What is Apache Spark?

Apache Spark Tutorial – Apache Spark is an Open source analytical processing engine for large-scale powerful distributed data processing and machine learning applications. Spark was Originally developed at the University of California, Berkeley's, and later donated to the Apache Software Foundation. In February 2014, Spark became a Top-Level Apache Project and has been contributed by thousands of engineers making Spark one of the most active open-source projects in Apache.

Apache Spark 3.5 is a framework that is supported in Scala, Python, R Programming, and Java. Below are different implementations of Spark.

- Spark – Default interface for Scala and Java
- PySpark – Python interface for Spark
- SparklyR – R interface for Spark.

Examples explained in this Spark tutorial are with Scala, and the same is also explained with PySpark Tutorial (Spark with Python) Examples. Python also supports Pandas which also contains Data Frame but this is not distributed.

Features of Apache Spark

- In-memory computation
- Distributed processing using parallelize
- Can be used with many cluster managers (Spark, Yarn, Mesos e.t.c)
- Fault-tolerant
- Immutable
- Lazy evaluation
- Cache & persistence
- Inbuild-optimization when using DataFrames
- Supports ANSI SQL

Advantages of Apache Spark

- Spark is a general-purpose, **in-memory**, fault-tolerant, **distributed processing** engine that allows you to process data efficiently in a distributed fashion.
- Applications running on Spark are **100x** faster than traditional systems.
- You will get great benefits from using Spark for data ingestion pipelines.

- Using Spark we can process data from Hadoop **HDFS**, **AWS S3**, **Databricks DBFS**, **Azure Blob Storage,** and many file systems.
- Spark also is used to process real-time data using Streaming and Kafka.
- Using Spark Streaming you can also stream files from the file system and also stream from the socket.
- Spark natively has machine learning and **graph libraries**.
- Provides connectors to store the data in NoSQL databases like MongoDB.

What Versions of Java & Scala Spark 3.5 Supports?

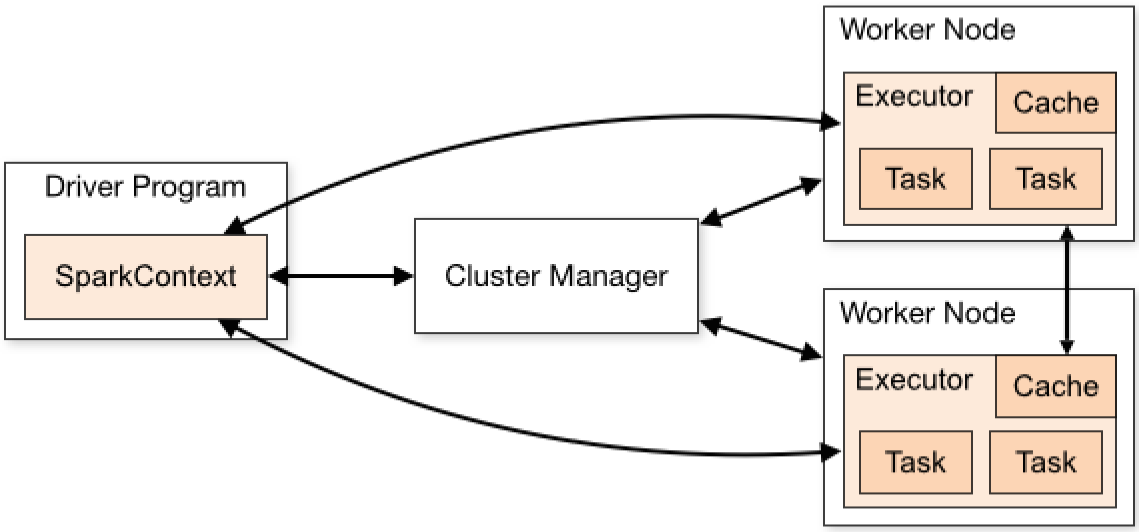
Apache Spark 3.5 is compatible with Java versions 8, 11, and 17, Scala versions 2.12 and 2.13, Python 3.8 and newer, as well as R 3.5 and beyond. However, it's important to note that support for Java 8 versions prior to 8u371 has been deprecated starting from Spark 3.5.0.

LANGUAGE	SUPPORTED VERSION
Python	3.8
Java	Java 8, 11, 13, 17, and the latest versions Java 8 versions prior to 8u371 have been deprecated
Scala	2.12 and 2.13
R	3.5

Apache Spark Tutorial – Versions Supported

Apache Spark Architecture

Spark works in a master-slave architecture where the master is called the "Driver" and slaves are called "Workers". When you run a Spark application, Spark Driver creates a context that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.



Source: https://spark.apache.org/

For additional learning on this topic, I would recommend reading the following.

- What is Spark Job
- What is the Spark Stage? Explained
- What is Spark Executor
- What is Apache Spark Driver?
- What is DAG in Spark or PySpark
- What is a Lineage Graph in Spark?
- How to Submit a Spark Job via Rest API?

Cluster Manager Types

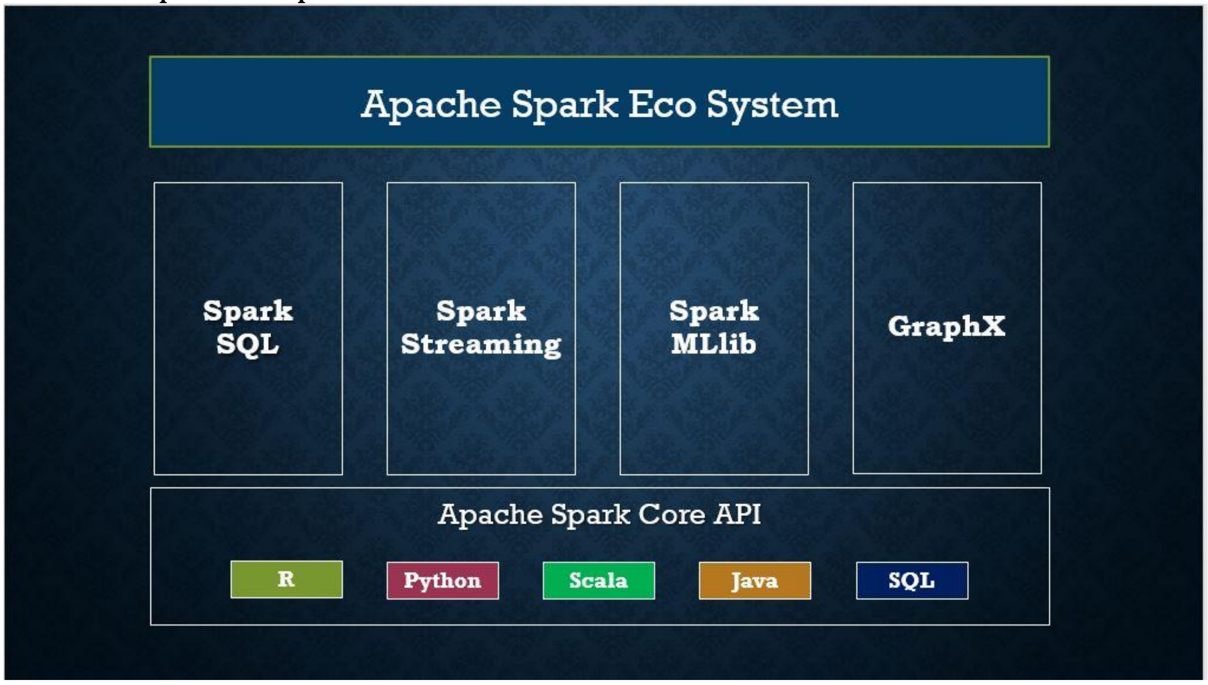
As of writing this Apache Spark Tutorial, Spark supports below cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – Mesons is a Cluster manager that can also run Hadoop MapReduce and Spark applications.
- Hadoop YARN – the resource manager in Hadoop 2. This is mostly used, a cluster manager.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

local – which is not really a cluster manager but still I wanted to mention that we use "local" for `master()` in order to run Spark on our laptop/computer.

Spark Modules

- Spark Core
- Spark SQL
- Spark Streaming
- Spark MLlib
- Spark GraphX



Spark Modules

Spark Core

In this section of the Apache Spark Tutorial, you will learn different concepts of the Spark Core library with examples in Scala code. Spark Core is the main base library of Spark which provides the abstraction of how distributed task dispatching, scheduling, basic I/O functionalities etc.

Before getting your hands dirty on Spark programming, have your Development Environment Setup to run Spark Examples using IntelliJ IDEA

SparkSession

SparkSession introduced in version 2.0, is an entry point to underlying Spark functionality in order to programmatically use Spark RDD, DataFrame, and Dataset. It's object `spark` is default available in spark-shell.
Creating a SparkSession instance would be the first statement you would write to the program with RDD, DataFrame and Dataset. SparkSession will be created using `SparkSession.builder()` builder pattern.

```
// Create SparkSession
import  org.apache.spark.sql.SparkSession
val spark:SparkSession = SparkSession.builder()
      .master("local[1]")
      .appName("SparkByExamples.com")
      .getOrCreate()
```

Spark Context

SparkContext is available since Spark 1.x (JavaSparkContext for Java) and is used to be an entry point to Spark and PySpark before introducing SparkSession in 2.0. Creating SparkContext was the first step to the program with RDD and to connect to Spark Cluster. It's object `sc` by default available in `spark-shell`.
Since Spark 2.x version, When you create SparkSession, SparkContext object is by default created and it can be accessed using `spark.sparkContext`
Note that you can create just one SparkContext per JVM but can create many SparkSession objects.

RDD Spark Tutorial

RDD (Resilient Distributed Dataset) is a fundamental data structure of Spark and it is the primary data abstraction in Apache Spark and the Spark Core. RDDs are fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.
This Apache Spark RDD Tutorial will help you start understanding and using Apache Spark RDD (Resilient Distributed Dataset) with Scala code examples. All RDD examples provided in this tutorial were also tested in our development environment and are available at GitHub spark scala examples project for quick reference.
In this section of the Apache Spark tutorial, I will introduce the RDD and explain how to create them and use their transformation and action operations. Here is the full article on Spark RDD in case you want to learn more about it and get your fundamentals strong.

RDD creation

RDDs are created primarily in two different ways, first parallelizing an existing collection and secondly referencing a dataset in an external storage system (`HDFS`, `HDFS`, `S3` and many more).

sparkContext.parallelize()

sparkContext.parallelize is used to parallelize an existing collection in your driver program. This is a basic method to create RDD.

```
//Create RDD from parallelize
val dataSeq = Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000))
val rdd=spark.sparkContext.parallelize(dataSeq)
```

sparkContext.textFile()

Using textFile() method we can read a text (.txt) file from many sources like HDFS, S#, Azure, local e.t.c into RDD.

```
//Create RDD from external Data source
val rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

RDD Operations

On Spark RDD, you can perform two kinds of operations.

RDD Transformations

Spark RDD Transformations are lazy operations meaning they don't execute until you call an action on RDD. Since RDDs are immutable, When you run a transformation(for example map()), instead of updating a current RDD, it returns a new RDD.
Some transformations on RDDs are `flatMap()`, `map()`, `reduceByKey()`, `filter()`, `sortByKey()` and all these return a new RDD instead of updating the current.

RDD Actions

RDD Action operation returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action. RDD operations trigger the computation and return RDD in a List to the driver program.
Some actions on RDDs are `count()`, `collect()`, `first()`, `max()`, `reduce()` and more.

RDD Examples

- Read CSV file into RDD
- RDD Pair Functions
- Generate DataFrame from RDD

DataFrame Spark Tutorial with Basic Examples

DataFrame definition is very well explained by Databricks hence I do not want to define it again and confuse you. Below is the definition I took from Databricks.

DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs.

– Databricks

DataFrame creation

The simplest way to create a Spark DataFrame is from a seq collection. Spark DataFrame can also be created from an RDD and by reading files from several sources.
Related: Spark Word Count Explained with Example

using createDataFrame()

By using `createDataFrame()` function of the SparkSession you can create a DataFrame.

```
// Create DataFrame
val data = Seq(('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)
)

val columns = Seq("firstname","middlename","lastname","dob","gender","salary")
df = spark.createDataFrame(data), schema = columns).toDF(columns:_*)
```

Since DataFrames are structure format that contains names and column, we can get the schema of the DataFrame using the `df.printSchema()`
`df.show()` shows the 20 elements from the DataFrame.

```
+---------+----------+--------+----------+------+------+
|firstname|middlename|lastname|dob       |gender|salary|
+---------+----------+--------+----------+------+------+
|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    |
+---------+----------+--------+----------+------+------+
```

In this Apache Spark SQL DataFrame Tutorial, I have explained several mostly used operation/functions on DataFrame & DataSet with working Scala examples.

- Spark DataFrame – Rename nested column
- How to add or update a column on DataFrame
- How to drop a column on DataFrame
- Spark when otherwise usage
- How to add literal constant to DataFrame
- Spark Data Types explained
- How to change column data type
- How to Pivot and Unpivot a DataFrame
- Create a DataFrame using StructType & StructField schema
- How to select the first row of each group
- How to sort DataFrame
- How to union DataFrame
- How to drop Rows with null values from DataFrame
- How to split single to multiple columns

- How to concatenate multiple columns
- How to replace null values in DataFrame
- How to remove duplicate rows on DataFrame
- How to remove distinct on multiple selected columns
- Spark map() vs mapPartitions()

Spark DataFrame Advanced concepts

- Spark Partitioning, Repartitioning and Coalesce
- How does Spark shuffle work?
- Spark Cache and Persistence
- Spark Persistance Storage levels
- Spark Broadcast shared variable
- Spark Accumulator shared variable
- Spark UDF

Spark Array and Map operations

- How to create an Array (ArrayType) column on DataFrame
- How to create a Map (MapType) column on DataFrame
- How to convert an Array to columns
- How to create an Array of struct column
- How to explode an Array and map columns
- How to explode an Array of structs
- How to explode an Array of map columns to rows
- How to create a DataFrame with nested Array
- How to explode nested Arrays to rows
- How to flatten nested Array to single Array
- Spark – Convert array of String to a String column

Spark Aggregate

- How to group rows in DataFrame
- How to get Count distinct on DataFrame
- How to add row number to DataFrame
- How to select the first row of each group

Spark SQL Joins

- Spark SQL Join

Spark Performance

- Spark Performance Improvement

Other Helpful topics on DataFrame

- How to stop DEBUG & INFO log messages
- Print DataFrame full column contents
- Unstructured vs semi-structured vs structured files

Spark SQL Schema & StructType

- How to convert case class to a schema
- Spark Schema explained with examples
- How to create array of struct column
- Spark StructType & StructField
- How to flatten nested column

Spark SQL Functions

Spark SQL provides several built-in functions, When possible try to leverage the standard library as they are a little bit more compile-time safe, handle null, and perform better when compared to UDF. If your application is critical on performance try to avoid using custom UDF at all costs as these are not guaranteed on performance.

In this section, we will see several Spark SQL functions Tutorials with Scala examples.

- Spark Date and Time Functions
- Spark String Functions
- Spark Array Functions
- Spark Map Functions
- Spark Aggregate Functions
- Spark Window Functions
- Spark Sort Functions

Spark Data Source with Examples

Spark SQL supports operating on a variety of data sources through the DataFrame interface. This section of the tutorial describes reading and writing data using the Spark Data Sources with Scala examples. Using Data source API we can load from or save data to RDMS databases, Avro, parquet, XML etc.

Text

- Spark process Text file
- How to process JSON from a Text file

CSV

- How to process CSV file
- How to convert Parquet file to CSV file
- How to process JSON from a CSV file
- How to Convert Avro file to CSV file
- How to convert CSV file to Avro, Parquet & JSON

JSON

JSON's readability, flexibility, language-agnostic nature, and support for semi-structured data make it a preferred choice in big data Spark applications where diverse sources, evolving schemas, and efficient data interchange are common requirements.

Key characteristics and reasons why JSON is used in big data include Human-Readable Format, Language-Agnostic, Semi-Structured Data, Schema Evolution e.t.c

- JSON Example (Read & Write)
- How to Read JSON from multi-line
- How to read JSON file with custom schema
- How to process JSON from a CSV file
- How to process JSON from a Text file
- How to convert Parquet file to JSON file
- How to convert Avro file to JSON file
- How to convert JSON to Avro, Parquet, CSV file

Parquet

- Parquet Example (Read and Write)
- How to convert Parquet file to CSV file
- How to convert Parquet file to Avro file
- How to convert Avro file to Parquet file

Avro

- Avro Example (Read and Write)
- Spark 2.3 – Apache Avro Example
- How to Convert Avro file to CSV file
- How to convert Parquet file to Avro file
- How to convert Avro file to JSON file
- How to convert Avro file to Parquet file

ORC

- Spark Read & Write ORC

XML

- Processing Nested XML structured files
- How to validate XML with XSD

Hive & Tables

- Spark Save DataFrame to Hive Table
- Spark JDBC Parallel Read
- Read JDBC Table to Spark DataFrame
- Spark saveAsTable() with Examples
- Spark Query Table using JDBC
- Spark Read and Write MySQL Database Table
- Spark with SQL Server – Read and Write Table
- Spark spark.table() vs spark.read.table()

SQL Spark Tutorial

Spark SQL is one of the most used **Spark** modules which is used for processing structured columnar data format. Once you have a DataFrame created, you can interact with the data by using SQL syntax. In other words, Spark SQL brings native RAW SQL queries on Spark meaning you can run traditional ANSI SQL on Spark Dataframe. In the later section of this Apache Spark tutorial, you will learn in detail using SQL `select`, `where`, `group by`, `join`, `union` e.t.c

In order to use SQL, first, we need to create a temporary table on DataFrame using createOrReplaceTempView() function. Once created, this table can be accessed throughout the SparkSession and it will be dropped along with your SparkContext termination.

On a table, SQL query will be executed using `sql()` method of the SparkSession and this method returns a new DataFrame.

```
df.createOrReplaceTempView("PERSON_DATA")
val df2 = spark.sql("SELECT * from PERSON_DATA")
df2.printSchema()
df2.show()
```

Let's see another example using `group by`.

```
val groupDF = spark.sql("SELECT gender, count(*) from PERSON_DATA group by gender")
groupDF.show()
```

This yields the below output

```
+------+--------+
|gender|count(1)|
+------+--------+
|     F|       2|
|     M|       3|
+------+--------+
```

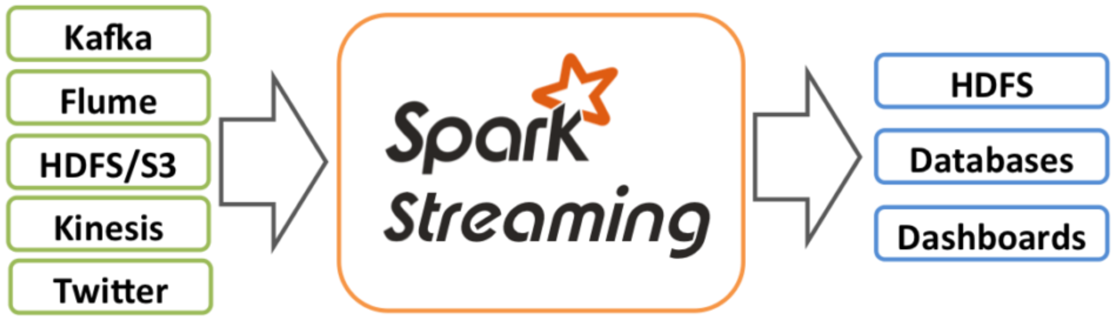
Similarly, you can run any traditional SQL queries on DataFrames using Spark SQL.

Spark HDFS & S3 Tutorial

- Processing files from Hadoop HDFS (TEXT, CSV, Parquet, Avro, JSON)
- Processing TEXT files from Amazon S3 bucket
- Processing JSON files from Amazon S3 bucket
- Processing CSV files from Amazon S3 bucket
- Processing Parquet files from Amazon S3 bucket
- Processing Avro files from Amazon S3 bucket

Spark Streaming Tutorial & Examples

Spark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is used to process real-time data from sources like file system folders, TCP sockets, S3, Kafka, Flume, Twitter, and Amazon Kinesis to name a few. The processed data can be pushed to databases, Kafka, live dashboards e.t.c



source: https://spark.apache.org/

- Spark Streaming – OutputModes Append vs Complete vs Update
- Spark Streaming – Read JSON Files From Directory with Scala Example
- Spark Streaming – Read data From TCP Socket with Scala Example
- Spark Streaming – Consuming & Producing Kafka messages in JSON format
- Spark Streaming – Consuming & Producing Kafka messages in Avro format
- Using from_avro and to_avro functions
- Reading Avro data from Kafka topic using from_avro() and to_avro()
- Spark Batch Processing using Kafka Data Source

Spark with Kafka Tutorials

- Spark Streaming – Consuming & Producing Kafka messages in JSON format
- Spark Streaming – Consuming & Producing Kafka messages in Avro format
- Using from_avro and to_avro functions
- Reading Avro data from Kafka topic using from_avro() and to_avro()
- Spark Batch Processing using Kafka Data Source

Spark – HBase Tutorials & Examples

In this section of the Spark Tutorial, you will learn several Apache HBase spark connectors and how to read an HBase table to a Spark DataFrame and write DataFrame to HBase table.
Apache HBase is an open-source, distributed, and scalable NoSQL database that runs on top of the Hadoop Distributed File System (HDFS). It provides real-time read and write access to large datasets and is designed for handling massive amounts of unstructured or semi-structured data, making it suitable for big data applications.

- Spark HBase Connectors explained
- Writing Spark DataFrame to HBase table using shc-core Hortonworks library
- Creating Spark DataFrame from Hbase table using shc-core Hortonworks library

Spark – Hive Tutorials

In this section, you will learn what is Apache Hive and several examples of connecting to Hive, creating Hive tables, reading them into DataFrame
Apache Hive is a data warehousing and SQL-like query language tool built on top of Hadoop. It facilitates querying and managing large datasets stored in Hadoop Distributed File System (HDFS) using a familiar SQL syntax. Hive translates SQL-like queries into MapReduce or Apache Tez jobs, enabling users without extensive programming skills to analyze big data. It supports schema-on-read, allowing flexible data structures, and integrates with HBase and other Hadoop ecosystem components. Hive is particularly useful for batch processing, data summarization, and analysis tasks, making big data analytics accessible to a broader audience within the Apache Hadoop ecosystem.

- Start HiveServer2 and connect to hive beeline

Spark GraphX and GraphFrames

Spark GraphFrames are introduced in Spark 3.0 version to support Graphs on DataFrames. Prior to 3.0, Spark had GraphX library which ideally runs on RDD, and lost all Data Frame capabilities.

GraphFrames is a graph processing library for Apache Spark that provides high-level abstractions for working with graphs and performing graph analytics. It extends Spark’s DataFrame API to support graph operations, allowing users to express complex graph queries using familiar DataFrame operations.

Below is an example of how to create and use Spark GraphFrame.

```scala
// Import necessary libraries
import org.apache.spark.sql.SparkSession
import org.graphframes.GraphFrame

// Create a Spark session
val spark =
SparkSession.builder.appName("GraphFramesExample").getOrCreate()

// Define vertices and edges as DataFrames
val vertices = spark.createDataFrame(Seq(
  (1, "Scott", 30),
  (2, "David", 40),
  (3, "Mike", 45)
)).toDF("id", "name", "age")

val edges = spark.createDataFrame(Seq(
  (1, 2, "friend"),
  (2, 3, "follow")
)).toDF("src", "dst", "relationship")

// Create a GraphFrame
val graph = GraphFrame(vertices, edges)

// Display vertices and edges
graph.vertices.show()
graph.edges.show()

// Perform Graph Queries
val aliceFriends = graph.edges.filter("src = 1").join(graph.vertices,
"dst").select("dst", "name")
aliceFriends.show()

// Graph Analytics - In-degrees
val inDegrees = graph.inDegrees
inDegrees.show()

// Subgraph Creation
val subgraph = graph.filterVertices("age >= 40").filterEdges("relationship =
'friend'")
subgraph.vertices.show()
subgraph.edges.show()

// Graph Algorithms - PageRank
val pageRankResults =
graph.pageRank.resetProbability(0.15).maxIter(10).run()
pageRankResults.vertices.show()
pageRankResults.edges.show()

// Stop the Spark session
```

```
spark.stop()
```

What are the key features and improvements released in Spark 3.5.0

Following are some of the key features and improvements in Spark 3.5

- **Spark Connect**: This release extends the general availability of Spark Connect with support for Scala and Go clients, distributed training and inference support, and enhanced compatibility for Structured streaming.
- **PySpark and SQL Functionality**: New functionality has been introduced in PySpark and SQL, including the SQL IDENTIFIER clause, named argument support for SQL function calls, SQL function support for HyperLogLog approximate aggregations, and Python user-defined table functions.
- **Distributed Training with DeepSpeed**: The release simplifies distributed training with DeepSpeed, making it more accessible.
- **Structured Streaming**: It introduces watermark propagation among operators and dropDuplicatesWithinWatermark operations in Structured Streaming, enhancing its capabilities.
- **English SDK:** Apache Spark for English SDK integrates the extensive expertise of Generative AI in Apache Spark.

Spark Architecture

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

- Resilient Distributed Dataset (RDD)
- Directed Acyclic Graph (DAG)

Resilient Distributed Datasets (RDD)

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

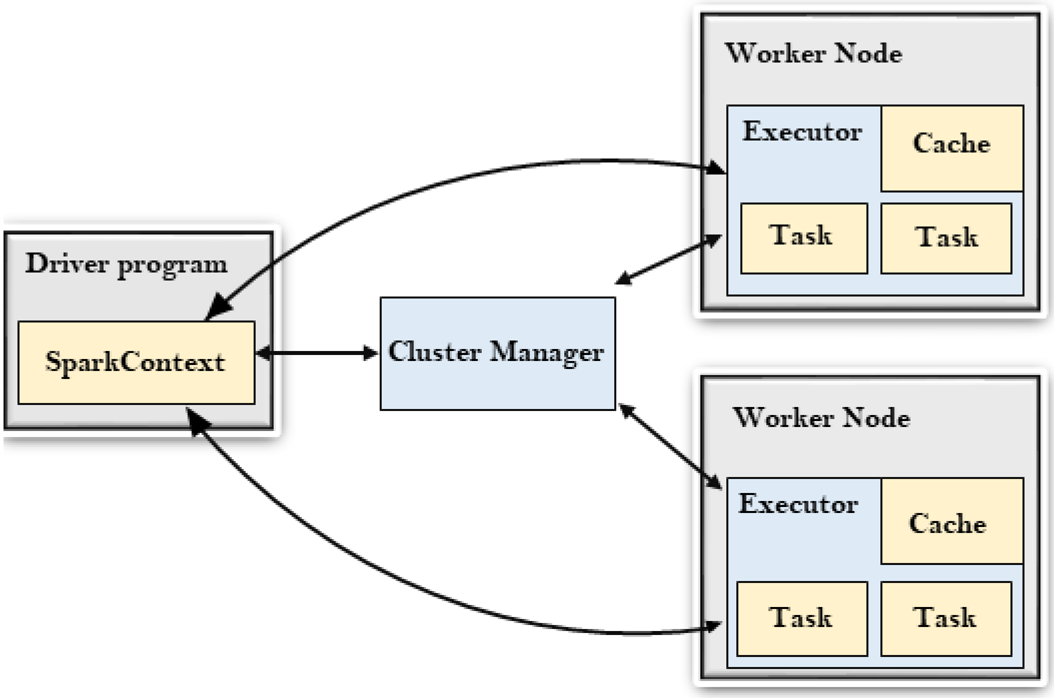
- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

We will learn about RDD later in detail.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.

Let's understand the Spark architecture.



Driver Program

The Driver Program is a process that runs the main() function of the application and creates the **SparkContext** object. The purpose of **SparkContext** is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the **SparkContext** connects to a different type of cluster managers and then perform the following tasks: -

- It acquires executors on nodes in the cluster.
- Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
- At last, the SparkContext sends tasks to the executors to run.

Cluster Manager

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

- The worker node is a slave node
- Its role is to run the application code in the cluster.

Executor

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.
- Every application contains its executor.

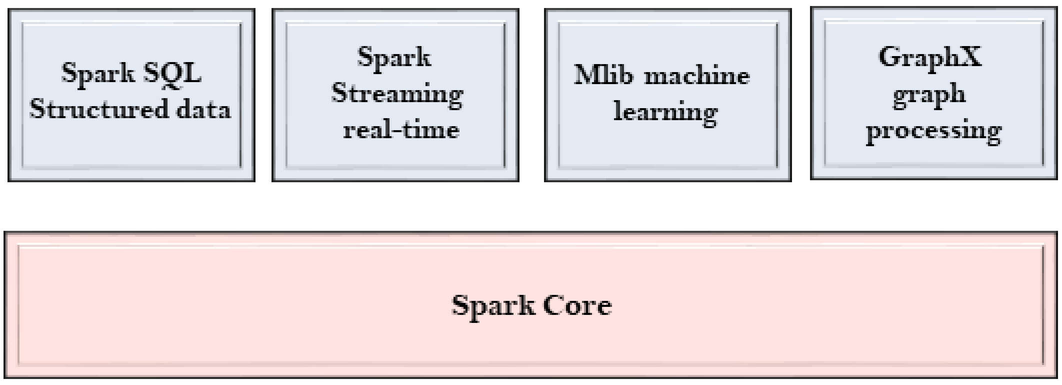
Task

- A unit of work that will be sent to one executor.

Spark Components

The Spark project consists of different types of tightly integrated components. At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.

Let's understand each Spark component in detail.



Spark Core

- The Spark Core is the heart of Spark and performs the core functionality.
- It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.

Spark SQL

- The Spark SQL is built on the top of Spark Core. It provides support for structured data.
- It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language).
- It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.
- It also supports various sources of data like Hive tables, Parquet, and JSON.

Spark Streaming

- Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
- It uses Spark Core's fast scheduling capability to perform streaming analytics.
- It accepts data in mini-batches and performs RDD transformations on that data.

- Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
- The log files generated by web servers can be considered as a real-time example of a data stream.

MLlib

- The MLlib is a Machine Learning library that contains various machine learning algorithms.
- These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
- It is nine times faster than the disk-based implementation used by Apache Mahout.

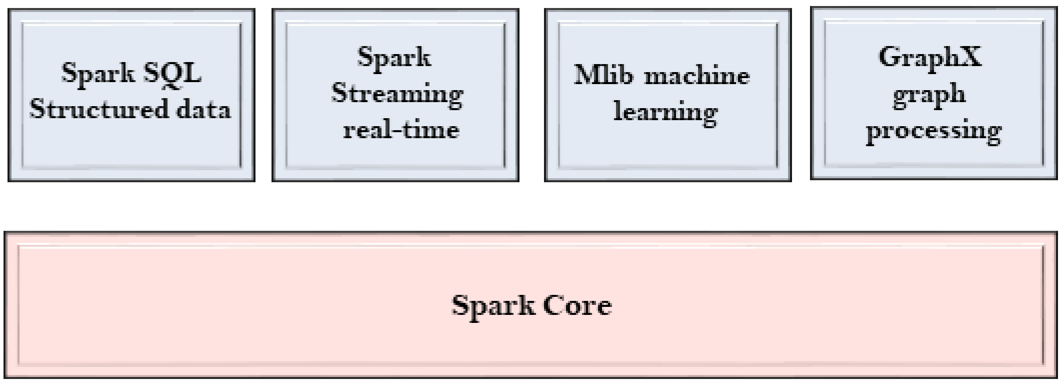
GraphX

- The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
- It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
- To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

Spark Components

The Spark project consists of different types of tightly integrated components. At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.

Let's understand each Spark component in detail.



Spark Core

- The Spark Core is the heart of Spark and performs the core functionality.
- It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.

Spark SQL

- The Spark SQL is built on the top of Spark Core. It provides support for structured data.
- It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language).
- It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.
- It also supports various sources of data like Hive tables, Parquet, and JSON.

Spark Streaming

- Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
- It uses Spark Core's fast scheduling capability to perform streaming analytics.
- It accepts data in mini-batches and performs RDD transformations on that data.
- Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
- The log files generated by web servers can be considered as a real-time example of a data stream.

MLlib

- The MLlib is a Machine Learning library that contains various machine learning algorithms.
- These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
- It is nine times faster than the disk-based implementation used by Apache Mahout.

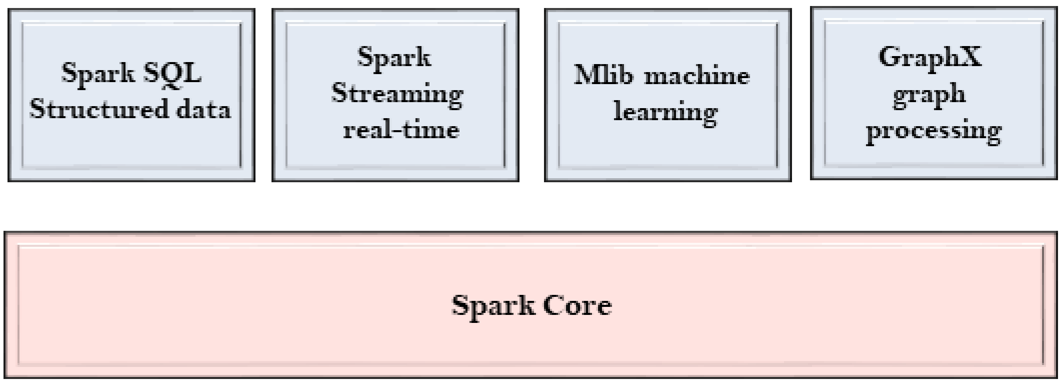
GraphX

- The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
- It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
- To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

Spark Components

The Spark project consists of different types of tightly integrated components. At its core, Spark is a computational engine that can schedule, distribute and monitor multiple applications.

Let's understand each Spark component in detail.



Spark Core

- The Spark Core is the heart of Spark and performs the core functionality.

- It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.

Spark SQL

- The Spark SQL is built on the top of Spark Core. It provides support for structured data.
- It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language).
- It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools.
- It also supports various sources of data like Hive tables, Parquet, and JSON.

Spark Streaming

- Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
- It uses Spark Core's fast scheduling capability to perform streaming analytics.
- It accepts data in mini-batches and performs RDD transformations on that data.
- Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
- The log files generated by web servers can be considered as a real-time example of a data stream.

MLlib

- The MLlib is a Machine Learning library that contains various machine learning algorithms.
- These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
- It is nine times faster than the disk-based implementation used by Apache Mahout.

GraphX

- The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.

- It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
- To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

What is RDD?

The RDD (Resilient Distributed Dataset) is the Spark's core abstraction. It is a collection of elements, partitioned across the nodes of the cluster so that we can execute various parallel operations on it.

There are two ways to create RDDs:

- Parallelizing an existing data in the driver program
- Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

Parallelized Collections

To create parallelized collection, call **SparkContext's** parallelize method on an existing collection in the driver program. Each element of collection is copied to form a distributed dataset that can be operated on in parallel.

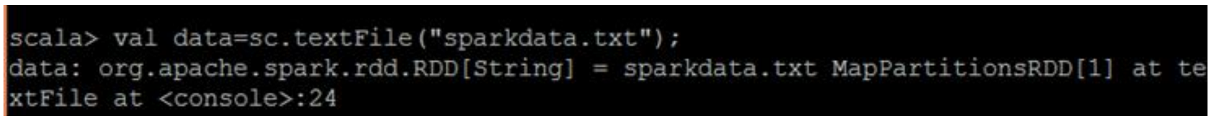
```
val info = Array(1, 2, 3, 4)
val distinfo = sc.parallelize(info)
```

Now, we can operate the distributed dataset (distinfo) parallel such like distinfo.reduce((a, b) => a + b).

External Datasets

In Spark, the distributed datasets can be created from any type of storage sources supported by Hadoop such as HDFS, Cassandra, HBase and even our local file system. Spark provides the support for text files, **SequenceFiles**, and other types of Hadoop **InputFormat**.

SparkContext's textFile method can be used to create RDD's text file. This method takes a URI for the file (either a local path on the machine or a hdfs://) and reads the data of the file.



```
scala> val data=sc.textFile("sparkdata.txt");
data: org.apache.spark.rdd.RDD[String] = sparkdata.txt MapPartitionsRDD[1] at te
xtFile at <console>:24
```

Now, we can operate data on by dataset operations such as we can add up the sizes of all the lines using the map and reduceoperations as follows: data.map(s => s.length).reduce((a, b) => a + b).

RDD Operations

The RDD provides the two types of operations:

- Transformation
- Action

Transformation

In Spark, the role of transformation is to create a new dataset from an existing one. The transformations are considered lazy as they only computed when an action requires a result to be returned to the driver program.

Let's see some of the frequently used RDD Transformations.

Transformation	Description
map(func)	It returns a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	It returns a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Here, each input item can be mapped to zero or more output items, so func should return a sequence rather than a single item.
mapPartitions(func)	It is similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> =>

	Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(func)	It is similar to mapPartitions that provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	It samples the fraction fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	It returns a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	It returns a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numPartitions]))	It returns a new dataset that contains the distinct elements of the source dataset.
groupByKey([numPartitions])	It returns a dataset of (K, Iterable) pairs when called on a dataset of (K, V) pairs.
reduceByKey(func, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.

sortByKey([ascending], [numPartitions])	It returns a dataset of key-value pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable, Iterable)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script.
coalesce(numPartitions)	It decreases the number of partitions in the RDD to numPartitions.
repartition(numPartitions)	It reshuffles the data in the RDD randomly to create either more or fewer partitions and balance it across them.
repartitionAndSortWithinPartitions(partitioner)	It repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

Action

In Spark, the role of action is to return a value to the driver program after running a computation on the dataset.

Action	Description
reduce(func)	It aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	It returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	It returns the number of elements in the dataset.
first()	It returns the first element of the dataset (similar to take(1)).
take(n)	It returns an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	It returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	It returns the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	It is used to write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	It is used to write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system.
saveAsObjectFile(path) (Java and Scala)	It is used to write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().

countByKey()	It is only available on RDDs of type (K, V). Thus, it returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	It runs a function func on each element of the dataset for side effects such as updating an Accumulator or interacting with external storage systems.

Spark Map function

In Spark, the Map passes each element of the source through a function and forms a new distributed dataset.

- Create an RDD using parallelized collection.

```
scala> val data = sc.parallelize(List(10,20,30))
```

- Now, we can read the generated result by using the following command.

```
scala> data.collect
```

- Apply the map function and pass the expression required to perform.

```
scala> val mapfunc = data.map(x => x+10)
```

- Now, we can read the generated result by using the following command.

```
scala> mapfunc.collect
```