA05BA0>

2. rdd=spark.sparkContext.parallelize([1,2,3,4,5])

print(rdd)

output :-

ParallelCollectionRDD[63] at readRDDFromFile at PythonRDD.scala:274

3. #Create RDD from parallelize

dataList = [("Java", 20000), ("Python", 100000), ("Scala", 3000)]

rdd=spark.sparkContext.parallelize(dataList)

print(dataList)

output :-

[('Java', 20000), ('Python', 100000), ('Scala', 3000)]

4. data = [('James','','Smith','1991-04-01','M',3000),

('Michael','Rose','','2000-05-19','M',4000),

('Robert','','Williams','1978-09-05','M',4000),

('Maria','Anne','Jones','1967-12-01','F',4000),

('Jen','Mary','Brown','1980-02-17','F',-1)

]

columns = ["firstname","middlename","lastname","dob","gender","salary"]

df = spark.createDataFrame(data=data, schema = columns)

print(df)

output :-

DataFrame[firstname: string, middlename: string, lastname: string, dob: string, gender: string, salary: bigint]

5. df.createOrReplaceTempView("PERSON\_DATA")

df2 = spark.sql("SELECT \* from PERSON\_DATA")

df2.printSchema()

output:-

root

|-- firstname: string (nullable = true)

|-- middlename: string (nullable = true)

|-- lastname: string (nullable = true)

|-- dob: string (nullable = true)

|-- gender: string (nullable = true)

|-- salary: long (nullable = true)

6. groupDF = spark.sql("SELECT gender, count(\*) from PERSON\_DATA group by gender")

print(groupDF)

output :-

DataFrame[gender: string, count(1): bigint]