Table of Contents

1. SPARK SQL – INTRODUCTION

Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.

As against a common belief, **Spark is not a modified version of Hadoop** and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Spark uses Hadoop in two ways – one is **storage** and second is **processing**. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

Apache Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia. It was Open Sourced in 2010 under a BSD license. It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

Features of Apache Spark

Apache Spark has following features.

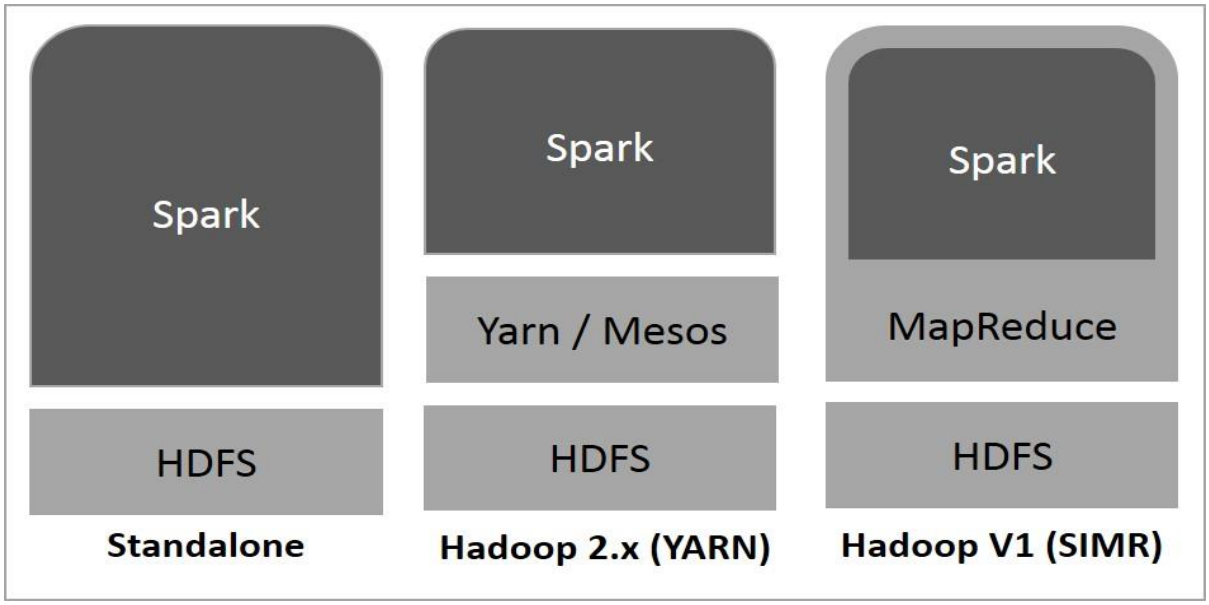
- **Speed:** Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.

- **Supports multiple languages:** Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.

- **Advanced Analytics:** Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

Spark Built on Hadoop

The following diagram shows three ways of how Spark can be built with Hadoop components.



There are three ways of Spark deployment as explained below.

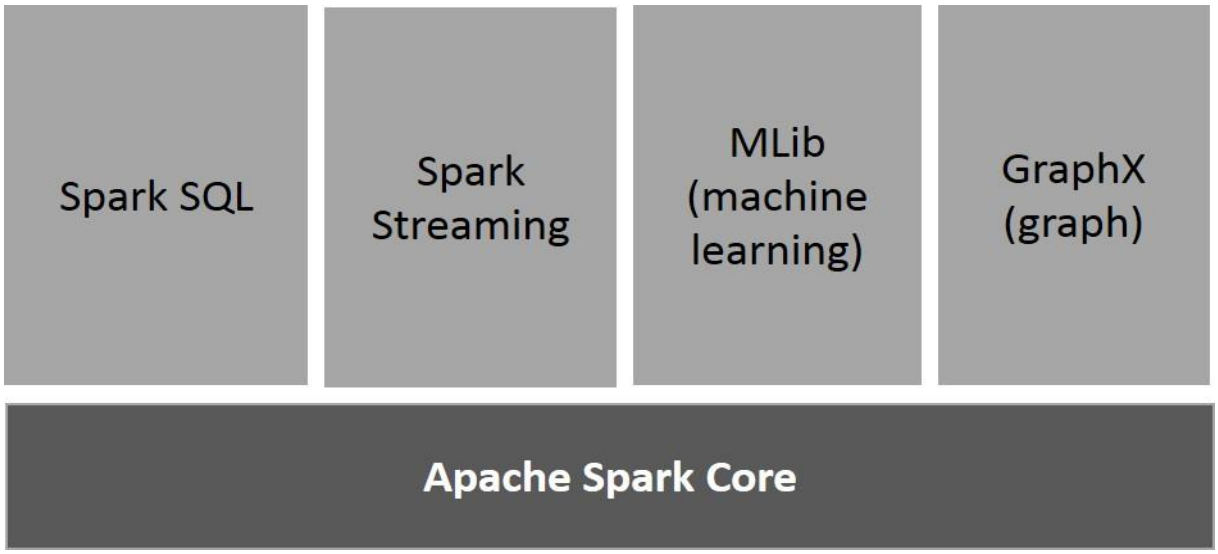
- **Standalone:** Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.

- **Hadoop Yarn:** Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.

- **Spark in MapReduce (SIMR):** Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

Components of Spark

The following illustration depicts the different components of Spark.



Apache Spark Core

Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL

Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

MLlib (Machine Learning Library)

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

GraphX

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

2. SPARK SQL – RDD

Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs: **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

Data Sharing is Slow in MapReduce

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

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex: between two MapReduce jobs) is to write it to an external stable storage system (Ex: HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication**, **serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.

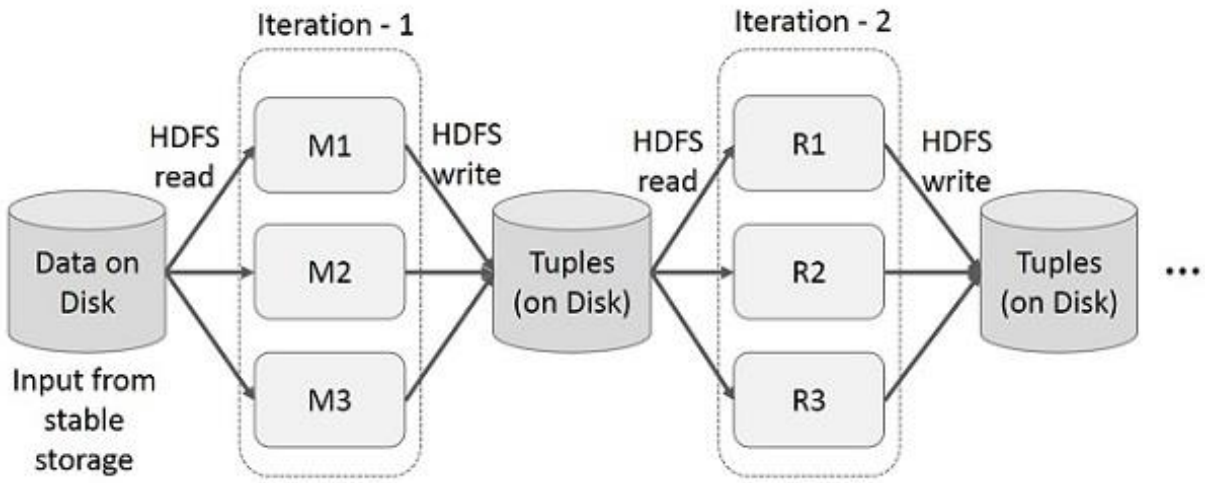


Figure: Iterative operations on MapReduce

Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominates application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.

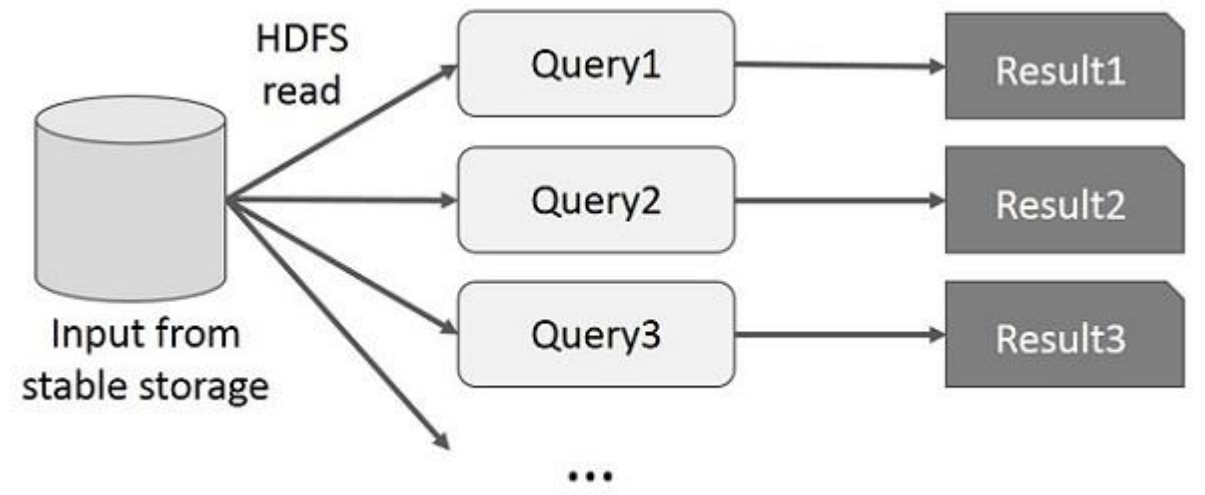


Figure: Interactive operations on MapReduce

Data Sharing using Spark RDD

Data sharing is slow in MapReduce due to **replication**, **serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

Iterative Operations on Spark RDD

The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

Note: If the Distributed memory (RAM) is sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.

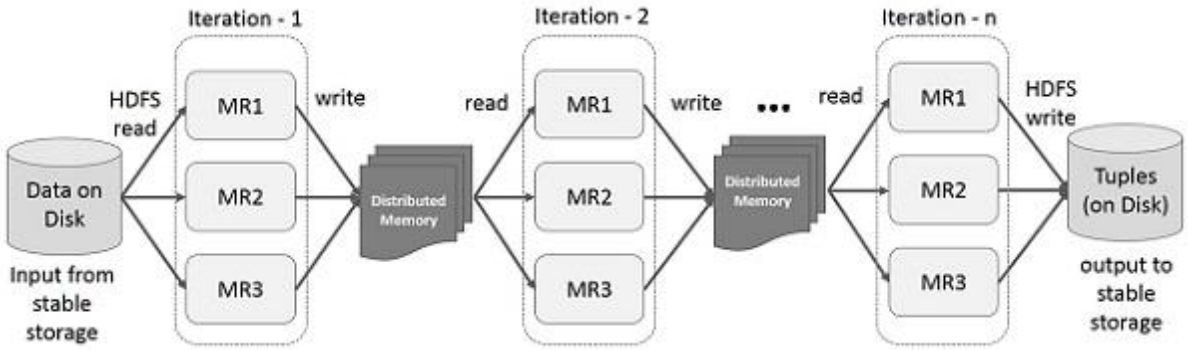


Figure: Iterative operations on Spark RDD

Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.

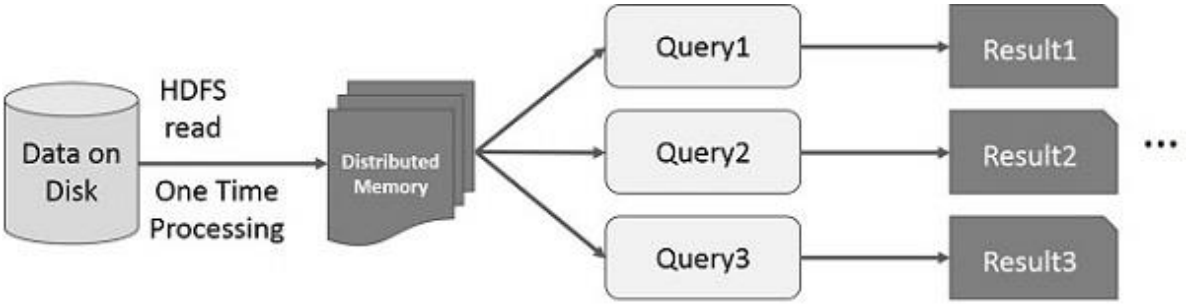


Figure: Interactive operations on Spark RDD

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

3. SPARK SQL – INSTALLATION

Spark is Hadoop's sub-project. Therefore, it is better to install Spark into a Linux based system. The following steps show how to install Apache Spark.

Step 1: Verifying Java Installation

Java installation is one of the mandatory things in installing Spark. Try the following command to verify the JAVA version.

```
$java -version
```

If Java is already, installed on your system, you get to see the following response –

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

In case you do not have Java installed on your system, then Install Java before proceeding to next step.

Step 2: Verifying Scala installation

You should Scala language to implement Spark. So let us verify Scala installation using following command.

```
$scala -version
```

If Scala is already installed on your system, you get to see the following response –

```
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

In case you don't have Scala installed on your system, then proceed to next step for Scala installation.

Step 3: Downloading Scala

Download the latest version of Scala by visit the following link Download Scala. For this tutorial, we are using scala-2.11.6 version. After downloading, you will find the Scala tar file in the download folder.

Step 4: Installing Scala

Follow the below given steps for installing Scala.

Extract the Scala tar file

Type the following command for extracting the Scala tar file.

```
$ tar xvf scala-2.11.6.tgz
```

Move Scala software files

Use the following commands for moving the Scala software files, to respective directory **(/usr/local/scala**).

```
$ su –
Password:
# cd /home/Hadoop/Downloads/
# mv scala-2.11.6 /usr/local/scala
# exit
```

Set PATH for Scala

Use the following command for setting PATH for Scala.

```
$ export PATH = $PATH:/usr/local/scala/bin
```

Verifying Scala Installation

After installation, it is better to verify it. Use the following command for verifying Scala installation.

```
$scala -version
```

If Scala is already installed on your system, you get to see the following response –

```
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

Step 5: Downloading Apache Spark

Download the latest version of Spark by visiting the following link Download Spark. For this tutorial, we are using **spark-1.3.1-bin-hadoop2.6** version. After downloading it, you will find the Spark tar file in the download folder.

Step 6: Installing Spark

Follow the steps given below for installing Spark.

Extracting Spark tar

The following command for extracting the spark tar file.

```
$ tar xvf spark-1.3.1-bin-hadoop2.6.tgz
```

Moving Spark software files

The following commands for moving the Spark software files to respective directory **(/usr/local/spark**).

```
$ su –
Password:

# cd /home/Hadoop/Downloads/
# mv spark-1.3.1-bin-hadoop2.6 /usr/local/spark
# exit
```

Setting up the environment for Spark

Add the following line to **~/.bashrc** file. It means adding the location, where the spark software file are located to the PATH variable.

```
export PATH = $PATH:/usr/local/spark/bin
```

Use the following command for sourcing the ~/.bashrc file.

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

Step 7: Verifying the Spark Installation

Write the following command for opening Spark shell.

```
$spark-shell
```

If spark is installed successfully then you will find the following output.

```
Spark assembly has been built with Hive, including Datanucleus jars on
classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/06/04 15:25:22 INFO SecurityManager: Changing view acls to: hadoop
15/06/04 15:25:22 INFO SecurityManager: Changing modify acls to: hadoop
```

```
15/06/04 15:25:22 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users with view permissions: Set(hadoop); users
with modify permissions: Set(hadoop)

15/06/04 15:25:22 INFO HttpServer: Starting HTTP Server

15/06/04 15:25:23 INFO Utils: Successfully started service 'HTTP class server'
on port 43292.

Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.0
      /_/


Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)

Type in expressions to have them evaluated.

Spark context available as sc


scala>
```

4. SPARK SQL – FEATURES AND ARCHITECTURE

Spark introduces a programming module for structured data processing called Spark SQL. It provides a programming abstraction called DataFrame and can act as distributed SQL query engine.

Features of Spark SQL

The following are the features of Spark SQL:

- **Integrated**: Seamlessly mix SQL queries with Spark programs. Spark SQL lets you query structured data as a distributed dataset (RDD) in Spark, with integrated APIs in Python, Scala and Java. This tight integration makes it easy to run SQL queries alongside complex analytic algorithms.
- **Unified Data Access**: Load and query data from a variety of sources. Schema-RDDs provide a single interface for efficiently working with structured data, including Apache Hive tables, parquet files and JSON files.
- **Hive Compatibility**: Run unmodified Hive queries on existing warehouses. Spark SQL reuses the Hive frontend and MetaStore, giving you full compatibility with existing Hive data, queries, and UDFs. Simply install it alongside Hive.
- **Standard Connectivity**: Connect through JDBC or ODBC. Spark SQL includes a server mode with industry standard JDBC and ODBC connectivity.
- **Scalability**: Use the same engine for both interactive and long queries. Spark SQL takes advantage of the RDD model to support mid-query fault tolerance, letting it scale to large jobs too. Do not worry about using a different engine for historical data.

Spark SQL Architecture

The following illustration explains the architecture of Spark SQL:

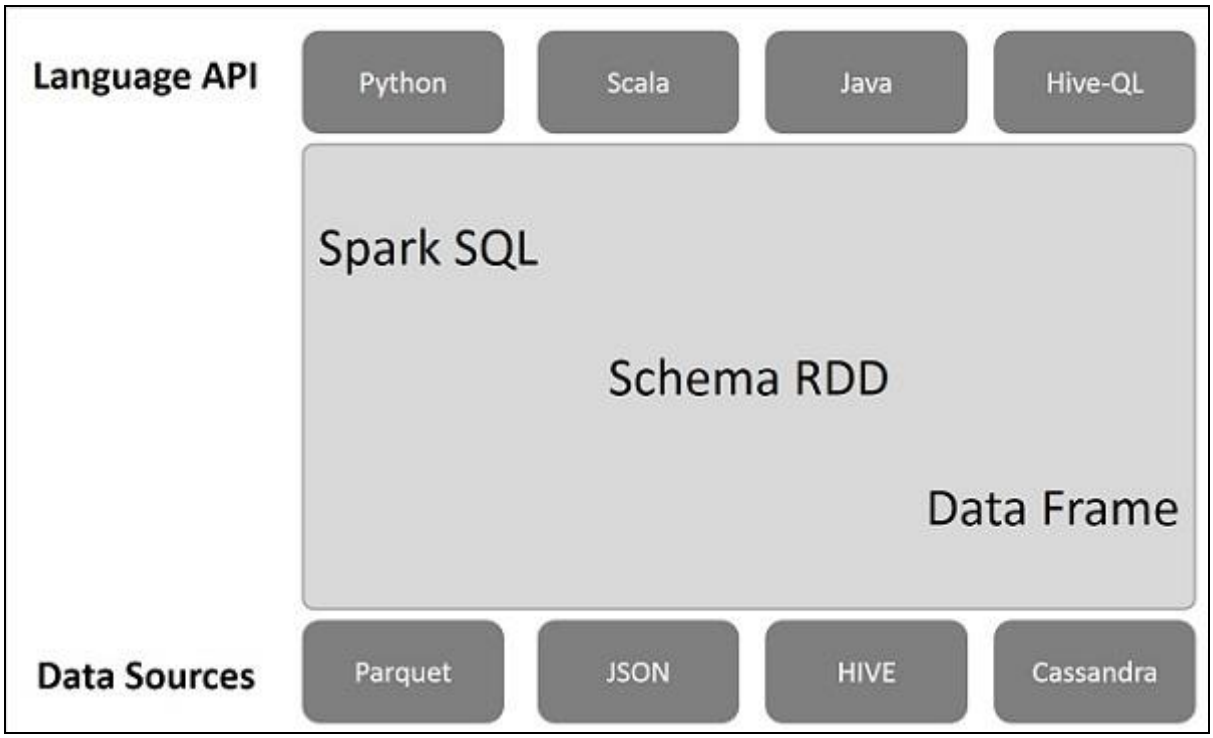


Figure : Spark SQL architecture

This architecture contains three layers namely, Language API, Schema RDD, and Data Sources.

- **Language API:** Spark is compatible with different languages and Spark SQL. It is also, supported by these languages- API (python, scala, java, HiveQL).

- **Schema RDD:** Spark Core is designed with special data structure called RDD. Generally, Spark SQL works on schemas, tables, and records. Therefore, we can use the Schema RDD as temporary table. We can call this Schema RDD as Data Frame.

- **Data Sources:** Usually the Data source for spark-core is a text file, Avro file, etc. However, the Data Sources for Spark SQL is different. Those are Parquet file, JSON document, HIVE tables, and Cassandra database.

We will discuss more about these in the subsequent chapters.

5. SPARK SQL – DATAFRAMES

A DataFrame is a distributed collection of data, which is organized into named columns. Conceptually, it is equivalent to relational tables with good optimization techniques.

A DataFrame can be constructed from an array of different sources such as Hive tables, Structured Data files, external databases, or existing RDDs. This API was designed for modern Big Data and data science applications taking inspiration from **DataFrame in R Programming** and **Pandas in Python**.

Features of DataFrame

Here is a set of few characteristic features of DataFrame:

- Ability to process the data in the size of Kilobytes to Petabytes on a single node cluster to large cluster.
- Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).
- State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
- Provides API for Python, Java, Scala, and R Programming.

SQLContext

SQLContext is a class and is used for initializing the functionalities of Spark SQL. SparkContext class object (sc) is required for initializing SQLContext class object.

The following command is used for initializing the SparkContext through spark-shell.

```
$ spark-shell
```

By default, the SparkContext object is initialized with the name **sc** when the spark-shell starts.

Use the following command to create SQLContext.

```
scala> val sqlcontext = new org.apache.spark.sql.SQLContext(sc)
```

Example

Let us consider an example of employee records in a JSON file named **employee.json**. Use the following commands to create a DataFrame (df) and read a JSON document named **employee.json** with the following content.

employee.json – Place this file in the directory where the current **scala>** pointer is located.

```
{

   {"id" : "1201", "name" : "satish", "age" : "25"}
   {"id" : "1202", "name" : "krishna", "age" : "28"}
   {"id" : "1203", "name" : "amith", "age" : "39"}
   {"id" : "1204", "name" : "javed", "age" : "23"}
   {"id" : "1205", "name" : "prudvi", "age" : "23"}

}
```

DataFrame Operations

DataFrame provides a domain-specific language for structured data manipulation. Here, we include some basic examples of structured data processing using DataFrames.

Follow the steps given below to perform DataFrame operations:

Read the JSON Document

First, we have to read the JSON document. Based on this, generate a DataFrame named (dfs).

Use the following command to read the JSON document named **employee.json.** The data is shown as a table with the fields – id, name, and age.

```
scala> val dfs = sqlContext.read.json("employee.json")
```

Output: The field names are taken automatically from **employee.json**.

```
dfs: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string]
```

Show the Data

If you want to see the data in the DataFrame, then use the following command.

```
scala> dfs.show()
```

Output: You can see the employee data in a tabular format.

```
<console>:22, took 0.052610 s
+---+----+-------+
|age|  id|   name|
+---+----+-------+
| 25|1201| satish|
```

```
| 28|1202|krishna|
| 39|1203|  amith|
| 23|1204|  javed|
| 23|1205| prudvi|
+---+----+-------+
```

Use printSchema Method

If you want to see the Structure (Schema) of the DataFrame, then use the following command.

```
scala> dfs.printSchema()
```

Output

```
root
 |-- age: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
```

Use Select Method

Use the following command to fetch **name-**column among three columns from the DataFrame.

```
scala> dfs.select("name").show()
```

Output: You can see the values of the **name** column.

```
<console>:22, took 0.044023 s
+-------+
|   name|
+-------+
| satish|
|krishna|
|  amith|
|  javed|
| prudvi|
+-------+
```

Use Age Filter

Use the following command for finding the employees whose age is greater than 23 (age > 23).

```
scala> dfs.filter(dfs("age") > 23).show()
```

Output

```
<console>:22, took 0.078670 s
+---+----+-------+
|age|  id|   name|
+---+----+-------+
| 25|1201| satish|
| 28|1202|krishna|
| 39|1203|  amith|
+---+----+-------+
```

Use groupBy Method

Use the following command for counting the number of employees who are of the same age.

```
scala> dfs.groupBy("age").count().show()
```

Output: two employees are having age 23.

```
<console>:22, took 5.196091 s
+---+-----+
|age|count|
+---+-----+
| 23 |    2  |
| 25 |    1  |
| 28 |    1  |
| 39 |    1  |
+---+-----+
```

Running SQL Queries Programmatically

An SQLContext enables applications to run SQL queries programmatically while running SQL functions and returns the result as a DataFrame.

Generally, in the background, SparkSQL supports two different methods for converting existing RDDs into DataFrames:

1. Inferring the Schema using Reflection

2. Programmatically specifying the Schema

Inferring the Schema using Reflection

This method uses reflection to generate the schema of an RDD that contains specific types of objects. The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The **case class** defines the schema of the table. The names of the arguments to the case class are read using reflection and they become the names of the columns.

Case classes can also be nested or contain complex types such as Sequences or Arrays. This RDD can be implicitly be converted to a DataFrame and then registered as a table. Tables can be used in subsequent SQL statements.

Example

Let us consider an example of employee records in a text file named **employee.txt**. Create an RDD by reading the data from text file and convert it into DataFrame using Default SQL functions.

Given Data: Take a look into the following data of a file named **employee.txt** placed it in the current respective directory where the spark shell point is running.

```
1201, satish, 25
1202, krishna, 28
1203, amith, 39
1204, javed, 23
1205, prudvi, 23
```

The following examples explain how to generate a schema using Reflections.

Start the Spark Shell

Start the Spark Shell using following command.

```
$ spark-shell
```

Create SQLContext

Generate SQLContext using the following command. Here, **sc** means SparkContext object.

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Import SQL Functions

Use the following command to import all the SQL functions used to implicitly convert an RDD to a DataFrame.

```
scala> import sqlContext.implicts._
```

Create Case Class

Next, we have to define a schema for employee record data using a case class. The following command is used to declare the case class based on the given data (id, name, age).

```
scala> case class Employee(id: Int, name: String, age: Int)


defined class Employee
```

Create RDD and Apply Transformations

Use the following command to generate an RDD named **empl** by reading the data from **employee.txt** and converting it into DataFrame, using the Map functions.

Here, two map functions are defined. One is for splitting the text record into fields (**.map(_.split(“,”))**) and the second map function for converting individual fields (id, name, age) into one case class object (**.map(e(0).trim.toInt, e(1), e(2).trim.toInt)**).

At last, **toDF()** method is used for converting the case class object with schema into a DataFrame.

```
scala> val empl=sc.textFile("employee.txt")
.map(_.split(","))
.map(e=> employee(e(0).trim.toInt,e(1), e(2).trim.toInt))
.toDF()
```

Output

```
empl: org.apache.spark.sql.DataFrame = [id: int, name: string, age: int]
```

Store the DataFrame Data in a Table

Use the following command to store the DataFrame data into a table named **employee**. After this command, we can apply all types of SQL statements into it.

```
scala> empl.registerTempTable("employee")
```

The employee table is ready. Let us now pass some sql queries on the table using **SQLContext.sql()** method.

Select Query on DataFrame

Use the following command to select all the records from the **employee** table. Here, we use the variable **allrecords** for capturing all records data. To display those records, call **show()** method on it.

```
scala> val allrecords = sqlContext.sql("SELeCT * FROM employee")
```

To see the result data of **allrecords** DataFrame, use the following command.

```
scala> allrecords.show()
```

Output

```
+----+--------+---+
|  id|    name|age|
+----+--------+---+
|1201|  satish| 25|
|1202| krishna| 28|
|1203|   amith| 39|
|1204|   javed| 23|
|1205|  prudvi| 23|
+----+--------+---+
```

Where Clause SQL Query on DataFrame

Use the following command for applying **where** statement in a table. Here, the variable **agefilter** stores the records of employees whose age are between 20 and 35.

```
scala> val agefilter = sqlContext.sql("SELeCT * FROM employee WHERE age>=20 AND
age <= 35")
```

To see the result data of **agefilter** DataFrame, use the following command.

```
scala> agefilter.show()
```

Output

```
<console>:25, took 0.112757 s
+----+--------+---+
|  id|    name|age|
+----+--------+---+
|1201|  satish| 25|
|1202| krishna| 28|
|1204|   javed| 23|
```

```
|1205|  prudvi| 23|
+----+--------+---+
```

The previous two queries were passed against the whole table DataFrame. Now let us try to fetch data from the result DataFrame by applying **Transformations** on it.

Fetch ID values from agefilter DataFrame using column index-

The following statement is used for fetching the ID values from **agefilter** RDD result, using field index.

```
scala> agefilter.map(t=>"ID: "+t(0)).collect().foreach(println)
```

Output

```
<console>:25, took 0.093844 s
ID: 1201
ID: 1202
ID: 1204
ID: 1205
```

This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

Programmatically Specifying the Schema

The second method for creating DataFrame is through programmatic interface that allows you to construct a schema and then apply it to an existing RDD. We can create a DataFrame programmatically using the following three steps.

- Create an RDD of Rows from an Original RDD.
- Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
- Apply the schema to the RDD of Rows via createDataFrame method provided bySQLContext.

Example

Let us consider an example of employee records in a text file named **employee.txt**. Create a Schema using DataFrame directly by reading the data from text file.

Given Data: Look at the following data of a file named **employee.txt** placed in the current respective directory where the spark shell point is running.

```
1201, satish, 25
1202, krishna, 28
1203, amith, 39
```

```
1204, javed, 23
1205, prudvi, 23
```

Follow the steps given below to generate a schema programmatically.

Open Spark Shell

Start the Spark shell using following example.

```
$ spark-shell
```

Create SQLContext Object

Generate SQLContext using the following command. Here, **sc** means SparkContext object.

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Read Input from Text File

Create an RDD DataFrame by reading a data from the text file named **employee.txt** using the following command.

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

Create an Encoded Schema in a String Format

Use the following command for creating an encoded schema in a string format. That means, assume the field structure of a table and pass the field names using some delimiter.

```
scala> val schemaString = "id name age"
```

Output

```
schemaString: String = id name age
```

Import Respective APIs

Use the following command to import Row capabilities and SQL DataTypes.

```
scala> import org.apache.spark.sql.Row;
scala> import org.apache.spark.sql.types.{StructType, StructField, StringType};
```

Generate Schema

The following command is used to generate a schema by reading the **schemaString** variable. It means you need to read each field by splitting the whole string with space as a delimiter and take each field type is String type, by default.

```
scala> val schema = StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
```

Apply Transformation for Reading Data from Text File

Use the following command to convert an RDD (employee) to Rows. It means, here we are specifying the logic for reading the RDD data and store it into rowRDD. Here we are using two map functions: one is a delimiter for splitting the record string (**.map(_.split(","))**) and the second map function for defining a Row with the field index value (**.map(e => Row(e(0).trim.toInt, e(1), e(2).trim.toInt))**).

```
scala> val rowRDD = employee.map(_.split(",")).map(e => Row(e(0).trim.toInt,
e(1), e(2).trim.toInt))
```

Apply RowRDD in Row Data based on Schema

Use the following statement for creating a DataFrame using **rowRDD** data and **schema** (SCHEMA) variable.

```
scala> val employeeDF = sqlContext.createDataFrame(rowRDD, schema)
```

Output

```
employeeDF: org.apache.spark.sql.DataFrame = [id: string, name: string, age:
string]
```

Store DataFrame Data into Table

Use the following command to store the DataFrame into a table named **employee**.

```
scala> employeeDF.registerTempTable("employee")
```

The **employee** table is now ready. Let us pass some SQL queries into the table using the method **SQLContext.sql()**.

Select Query on DataFrame

Use the following statement for selecting all records from the **employee** table. Here we use the variable **allrecords** for capturing all records data. To display those records, call **show()** method on it.

```
scala> val allrecords = sqlContext.sql("SELECT * FROM employee")
```

To see the result data of **allrecords** DataFrame, use the following command.

```
scala> allrecords.show()
```

Output

```
+----+--------+---+
```

```
|  id|    name|age|
+----+--------+---+
|1201|  satish| 25|
|1202| krishna| 28|
|1203|   amith| 39|
|1204|   javed| 23|
|1205|  prudvi| 23|
+----+--------+---+
```

The method **sqlContext.sql** allows you to construct DataFrames when the columns and their types are not known until runtime. Now you can run different SQL queries into it.

6. SPARK SQL – DATA SOURCES

A DataFrame interface allows different DataSources to work on Spark SQL. It is a temporary table and can be operated as a normal RDD. Registering a DataFrame as a table allows you to run SQL queries over its data.

In this chapter, we will describe the general methods for loading and saving data using different Spark DataSources. Thereafter, we will discuss in detail the specific options that are available for the built-in data sources.

There are different types of data sources available in SparkSQL, some of which are listed below:

- JSON Datasets
- Hive Tables
- Parquet Files

JSON Datasets

Spark SQL can automatically capture the schema of a JSON dataset and load it as a DataFrame. This conversion can be done using **SQLContext.read.json()** on either an RDD of String or a JSON file.

Spark SQL provides an option for querying JSON data along with auto-capturing of JSON schemas for both reading and writing data. Spark SQL understands the nested fields in JSON data and allows users to directly access these fields without any explicit transformations.

Example

Let us consider an example of **employee** records in a text file named **employee.json**. Use the following commands to create a DataFrame (df).

Read a JSON document named **employee.json** with the following content and generate a table based on the schema in the JSON document.

employee.json – Place this file into the directory where the current **scala>** pointer is located.

```
{
   {"id" : "1201", "name" : "satish", "age" : "25"}
   {"id" : "1202", "name" : "krishna", "age" : "28"}
   {"id" : "1203", "name" : "amith", "age" : "39"}
   {"id" : "1204", "name" : "javed", "age" : "23"}
   {"id" : "1205", "name" : "prudvi", "age" : "23"}
}
```

Let us perform some Data Frame operations on given data.

DataFrame Operations

DataFrame provides a domain-specific language for structured data manipulation. Here we include some basic examples of structured data processing using DataFrames.

Follow the steps given below to perform DataFrame operations:

Read JSON Document

First of all, we have to read the JSON document. Based on that, generate a DataFrame named **dfs**.

Use the following command to read the JSON document named **employee.json** containing the fields – id, name, and age. It creates a DataFrame named **dfs**.

```
scala> val dfs = sqlContext.read.json("employee.json")
```

Output: The field names are automatically taken from **employee.json**.

```
dfs: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string]
```

Use printSchema Method

If you want to see the Structure (Schema) of the DataFrame, then use the following command.

```
scala> dfs.printSchema()
```

Output

```
root
 |-- age: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
```

Show the data

If you want to show the data in the DataFrame, then use the following command.

```
scala> dfs.show()
```

Output: You can see the employee data in a tabular format.

```
<console>:22, took 0.052610 s
+---+----+-------+
|age|  id|   name|
+---+----+-------+
```

```
| 25|1201| satish|
| 28|1202|krishna|
| 39|1203|  amith|
| 23|1204|  javed|
| 23|1205| prudvi|
+---+----+-------+
```

Then we can run different SQL statements in it. Users can migrate data into JSON format with minimal effort, regardless of the origin of the data source.

Hive Tables

Hive comes bundled with the Spark library as **HiveContext**, which inherits from **SQLContext**. Using HiveContext, you can create and find tables in the HiveMetaStore and write queries on it using HiveQL. Users who do not have an existing Hive deployment can still create a HiveContext. When not configured by the **hive-site.xml**, the context automatically creates a metastore called **metastore_db** and a folder called **warehouse** in the current directory.

Consider the following example of **employee** record using Hive tables. All the recorded data is in the text file named **employee.txt**. Here, we will first initialize the HiveContext object. Using that, we will create a table, load the employee record data into it using HiveQL language, and apply some queries on it.

employee.txt: Place it in the current directory where the spark-shell is running.

```
1201, satish, 25
1202, krishna, 28
1203, amith, 39
1204, javed, 23
1205, prudvi, 23
```

Start the Spark Shell

First, we have to start the Spark Shell. Working with HiveTables means we are working on Hive MetaStore. Hence, the system will automatically create a warehouse for storing table data. Therefore, it is better to run Spark Shell on super user. Consider the following command.

```
$ su
password:
#spark-shell
scala>
```

Create SQLContext Object

Use the following command for initializing the HiveContext into the Spark Shell.

```
scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

Create Table using HiveQL

Use the following command for creating a table named **employee** with the fields **id**, **name**, and **age**. Here, we are using the **Create** statement of **HiveQL** syntax.

```
scala> sqlContext.sql("CREATE TABLE IF NOT EXISTS employee(id INT, name STRING,
age INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY
'\n'")
```

Load Data into Table using HiveQL

Use the following command for loading the employee record data into the employee table. If it is executed successfully, the given employee records are stored into the **employee** table as per the schema.

```
scala> sqlContext.sql("LOAD DATA LOCAL INPATH 'employee.txt' INTO TABLE
employee")
```

Select Fields from the Table

We can execute any kind of SQL queries into the table. Use the following command for fetching all records using HiveQL select query.

```
scala> val result = sqlContext.sql("FROM employe SELECT id, name, age")
```

To display the record data, call the **show()** method on the result DataFrame.

```
scala> result.show()
```

Output

```
<console>:26, took 0.157137 s
+----+---------+---+
|  id|   name   |age|
+----+---------+---+
|1201| Satish  | 25|
|1202|Krishna | 28|
|1203|  amith  | 39|
|1204|  javed  | 23|
|1205| prudvi  | 23|
```

```
+----+---------+---+
```

Parquet Files

Parquet is a columnar format, supported by many data processing systems. The advantages of having a columnar storage are as follows:

- Columnar storage limits IO operations.
- Columnar storage can fetch specific columns that you need to access.
- Columnar storage consumes less space.
- Columnar storage gives better-summarized data and follows type-specific encoding.

Spark SQL provides support for both reading and writing parquet files that automatically capture the schema of the original data. Like JSON datasets, parquet files follow the same procedure.

Let's take another look at the same example of **employee** record data named **employee.parquet** placed in the same directory where spark-shell is running.

Given data: Do not bother about converting the input data of employee records into parquet format. We use the following commands that convert the RDD data into Parquet file. Place the **employee.json** document, which we have used as the input file in our previous examples.

```
$ spark-shell

Scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)

Scala> val employee = sqlContext.read.json("emplaoyee")

Scala> employee.write.parquet("employee.parquet")
```

It is not possible to show you the parquet file. It is a directory structure, which you can find in the current directory. If you want to see the directory and file structure, use the following command.

```
$ cd employee.parquet/


$ ls

_common_metadata

Part-r-00001.gz.parquet

_metadata

_SUCCESS
```

The following commands are used for reading, registering into table, and applying some queries on it.

Open Spark Shell

Start the Spark shell using following example.

```
$ spark-shell
```

Create SQLContext Object

Generate SQLContext using the following command. Here, **sc** means SparkContext object.

```
scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Read Input from Text File

Create an RDD DataFrame by reading a data from the parquet file named **employee.parquet** using the following statement.

```
scala> val parqfile = sqlContext.read.parquet(“employee.parquet”)
```

Store the DataFrame into the Table

Use the following command for storing the DataFrame data into a table named **employee**. After this command, we can apply all types of SQL statements into it.

```
scala> Parqfile.registerTempTable(“employee”)
```

The employee table is ready. Let us now pass some SQL queries on the table using the method **SQLContext.sql()**.

Select Query on DataFrame

Use the following command for selecting all records from the **employee** table. Here, we use the variable **allrecords** for capturing all records data. To display those records, call **show()** method on it.

```
scala> val allrecords = sqlContext.sql("SELeCT * FROM employee")
```

To see the result data of **allrecords** DataFrame, use the following command.

```
scala> allrecords.show()
```

Output

```
+----+--------+---+
|  id|    name|age|
+----+--------+---+
|1201|  satish| 25|
|1202| krishna| 28|
|1203|   amith| 39|
|1204|   javed| 23|
```

```
|1205|  prudvi| 23|
+----+--------+---+
```