# Introduction to Apache Spark

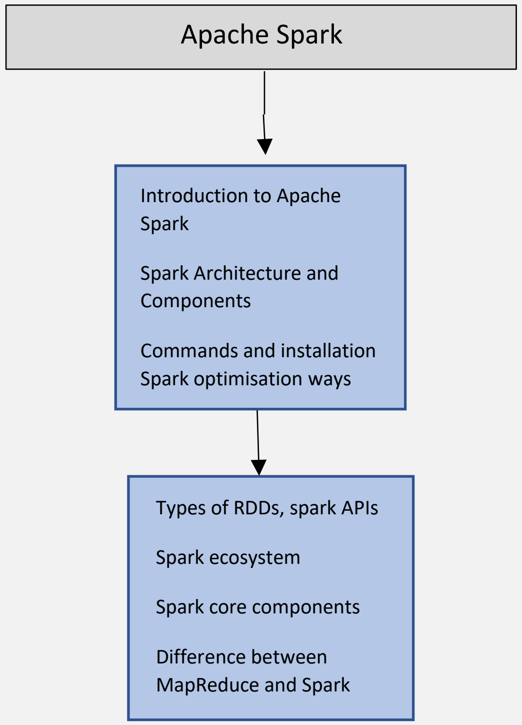
Apache Spark is an open-source data-processing engine for large data sets. It scales by distributing processing work across large clusters of computers, with built-in parallelism and fault tolerance.

As a part of Introduction to Apache Spark, you covered:

- Introduction to Apache Spark and Features
- Apache Spark Ecosystem, Architecture and Components
- Apache Spark commands, optimisation, types of RDDs and APIs
- Difference Between MapReduce and Spark

## Common Interview Questions:

1. What are some of the features of Spark?
2. Explain how Spark runs applications with the help of its architecture
3. What are the different cluster managers available in Apache Spark?
4. How is Apache Spark different from MapReduce?
5. What are the various functionalities supported by Spark Core?
6. How do you convert a Spark RDD into a DataFrame?
7. What are the functions of Spark SQL?
8. What's Spark Driver?


Apache Spark
Introduction to Apache Spark
Spark Architecture and Components
Commands and installation
Spark optimisation ways
Types of RDDs, spark APIs
Spark ecosystem
Spark core components
Difference between MapReduce and Spark

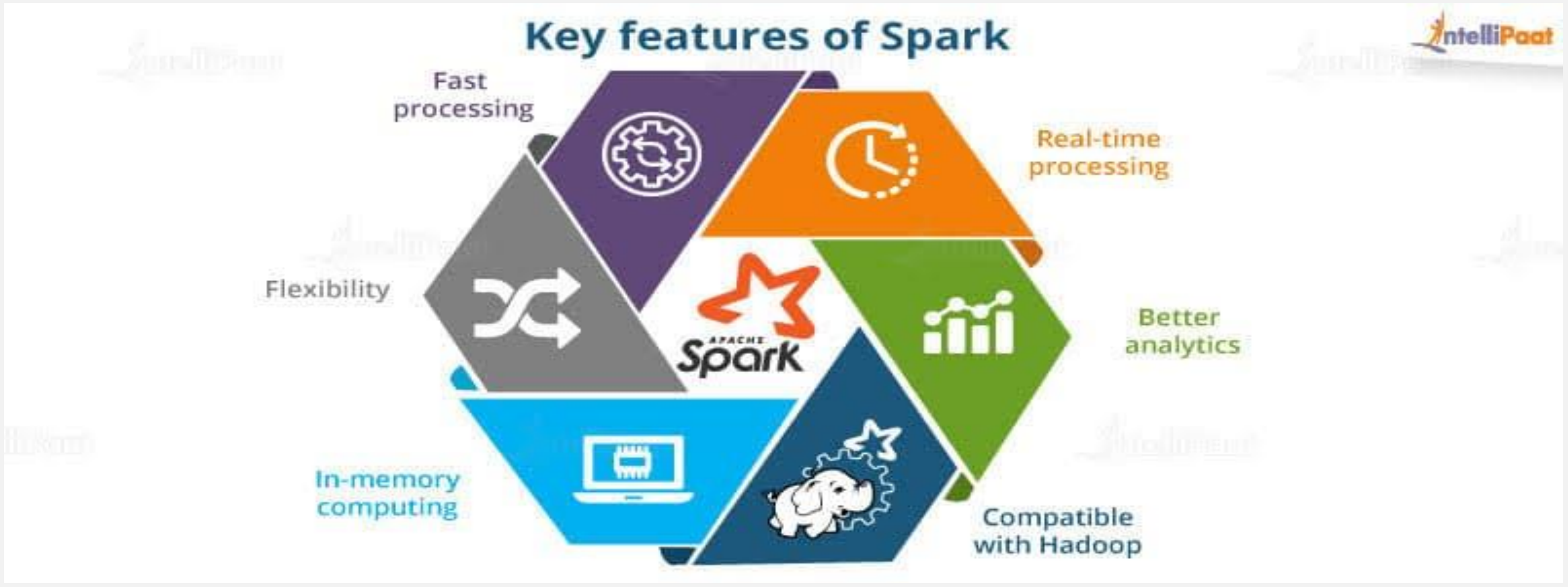
# Introduction to Apache Spark

## Apache Spark:

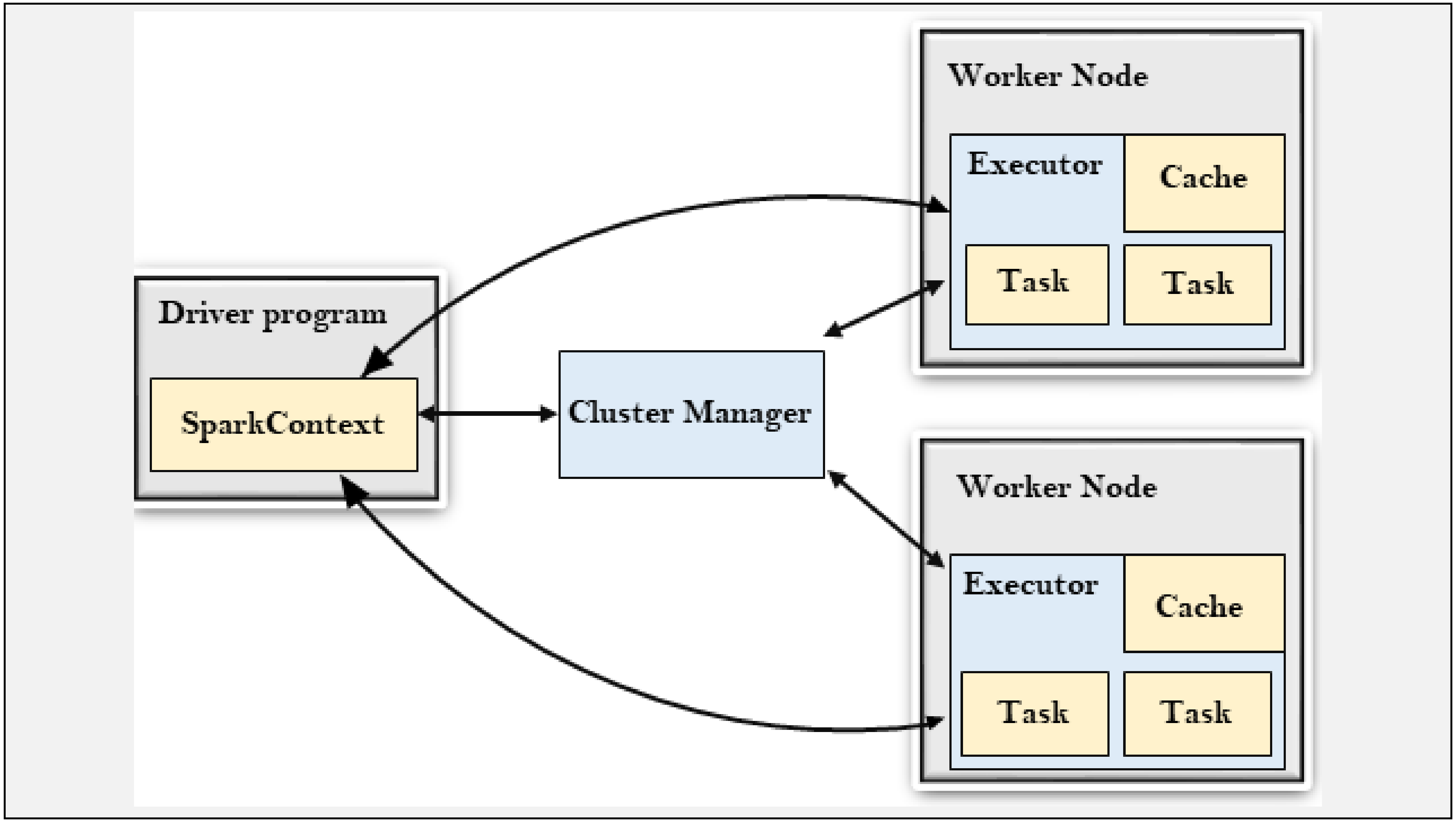
Apache Spark is an open-source data-processing engine for large data sets. It is designed to deliver the computational speed, scalability, and programmability required for Big Data, specifically for streaming data, graph data, machine learning, and artificial intelligence (AI) applications.

It scales by distributing processing work across large clusters of computers, with built-in parallelism and fault tolerance. It even includes APIs for programming languages that are popular among data analysts and data scientists, including Scala, Java, Python, and R.

## Features of Apache Spark:



## Apache Spark Architecture:



# Introduction to Apache Spark

## Apache Spark Architecture Abstraction:

### Resilient Distributed Datasets (RDDs)

**Resilient Distributed Datasets (RDDs)** are fault-tolerant collections of elements that can be distributed among multiple nodes in a cluster and worked on in parallel. RDDs are a fundamental structure in Apache Spark.

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

### Directed Acyclic Graph (DAG)

Spark creates a **Directed Acyclic Graph (DAG)** to schedule tasks and the orchestration of worker nodes across the cluster. As Spark acts and transforms data in the task execution processes, the DAG scheduler facilitates efficiency by orchestrating the worker nodes across the cluster. This task-tracking makes fault tolerance possible, as it reapplies the recorded operations to the data from a previous state.

## Components of Apache Spark Architecture:

**Driver Program:**

The Driver Program is a process that runs the main() function of the application and creates the SparkContext object.

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.

**Worker Node:**

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

# Introduction to Apache Spark

## Components of Apache Spark Architecture:

**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.

**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.

**Task:**

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

## Apache Spark data types: DataFrames and Datasets:

**DataFrames and Datasets in Spark:**

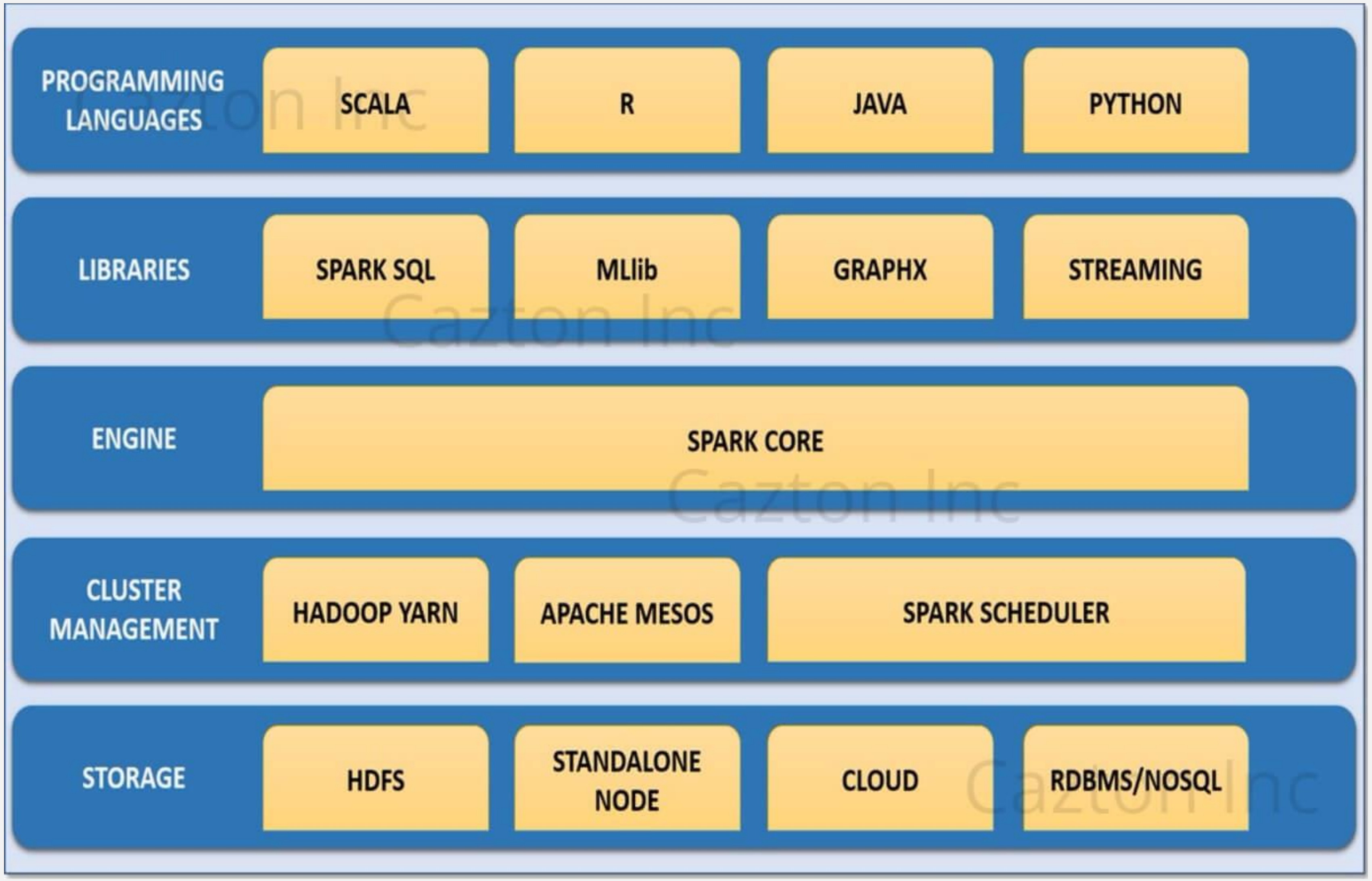
**DataFrames:**

DataFrames are the most common structured application programming interfaces (APIs) and represent a table of data with rows and columns. Because of the popularity of Spark's Machine Learning Library (MLlib), DataFrames have taken on the lead role as the primary API for MLlib. This is important to note when using the MLlib API, as DataFrames provide uniformity across the different languages, such as Scala, Java, Python, and R.

**Datasets:**

Datasets are an extension of DataFrames that provide a type-safe, object-oriented programming interface. Datasets are, by default, a collection of strongly typed JVM objects, unlike DataFrames.

## Apache Spark Ecosystem:



## Apache Spark Core Components:

### 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

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.

### 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

### 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.

# Introduction to Apache Spark

## Spark Installation steps:

**Download spark:**

Download the latest version of spark

**Install 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*

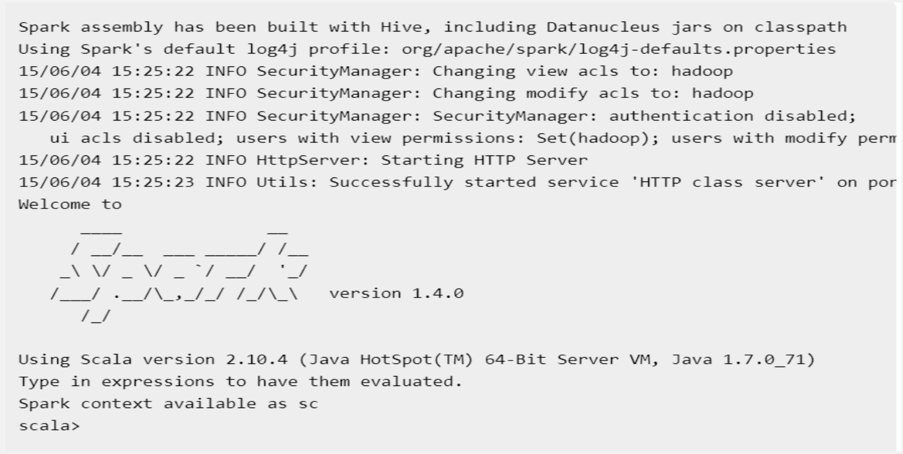
Use the following command for sourcing the *~/.bashrc* file.

**Verifying the Spark Installation:**

Write the following command for opening Spark shell.

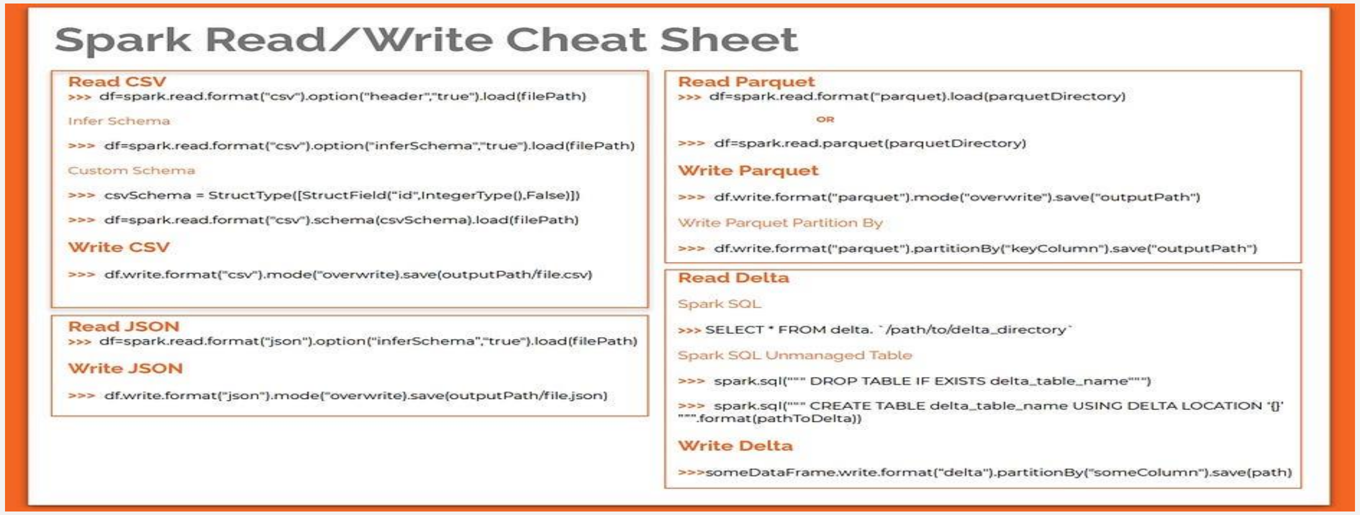
$spark-shell

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 perm
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 por
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>
```

## Spark read write commands:



Spark Read/Write Cheat Sheet

**Read CSV**

```
>>> df=spark.read.format("csv").option("header","true").load(filePath)
```

Infer Schema

```
>>> df=spark.read.format("csv").option("inferSchema","true").load(filePath)
```

Custom Schema

```
>>> csvSchema = StructType([StructField("id",IntegerType(),False)])
>>> df=spark.read.format("csv").schema(csvSchema).load(filePath)
```

**Write CSV**

```
>>> df.write.format("csv").mode("overwrite).save(outputPath/file.csv)
```

**Read JSON**

```
>>> df=spark.read.format("json").option("inferSchema","true").load(filePath)
```

**Write JSON**

```
>>> df.write.format("json").mode("overwrite).save(outputPath/file.json)
```

**Read Parquet**

```
>>> df=spark.read.format("parquet).load(parquetDirectory)
```

OR

```
>>> df=spark.read.parquet(parquetDirectory)
```

**Write Parquet**

```
>>> df.write.format("parquet").mode("overwrite").save("outputPath")
```

Write Parquet Partition By

```
>>> df.write.format("parquet").partitionBy("keyColumn").save("outputPath")
```

**Read Delta**

Spark SQL

```
>>> SELECT * FROM delta. `/path/to/delta_directory`
```

Spark SQL Unmanaged Table

```
>>> spark.sql(""" DROP TABLE IF EXISTS delta_table_name""")
>>> spark.sql(""" CREATE TABLE delta_table_name USING DELTA LOCATION '{}'
""".format(pathToDelta))
```

**Write Delta**

```
>>>someDataFrame.write.format("delta").partitionBy("someColumn").save(path)
```

## Spark Import command:

**Spark Import CSV command:**

*csv_2_df = spark.read.csv("gs://my_buckets/poland_ks")*

**Spark Import JSON command:**

*json_to_df = spark.read.json("gs://my_bucket/poland_ks_json")*

**Spark Import Parquet command:**

*parquet_to_df = spark.read.parquet("gs://my_bucket/poland_ks_parquet")*

**Spark Import Avro command:**

*df = spark.read.format("com.databricks.spark.avro").load("gs://alex_precopro/poland_ks_avro", header = 'true')*

## Types of operation in RDDs:

**Transformations** are a kind of operation that takes an RDD as input and produces another RDD as output. Once a transformation is applied to an RDD, it returns a new RDD, the original RDD remains the same and thus are immutable. After applying the transformation, it creates a Directed Acyclic Graph or DAG for computations and ends after applying any actions on it. This is the reason they are called lazy evaluation processes.

Methods:

**.flatMap()** transformation peforms same as the .map() transformation except the fact that .flatMap() transformation return seperate values for each element from original RDD.

**map()** transformation takes in an anonymous function and applies this function to each of the elements in the RDD.

**.filter()** transformation is an operation in PySpark for filtering elements from a PySpark RDD.

**.union()** transformation combines two RDDs and returns the union of the input two RDDs.

**Actions** are a kind of operation which are applied on an RDD to produce a single value. These methods are applied on a resultant RDD and produces a non-RDD value, thus removing the laziness of the transformation of RDD.

Methods:

**.collect()** action on an RDD returns a list of all the elements of the RDD. It's a great asset for displaying all the contents of our RDD.

**.count()** action on an RDD is an operation that returns the number of elements of our RDD.

**.saveAsTextFile()** Action is used to serve the resultant RDD as a text file. We can also specify the path to which file needed to be saved.

**.reduce()** Actiontakes two elements from the given RDD and operates.

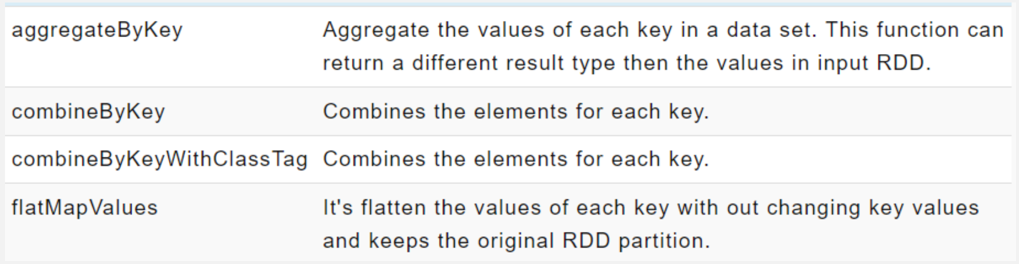
**.take(n)** action on an RDD returns n number of elements from the RDD. The 'n' argument takes an integer which refers to the number of elements we want to extract from the RDD.

# Introduction to Apache Spark

## Paired RDDs:

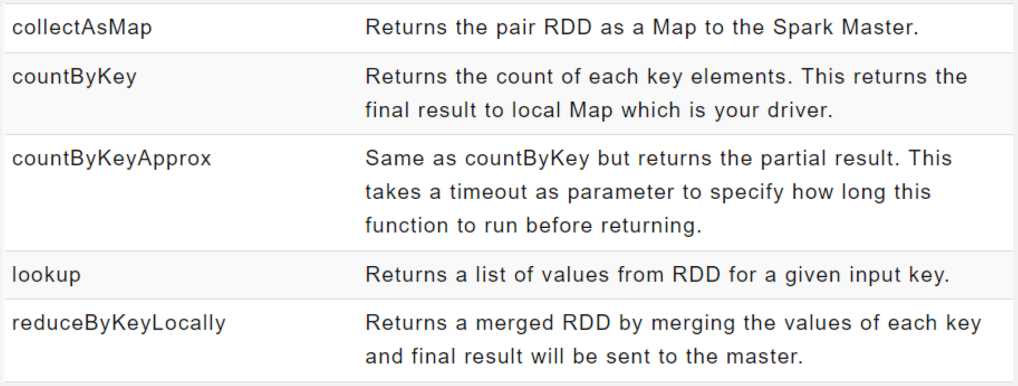
PySpark has a dedicated set of operations for Pair RDDs. Pair RDDs are a special kind of data structure in PySpark in the form of key-value pairs, and that's how it got its name. Practically, the Pair RDDs are used more widely because of the reason that most of the real-world data is in the form of Key/Value pairs.

**Spark transformation function**:



| | |
|---|---|
| aggregateByKey | Aggregate the values of each key in a data set. This function can return a different result type then the values in input RDD. |
| combineByKey | Combines the elements for each key. |
| combineByKeyWithClassTag | Combines the elements for each key. |
| flatMapValues | It's flatten the values of each key with out changing key values and keeps the original RDD partition. |

**Spark action functions:**



| | |
|---|---|
| collectAsMap | Returns the pair RDD as a Map to the Spark Master. |
| countByKey | Returns the count of each key elements. This returns the final result to local Map which is your driver. |
| countByKeyApprox | Same as countByKey but returns the partial result. This takes a timeout as parameter to specify how long this function to run before returning. |
| lookup | Returns a list of values from RDD for a given input key. |
| reduceByKeyLocally | Returns a merged RDD by merging the values of each key and final result will be sent to the master. |

## Spark optimisation ways:

**Spark optimisation ways:**

**Serialization:** Serialization plays an important role in the performance for any distributed application. By default, Spark uses Java serializer.

**API selection:** Spark introduced three types of API to work upon – RDD, DataFrame, DataSet

**Advance Variable:** Broadcasting plays an important role while tuning Spark jobs. Broadcast variable will make small datasets available on nodes locally.

**Cache and persist:** .cache() and persist() will store the dataset in memory.When you have a small dataset which needs be used multiple times in your program, we cache that dataset.

**File Format selection:** Spark supports many formats, such as CSV, JSON, XML, PARQUET, ORC, AVRO, etc. Spark jobs can be optimized by choosing the parquet file with snappy compression which gives the high performance and best analysis.

**Level of Parallelism:** it plays a very important role while tuning spark jobs.

# Introduction to Apache Spark

## Spark Disk:

Spark's operators spill data to disk if it does not fit in memory, allowing it to run well on any sized data. Likewise, cached datasets that do not fit in memory are either spilled to disk or recomputed on the fly when needed, as determined by the RDD's storage level.

## Pandas API:

Using Pandas API on PySpark enables data scientists and data engineers who have prior knowledge of pandas more productive by running the pandas DataFrame API on PySpark by utilizing its capabilities and running pandas operations 10 x faster for big data sets.

By running pandas API on PySpark you will overcome the following challenges.

- Avoids learning a new framework
- More productive
- Maintain single codebase
- Time-consuming to rewrite & testing
- Confusion between pandas vs Spark API
- Finally Error prone

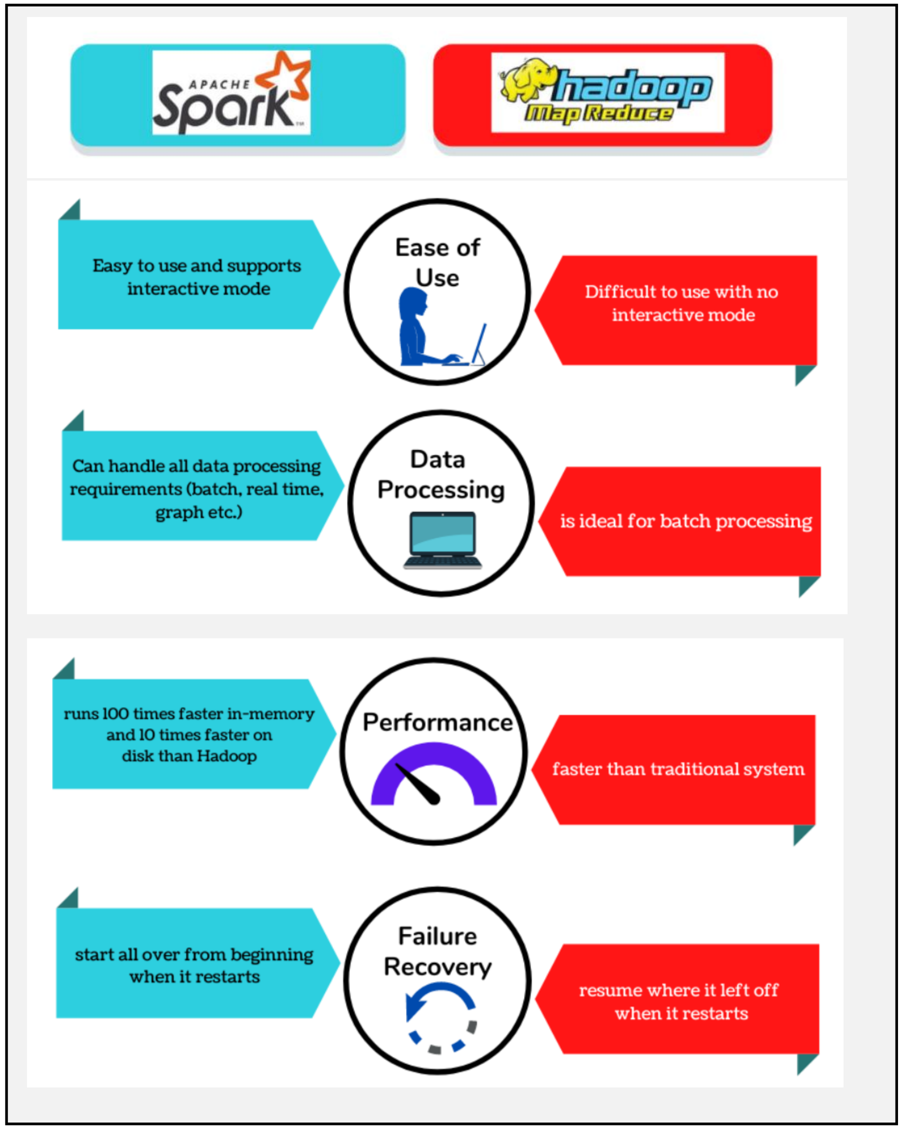
## Dataframe API:

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 or in the Python pandas library. You can construct DataFrames from a wide array of sources, including structured data files, Apache Hive tables, and existing Spark resilient distributed datasets (RDD). The Spark DataFrame API is available in Scala, Java, Python, and R.

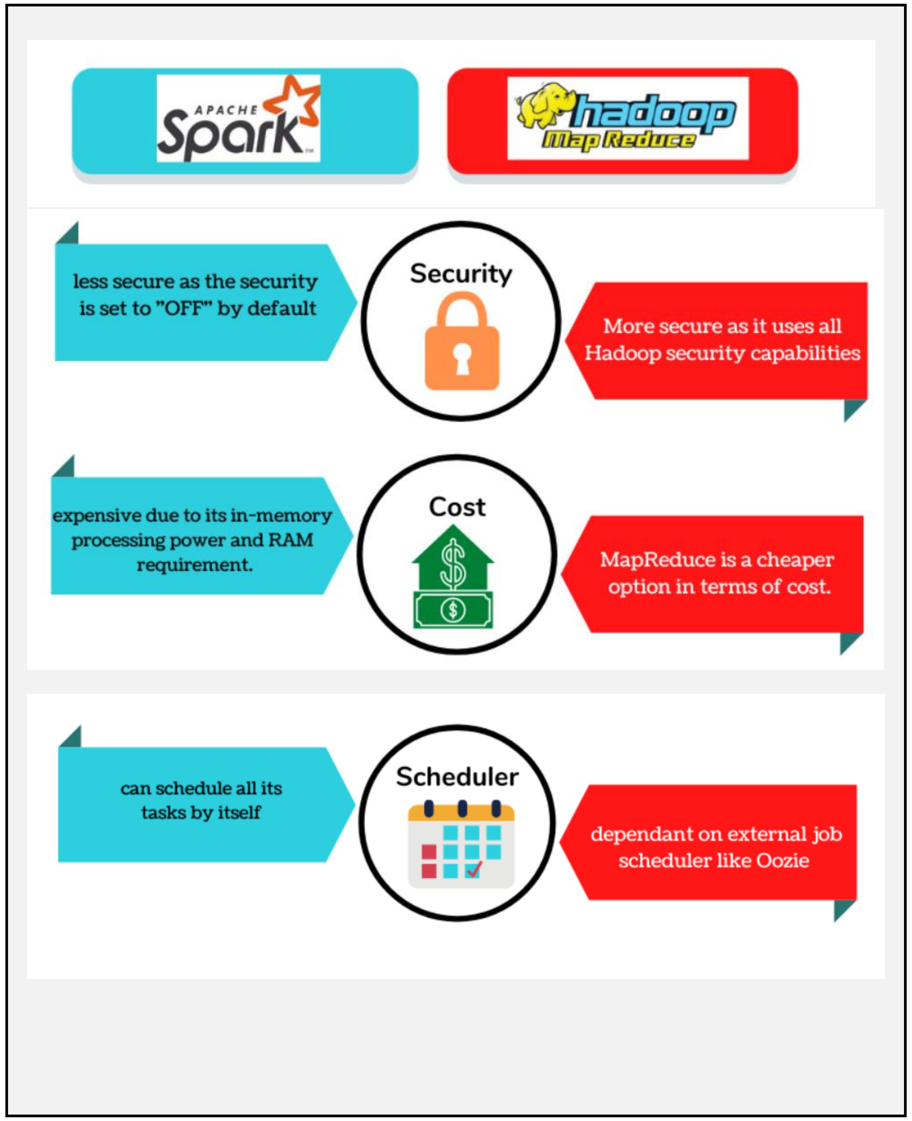
Example on DataFrame API

```
import org.apache.spark.sql._

val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val people = sc.textFile("people.txt")
val schemaString = "name age"

import org.apache.spark.sql.types.{StructType,StructField,StringType}

val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

peopleDataFrame.registerTempTable("people")

val results = sqlContext.sql("SELECT name FROM people")

results.map(t => "Name: " + t(0)).collect().foreach(println)
```

## Difference between Apache Spark and Hadoop MapReduce:



| Apache Spark | | Hadoop MapReduce |
|---|---|---|
| Easy to use and supports interactive mode | Ease of Use | Difficult to use with no interactive mode |
| Can handle all data processing requirements (batch, real time, graph etc.) | Data Processing | is ideal for batch processing |
| runs 100 times faster in-memory and 10 times faster on disk than Hadoop | Performance | faster than traditional system |
| start all over from beginning when it restarts | Failure Recovery | resume where it left off when it restarts |



| Apache Spark | | Hadoop MapReduce |
|---|---|---|
| less secure as the security is set to "OFF" by default | Security | More secure as it uses all Hadoop security capabilities |
| expensive due to its in-memory processing power and RAM requirement. | Cost | MapReduce is a cheaper option in terms of cost. |
| can schedule all its tasks by itself | Scheduler | dependant on external job scheduler like Oozie |