PART 6

Apache Spark

AGENDA

- Apache Spark
- Pre- Requisites of Spark
- Uses of Spark
- Features / Why Spark
- Cluster Managers
- Storage layers of Spark
- Quick Introduction
- RDD
- RDD Operations
- Types of RDD
- Spark Architecture
- Spark Program execution
- Case Study
- Sample Transformations- Scala
- Actions Execution - Scala
- Defining Schema/ DF
- Non _-Defining Schema/DF
- Parquet files

What is Spark

- Open source software tool
- Distributed cluster computing
- Batch/Stream processing
- Integrated with various big data tools
- Interactive querying
- Cluster management system

What is APACHE SPARK?

Limitations of MapReduce

- Use only Java for application building.
- Opt **only for batch processing**. Does not support stream processing.
- Hadoop MapReduce **uses disk-based processing**.

Why Spark?

In the industry, there is a need for a general-purpose cluster computing tool as:

•Hadoop MapReduce can only perform batch processing.

•Apache Storm / S4 can only perform stream processing.

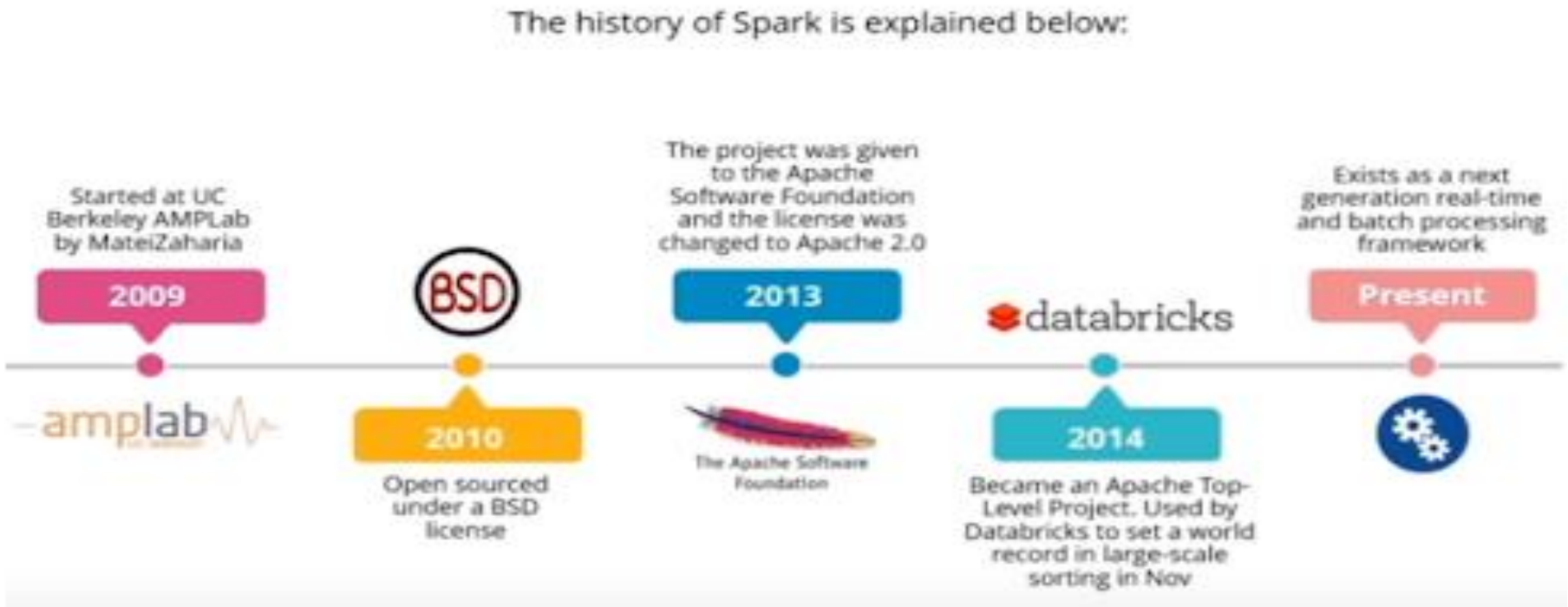
•Apache Impala / Apache Tez can only perform interactive processing

•Neo4j / Apache Giraph can only perform graph processing

- There was a big demand for a powerful engine that can process the **data in real-time (streaming) as well as in batch mode.**

- There was a need for an engine that can respond in sub-second and perform in-memory processing.

History of Spark?

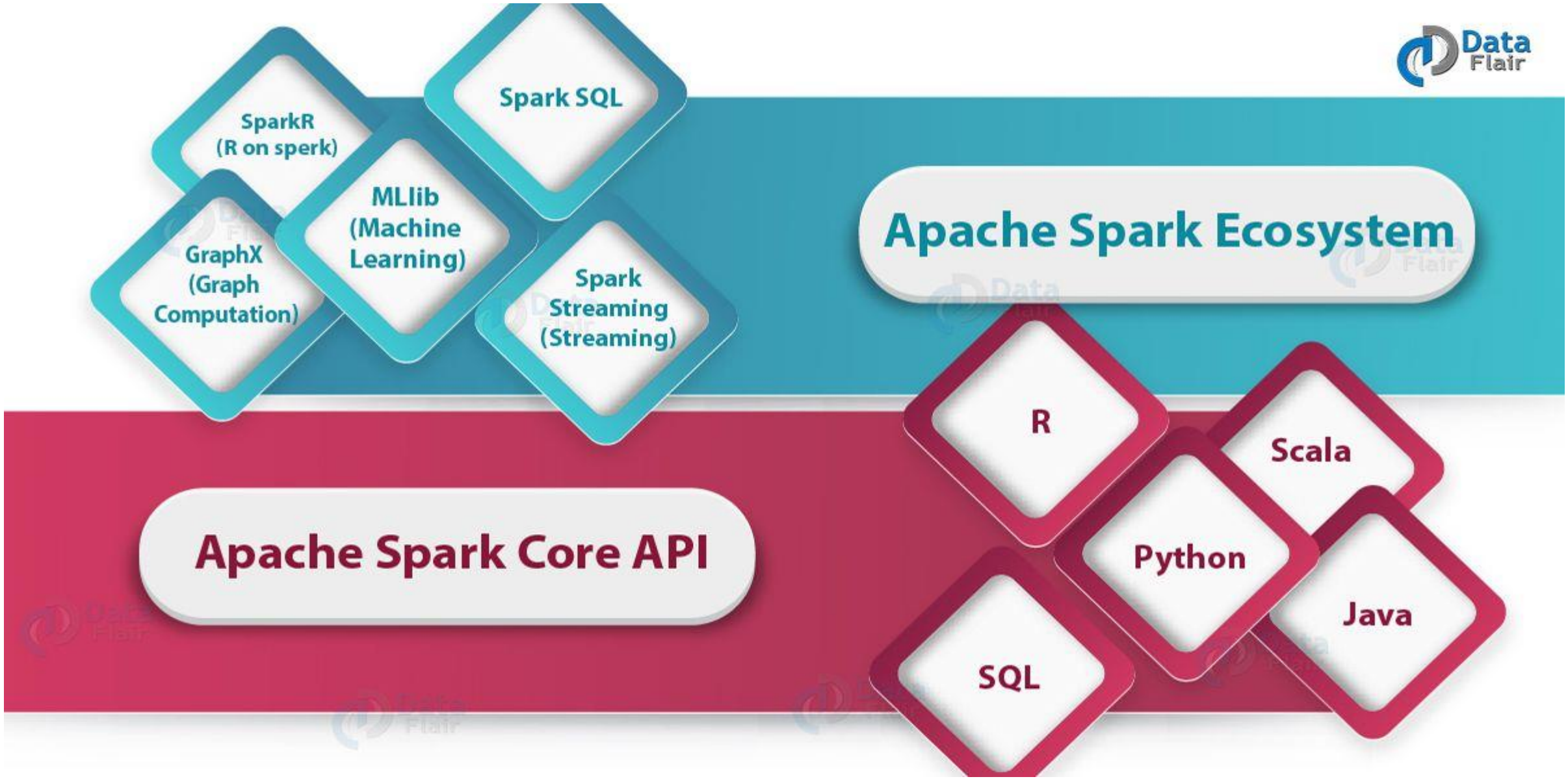


- The Spark was initiated by Matei Zaharia at UC Berkeley's AMPLab in 2009.
- It was open sourced in 2010 under a BSD license.
- In 2013, the project was acquired by Apache Software Foundation.
- In 2014, the Spark emerged as a Top-Level Apache Project.

Apache Spark:

- Cluster computing platform designed to be fast and general purpose.
- In memory computation.
- Designed to cover wide range of workloads(batch applications, machine learning, interactive queries, streaming).
- Combines different processing types.
- Integrates closely with other Big Data tools.

Apache Spark Ecosystem



Hadoop V/S Spark

Basis	Hadoop	Spark
Processing Speed & Performance	Hadoop's MapReduce model reads and writes from a disk, thus slowing down the processing speed.	Spark reduces the number of read/write cycles to disk and stores intermediate data in memory, hence faster-processing speed.
Usage	Hadoop is designed to handle batch processing efficiently.	Spark is designed to handle real-time data efficiently.
Latency	Hadoop is a high latency computing framework, which does not have an interactive mode.	Spark is a low latency computing and can process data interactively.
Data	With Hadoop MapReduce, a developer can only process data in batch mode only.	Spark can process real-time data, from real-time events like Twitter, and Facebook.
Cost	Hadoop is a cheaper option available while comparing it in terms of cost	Spark requires a lot of RAM to run in-memory, thus increasing the cluster and hence cost.
Algorithm Used	The PageRank algorithm is used in Hadoop.	Graph computation library called GraphX is used by Spark.

Hadoop V/S Spark:

Fault Tolerance	Hadoop is a highly fault-tolerant system where Fault-tolerance achieved by replicating blocks of data. If a node goes down, the data can be found on another node	Fault-tolerance achieved by storing chain of transformations If data is lost, the chain of transformations can be recomputed on the original data
Security	Hadoop supports LDAP, ACLs, SLAs, etc and hence it is extremely secure.	Spark is not secure, it relies on the integration with Hadoop to achieve the necessary security level.
Machine Learning	Data fragments in Hadoop can be too large and can create bottlenecks. Thus, it is slower than Spark.	Spark is much faster as it uses MLib for computations and has in-memory processing.
Scalability	Hadoop is easily scalable by adding nodes and disk for storage. It supports tens of thousands of nodes.	It is quite difficult to scale as it relies on RAM for computations. It supports thousands of nodes in a cluster.
Language support	It uses Java or Python for MapReduce apps.	It uses Java, R, Scala, Python, or Spark SQL for the APIs.
User-friendliness	It is more difficult to use.	It is more user-friendly.
Resource Management	YARN is the most common option for resource management.	It has built-in tools for resource management.

Hadoop V/S Apache Spark

Features	Hadoop	Apache Spark
Data Processing	Apache Hadoop provides batch processing	Apache Spark provides both batch processing and stream processing
Memory usage	Hadoop is disk-bound	Spark uses large amounts of RAM
Security	Better security features	Its security is currently in its infancy
Fault Tolerance	Replication is used for fault tolerance.	RDD and various data storage models are used for fault tolerance.
Graph Processing	Algorithms like PageRank is used.	Spark comes with a graph computation library called GraphX.
Ease of Use	Difficult to use.	Easier to use.
Real-time data processing	It fails when it comes to real-time data processing.	It can process real-time data.
Speed	Hadoop's MapReduce model reads and writes from a disk, thus it slows down the processing speed.	Spark reduces the number of read/write cycles to disk and store intermediate data in memory, hence faster-processing speed.
Latency	It is high latency computing framework.	It is a low latency computing and can process data interactively

Pre-Requisites For Spark

- Basic knowledge of Python, Scala, SQL, Linux, Hadoop, Statistics.

Uses of Spark:

- Data processing applications
  - Batch Applications
  - SQL
  - Machine Learning
  - Streaming data processing
  - Graph data processing

Feature of Spark / Why Spark?

- In memory processing.
- Tight integration of components.
- Easy and inexpensive.

Features of Spark

Features of Apache Spark

Swift Processing: swift processing speed of up to 100x faster in memory and 10x faster than Hadoop even when running on disk.

Dynamic: 80 high-level operators, Scala being defaulted language for Spark. We can also work with Java, Python, R.

In – Memory Processing: Disk seeks is becoming very costly, Spark keeps data in memory for faster access. Spark owns advanced DAG execution engine.

Reusability: Apache Spark provides the provision of code reusability for batch processing, join streams against historical data, or run adhoc queries on stream state.

Fault Tolerance: Spark RDD (Resilient Distributed Dataset), abstraction are designed to seamlessly handle failures of any worker nodes in the cluster.
Real-Time Stream Processing

Features of Spark

Polyglot:

Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. It provides a shell in Scala and Python. The Scala shell can be accessed through **./bin/spark-shell** and Python shell through **./bin/pyspark** from the installed directory.

Speed:

Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. Spark is able to achieve this speed through controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.

Features of Spark

Multiple Formats:

Spark supports multiple data sources such as Parquet, JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables. The Data Source API provides a pluggable mechanism for accessing structured data though Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.

Real Time Computation:

Spark's computation is real-time and has low latency because of its in-memory computation. Spark is designed for massive scalability and the Spark team has documented users of the system running production clusters with thousands of nodes and supports several computational models.

Features of Spark

Hadoop Integration:

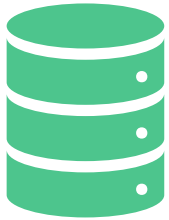
Apache Spark provides smooth compatibility with Hadoop. This is a boon for all the Big Data engineers who started their careers with Hadoop. Spark is a potential replacement for the MapReduce functions of Hadoop, while Spark has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling.

Machine Learning:

Spark's MLlib is the machine learning component which is handy when it comes to big data processing. It eradicates the need to use multiple tools, one for processing and one for machine learning. Spark provides data engineers and data scientists with a powerful, unified engine that is both fast and easy to use.

Cluster Managers:

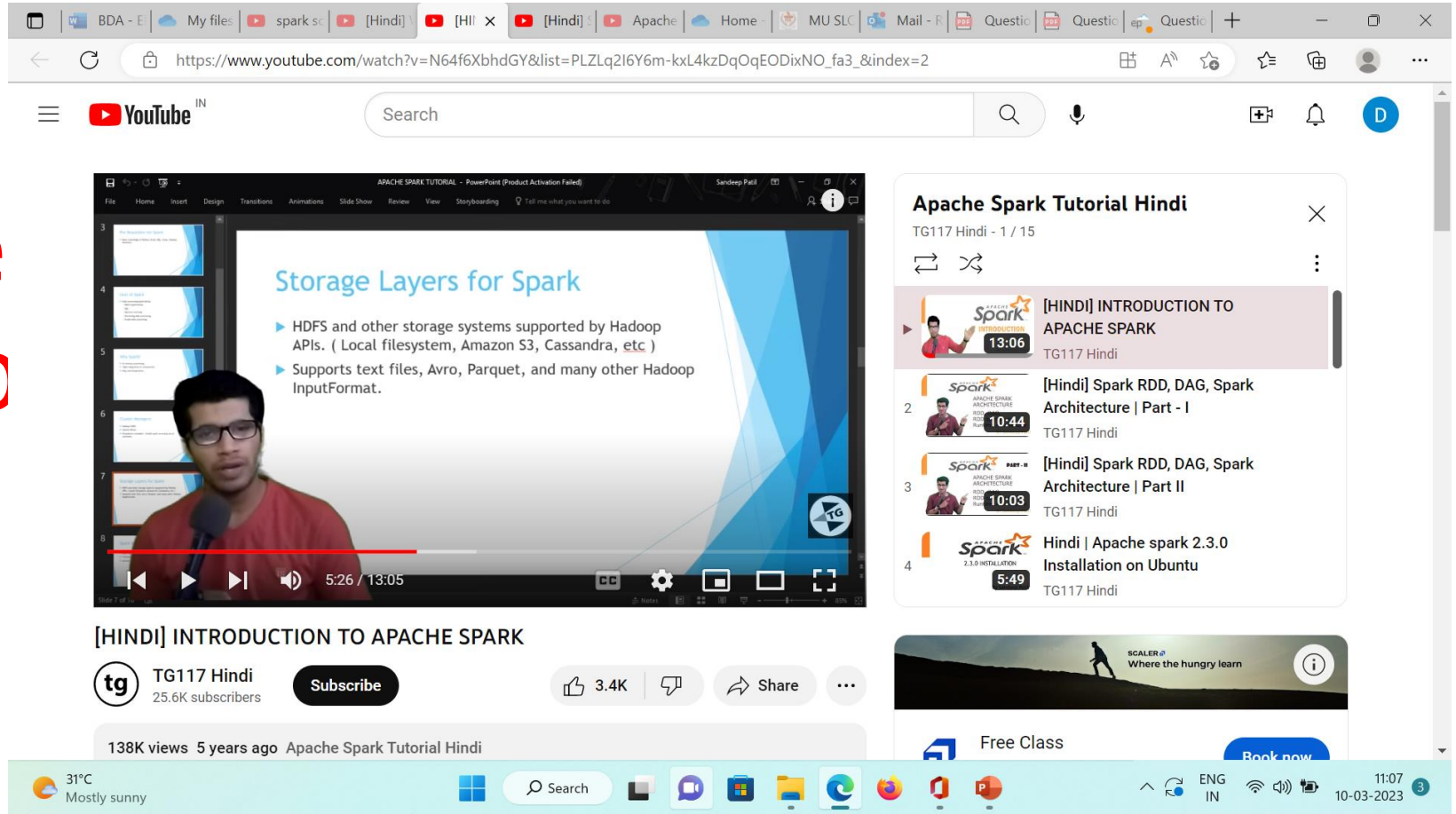
Hadoop YARN



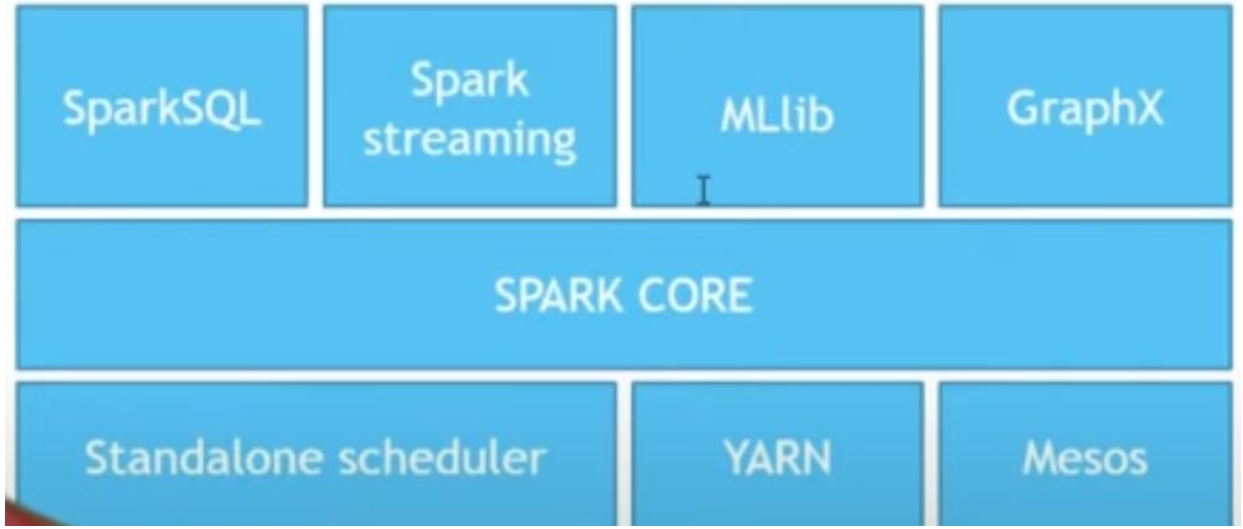
Apache Mesos

Standalone Scheduler : Install Spark on empty set of machines.

Storage of Sp



Unified Stack:



- Base layer = Cluster Managers
- Middle layer = Spark Core (Engine of spark)
  - Spark Core = It will manage:
  - Task Scheduling, Distributing, Monitoring applications
- Top Layer = Own components of spark to perform various operations

SL.NO	Hadoop	Apache Spark
1	It does not have its own components to handle : structured data (Hive), machine learning (Install Mahout)	It has its own components to perform various processing types.
2	Lot of time required to install, deploy & maintain	Installation Time, Deployment time, Maintenance time is reduced.
3	Not much suitable	If new application to be developed – it should perform various workloads like query streaming, ML, graph data processing etc. So spark is suitable

Spark Components

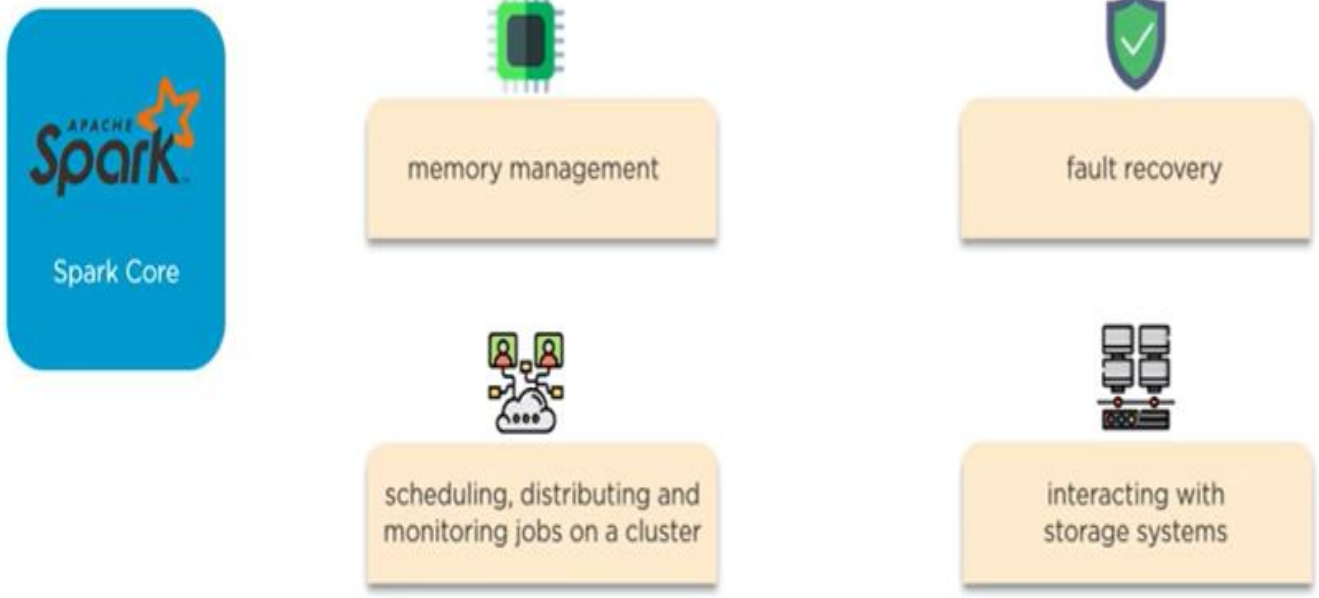
- Spark components are what make Apache Spark fast and reliable. A lot of these Spark components were built to resolve the issues that cropped up while using Hadoop MapReduce. Apache Spark has the following components:

- **Spark Core**
- **Spark SQL**
- **Spark Streaming**
- **Spark GraphX**
- **MLlib (Machine Learning)**

Spark Core:

- Contains basic functionality of Spark. (task scheduling, memory management, fault recovery, interacting with storage systems)
- Home to API that defines RDDs

Spark Core:



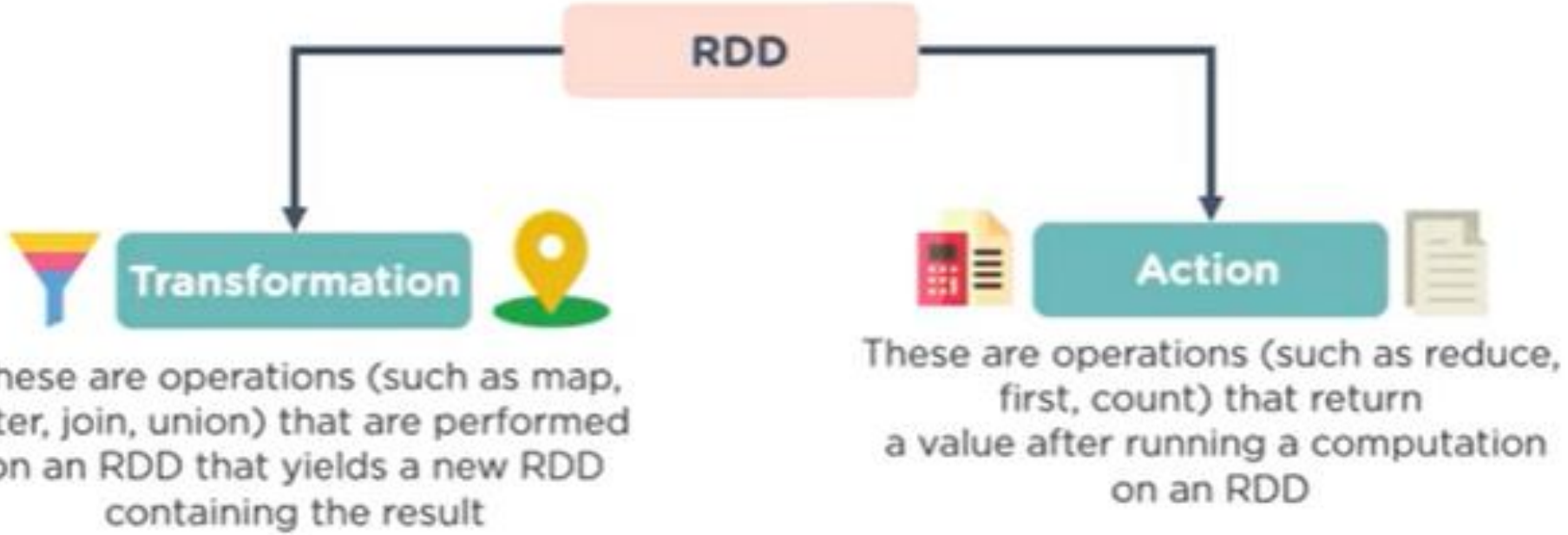
1. Spark Core

- *Spark Core* is the base engine for large-scale parallel and distributed data processing.

- The core is the distributed execution engine and the Java, Scala, and Python APIs offer a platform for distributed ETL application development.
- Further, additional libraries which are built on the core allow diverse workloads for streaming, SQL, and machine learning.
- It is responsible for:
  - ✓ Memory management and fault recovery
  - ✓ Scheduling, distributing and
  - ✓ monitoring jobs on a cluster
  - ✓ Interacting with storage systems

Spark Core:

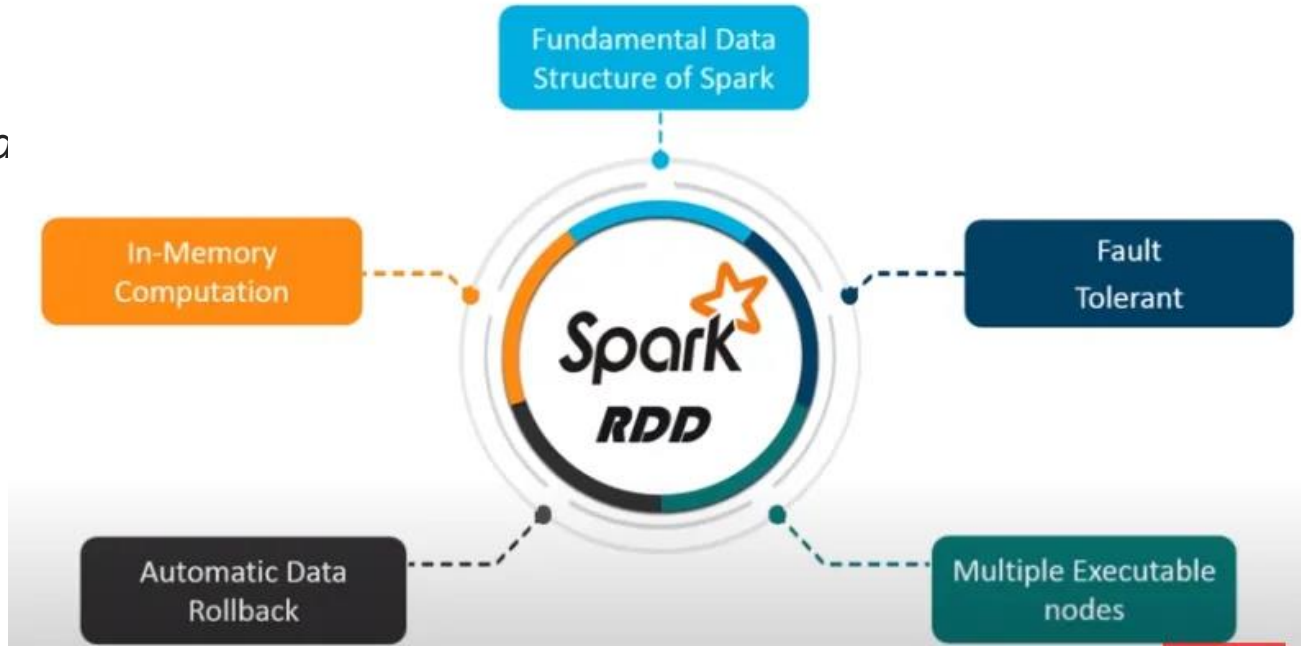
Spark Core is embedded with RDDs (Resilient Distributed Datasets), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel



What are RDDs?

- **RDD** or (**Resilient Distributed Data set**) is a fundamental **data structure** in Spark.
- The term **Resilient** defines the ability that generates the data automatically or data **rolling back** to the **original state** when an unexpected calamity occurs with a probability of data loss.

  - RDD does **not need** something like **Hard Disks** or any other secondary storage. **It needs only RAM**.
  - *RDD is not a Distributed File System instead it is Distributed Memory System.*
  - **Data can be loaded from any source to Apache Spark like Hadoop, HBase, Hive, SQL, S3**
  - **They can process any type of data such as Structured, Unstructured, Semi-Structured data**.



Operations performed on RDDs

Transformations :

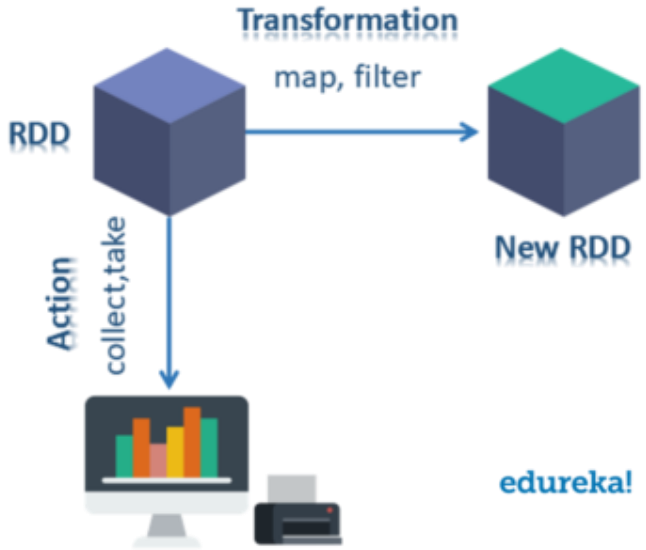
- **filter, access** and **modify** the data in parent RDD to generate a **successive RDD**
- The new RDD returns a pointer to the previous RDD ensuring the dependency between them.
- Transformations are **Lazy Evaluations**
- **1. Narrow Transformations**

  - **map() , filter()**
  - **flatMap(), partition()**
  - **mapPartitions()**

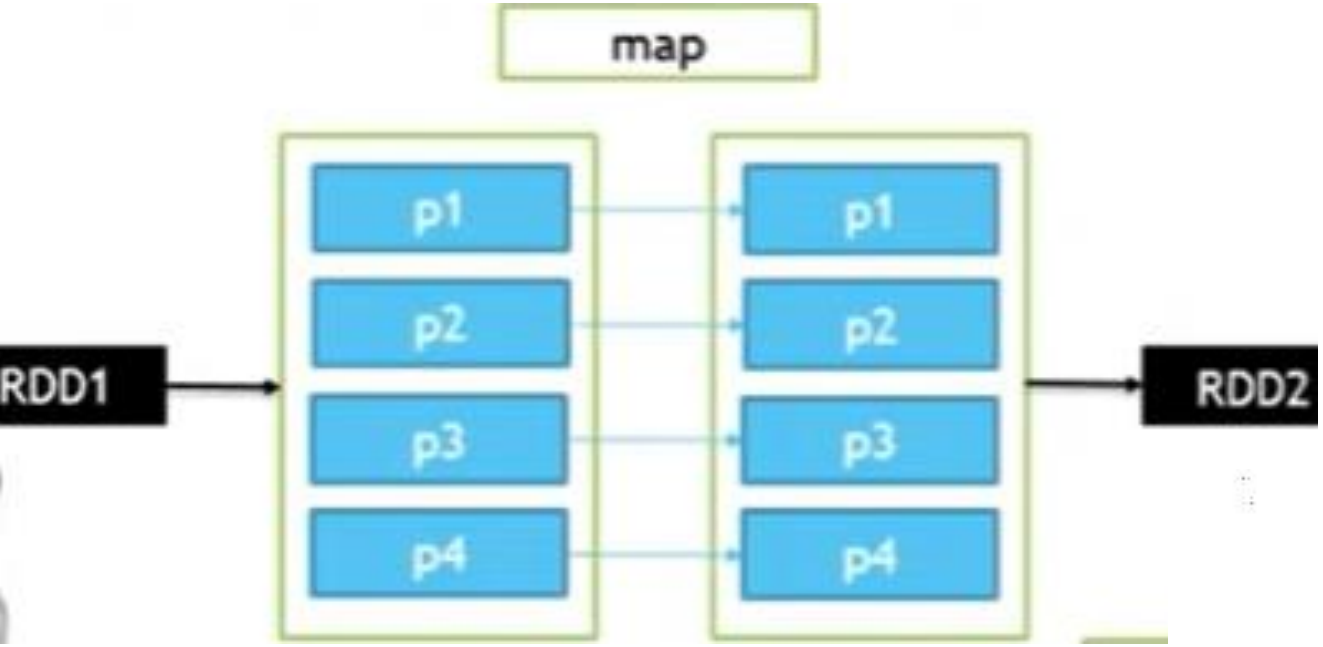
2. Wide Transformations

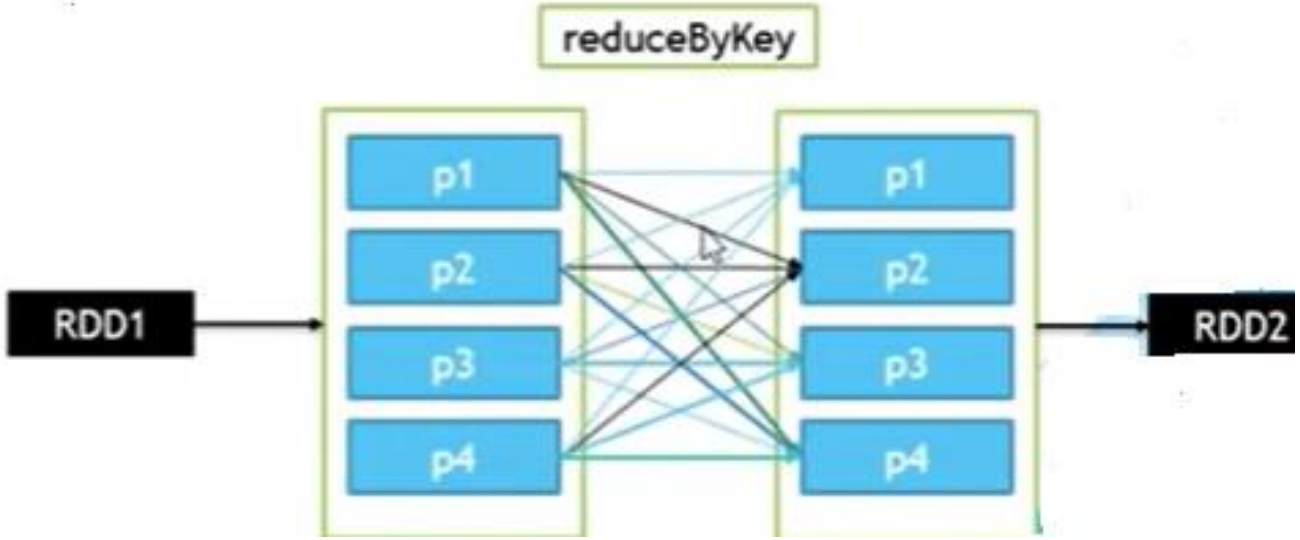
- **reduceBy()**
- **union()**
- **Intersection()**
- **Join()**

Actions: Actions instruct Apache Spark to apply **computation** and pass the result or an exception back to the driver RDD. Few of the actions include:**collect()**, **count(), take()**, **first()**



Narrow & Wide Transformations





RDD: Resilient Distributed Dataset

RDD Once done – It is immutable but can overridden

If u want to modify “data” which is already there in specific RDD it cant be done , instead need to make “new RDD”

If data to be overridden –it can be successfully performed (means data from another file can be called in to current RDD, once data is overridden the content of current RDD will be updated with new called file data)

```
Transformations :

var a = sc.texFile("hdfs://.....")
var b = a.flatMap( .... )
var c = b.distinct( .... )
```

- Here a is “RDD”
- Transformation = Process of making RDD’s
  - In above example; b & c are new RDD.
  - Flatmap = Is Transformation
- Actions = Processing the data inside the cluster

RDD:

- Fault tolerant distributed dataset.
- Lazy Evaluation
- Caching
- In memory computation
- Immutability
- Partitioning

RDD

- Unless Actions are not called no Transformations get executed – lazy Evaluation.
- RDD (2 operations)
  - Transformations = process of making chain of RDD's
  - Ex: From a RDD making b RDD
  - During Transformations = No executions take place (in cluster)
  - Actions = Actual Executions take place
  - During this time of transformations – spark makes (RDD's Map) using DAG (which RDD made – sequence of RDD's).

```
Transformations :

var a = sc.texFile("hdfs://...../abc.txt")      --> RDD0
var b = a.filter( .... )                        --> RDD1
var c = b.distinct( .... )                      --> RDD2


Actions

c.collect()
```

Caching:

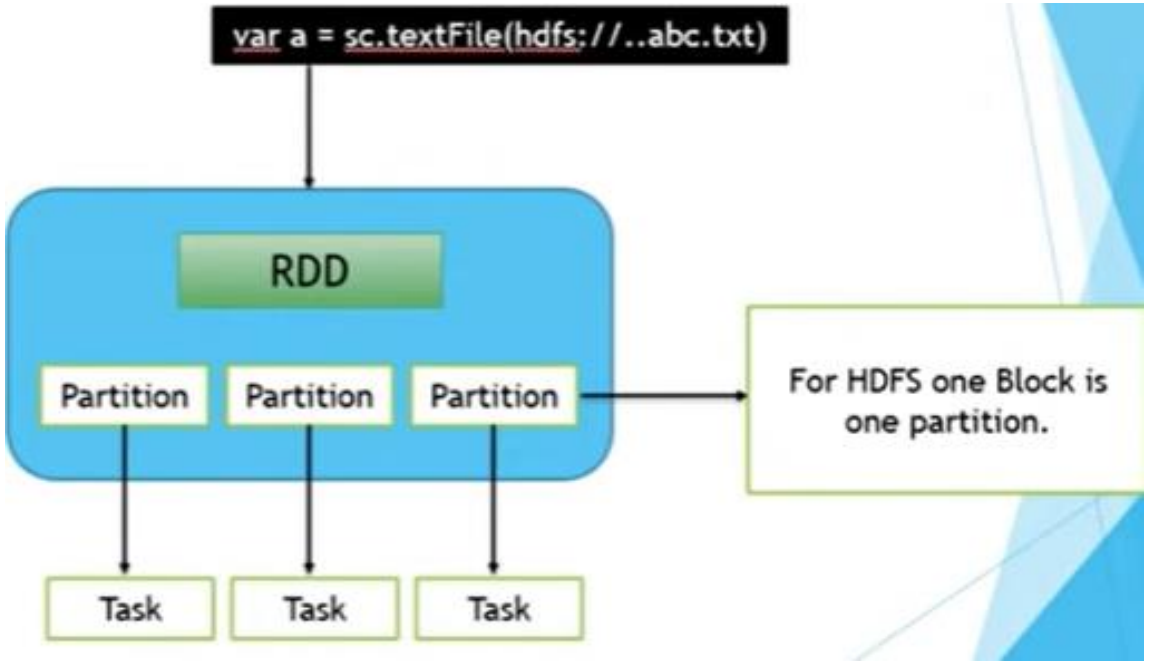
- If any modifications to be made in data we need to make new RDD always .
- Dataset used frequently is transferred to cache.

Immutable:

- If any modifications to be made in data we need to make new RDD always (Immutable).
- Means RDD’s are immutable(we cant change RDD).

Partitioning

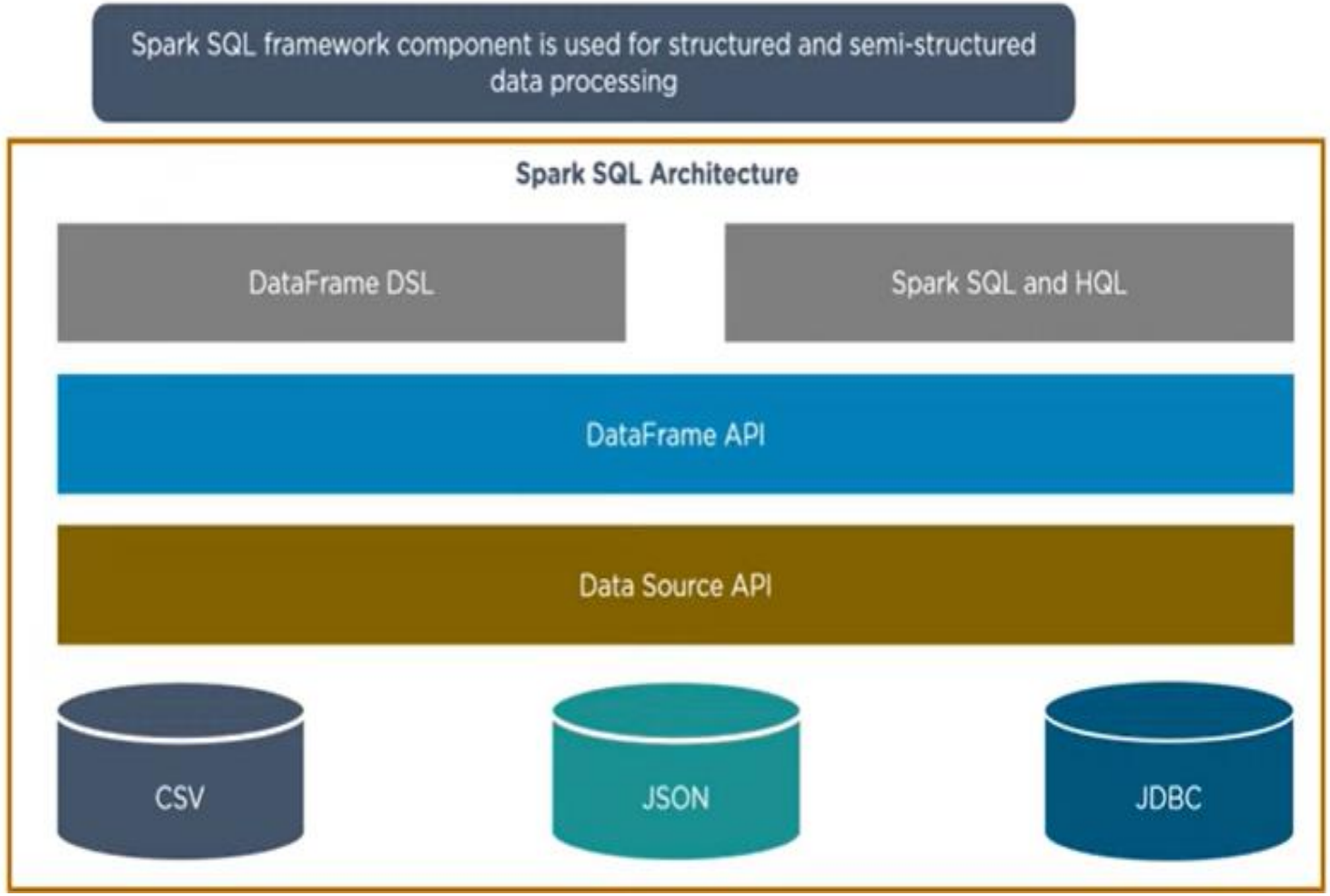
- Data is split up into partitions.
- Partition size depends on data source you are using.
- For HDFS one Block is one Partition.
- Single partition -- > Single Task



Spark SQL – Structured Data

- Spark's package for working with structured data.
- Supports many sources of data including Hive tables, parquet, JSON.
- Allows developers to intermix SQL with programmatic data manipulations supported by RDDs in Python, Scala and Java.
- Shark was older SQL on Spark project.

Spark SQL:



2. Spark SQL

Spark SQL is a new module in Spark which integrates relational processing with Spark's functional programming API. It supports querying data either via SQL or via the Hive Query Language.

Spark SQL integrates relational processing with Spark's functional programming. Further, it provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

The following are the four libraries of Spark SQL.

1.Data Source API
2.DataFrame API
3.Interpreter & Optimizer
4.SQL Service

Spark SQL

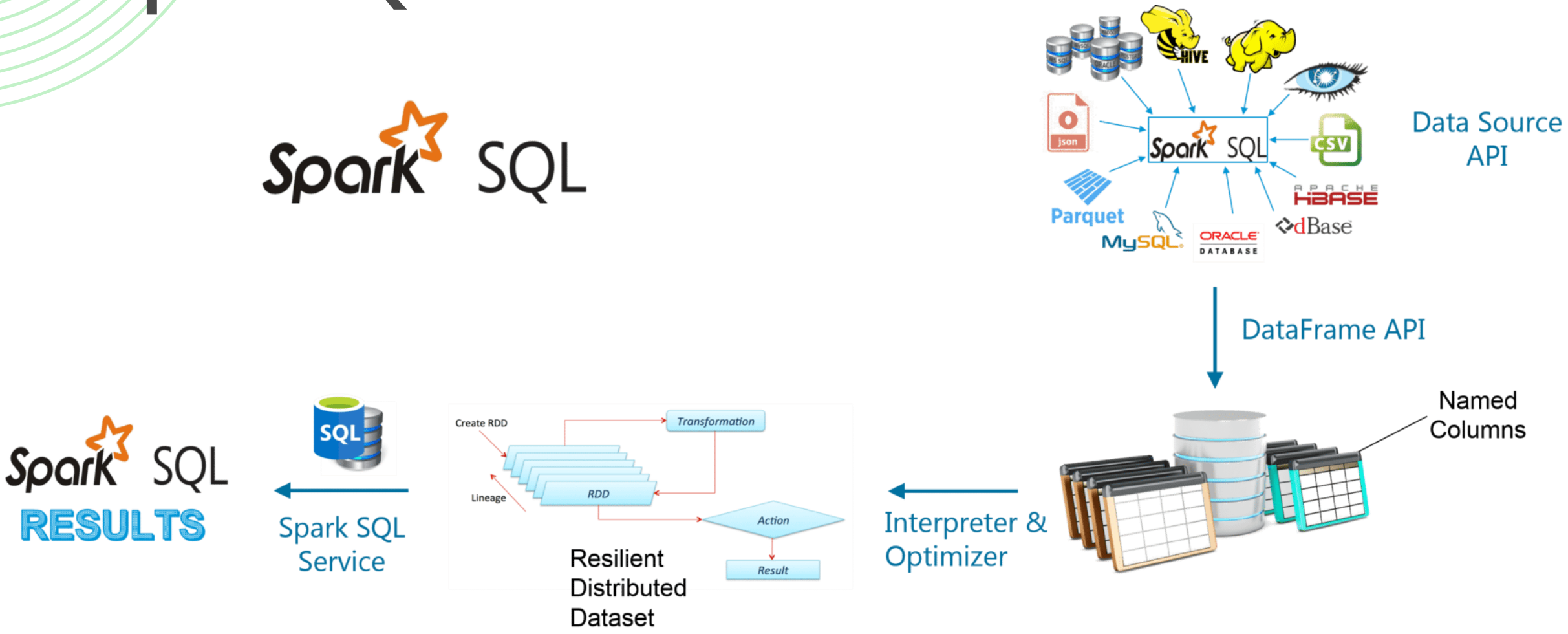
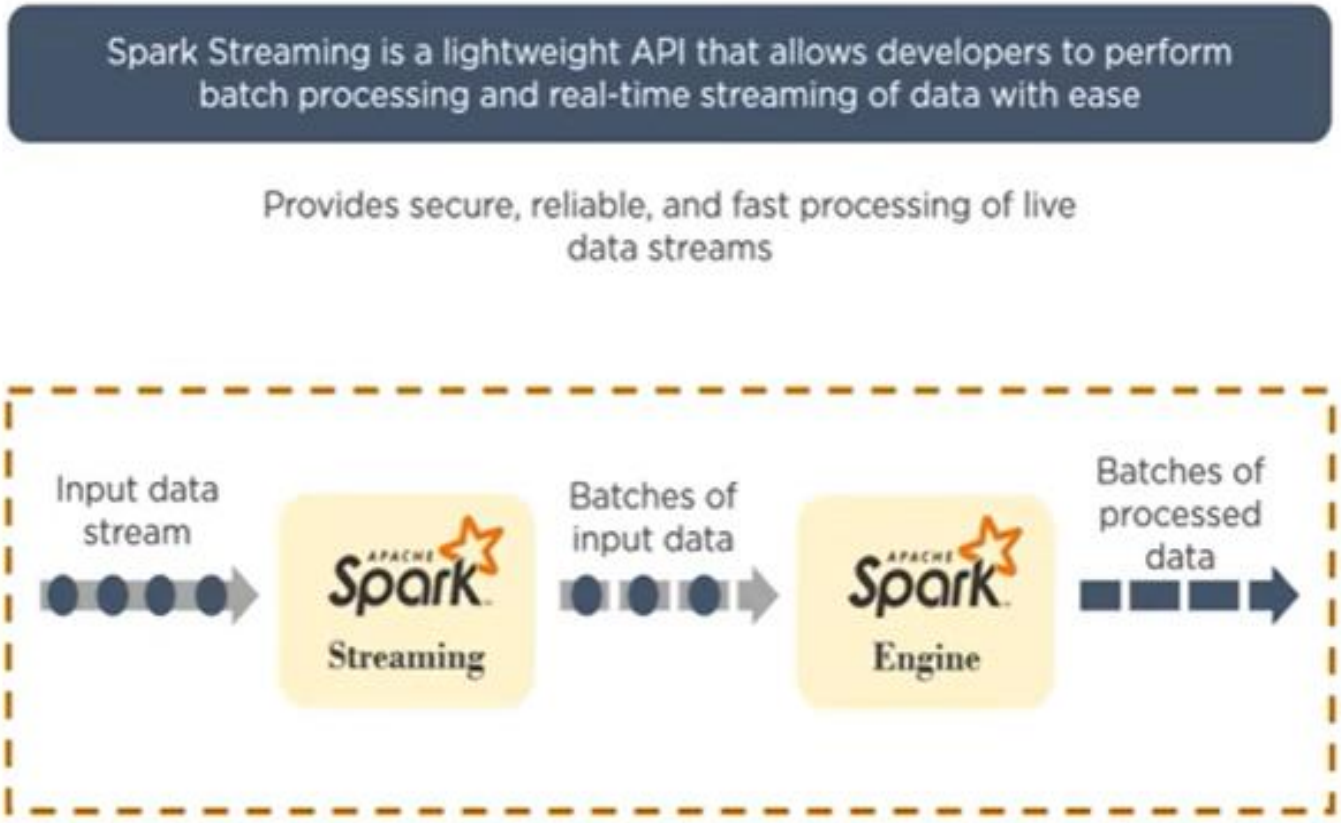


Figure: *The flow diagram represents a Spark SQL process using all the four libraries in sequence*

Spark Streaming:

- Enables processing of live streams of data e.g. log files generated by production web servers.
- APIs are quite similar to spark core's RDD APIs.

Spark Streaming:



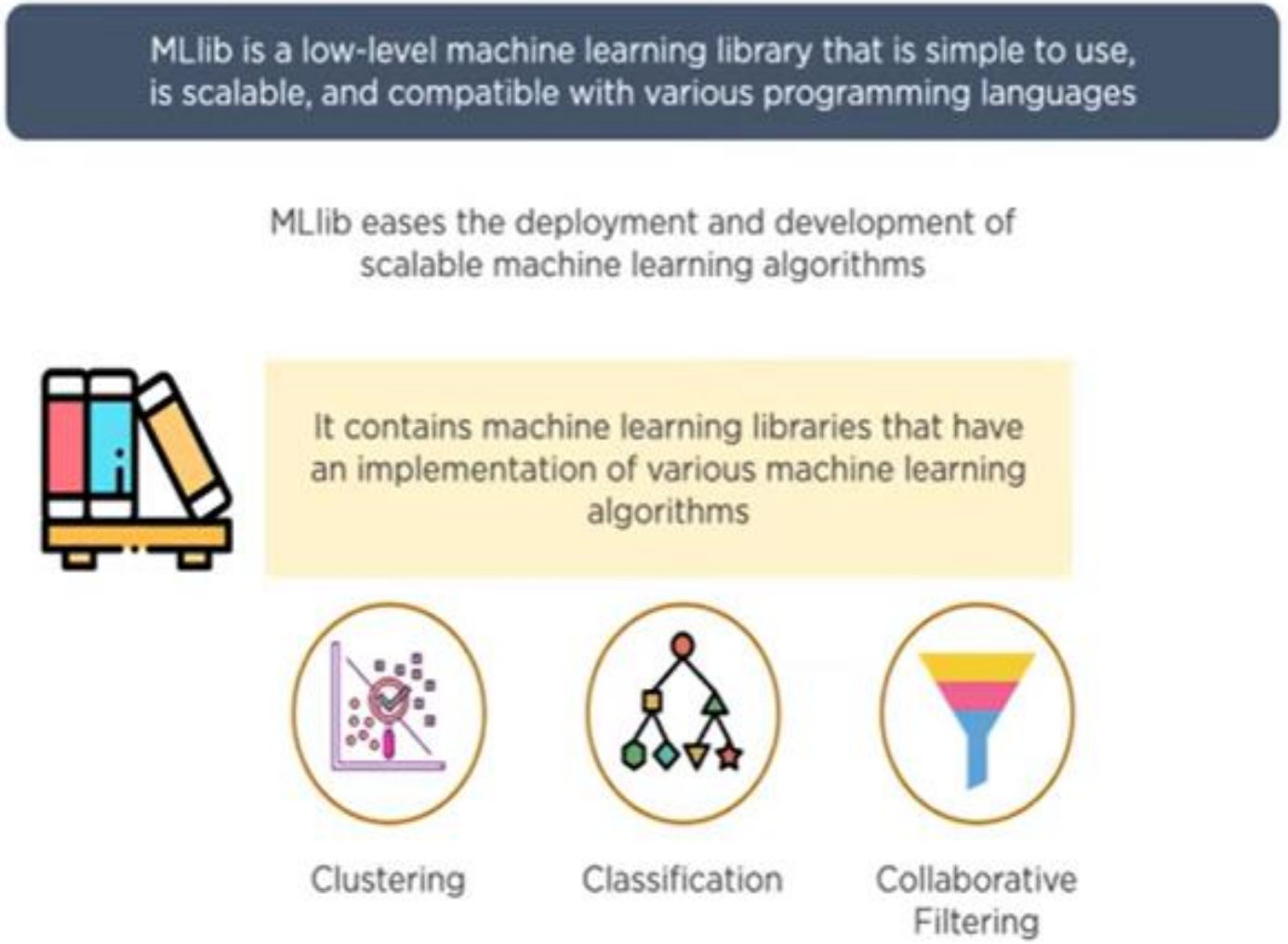
3. Spark Streaming

Spark Streaming is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.

MLLib

- Provides multiple types of machine learning algorithms.

Spark MLlib:

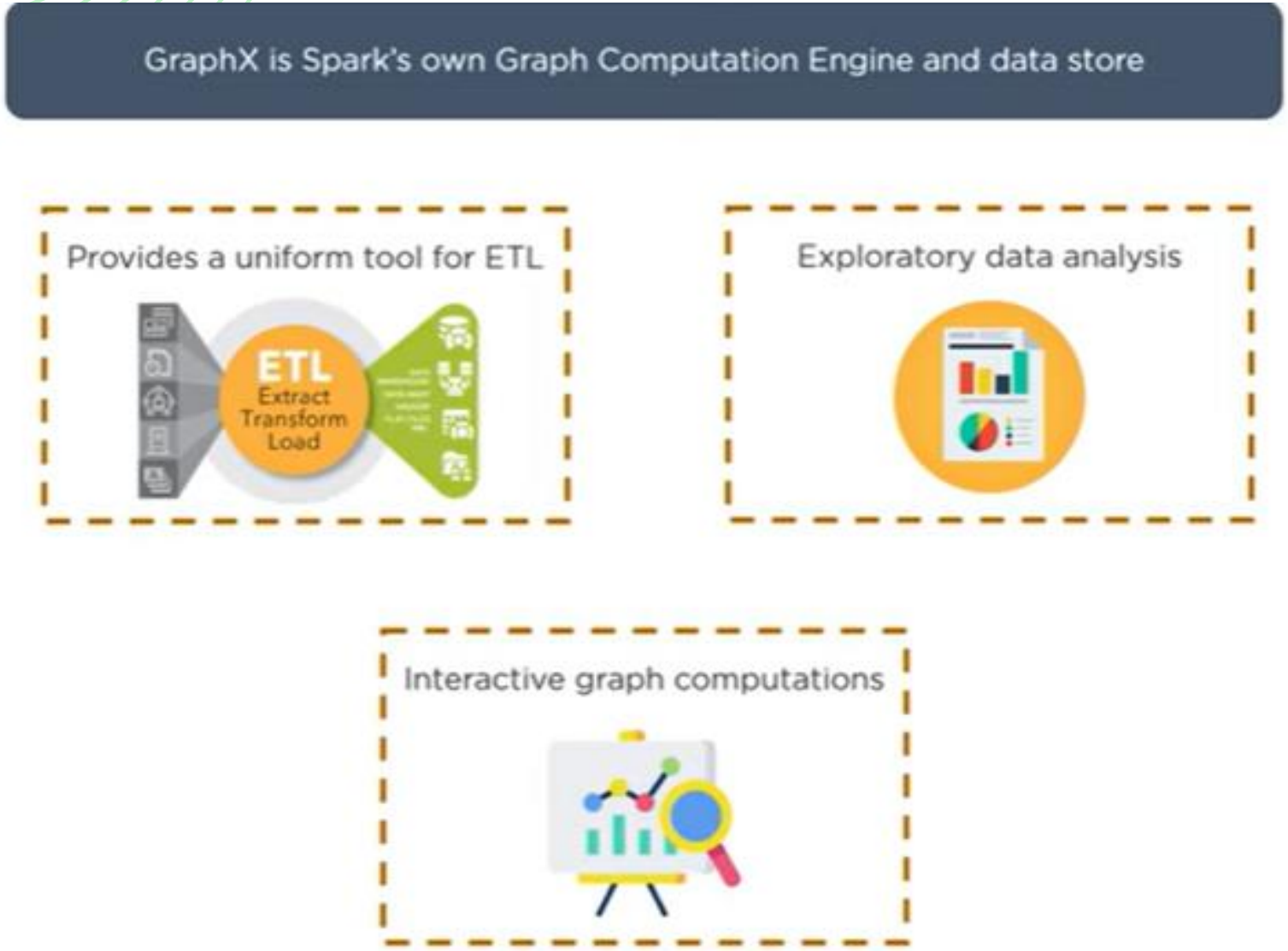


4. MlLib (Machine Learning)
MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

Graphx

- Library for manipulating Graphs.
- Extends Spark RDD API.
- Provides various operators for manipulating Graphs.

Spark GraphX:



5. GraphX

- *GraphX* is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph.
- The property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it.
- Here, the parallel edges allow multiple relationships between the same vertices. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.

Transformations

Function	Description
map()	Returns a new RDD by applying the function on each data element
filter()	Returns a new RDD formed by selecting those elements of the source on which the function returns true
reduceByKey()	Aggregates the values of a key using a function
groupByKey()	Converts a (key, value) pair into a (key, <iterable value>) pair
union()	Returns a new RDD that contains all elements and arguments from the source RDD
intersection()	Returns a new RDD that contains an intersection of the elements in the datasets

Actions

Function	Description
count()	Gets the number of data elements in an RDD
collect()	Gets all the data elements in an RDD as an array
reduce()	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(operation)	Executes the operation for each data element in an RDD
first()	Retrieves the first data element of an RDD

Spark Context

1. **Entry Point**: SparkContext is the entry point for Spark functionality in any Spark application.
2. **Connection to Cluster**: It establishes a connection to the Spark cluster.
3. **Resource Management**: SparkContext manages the resources allocated to the Spark application on the cluster.
4. **RDD Creation**: It is used to create RDDs (Resilient Distributed Datasets), which are the fundamental data structures in Spark.
5. **Driver Program Coordination**: The SparkContext is created by the driver program to coordinate the execution of Spark jobs on the cluster.
6. **Access to Services**: It provides access to various Spark services like broadcast variables, accumulators, and shared variables.
7. **Configuration**: SparkContext allows setting various configurations for Spark application execution.
8. **Lifecycle Management**: It manages the lifecycle of the Spark application, including starting, running, and stopping the application.
9. **Fault Tolerance**: SparkContext ensures fault tolerance by monitoring the execution of tasks and recovering from failures.
10. **Parallel Operations**: SparkContext enables parallel operations on distributed data by leveraging the capabilities of the Spark cluster.

How to Create RDD in Spark?

There are following three processes to create Spark RDD.

1. Using **parallelized collections**

2. From **external datasets** (viz . other external storage systems like a shared file system, HBase or HDFS)

3. From existing **Apache Spark RDDs**

1.Parallelized Collections

create parallelized collections by calling **parallelize method** of **SparkContext interface** on the existing collection of driver program in Java, Scala or Python.

Example:
To hold the numbers 2 to 6 as parallelized collection

```
val collection = Array(2, 3, 4, 5,6)
val prData =
spark.sparkContext.parallelize(collection)
```

How to Create RDD in Spark?

2. External Datasets

Apache Spark can create distributed datasets from any **Hadoop supported file storage** which may include:

- Local file system
- HDFS
- Cassandra
- HBase
- Amazon S3

Spark supports file formats like

- Text files
- Sequence Files
- CSV
- JSON
- Any Hadoop Input Format

How to Create RDD in Spark?

3. From Existing RDDs

RDD is immutable; hence you can't change it. However, using transformation, you can create a new RDD from an existing RDD. As no change takes place due to mutation, it maintains the consistency over the cluster. Few of the operations used for this purpose are:

- map
- filter
- count
- distinct
- flatmap

Practical demo of RDD operations

Create RDD

```
scala> val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
 <console>:27
```

Here, sc denotes SparkContext and each element is copied to form RDD.

Read result

We can read the result generated by RDD by using the collect operation.

```
scala> rdd1.collect
res0: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)

scala>
```

Count

The count action is used to get the total number of elements present in the particular RDD.

```
scala> rdd1.count
res1: Long = 10

scala>
```

Distinct

Distinct is a type of transformation that is used to get the unique elements in the RDD.

```
scala> rdd1.distinct.collect
res3: Array[Int] = Array(82, 86, 78, 27, 23, 45, 67)

scala>
```

Filter

Filter transformation creates a new dataset by selecting the elements according to the given

```
scala> rdd1.filter(x => x < 50).collect
res5: Array[Int] = Array(23, 45, 27, 45)

scala>
```

sortBy

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

```
scala> rdd1.sortBy(x => x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)

scala> rdd1.sortBy(x => x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)

scala>
```

Reduce

Reduce action is used to summarize the RDD based on the given formula.

```
scala> rdd1.reduce((x, y) => x + y)
res8: Int = 606

scala>
```

Map

Map transformation processes each element in the RDD according to the given condition and creates a new RDD.

```
scala> rdd1.map(x => x + 1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)

scala>
```

Union, intersection, and cartesian

Let’s create another RDD.
val rdd2 = sc.parallelize(List(25,73, 97, 78, 27, 82))

- Union operation combines all the elements of the given two RDDs.
- Intersection operation forms a new RDD by taking the common elements in the given RDDs.
- Cartesian operation is used to create a cartesian product of the required RDDs.

```
scala> rdd1.union(rdd2).collect
res12: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86, 25, 73, 97, 78
, 27, 82)

scala> rdd1.intersection(rdd2).collect
res13: Array[Int] = Array(82, 78, 27)

scala> rdd1.cartesian(rdd2).collect
res14: Array[(Int, Int)] = Array((23,25), (23,73), (23,97), (45,25), (45,73), (4
5,97), (67,25), (67,73), (67,97), (86,25), (86,73), (86,97), (78,25), (78,73), (
78,97), (23,78), (23,27), (23,82), (45,78), (45,27), (45,82), (67,78), (67,27),
(67,82), (86,78), (86,27), (86,82), (78,78), (78,27), (78,82), (27,25), (27,73),
 (27,97), (82,25), (82,73), (82,97), (45,25), (45,73), (45,97), (67,25), (67,73)
, (67,97), (86,25), (86,73), (86,97), (27,78), (27,27), (27,82), (82,78), (82,27
), (82,82), (45,78), (45,27), (45,82), (67,78), (67,27), (67,82), (86,78), (86,2
7), (86,82))

scala>
```

First

First is a type of action that always returns the first element of the RDD.

```
scala> rdd1.first()
res15: Int = 23

scala>
```

Take

Take action returns the first n elements in the RDD.

```
scala> rdd1.take(5)
res16: Array[Int] = Array(23, 45, 67, 86, 78)

scala>
```

Pair RDD Operations

- Pair RDDs are a unique class of data structure in Spark that take the form **of key-value pairs**
- most **real-world data** is in the form **of Key/Value pairs**, Pair RDDs are practically employed more frequently.
- The terms "key" and "value" are different by the Pair RDDs. The **value is data**, whereas the **key is an identifier.**

Transformations in Pair RDDs

We must utilize operations that use keys and values since Pair RDDs are built from many tuples.

Following are the widely used Transformation on a Pair RDD:

1. The .reduceByKey() Transformation

For each key in the data, the.reduceByKey() transformation runs multiple parallel operations, **combining the results for the same keys. The task is carried out using a lambda or anonymous function**. Since it is a transformation, the outcome is an RDD.

Pair RDD Operations

- **The .sortByKey() Transformation**

Using the keys from key-value pairs, the.sortByKey() transformation **sorts the input data in ascending or descending order.** It returns a unique RDD as a result.

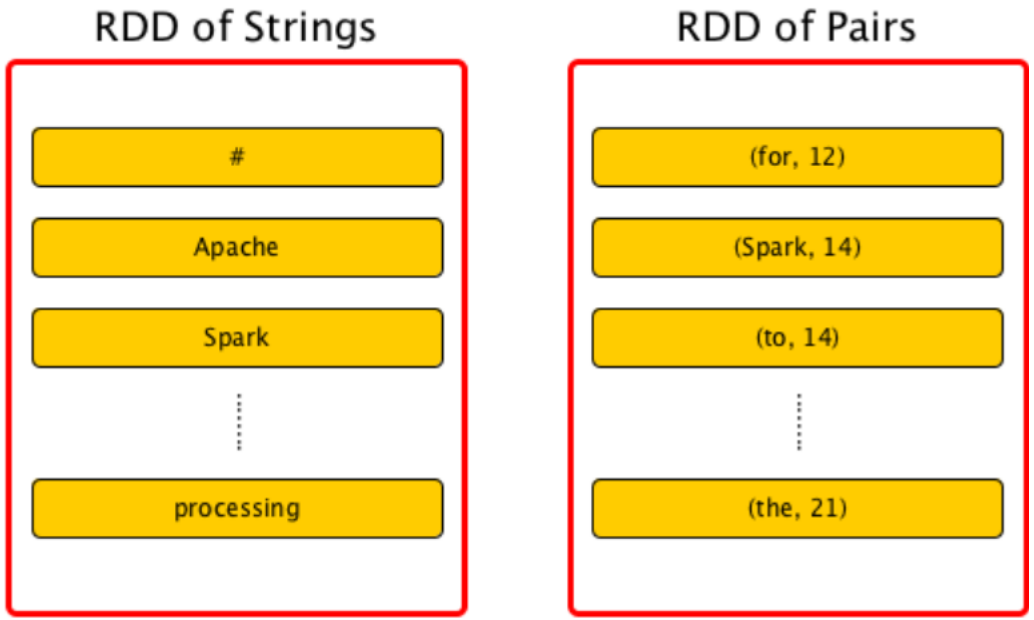
- **The .groupByKey() Transformation**

The.groupByKey() transformation **groups all values in the given data with the same key**. As a result, a **new returned**. For instance, the.groupByKey() function will be useful if we need to extract all the Cultural Members from a list of committee members.

Actions in Pair RDDs

The countBykey() Action

The number of values linked with each key in the provided data is counted using the.countByKey() action. This operation returns a dictionary, which can be iterated using loops to retrieve the keys and values. We can also utilize dictionary methods like.keys(),.values(), and.items because the result is a dictionary ().



DataFrame:

•A Data Frame is used for storing data in tables.

•It is equivalent to a table in a relational database but with richer optimization.

•It is a data abstraction and domain-specific language (DSL) applicable to a structure and semi-structured data.

•It is a distributed collection of data in the form of named column and row.

•It has a matrix-like structure whose column may be different types (numeric, logical, factor, or character).

•We can say the data frame has a two-dimensional array-like structure where each column contains the value of one variable and row contains one set of values for each column.

•It combines features of lists and matrices.

RDD:

•It is the representation of a set of records, an immutable collection of objects with distributed computing.

•RDD is a large collection of data or RDD is an array of references for partitioned objects.

•Each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

•RDDs are fault-tolerant i.e. self-recovered/recomputed in the case of failure.

•The dataset could be data loaded externally by the users which can be in the form of JSON file, CSV file, text file or database via JDBC with no specific data structure.

Differentiation: RDD vs Datasets vs DataFrame

Basis	RDD	Datasets	DataFrame
Inception Year	RDD came into existence in the year 2011.	Datasets entered the market in the year 2013.	DataFrame came into existence in the year 2015.
Meaning	RDD is a collection of data where the data elements are distributed without any schema	Datasets are distributed collections where the data elements are organized into the named columns.	Datasets are basically the extension of DataFrames with added features
Optimization	In case of RDDs, the developers need to manually write the optimization codes.	Datasets use catalyst optimizers for optimization.	Even in the case of DataFrames, catalyst optimizers are used for optimization.
Defining the Schema	In RDDs, the schema needs to be defined manually.	The schema is automatically defined in case of Datasets	The schema is automatically defined in DataFrame

Advantages of RDD

There are multiple advantages of RDD in Spark. We have covered few of the important ones here

•RDD aids in increasing the **execution speed** of Spark.

•RDDs are the **basic unit of parallelism** and hence help in achieving the **consistency of data.**

•RDDs help in **performing and saving the actions** separately

•They are **persistent** as they can be used repeatedly.

Limitation of RDD

- There is **no input optimization** available in RDDs
- One of the biggest limitations of RDDs is that the **execution process does not start instantly**.
- **No changes can be made** in RDD once it is **created**.
- RDD **lacks enough storage memory**.
- The **run-time type safety** is absent in RDDs.

RDD Persistence

- Spark RDD persistence is an **optimization technique** which **saves** the result of RDD evaluation in **cache memory**.
- Spark provides a convenient way to work on the dataset by **persisting it in memory across operations.**

- we can **also reuse** them in other tasks on that dataset.

- We can use either **persist() or cache() method** to mark an RDD **to be persisted**.

- In any case, if the partition of an RDD is lost, it will automatically be **recomputed** using the **transformations that originally created it.**

- There is an **availability of different storage levels** which are used to **store persisted RDDs**.

- Use these levels by passing a **StorageLevel** object (Scala, Java, Python) to persist().

- The **cache() method** is used for the **default storage level**, which is **StorageLevel.MEMORY_ONLY**.

Example

RDD1: Contains a list of numbers [1, 2, 3, 4, 5].
RDD2: Created by doubling each element of RDD1.
RDD3: Created by squaring each element of RDD2.

```
# Create RDD1
rdd1 = sc.parallelize([1, 2, 3, 4, 5])

# Create RDD2 by performing a transformation on RDD1
(doubling each element)
rdd2 = rdd1.map(lambda x: x * 2)

# Create RDD3 by performing a transformation on RDD2
(squaring each element)
rdd3 = rdd2.map(lambda x: x ** 2)
```

Now, let's say we perform an action on RDD3, such as counting the number of elements:

```
# Perform an action on RDD3 (e.g., count the number of elements)

count = rdd3.count()
```

Example

Here's an example of how you can cache RDD3:

```
# Assuming RDD3 is already created
rdd3.persist()

# Perform actions on RDD3
count = rdd3.count()
# Other operations on RDD3...
```

- Intermediate results are stored in memory or disk, leading to **faster and more efficient** computations when accessing RDD3 multiple times.

Note: rdd.cache() is same as rdd.persist()

Need of Persistence in Apache Spark

- In Spark, we can use some RDD's **multiple times**.
- We repeat the same process of **RDD evaluation** each time it required or brought into action.
- This task can be **time and memory consuming**, especially for iterative algorithms that look at data multiple times.
- **To solve** the problem of **repeated computation** the technique of persistence came into the picture.
- There are some **advantages of RDD caching** and persistence mechanism in spark. It makes the whole system

1. Time efficient
2. Cost efficient
3. Lessen the execution time.

The set of storage levels:

Storage Level	Description
MEMORY_ONLY	It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed.
MEMORY_AND_DISK	It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	It stores RDD as serialized Java objects (i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.
MEMORY_AND_DISK_SER (Java and Scala)	It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.
DISK_ONLY	It stores the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	It is the same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.

Persistence Storage Levels

– MEMORY_ONLY (default) – same as cache
rdd.persist(StorageLevel.MEMORY_ONLY) or rdd.persist()

– MEMORY_AND_DISK – Stores partitions on disk which do not fit in memory (This is also called Spilling)
rdd.persist(StorageLevel.MEMORY_AND_DISK)

– DISK_ONLY – Stores all partitions on the disk
rdd.persist(StorageLevel.DISK_ONLY)

-**MEMORY_ONLY_SER and MEMORY_AND_DISK_SER**

- Persisting the **RDD in a serialized** (binary) form helps to **reduce the size of the RDD**, thus making space for more RDD to be persisted in the cache memory. So these two memory formats are **space-efficient**.

- They are **not time-efficient** because we need **to incur the cost of time** involved in **deserializing the data.**

Partition Replication

Stores the partition on two nodes.

DISK_ONLY_2

MEMORY_AND_DISK_2

MEMORY_ONLY_2

MEMORY_AND_DISK_SER_2

MEMORY_ONLY_SER_2

- These options stores a replicated copy of the RDD into some other Worker Node's cache memory as well.
- it helps to recompute the RDD if the other worker node goes down.

Unpersisting the RDD

– To stop persisting and remove from memory or disk
– To change an RDD's persistence level

rdd.unpersist()

Representation of the Storage level

Storage Level	Space used	CPU time	In memory	On-disk	Serialized	Recompute some partitions
MEMORY_ONLY	High	Low	Y	N	N	Y
MEMORY_ONLY_SER	Low	High	Y	N	Y	Y
MEMORY_AND_DISK	High	Medium	Some	Some	Some	N
MEMORY_AND_DISK_SER	Low	High	Some	Some	Y	N
DISK_ONLY	Low	High	N	Y	Y	N

Serialization

- In distributed systems, **data transfer over the network is the most common task**.

- If this is **not handled efficiently**, you may end up facing numerous problems, like **high memory usage, network bottlenecks, and performance issues.**

- **Serialization** refers to **converting objects into a stream of bytes** and vice-versa (de-serialization) in an optimal way to transfer it over nodes of network or store it in a file/memory buffer.

- Spark provides **two serialization libraries** and modes are supported and configured through **spark.serializer** property.

Two serialization libraries

1. Java serialization (default)

The serialization of a class is enabled by the class implementing the java.io.Serializable interface.

Java serialization is slow and leads to large serialized formats for many classes. We can fine-tune the performance by extending java.io.Externalizable.

2. Kryo serialization (recommended by Spark)

- Kryo is a Java serialization framework that focuses on **speed, efficiency, and a user-friendly** API.
- Kryo has **less memory footprint**, which becomes very important when you are shuffling and caching a large amount of data.
  - It is **not natively supported to serialize to the disk**.
  - A class is never serialized **only object of a class is serialized**.

```
from pyspark import SparkConf, SparkContext
from pyspark.serializers import KryoSerializer


conf = SparkConf().setAppName("my_app").setMaster("local")

conf.set("spark.serializer", KryoSerializer.getName())

sc = SparkContext(conf=conf)

# Now we can create and process RDDs using Kryo serialization

# Create RDD1
rdd1 = sc.parallelize([1, 2, 3, 4, 5])
```

Shared Variable in Spark

- Variables those we want to share **throughout our cluster**

- Transformation functions passed on variable, it executes on the specific remote cluster node.
- Usually, it works on separate copies of all the variables those we use in functions.
- These specific variables are precisely copied to each machine.
- on the remote machine, no updates to the variables sent back to the driver program.
- Therefore, it would be inefficient to support general, read-write shared variables across tasks.

Two types of shared variables, such as:

1. Broadcast Variables
2. Accumulators

Broadcast Variables:

- Broadcast variables allow you to efficiently **distribute read-only data to all the worker nodes** in a Spark cluster.
- They are useful when you have large datasets or lookup tables that need to be shared across all nodes but don't change frequently.
- Broadcast variables are **broadcasted to all the worker nodes** once and then cached locally on each node for future reference, which **reduces network overhead**.
- They are **immutable** and can be safely used in parallel operations.

```
# Creating a broadcast variable
broadcast_var = sc.broadcast([1, 2, 3, 4, 5])

# Accessing the broadcast variable value on each worker node
def process_data(element):
    broadcast_value = broadcast_var.value

    processed_data = [x * 2 for x in element if x in broadcast_value]
    return processed_data

# Applying a transformation operation using the broadcast variable
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)
```

Accumulators:

•Accumulators are variables that are only "added" to through an **associative and commutative operation** and are only **"read" by the driver program.**

•They are primarily used for **aggregating information across all the worker nodes** during parallel computations, such as **counting occurrences or summing values**.

•Accumulators are useful for tasks like **collecting statistics or monitoring the progress of a job.**

```
# Creating an accumulator for counting occurrences
accum = sc.accumulator(0)

# Defining a function to increment the accumulator
def process_data(element):
    global accum
    if 3 in element:
        accum += 1
    return element

# Applying a transformation operation using the accumulator
rdd = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
processed_rdd = rdd.map(process_data)

# Performing an action to trigger the accumulation
processed_rdd.count()

# Accessing the value of the accumulator in the driver program
print("Occurrences of 3:", accum.value)
```

DAG: Directed Acyclic Graph

Spark creates a graph when you enter code in spark console.

When an action is called on Spark RDD, Spark submits graph to DAG scheduler.

Operators are divided into stages of task in DAG Scheduler.

The Stages are passed on Task Scheduler, Which launches task through CM.

Importance of DAG in Spark

The DAG plays a critical role in this process by providing **a logical execution plan for the job**.

•The DAG breaks the job down into a **sequence of stages**, where **each stage** represents a **group of tasks** that can be **executed independently** of each other. The **tasks within each stage can be executed in parallel across the machines.**

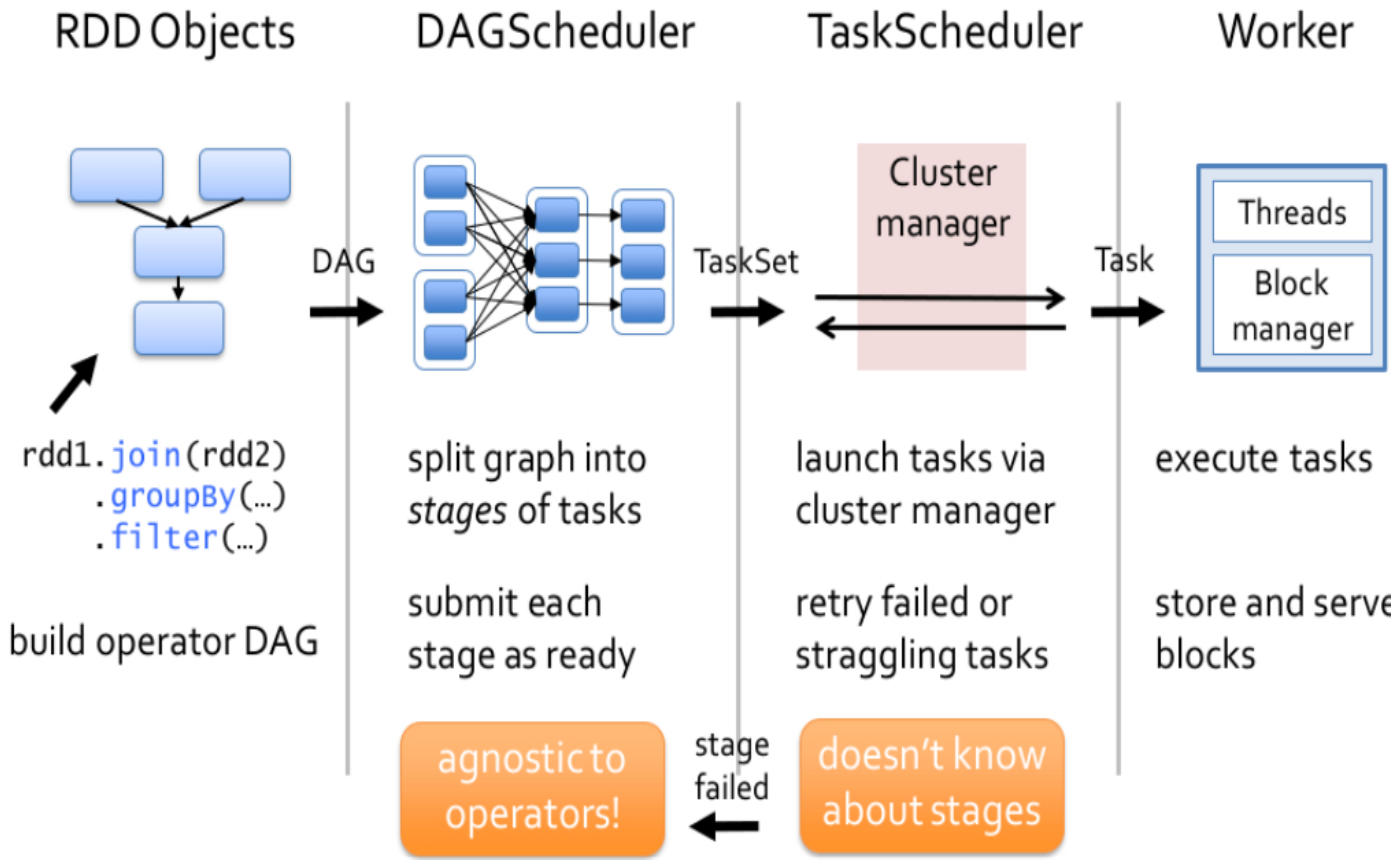
•The DAG allows Spark to perform various optimizations, such as **pipelining, task reordering**, and **pruning unnecessary operations**, to improve the efficiency of the job execution.

•By **breaking down the job into smaller stages** and tasks, Spark can **execute them in parallel** and **distribute them across a cluster of machines for faster processing**.

DAG Scheduler

DAG Scheduler is responsible for **transforming a sequence of RDD transformations and actions** into a directed acyclic graph (DAG) of **stages and tasks**

- Stages will be created.
- Which task need to put in which stage is decided by DAG Scheduler.
- Stage list will be created by DAG scheduler and submitted to Task scheduler.
- Each task will be executed via Cluster manager and cluster manager (CM) will launch it.
- To perform particular task, it needs data and resources.
- Cluster manager will help task scheduler to provide information from where the data need to be fetched in order to execute the tasks list and required resources check will be done by cluster manager.
- Later on task scheduler requests the CM to provide some slot to execute task .
- Executer run on slave machine. TO execute particular task executer has required resources



DAGScheduler transforms a **logical execution plan** (RDD lineage of dependencies built using RDD transformations) to a **physical execution plan** (using stages).

Terminologies in DAG: Stages, Tasks, Dependencies

To work with the DAG Scheduler in Spark, you need to understand the following concepts:

Stages:

- A stage represents a set of tasks that can be executed in parallel.
- There are **two types of stages** in Spark: **shuffle stages** and **non-shuffle stages**.
- **Shuffle** stages involve the **exchange of data between nodes**, while non-shuffle stages do not.

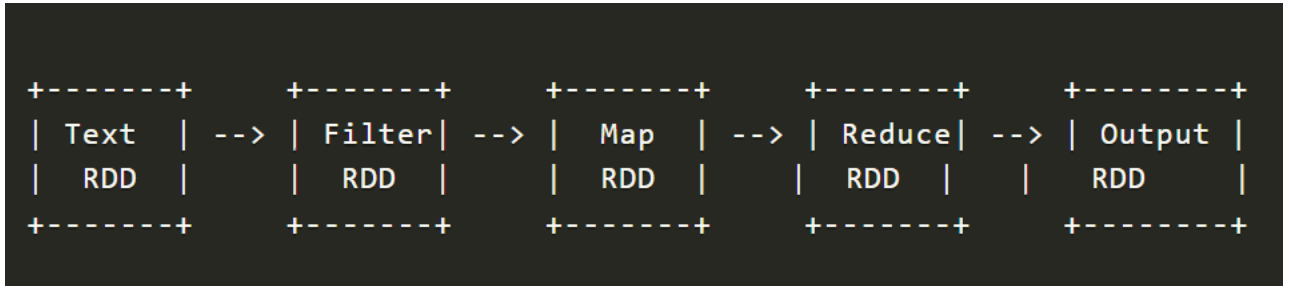
Tasks:

- A task represents a **single unit of work** that can be executed on a **single partition** of an **RDD**.
- Tasks are the smallest units of parallelism in Spark.

Dependencies:

- The dependencies between RDDs determine **the order in which tasks are executed**.
- There are two types of dependencies in Spark: **narrow dependencies and wide dependencies**.
- **Narrow** dependencies indicate that **each partition of the parent RDD** is used by **at most one partition of the child RDD.**
- while wide dependencies indicate **that each partition of the parent RDD** can be used by **multiple partitions of the child RDD.**

Example: DAG diagram



- By visualizing the DAG diagram, developers can better understand the logical execution plan of a Spark job and identify any **potential bottlenecks or performance issues.**

- The DAG diagram consists of five stages: Text RDD, Filter RDD, Map RDD, Reduce RDD, and Output RDD.

- The arrows indicate the dependencies between the stages, and each stage is made up of multiple tasks that can be executed in parallel.
- The Text RDD stage represents the initial loading of the data from a text file, and the subsequent stages involve applying transformations to the data to produce the final output.

- The Filter RDD stage applies a filter transformation to remove any unwanted data.

- The map RDD stage applies a map transformation to transform the remaining data.

- The reduce rdd stage applies a reduce transformation to aggregate the data and

- The output RDD stage writes the final output to a file.

Fault tolerance with the help of DAG

- Spark achieves **fault tolerance** using the DAG by using a technique called **lineage**, which is the **record of the transformations that were used to create an RDD**.

- When a partition of an **RDD is lost** due to a node failure, Spark can **use the lineage to rebuild the lost partition.**

- The lineage is built up as the DAG is constructed, and Spark uses it to recover from any failures during the job execution.

- Spark uses the lineage to recompute the lost partitions.

To achieve fault tolerance, Spark uses **two mechanisms:**

1. RDD Persistence
2. Checkpointing

Advantages of DAG in spark

Efficient execution:

- The DAG allows Spark to break down a large-scale data processing job into smaller, independent tasks that can be executed in parallel.

- By executing the tasks in parallel, Spark can distribute the workload across multiple machines and perform the job much faster than if it was executed sequentially.

Optimization:

- The DAG allows Spark to optimize the job execution by performing various optimizations, such as **pipelining, task reordering, and pruning unnecessary operations**.

- This helps to reduce the overall execution time of the job and improve performance.

Fault tolerance:

- The DAG allows Spark to achieve fault tolerance by using the lineage to recover from node failures during the job execution.

- This ensures that the job can continue running even if a node fails, without losing any data.

Reusability:

- The DAG allows Spark to reuse the intermediate results generated by a job.

- This means that if a portion of the data is processed once, it can be reused in subsequent jobs, thereby reducing the processing time and improving performance.

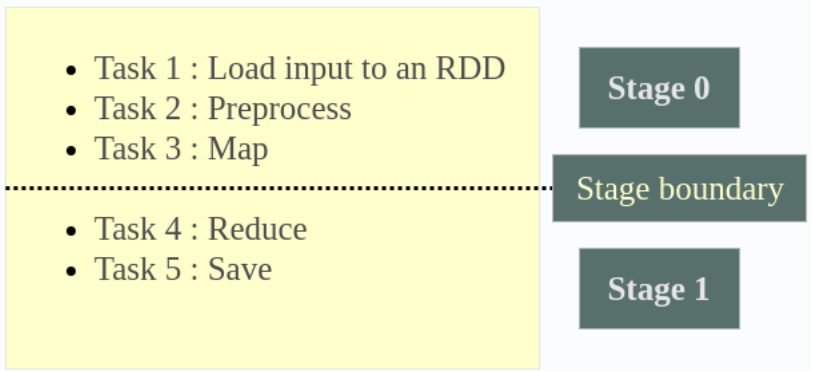
Advantages of DAG in spark

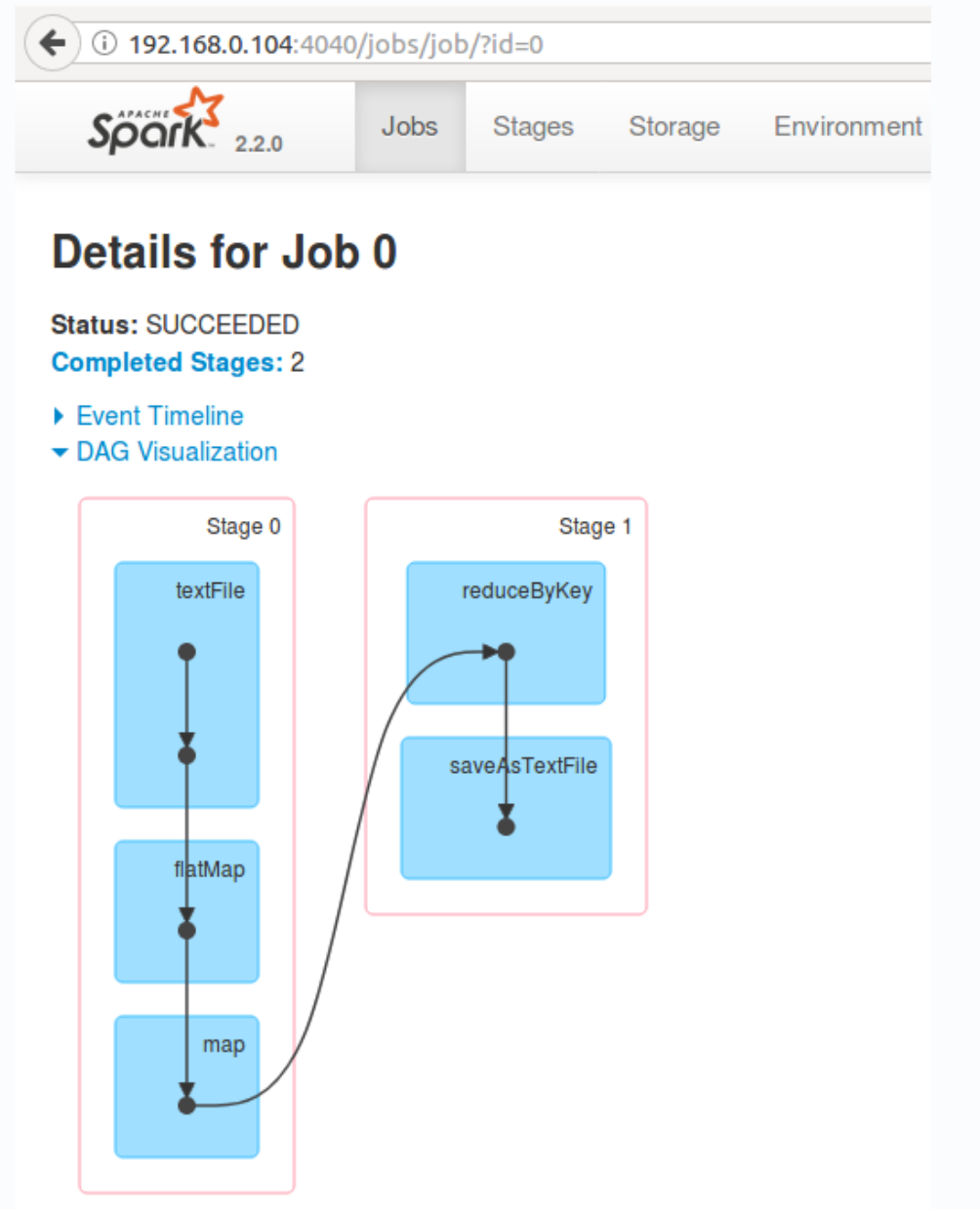
Visualization:

The DAG provides a visual representation of the **logical execution plan** of the job, which can help users to better understand the job and **identify any potential bottlenecks or performance issues.**

This could be visualized in **Spark Web UI**, once you run the WordCount example.

1. Hit the url 192.168.0.104:4040/jobs/
2. Click on the link under Job Description.
3. Expand ‘DAG Visualization’





To work with the DAG Scheduler, you can use the following approaches:

Visualize the DAG:

- You can use the Spark UI to visualize the DAG of a Spark job.
- This allows you to see the different stages and tasks that make up the job and identify any potential bottlenecks or performance issues.

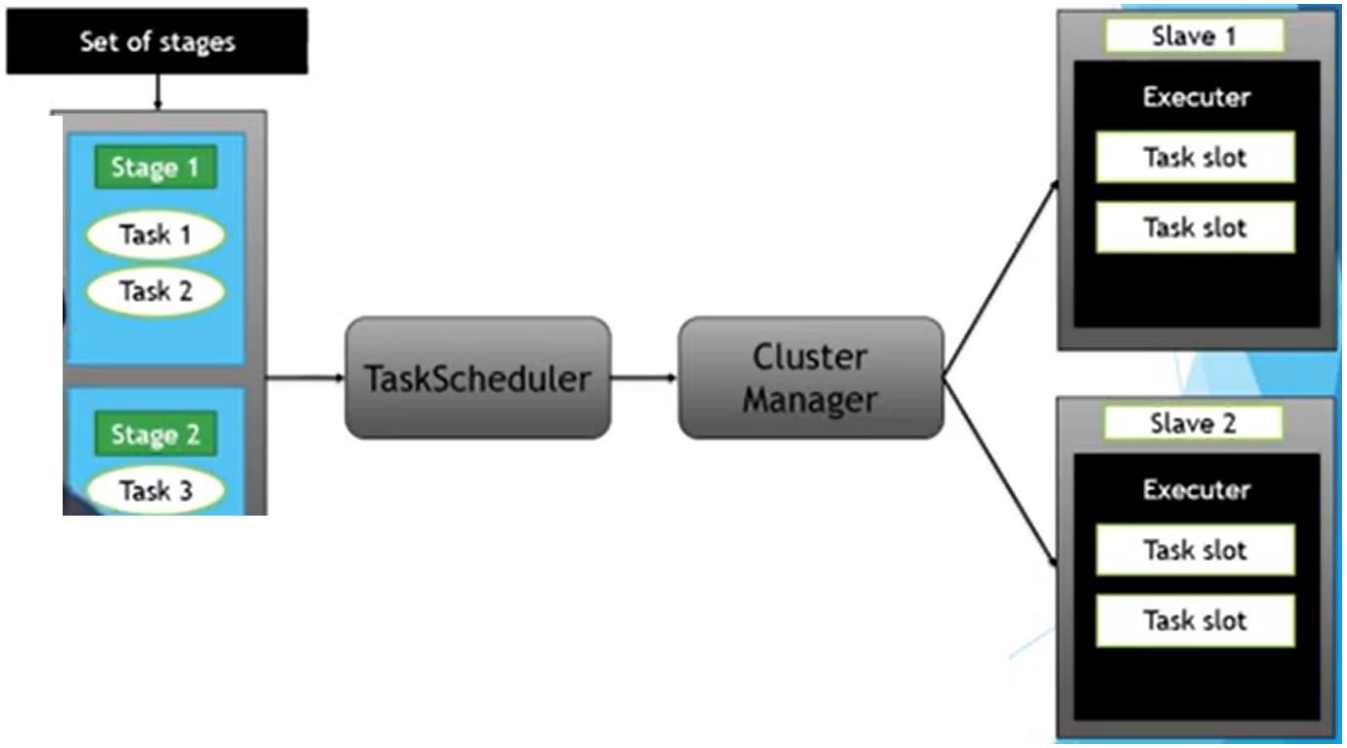
Optimize the DAG:

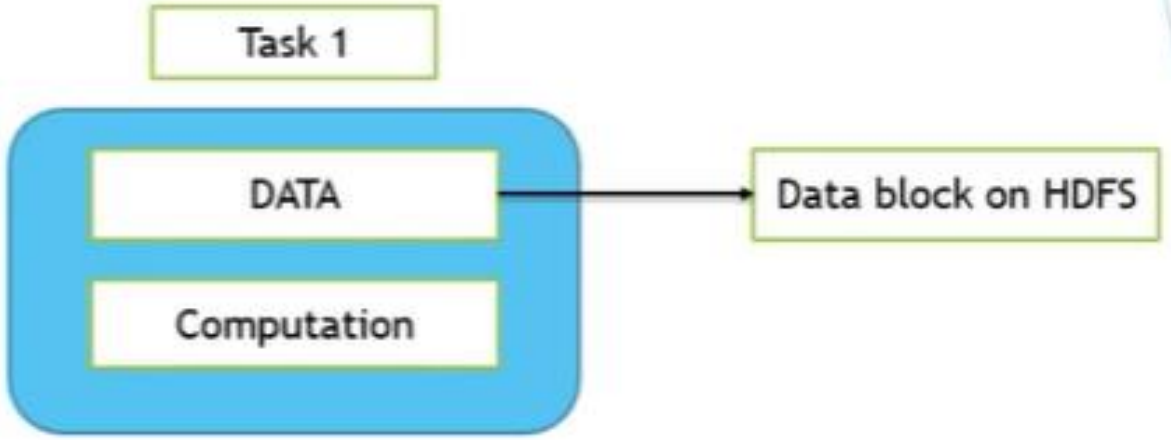
- You can optimize the DAG by using techniques such as pipelining, caching, and reordering of tasks to improve the performance of the job.

Debug issues:

- If you encounter issues with a Spark job, you can use the DAG Scheduler to **identify the root cause of the problem.**
- For example, you can use the Spark UI to identify any slow or failed stages and use this information to troubleshoot the issue.

DAG Scheduler:

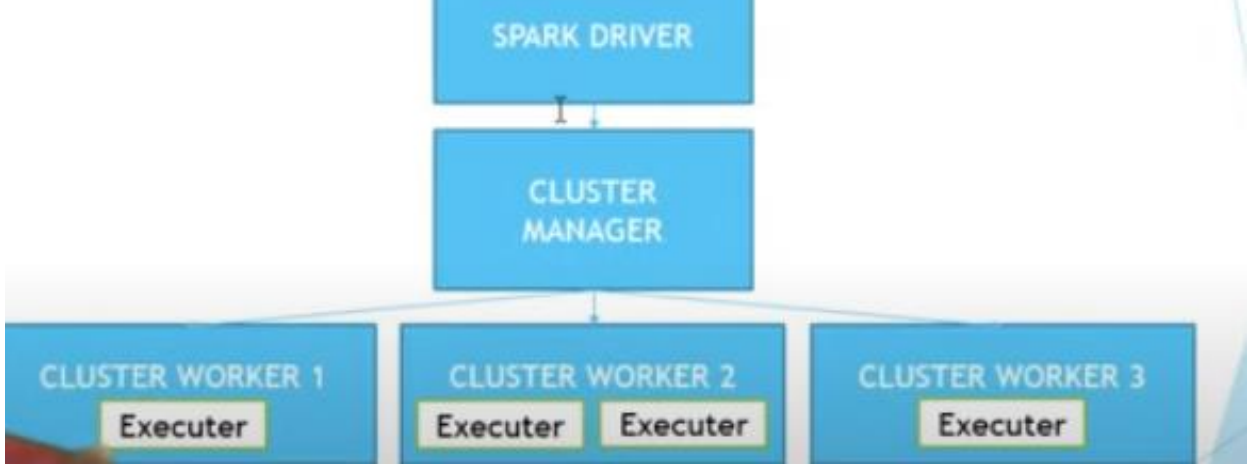




Spark Context (Ticket To Park)

- Establish a connection to spark execution environment.
- It can be used to create RDDs, accumulators and broadcast variables.

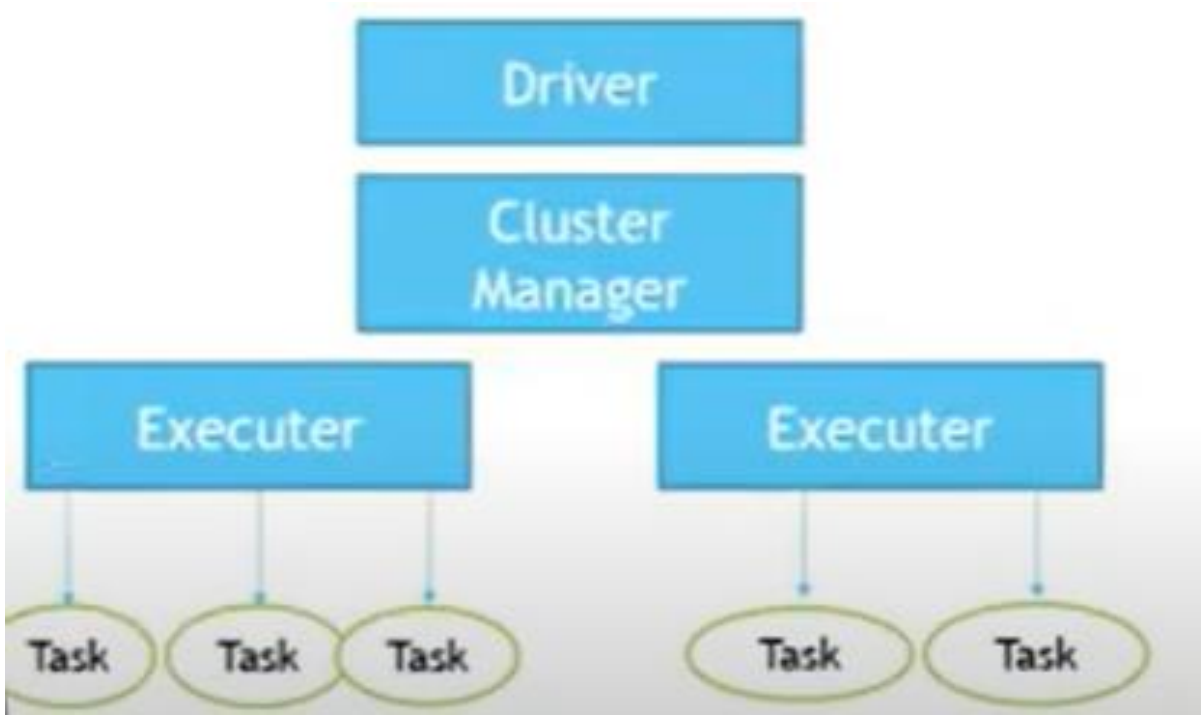
Spark Architecture



- **Spark Driver**
- It works on master slave concept.
- Master = Spark Driver
- Slave = Executer
- Spark Driver
- Runs on top of master system
- When we submit spark program , the main method execution is done by driver- (driver execution will be inside jvm)
- Creation of Context also takes place in driver
- Indirectly Dag scheduler & task scheduler are created at driver end itself.
- User spark program (application) will be converted into actual spark job by "Spark driver"
- **Cluster Manager- (CM)**
- It's a pluggable component
- Use – Yarn , Mesos, or Spark scheduler
- Jobs: Resource Allocate or Deallocate
- Driver – send Tasks – CM- To execute Task – CM checks on which system data is present – CM checks required resources are there or no (CM will check for availability of RAM on slave)- If RAM available – CM will assign the task to Specific Executer Present in worker Machine
- After the specific Task is executed –CM will Deallocate Resources

- **Executers:**
- Are distributed
- Run on slave machines
- Whenever particular task need to be executed it always executed insider executer.
- Executer>=1 can be present inside slave machine
- 1 Executer can have >=1Task
- Jobs:
  - Data Read
  - Data Process
  - Results Writing (memory/disk/cache)

Spark Program Execution:



- User - Spark application code is submitted to spark
- Spark Driver internal components - convert that program (user code) in to DAG (Map – execution graph)
- DAG will be give to DAG Scheduler
- DAG Scheduler – output (stages) – Task scheduler – Task list execution – Negotiate with CM – CM checks availability of resources & RAM - CM assigns Task list to Executer -Inside executer Task will be executed- Data read- data process and write results – Results will be sent from executer to driver .
- After completion of spark application- CM deallocates resources
- Before execution of Task in Executer- Executer has to Register in Driver (Driver will have information of running executers, quantity of executers present)

Working Process :

1. Let’s say a user submits a job using “spark-submit”.
2. “spark-submit” will in-turn launch the Driver which will execute the main() method of our code.
3. Driver contacts the cluster manager and requests for resources to launch the Executors.
4. The cluster manager launches the Executors on behalf of the Driver.
5. Once the Executors are launched, they establish a direct connection with the Driver.
6. The driver determines the total number of Tasks by checking the Lineage.
7. The driver creates the Logical and Physical Plan.
8. Once the Physical Plan is generated, Spark allocates the Tasks to the Executors.
9. Task runs on Executor and each Task upon completion returns the result to the Driver.
10. Finally, when all Task is completed, the main() method running in the Driver exits, i.e. main() method invokes sparkContext.stop().
11. Finally, Spark releases all the resources from the Cluster Manager.

How Apache Spark builds a DAG and Physical Execution Plan ?

1.User **submits a spark application** to the Apache Spark.

2.Driver is the module that takes in the **application from Spark side**.

3.Driver **identifies transformations and actions present** in the spark application. These identifications are the tasks.

4.Based on the flow of program, these **tasks are arranged in a graph like structure** with directed flow of execution from task to task forming **no loops** in the graph (also called DAG). DAG is pure logical.

5.This **logical DAG is converted to Physical Execution Plan**. Physical Execution Plan contains stages.

6.DAG Scheduler creates a **Physical Execution Plan from the logical DAG**. Physical Execution Plan contains tasks and are bundled to be sent to nodes of cluster.

Use Case: Earthquake Detection using Spark

Problem Statement: *To design a Real Time Earthquake Detection Model to send life saving alerts, which should improve its machine learning to provide near real-time computation results.*

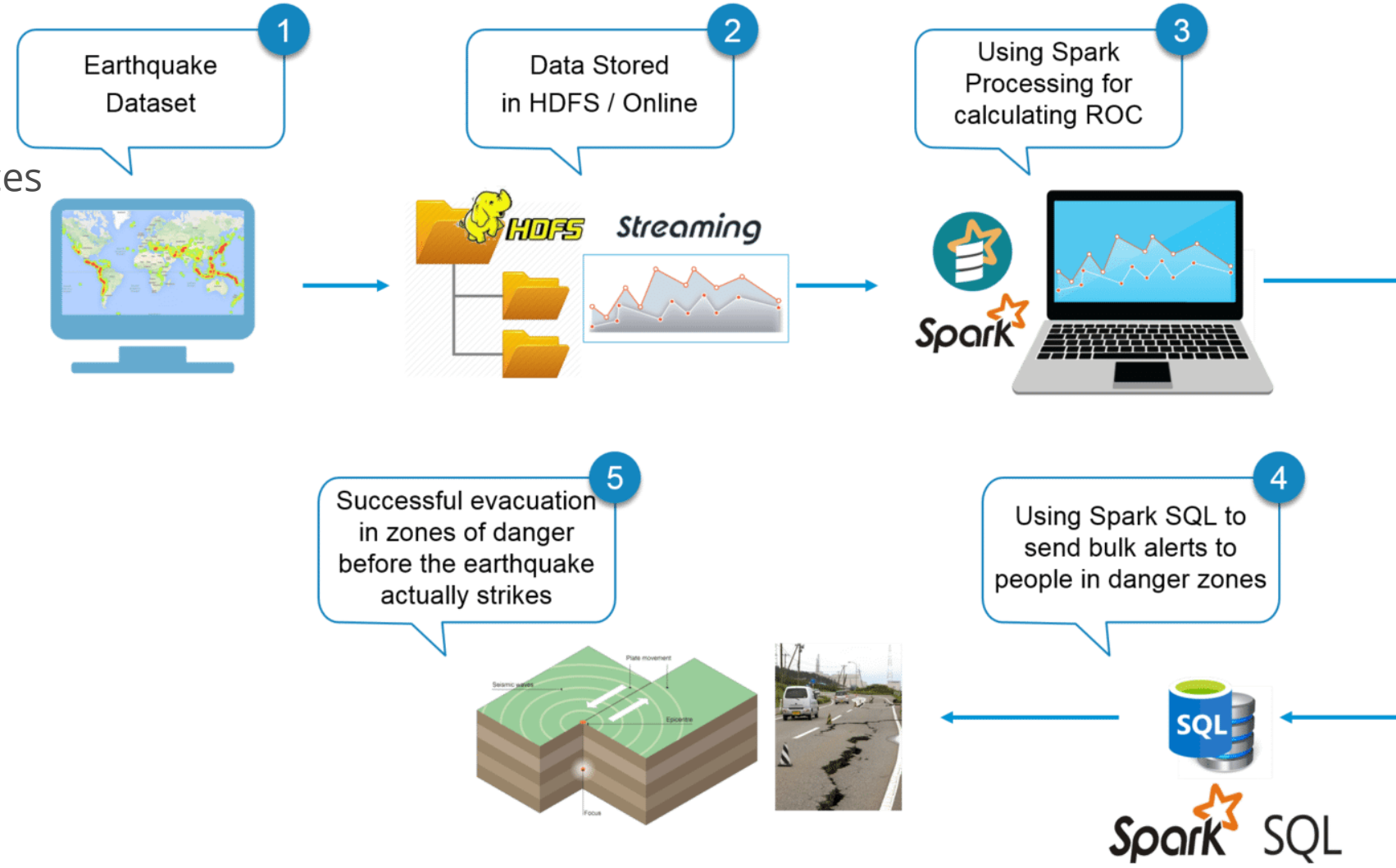
Use Case – Requirements:

1.Process data in real-time

2.Handle input from multiple sources

3.Easy to use system

4.Bulk transmission of alerts



Common Transformations & Actions

```
scala> val rdd1 = sc.parallelize(List(1,2,3,4))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at p

scala> val maprdd1 = rdd1.map(x => x+5)
maprdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map

scala> maprdd1.collect
res0: Array[Int] = Array(6, 7, 8, 9)
```

```
scala> val filterrdd1 = rdd1.filter(x => x!=3)
filterrdd1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at f

scala> filterrdd1.collect
res1: Array[Int] = Array(1, 2, 4)
```

```
scala> val rdd2 = sc.parallelize(List(1,2))
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:24

scala> val maprdd2 = rdd2.map(x=>x.to(3))
maprdd2: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = MapPartitionsRDD[4] at map at <console>:25

scala> maprdd2.collect
res2: Array[scala.collection.immutable.Range.Inclusive] = Array(Range(1, 2, 3), Range(2, 3))
```

```
scala> val flatmaprdd2 = rdd2.flatMap(x=>x.to(3))
flatmaprdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5]

scala> flatmaprdd2.collect
res3: Array[Int] = Array(1, 2, 3, 2, 3)
```

```
scala>  val rdd3 = sc.parallelize(List(1,1,2,3,2,4))
rdd3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at pa

scala> val distinctrdd3 = rdd3.distinct
distinctrdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at

scala> distinctrdd3.collect
res4: Array[Int] = Array(4, 1, 2, 3)
```

```
scala> val rdd4 = sc.parallelize(List(1,2,3,4))
rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at paralleli

scala>

scala>  val rdd5 = sc.parallelize(List(4,5))
rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at paralleli

scala> val unionrdd = rdd4.union(rdd5)
unionrdd: org.apache.spark.rdd.RDD[Int] = UnionRDD[12] at union at <console>

scala> unionrdd.collect
res5: Array[Int] = Array(1, 2, 3, 4, 4, 5)
```

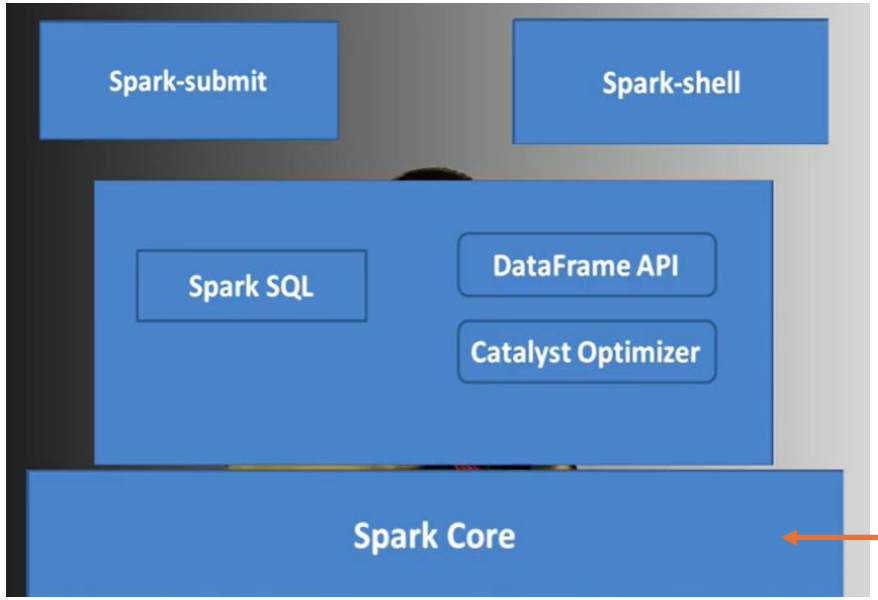
```
scala> subtractrdd.collect
res7: Array[Int] = Array(1, 2, 3)

scala> rdd4.reduce((x, y) => x + y)
res8: Int = 10

scala> rdd4.take(2)
res9: Array[Int] = Array(1, 2)

scala>  rdd4.top(3)
res10: Array[Int] = Array(4, 3, 2)
```

```
scala> val intersectionrdd = rdd4.intersection(rdd5)
intersectionrdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[

scala> intersectionrdd.collect
res6: Array[Int] = Array(4)

scala> val subtractrdd = rdd4.subtract(rdd5)
```

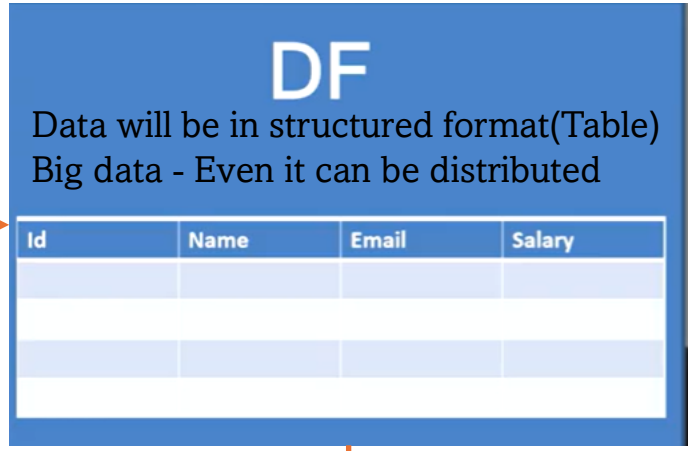
DATAFRAME (DF):

DF = Is RDD in which SQL Table is inserted

- RDD (Limitations)
  - Not Easy as SQL
  - No support of catalyst optimizer (CO – Helps to prioritize the operations = Increase performance, arranges the flow of work)



Execution of DF/RDD



Execution Process

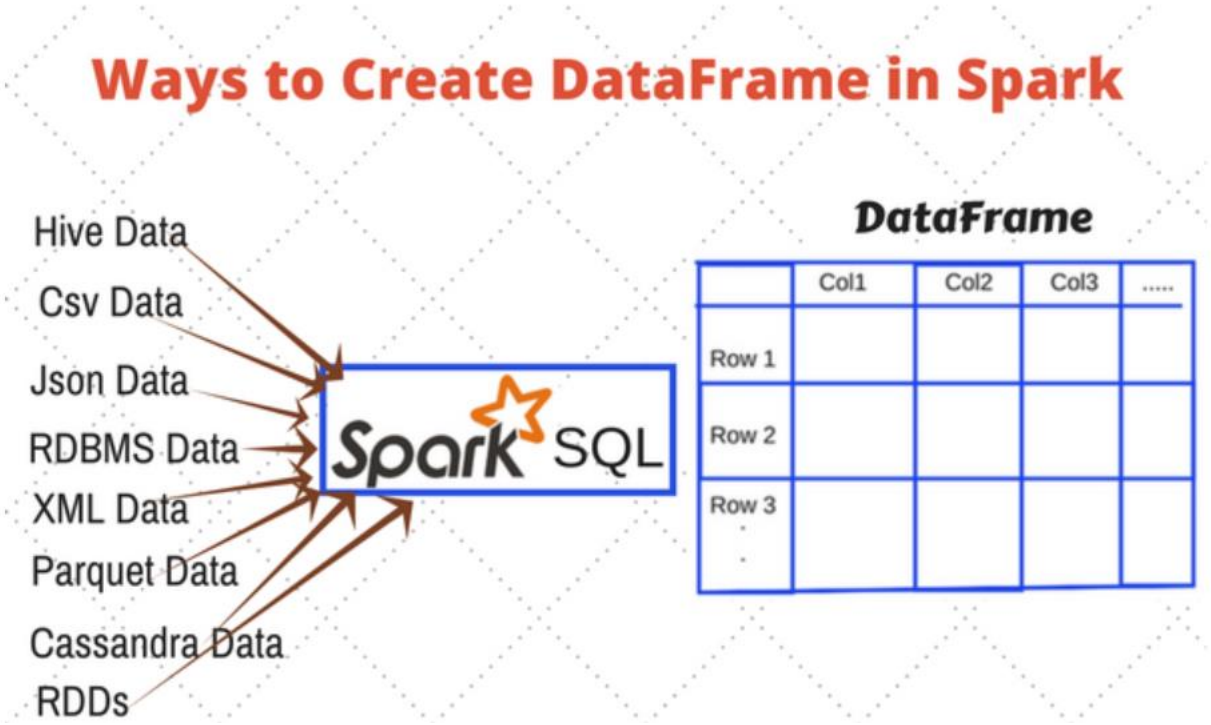
How to Create a DataFrame?

A DataFrame in Apache Spark can be created in multiple ways:

•It can be created using different data formats. For example, loading the data from JSON, CSV.

•Loading data from Existing RDD.

•Programmatically specifying schema



Creating DataFrame from RDD

Steps for creating a DataFrame from list of tuples:

- Create a list of tuples. Each tuple contains name of a person with age.
- Create a RDD from the list above.
- Convert each tuple to a row.
- Create a DataFrame by applying **createDataFrame** on RDD with the help of **sqlContext**.

```
from pyspark.sql import Row
l = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(l)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
schemaPeople = sqlContext.createDataFrame(people)
```

RDD To DF

```
hduser@ubuntu:~$ spark-shell --conf spark.sql.catalogImplementation=in-memory */
20/05/28 13:28:31 WARN Utils: Your hostname, ubuntu resolves to a loopback address:
```

```
scala> val pehlaRdd = sc.parallelize(1 to 10).map(x => (x,"df ka data"))
pehlaRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:24

scala> pehlaRdd.collect
20/05/28 13:31:11 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes
res0: Array[(Int, String)] = Array((1,df ka data), (2,df ka data), (3,df ka data), (4,df ka data), (5,df
ka data), (6,df ka data), (7,df ka data), (8,df ka data), (9,df ka data), (10,df ka data))
```

```
scala> val pehlaDf = pehlaRdd.toDF("id","simple_string")
pehlaDf: org.apache.spark.sql.DataFrame = [id: int, simple_string: string]

scala> pehlaRdd.collect.foreach(println)
(1,df ka data)
(2,df ka data)
(3,df ka data)
(4,df ka data)
(5,df ka data)
(6,df ka data)
(7,df ka data)
(8,df ka data)
(9,df ka data)
(10,df ka data)
```

- We didn't provide any schema
- Spark detected the data types by itself – Because of RDD.
- Assume if you were suppose to read data file from Hadoop (HDFS) or from local system – define RDD - then we need to define schema.
- AS we cannot use parallelize method to create RDD always.

```
scala> pehlaDf.show
+---+-------------+
| id|simple_string|
+---+-------------+
|  1|    df ka data|
|  2|    df ka data|
|  3|    df ka data|
|  4|    df ka data|
|  5|    df ka data|
|  6|    df ka data|
|  7|    df ka data|
|  8|    df ka data|
|  9|    df ka data|
| 10|    df ka data|
+---+-------------+
```

Creating DF – Providing Schema

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types
```

```
scala> vidyarthiDf.printSchema
root
 |-- rollNo: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- marks: integer (nullable = true)
```

```
scala> val vidyarthiRdd = sc.parallelize(Array(Row(1,"sdp",25),(Row(2,"xyz",31))))
vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = ParallelCollectionRD
e at <console>:28

scala> vidyarthiRdd
   val vidyarthiRdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]

scala> vidyarthiRdd.collect
20/05/28 15:39:27 WARN SizeEstimator: Failed to check whether UseCompressedOops is set;
res0: Array[org.apache.spark.sql.Row] = Array([1,sdp,25], [2,xyz,31])
```

```
scala> vidyarthi
vidyarthiDf   vidyarthiRdd

scala> vidyarthiDf.show
+------+----+-----+
|rollNo|name|marks|
+------+----+-----+
|     1| sdp|   25|
|     2| xyz|   31|
+------+----+-----+
```

```
scala> val schema009 = StructType(
     |                         Array(
     |                               StructField("rollNo",IntegerType,true),
     |                               StructField("name",StringType,true),
     |                               StructField("marks",IntegerType,true)
     |                              )
     |                        )
schema009: org.apache.spark.sql.types.StructType = StructType(StructField(rollNo,IntegerType,true)
tField(name,StringType,true), StructField(marks,IntegerType,true))

scala> val vidyarthiDf = spark.createDataFrame(vidyarthiRdd,schema009);
vidyarthiDf: org.apache.spark.sql.DataFrame = [rollNo: int, name: string ... 1 more field]
```

Creating DF using CSV & Parquet files

*spark_df_video

```
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007
```

- Transfer the file to spark (read from HDFS / Local system)
- Create DF & Define Schema
  - Create Class
  - Create RDD
  - Create DF

```
scala> case class Movies(rank:Int,movie:String,actor:String,release_year:Int)
defined class Movies

scala> val FavouriteMoviesRdd = sc.textFile("/home/hduser/Desktop/data/fav_movies")
FavouriteMoviesRdd: org.apache.spark.rdd.RDD[String] = /home/hduser/Desktop/data/fav_mov
RDD[7] at textFile at <console>:28

scala> FavouriteMoviesRdd.collect
collect   collectAsync

scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007
```

Class

RDD (Read file from local)

```
scala> FavouriteMoviesRdd.collect.foreach(println)
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2012
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007

scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
```

x(0) = 1

x(1) = logan

x(2) = hugh jackman

x(3) = 2017

Create DF

```
case class Movies(rank:Int,movie:String,
    actor:String,release_year:Int)
```

```
scala> val FavouriteMoviesDf = FavouriteMoviesRdd.map(x => x.split(',')).map(x => Movies(x(0).toInt,x(1),
x(2),x(3).toInt)).toDF
FavouriteMoviesDf: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> FavouriteMovies
FavouriteMoviesDf   FavouriteMoviesRdd

scala> FavouriteMoviesDf.printSchema
root
 |-- rank: integer (nullable = false)
 |-- movie: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- release_year: integer (nullable = false)
```

```
scala> FavouriteMovies
FavouriteMoviesDf   FavouriteMoviesRdd

scala> FavouriteMoviesDf.printSchema
root
 |-- rank: integer (nullable = false)
 |-- movie: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- release_year: integer (nullable = false)


scala> FavouriteMovies
FavouriteMoviesDf   FavouriteMoviesRdd

scala> FavouriteMoviesDf.show
+----+--------------------+------------------+------------+
|rank|               movie|             actor|release_year|
+----+--------------------+------------------+------------+
|   1|               logan|      hugh jackman|        2017|
|   2|     The Dark Knight|      Heath ledger|        2008|
|   3|            Wolverine|      hugh jackman|        2013|
|   4|            Deadpool|     ryan reynolds|        2016|
|   5|               Rocky|Sylvester Stallone|        1976|
|   6|         go goa gone|              saif|        2013|
|   7|               Joker|   joaquin phoenix|        2019|
|   8|            Avengers|               rdj|        2012|
|   9|     sherlock Holmes|               rdj|        2009|
|  10|pirates of the ca...|       johnny depp|        2003|
|  11|     The Expendables|sylvester stallone|        2010|
|  12|             dhamaal|      javed jafree|        2007|
+----+--------------------+------------------+------------+
```

Direct Load CSV file – No need to define Schema- No class creation

*spark_df_video

```
rank,movie,actor,release_year
1,logan,hugh jackman,2017
2,The Dark Knight,Heath ledger,2008
3,Wolverine,hugh jackman,2013
4,Deadpool,ryan reynolds,2016
5,Rocky,Sylvester Stallone,1976
6,go goa gone,saif,2013
7,Joker,joaquin phoenix,2019
8,Avengers,rdj,2008
9,sherlock Holmes,rdj,2009
10,pirates of the caribbean,johnny depp,2003
11,The Expendables,sylvester stallone,2010
12,dhamaal,javed jafree,2007
```

Define Column name in File itself
Detect type (schema by spark)

```
scala> val moviesDf1 = spark.read.option("header","true").option("inferSchema","true").csv("/home/hduser/
Desktop/data/fav_movies_with_header")
moviesDf1: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]

scala> moviesDf1.printSchema
root
 |-- rank: integer (nullable = true)
 |-- movie: string (nullable = true)
 |-- actor: string (nullable = true)
 |-- release_year: integer (nullable = true)


scala> moviesDf1.show
+----+--------------------+-----------------+------------+
|rank|               movie|            actor|release_year|
+----+--------------------+-----------------+------------+
|   1|               logan|      hugh jackman|        2017|
|   2|     The Dark Knight|      Heath ledger|        2008|
|   3|           Wolverine|      hugh jackman|        2013|
|   4|            Deadpool|     ryan reynolds|        2016|
|   5|               Rocky|Sylvester Stallone|        1976|
|   6|        go goa gone|             saif|        2013|
|   7|              Joker|   joaquin phoenix|        2019|
|   8|           Avengers|              rdj|        2008|
|   9|     sherlock Holmes|              rdj|        2009|
|  10|pirates of the ca...|       johnny depp|        2003|
|  11|     The Expendables|sylvester stallone|        2010|
|  12|            dhamaal|      javed jafree|        2007|
+----+--------------------+-----------------+------------+
```

Parquet File- No column Names, Schema

- Parquet File – Inside it will have information of “Column names, data types, Schema”- No need to define them.
- But file need to be with parquet extension (read from local/hdfs system)

```
scala>  val moviesDf2 = spark.read.load("/home/hduser/Desktop/data/movies/movies.parquet")
moviesDf2: org.apache.spark.sql.DataFrame = [rank: int, movie: string ... 2 more fields]
```

```
scala> moviesDf2.show
+----+--------------------+-----------------+------------+
|rank|               movie|            actor|release_year|
+----+--------------------+-----------------+------------+
|   1|               logan|      hugh jackman|        2017|
|   2|     The Dark Knight|      Heath ledger|        2008|
|   3|           Wolverine|      hugh jackman|        2013|
|   4|            Deadpool|     ryan reynolds|        2016|
|   5|               Rocky|Sylvester Stallone|        1976|
|   6|         go goa gone|             saif|        2013|
|   7|               Joker|   joaquin phoenix|        2019|
|   8|            Avengers|              rdj|        2008|
|   9|     sherlock Holmes|              rdj|        2009|
|  10|pirates of the ca...|       johnny depp|        2003|
|  11|     The Expendables|sylvester stallone|        2010|
|  12|            dhamaal|      javed jafree|        2007|
+----+--------------------+-----------------+------------+
```

Word Count Program using Spark

•Create a text file in your local machine and write some text into it.

$ nano sparkdata.txt

Create a directory in HDFS, where to kept text file.

$ hdfs dfs -mkdir /spark

Upload the sparkdata.txt file on HDFS in the specific directory.

$ hdfs dfs -put /home/codegyani/sparkdata.txt /spark

Now, follow the below command to open the spark in Scala mode.

$ spark-shell

•Let's create an RDD by using the following command.

scala> val data=sc.textFile("sparkdata.txt")

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

scala> data.collect;

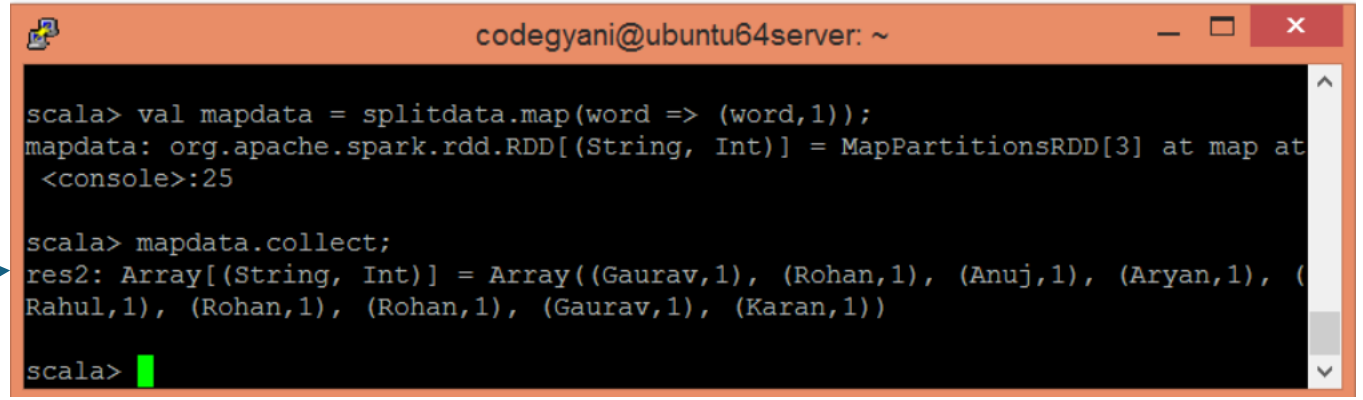
•Here, we split the existing data in the form of individual words by using the following command.

scala> val splitdata = data.flatMap(line => line.split(" "));

•Now, we can read the generated result by using the following command.
scala> splitdata.collect;

•Now, perform the map operation.
scala> val mapdata = splitdata.map(word => (word,1));

scala> mapdata.collect;



•Now, perform the reduce operation

scala> val reducedata = mapdata.reduceByKey(_+_);

scala> reducedata.collect;

Now, on to the WordCount script. For local testing, we will use a file from our file system.

val text = sc.textFile("mytextfile.txt")
val counts = text.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_+_) counts.collect

The next step is to run the script.

spark-shell -i WordCountscala.scala


Thank you