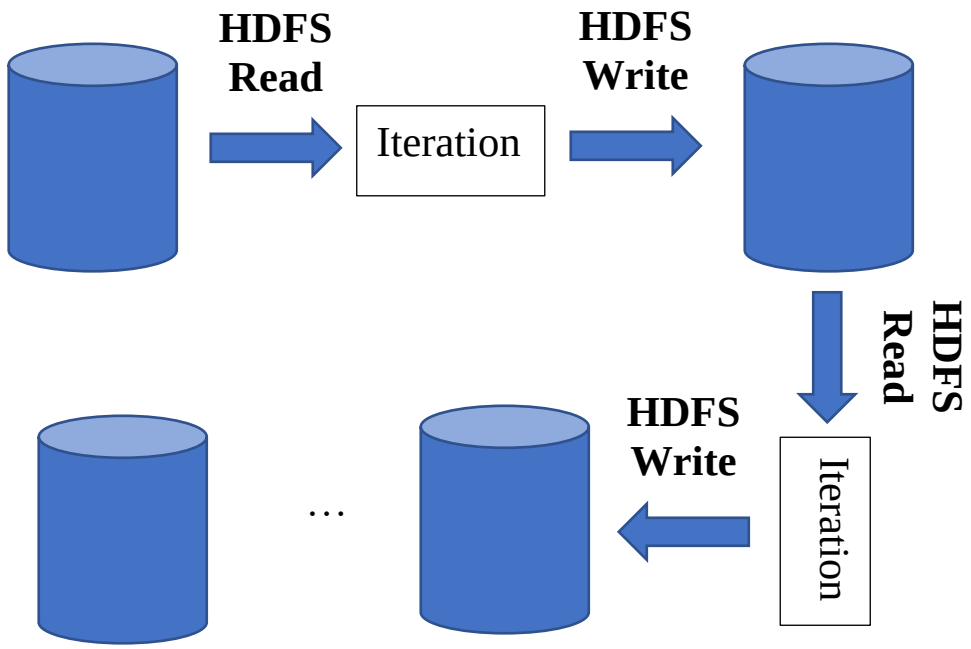
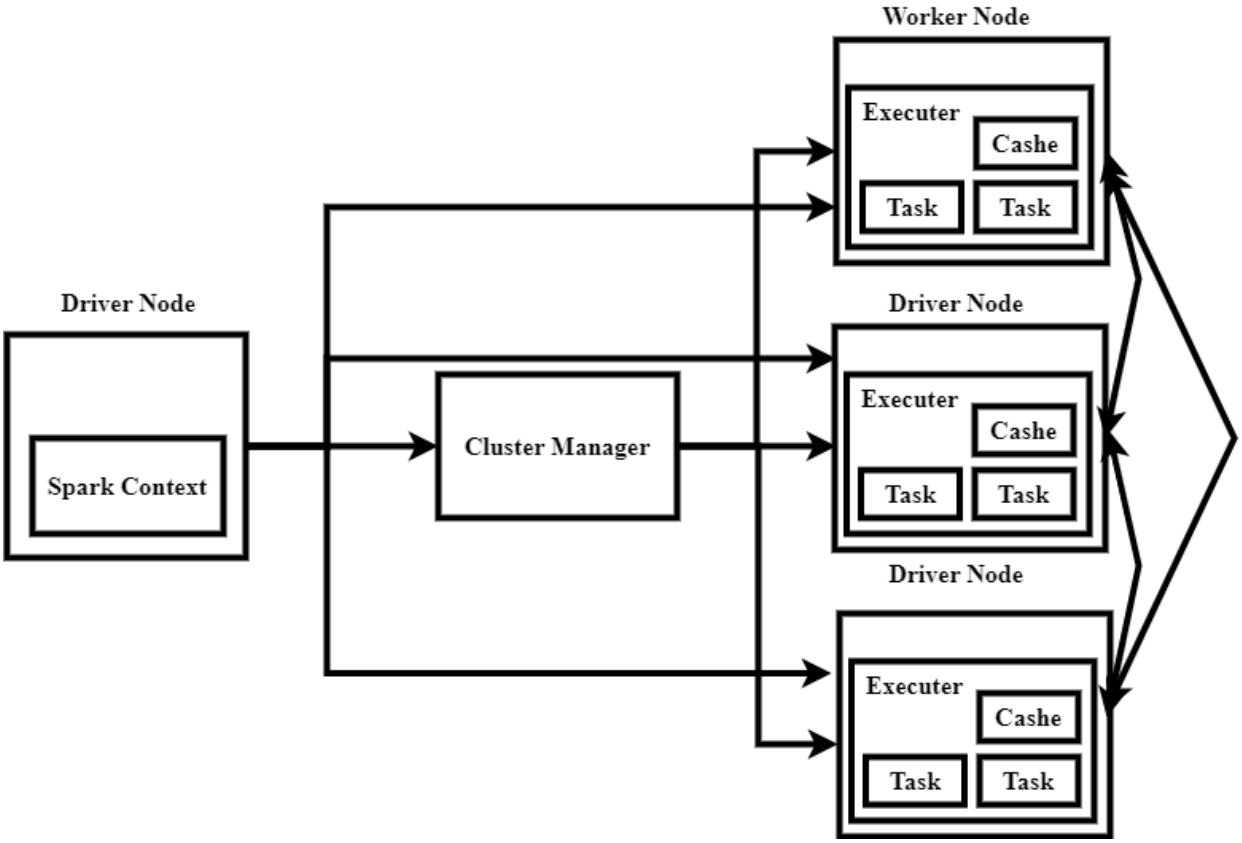
Spark Framework

- **Overview**
  - Apache Spark is an Open-source analytical processing engine for large scale powerful distributed data processing and machine learning applications.
  - It is framework that supports SQL queries, streaming data, machine learning (ML) and graph processing.
  - Spark run 100 times faster in-memory, and 10 times faster on disk.
  - Spark was used to sort 100 TB of data 3 times faster than Hadoop MapReduce on one-tenth of the machines.
  - Apache Spark is a framework that is supported in Scala, Python (PySpark), R Programming, and Java.
  - Hadoop data processing is slow: Too many disk I/Os

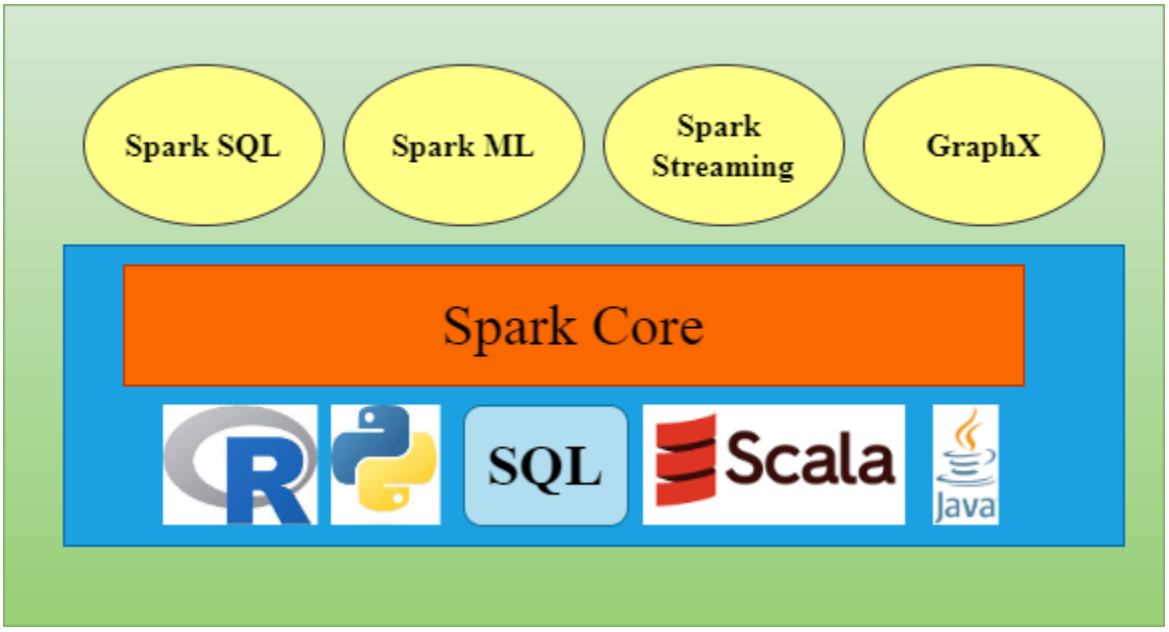


- **Spark Features**
    - **In-memory computation**:
        - Spark stores the data in the RAM of servers which allows quick access and in turn accelerates the speed of analytics.
    - **Distributed processing using parallelize.**
        - Can be used with many cluster managers (Spark, Yarn, Mesos etc.)
    - **Flexibility**:
        - Spark supports multiple languages and allows the developers to write applications in Java, Scala, R, or Python.
    - **Fault-tolerant:**
        - Ability to rebuild lost data automatically on failure
    - **Immutable:**
        - Having Immutable data is safer to share across processes.
    - **Lazy evaluation:**
        - Spark Transformations are lazy - they do not compute the results.
        - Spark computes these Transformations when an action requires a result for the driver program.
    - **Cache & persistence**
    - **Inbuild-optimization when using DataFrames.**

- **Spark Architecture**
  - Apache Spark works in a master-slave architecture where the master is called Master Node and slaves called Worker Nodes.
  - It had four components: **Spark Driver, Executors, cluster Manager, and Worker Nodes**
  - Driver Node:
    - It consists of a Driver program.
    - The Spark application behaves as a Driver program and create a SparkContext that provides access to all Spark functionalities.
    - The Spark Context talks to your Cluster Manager.
    - Spark application run as independent sets of tasks.
    - The driver program and Spark Context takes care o the job execution within the cluster.
  - Cluster Manager:
    - It can be YARN, Mesos, or your computer if you run your Spark as standalone.
    - The Spark manager in case of Yarn is your Resource manager.
    - Cluster Manager Types: Deployment
      - Spark supports the following 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, cluster manager.
        - Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.
  - Worker Nodes:
    - The Spark application will be split into multiple tasks allocated on multiple Worker nodes.


Worker Node
Executer
Cashe
Task
Task
Driver Node
Driver Node
Spark Context
Cluster Manager
Executer
Cashe
Task
Task
Driver Node
Executer
Cashe
Task
Task

- **Spark Ecosystem**
  - It supports several APIs for data processing and analysis.
  - It includes the following components:
    - Spark SQL
    - Spark Streaming
    - MLLib (Machine Learning)
    - GraphX (Graph Computation)



  - Spark Core
    - Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more.
    - Spark Core provides APIs that define and manipulate resilient distributed datasets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel.

  - Spark SQL

        - Spark SQL allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—
        - It also supports other sources of data, including Hive tables, Parquet, and JSON.

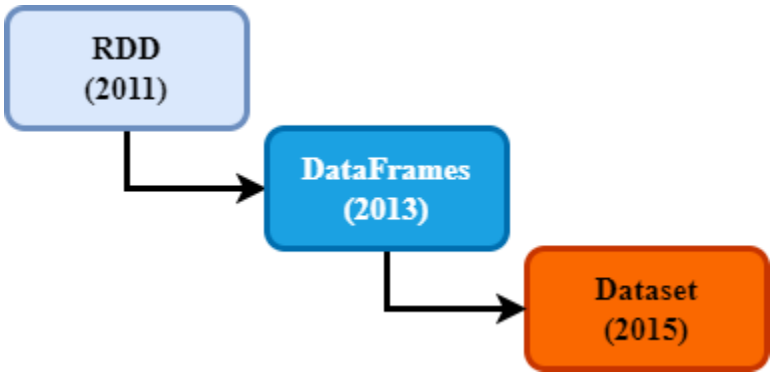
    - Spark Streaming
        - Spark Streaming is a Spark component that enables processing of live streams of data such as logfiles
        - Spark Streaming provides an API for manipulating data streams that closely matches the Spark DD APIs.

    - MLlib
        - Spark comes with a library containing common machine learning (ML) functionality, called MLlib.
        - MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering.

    - GraphX
        - GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations.
        - GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge.
        - GraphX also provides various operators for manipulating graphs (e.g., subgraph) and a library of common graph algorithms (e.g., PageRank and triangle counting).

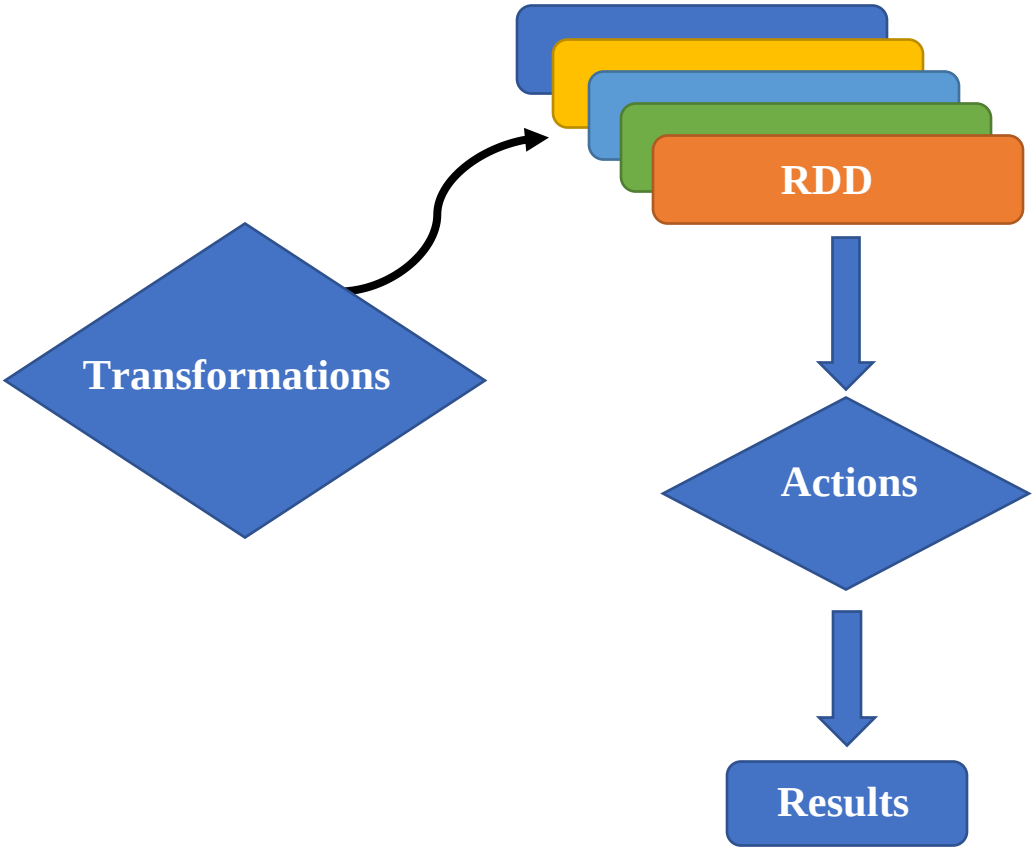
- **Spark Abstract Data Types**

- Spark provides different abstract data structures: Resilient Distributed Dataset (RDD), DataFrame, and Dataset.
  - 2011: Spark was introduced with the concept of RDDs
  - 2013: DataFrames were introduced
  - 2015: Datasets were introduced



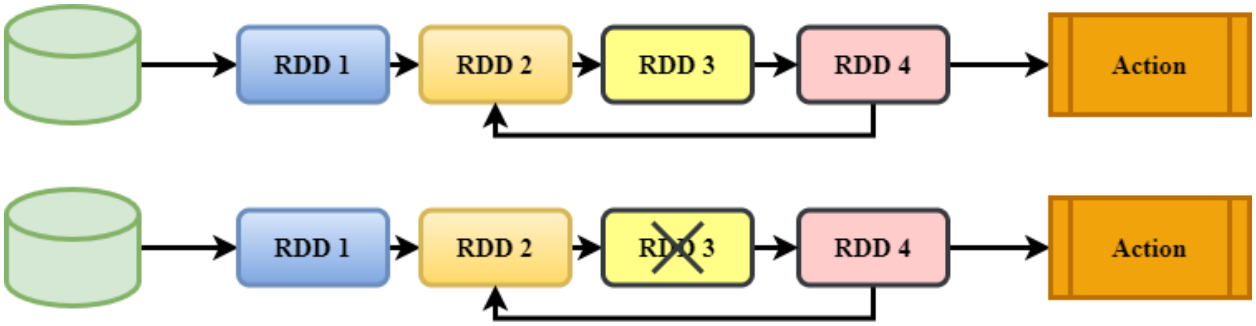
- **Resilient Distributed Dataset (RDD)**
  - RDD is the main data abstraction provided by Spark.
  - They are like list in python.
  - RDD is a fault-tolerant collection of data elements split across Worker Nodes that can be operated on in parallel.
  - RDD are immutable: once they are created, they cannot be modified.
  - **Resilient**:
    - Recover from node failures
    - An RDD keeps its lineage information ➔ It is created from its parent RDD.
  - **Distributed**:
    - Each RDD is composed of one or more partitions:
      more partitions ➔ more parallel

- **Spark Transformations and Actions**
  - Two types of operations:
    - Transformations
    - Actions



  - **Transformations**:
    - They are low level APIs.
    - They produce new RDD from the existing RDDs
    - They take RDD as input and produces one or more RDD as output.
    - Immutable: they cannot be changed.
    - **Lazy evaluation**:
      - Transformations are not executed immediately.
      - They are executed only when an action is called.

- **RDD Lineage**:
  - The sequence of RDDs forms a lineage
  - It is also known as RDD operator graph or RDD dependency graph.
  - It is a logical execution plan of a Spark program.
- Support:
  - Languages: RDD are available for all languages: R, Python, Java, Scala.
  - File Format: textFile, CSV, JSON, Parquet file format/
- Fault-tolerance:



- **Actions**:
  - Spark Actions trigger execution of Spark programs and return values from Spark.
  - They do not produce RDDs.
  - Actions include count, collect, save, etc. (See table below)
  - triggers execution

Action	Description

reduce(func)	aggregate the elements of the dataset using a function func (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to take(1)
take(n)	return an array with the first n elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
countByKey()	only available on RDDs of type (K, V). Returns a `Map` of (K, Int) pairs with the count of each key
foreach(func)	run a function func on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

- **RDD Creation**
  - Three ways to create RDDs:
    - From data in memory
    - From a file or a set of files
    - From another RDD

  - From Memory: Parallelize Collection
    from pyspark.sql. import SparkSession
    spark = SparkSession.builder.getOrCreate()
    myrdd = spark.sparkContext.parallelize([“Jan”, “Feb”, “March”, “April”])
    myrdd.collect()
    [“Jan”, “Feb”, “March”, “April”]

mrdd.count()

4

- From a text file:
  from pyspark.sql. import SparkSession
  spark = SparkSession.builder.getOrCreate()
  myrdd = spark.sparkContext.textFile(“input.txt”)
  or
  myrdd = spark.sparkContext.textFile(“input.txt”, minPartitions=5)

- From another RDD:
  from pyspark.sql. import SparkSession
  spark = SparkSession.builder.getOrCreate()
  myrdd = spark.sparkContext.parallelize([“Jan”, “Feb”, “April”, “Aug”])
  myrdd2 = myrdd.filter(lambfa x.startwith(‘A’))

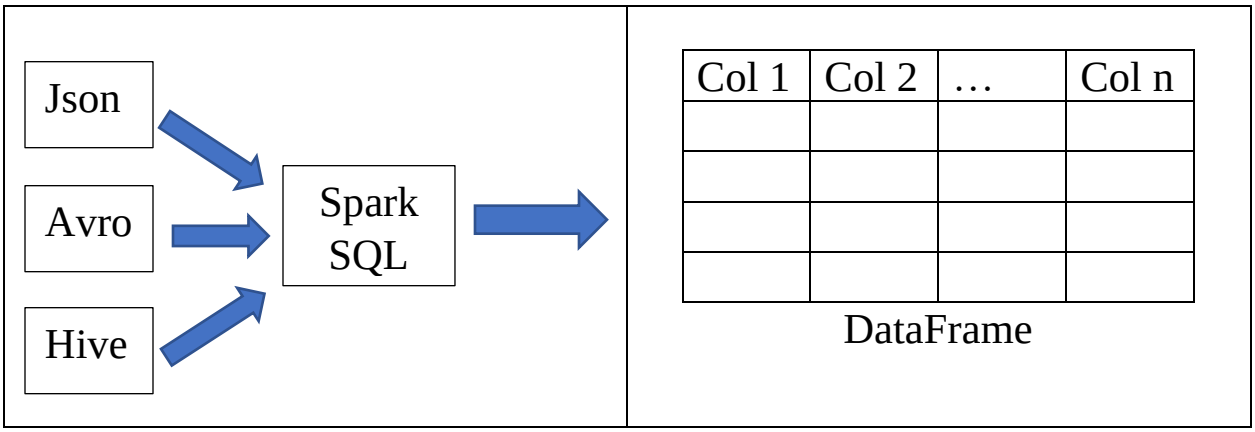
  [”April”,”Aug”]

- **Examples: Wordcount in Python**
  - Consider a word count example: It counts each word appearing in a document.

```
# Apache.org
# Create SparkSession and sparkContext
from pyspark.sql import SparkSession
spark = SparkSession.builder\
                .master("local")\
                .appName('Countprogram')\
                .getOrCreate()
sc=spark.sparkContext
//Read an input file
text_file = sc.textFile("wordfilein.txt")
```

```
counts = text_file.flatMap(lambda line: line.split(" ")) \ //Transformation-spl
each line by space and creating a //single work per record
        .map(lambda word: (word, 1)) \   //Transformation-add a column to
each work
        .reduceByKey(lambda a, b: a + b)  //Transformation
counts.saveAsTextFile("wordfileout.txt")  //Action
```

- **Spark DataFrames**
  - They are immutable high-level APIs
  - DataFrames handle structured and unstructured data.
  - Every DataFrame has a Schema.
    - Data is organized into named columns, like tables in RDMBS or a dataframes in R/Python
  - You can use SQL queries on DataFrame using spark SQL



  - Use **SparkSession** as an entry point to create and manipulate DataFrames. It is similar to SparkContext for RDDs.
  - Dataframes are supported by all languages:
    - APIs available in Scala, Java, Python, and R
  - DataFrame Creation:
    - Create a DataFrame from a list collection using the **toDataFrame**() method from the SparkSession.
      - Consider the following list:

```
students = [{"StdId": 100, "Name": “Mary“, "Grade": ‘A’},
        {"StdId": 200, "Name": “Paul”, "Grade": ‘B’},
        {"StdId": 300, "Name": “John“, "Grade": ‘C’}]

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

df = **spark.createDataFrame**(students)
//Check the type
pyspark.sql.datarame.**DataFrame**
//Display DataFrame
df.show()

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

df.show(1)

StdId	Name	Grade
100	Mary	A

- Convert an RDD to a DataFrame using the **toDF**() method:
  - Given the following list:
    students = [(100, "Mary", 'A'), (200, "Paul", 'B'), (100, "John", 'C')]
    stdRDD = spark.sparkContext.parallelize(students)
    print(stdRDD.collect())

    [(100, "Mary", 'A'), (200, "Paul", 'B'), (100, "John", 'C')]
    //Convert to DataFrame
    stdDF =stdRDD.toDF()
    stdDF.show()

_1	_2	_3
100	Mary	A
200	Paul	B
300	John	C

```
//Specify column names
stdDF =stdRDD.toDF(schema=['StdId', 'Name', 'Grade'])
stdDF.show()
```

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

```
//Another way to create DataFrame
stdDF = spark.createDataFrame(stdRDD, schema=['StdId',
'Name', 'Grade'])
stdDF.show()
```

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

- Import a file into a SparkSession as a **DataFrame** directly.
  - Given a csv file: students.csv

```
StdId,Name,Grade
100,Mary,A
200,Paul,B
300,John,C
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('df_csv_test').getOrCreate()
students = spark.read.csv('students', header=True, inferSchema=True)
students.show()
```

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

- Reading a Json file:
  Using the same data as a Json file:

```
[
    {“stdID”: “100”,
     “name”: “Mary”,
     “Grade”: ‘A’
    }
    {“stdID”: “200”,
     “name”: “Paul”,
     “Grade”: ‘B’
    }
    {“stdID”: “100”,
     “name”: “John”,
     “Grade”: ‘C’
    }
]
```

```
dfjson = spark.read.json(“students.json”)
dfjson.printSchema()
dfjson.show()
```

- Try to create a DataFrame for the following Json file

```
[{"name":"Mary",
  "Courses":[{
        "Algorithms": ‘A’},
     {
        "Data Structure": ‘B’}
    ]}
```

```
{"name":"Paul",
 "Courses":[{
            "Algorithms": ‘B’},
          {
            "Data Structure": ‘A’}
          ]}
{"name":"John",
 "Courses":[{
            "Algorithms": ‘A’},
          {
            "Data Structure": ‘C’}
          ]}
]
```

- DataFrame Operations:
  - In the following, we are going to use the students.csv DataFrame we created earlier:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('df_csv_test').getOrCreate()
students = spark.read.csv('students', header=True, inferSchema=True)
```

  - Show
    students.show()

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

  - Collect: returns the DataFrame as a set of rows
    Students.collect()

[Row (stdID =”100”, name=”Mary”, “Grade=”A”),
Row (stdID =”200”, name=”Paul”, “Grade=”B”),
Row (stdID =”300”, name=”John”, “Grade=”C”)
]

- Take: similar to collect, but you can specify the number of rows you want to return
  Students.take(2)
  [Row (stdID =”100”, name=”Mary”, “Grade=”A”),
  Row (stdID =”200”, name=”Paul”, “Grade=”B”),
  ]

- printSchema: returns the schema of the DataFrame

- count: returns the number of rows is the DataFrame
  students.count()
  3

- select: prints columns from the DataFrame
  students.select(“Name”).show(2)

Name
Mary
Paul

- filter: filters data from the DataFrame

students.filter(students[“Grade” = ‘B’)].show()

StdId	Name	Grade
200	Paul	B

- like: matches certain criteria and return data from the DataFrame

students.select(“Name”).filter(“Name like ‘M%’ “).show()

Name
Mary

- Sort: sorts the DataFrame based on a certain column
  students.sort("Name”).show()

StdId	Name	Grade
300	John	C
100	Mary	A
200	Paul	B

- **Spark Dataset**
  - A Dataset is also a SparkSQL structure and represents an extension of the DataFrame APIs.
  - Spark dataset provides a type-safe, object-oriented programming interface.
  - They provide compile-time safety-check for errors before they run.
  - Dataset APIs offers domain specific operations like sum(), join(), groupBy().
  - Dataset: Only JVM based languages: Java and Scala
  - Datasets can be converted to DataFrame and vice versa.

    DataFrame = Dataset[Row]

    DataFrame is a set of generic untyped row objects.
    Dataset provides typed APIs
  - Dataset creation:
    - Four ways to create a dataset:
      - Create a dataset from sequence of elements
      - Create a dataset from sequence of Case Classes

- Create a dataset from an RDD
- Create a dataset from DataFrame

- Create a dataset from a sequence:

```
scala> val evennum = Seq(0,2,4,6,8)
evennum: Seq[Int] = List(0,2,4,6,8)
scala> val evennumDs = evennum.toDS()
evennumDs: org.apache.spark.sql.Dataset[int] = [value: int]

scala> evennumDs.show
```

value
0
2
4
6
8

- Create a dataset from a sequence of Case Classes:

```
scala> case class students (stdID: Int, name: String, Grade: Char)

defined class students

scala> val studentsSeq = Seq(students(100, “Mary”, ‘A’),
                             students(200, “Paul”, ‘B’),
                             students(300, “John”, ‘C’))

studentsSeq: Seq[students] = List(students(100,Mary,A),
students(200,Paul,B), students(300,John,C))


scala> val studentsDs = studentsSeq.toDS()

studentsDs: org.apache.spark.sql.Dataset[students] = [stdID:Int, name:
String, Grade: Char]
```

scala> studentsDs.show

StdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

- Create a dataset from an RDD:

```
scala> val studentsSeq = Seq(students(100, “Mary”, ‘A’),
                             students(200, “Paul”, ‘B’),
                             students(300, “John”, ‘C’))
studentsSeq: Seq[students] = List(students(100,Mary,A),
students(200,Paul,B), students(300,John,C))
scala> val studentsRDD = spark.sparkContext.parallelize(studentsSeq)
```

scala> val studentsDS = studentsRDD.toDS()

scala> studentsDS.show

_1	_2	_3
100	Mary	A
200	Paul	B
300	John	C

- Create a dataset from a DataFrame:

```
scala> case class students (stdID: Int, name: String, Grade: Char)
defined class students
scala> val studentsSeq = Seq(students(100, “Mary”, ‘A’),
                             students(200, “Paul”, ‘B’),
```

students(300, “John”, ‘C’))

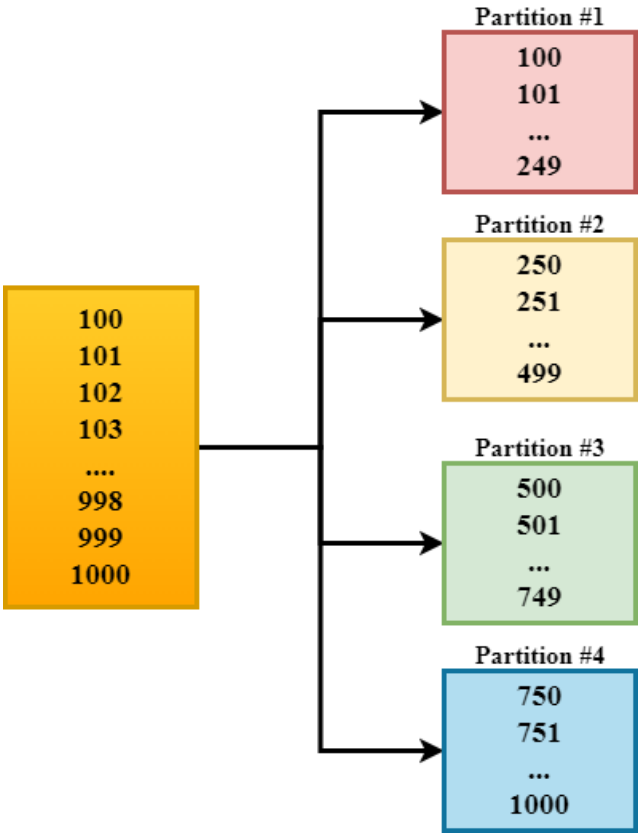
```
scala> val studentsRdd = spark.sparkContext.parallelize(studentsSeq)

scala> val studentsDd = studentsRdd.toDF()

scala> val studentsDs = studentsDf.as[students]

scala> studentsDs.show
```

stdId	Name	Grade
100	Mary	A
200	Paul	B
300	John	C

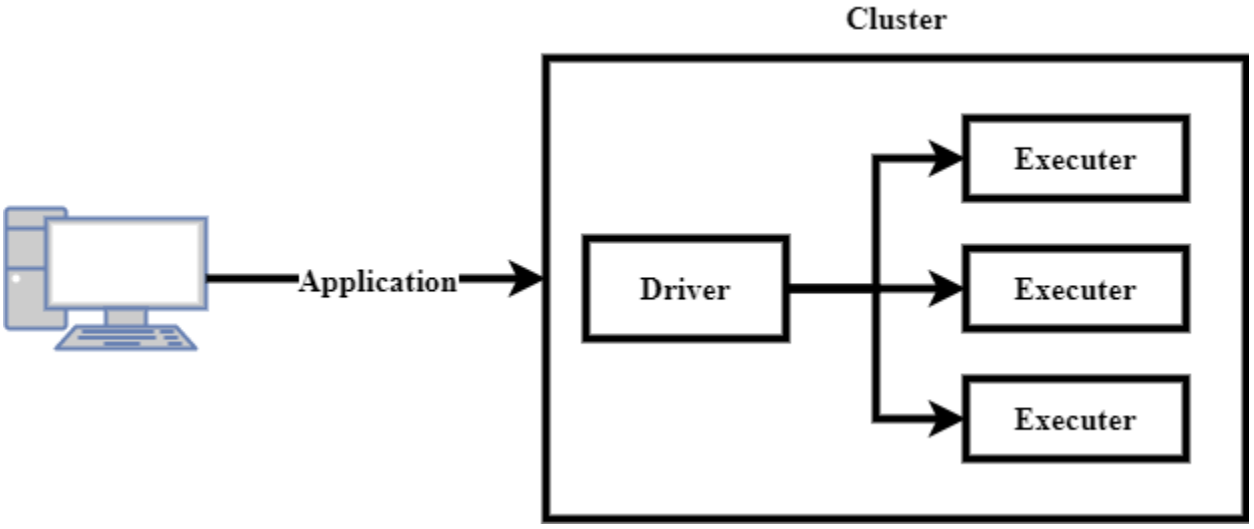
- **Spark RDD, DataFrame, and Dataset**

RDD	DataFrame	Dataset
Spark 1.0	Spark1.3	Spark 1.6
Low level APIs	High level APIs	High level APIs
Lazy evaluation	Lazy evaluation	Lazy evaluation
Type safe	Not type safe	Type sage
Developer takes care of optimization	Auto optimization using Catalyst Optimizer	Auto optimizer
Not good in performance	Not good in performance	Better performance
Not memory efficient	Not memory efficient	More memory efficient

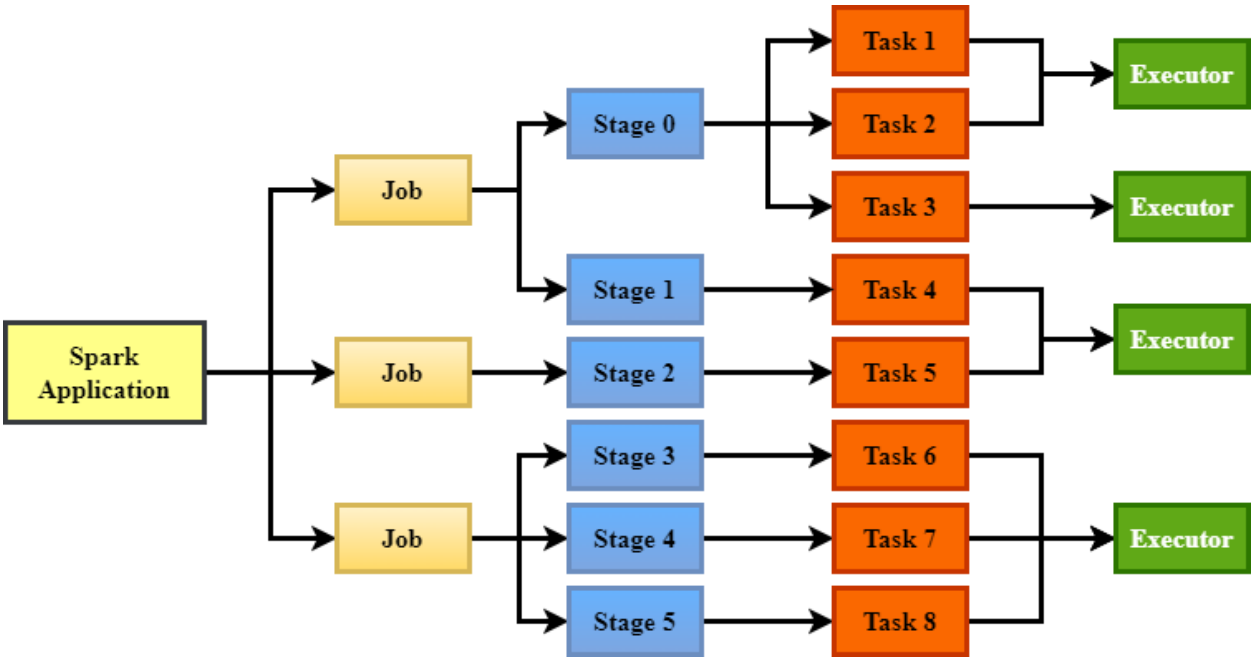
- **Spark Data Partition**
  - Data in a Spark cluster is split into multiple partitions
  - Every node contains one or more partitions.
  - The number of partitions can be configured by the application. By default, it is set to the total number of cores on all the executor nodes.
  - Partitions do not span multiple machines.
  - Spark tries to set the number of partitions automatically based on your cluster.
  - Users can also manually set the number of partitions:
    sc.parallelize(data, 10)).
  - Tuples in the same partition are guaranteed to be on the same machine.
  - Spark assigns one task per partition.
  - Ideal number of partitions:
    - Spark recommends to have 4x of partitions to the number of cores in cluster available for application.
  - Types of partitioning:
    - Hash partitioning
    - Range partitioning
  - Hash Partitioning:
    - Splits our data in such way that elements with the same hash (can be key, keys, or a function) will be in the same partition.
    - We can configure the number of partitions:
      Hash(key) % numPartitions
  - Range Partitioning:
    - RDD are partitioned is based on a range of values of the partitioning key. The key must follow a particular ordering.


100
101
102
103
....
998
999
1000
Partition #1
100
101
...
249
Partition #2
250
251
...
499
Partition #3
500
501
...
749
Partition #4
750
751
...
1000

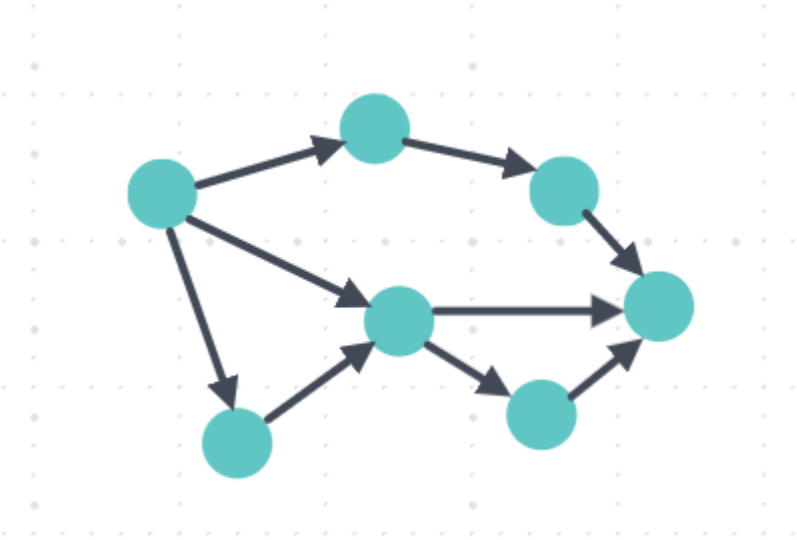
- **Spark Application Execution**
    - For each application, Spark create on driver and a set of executers.
    - The driver is the master: responsible for analyzing, distributing, and scheduling and monitoring work across executors.
    - Executers are responsible of executing the code (tasks)



    - When an application is submitted to Spark, the Driver program is initialed by Spark Context.
    - An application consistes of several job: one per each each action.
    - Once a job is created, it is submitted to a Spark Directed Acyclic Graph (DAG) scheduler (see next)
    - Each job can have multiple stages and each stage may have multiple tasks.



- **Directed Acyclic Graph (DAG)**
  - DAG a finite direct graph with no directed cycles.



  - DAG in Spark is a combination of vertices as well as edges. In DAG vertices represent the RDDs and the edges represent the operation to be applied on RDD.

- When we call an action is encountered, Spark create a DAG and submits it to the DAG Scheduler.
- The DAG scheduler splits the graph into multiple stages.
- Stages:
  - A stage is created for each partition.
  - A stage is divided into individual tasks and every task executes the same set of instructions.
  -
- Tasks:
  - It is the smallest execution unit in Sparke executing a series of narrow transformations inside an executor.
  - Example: Reading a file, filtering and applying map() on data can be combined into a task.

- **Spark Special Variables**
  - Spark supports two types of shared variables:
    - broadcast variables
    - accumulators

  - Broadcast variables:
    - Read-only
    - Should fit into memory on a single machine.
    - They are distributed to the cluster.
    - It can be used to cache a value in memory on all nodes.
    - Create a broadcast variable using Python:
      >> broadcastList = sc.broadcast([3.14, 9.8, 30])
      >> broadcastList.value //to print

      >> ////delete cached copies of the variable on all executors
      >> broadcastList.unpersist
      >> broadcastList.destroy

- Accumulator:
  - They are used by worker nodes to write some data.
  - They can only “added” to, such as counters and sums.
  - Create an accumulator variable to store a count by each task using Python

    >> accumCount = sc.accumulator(0)

    >> sc.parallelize([0,2,4,6,8]).foreach(lambda x: accumCount.add(x))

    >> accumCount.value //print

    20
  - Tasks can only add to an accumulator variable and cannot read its value.
  - Only the driver program can read the accumulator’s value.

- **Apache Spark Advantages**
  - Spark is a general-purpose, in-memory, fault-tolerant, distributed processing engine.
  - Spark is 100x faster than traditional systems.
  - Spark can process data from Hadoop HDFS, AWS S3, Azure Blob Storage, and many file systems.
  - Support for both batch and real-time streaming data.
  - Spark includes machine learning and graph libraries.

- **Spark vs. Hadoop**

	Hadoop	Spark
Cost	Less expensive: disk space is cheaper	Cost can increase: requires more RAM as it is in-memory processing.
Performance	Slow: Disk operations	Fast: in-memory operations. May be slower if YARN is used.
Fault Tolerance	High: data replication	RDD: fault-tolerant collections of elements.
Processing	Batch	Batch, stream, and graph processing
Ease of Use	No interactive mode	Interactive mode with support for API for multiple languages
Language Support	Java, Python, R, and C++	Scala, Python, R, and Java
Scalability	High	High
Machine Learning		
Scheduler	External like Zookeeper	Own scheduler