Apache Spark

Prepared By : Prof. Shital Pathar

Big Data Analytics

Big data analytics describes the process of uncovering trends, patterns, and correlations in large amounts of raw data to help make data-informed decisions.

Types of Big Data Analytics

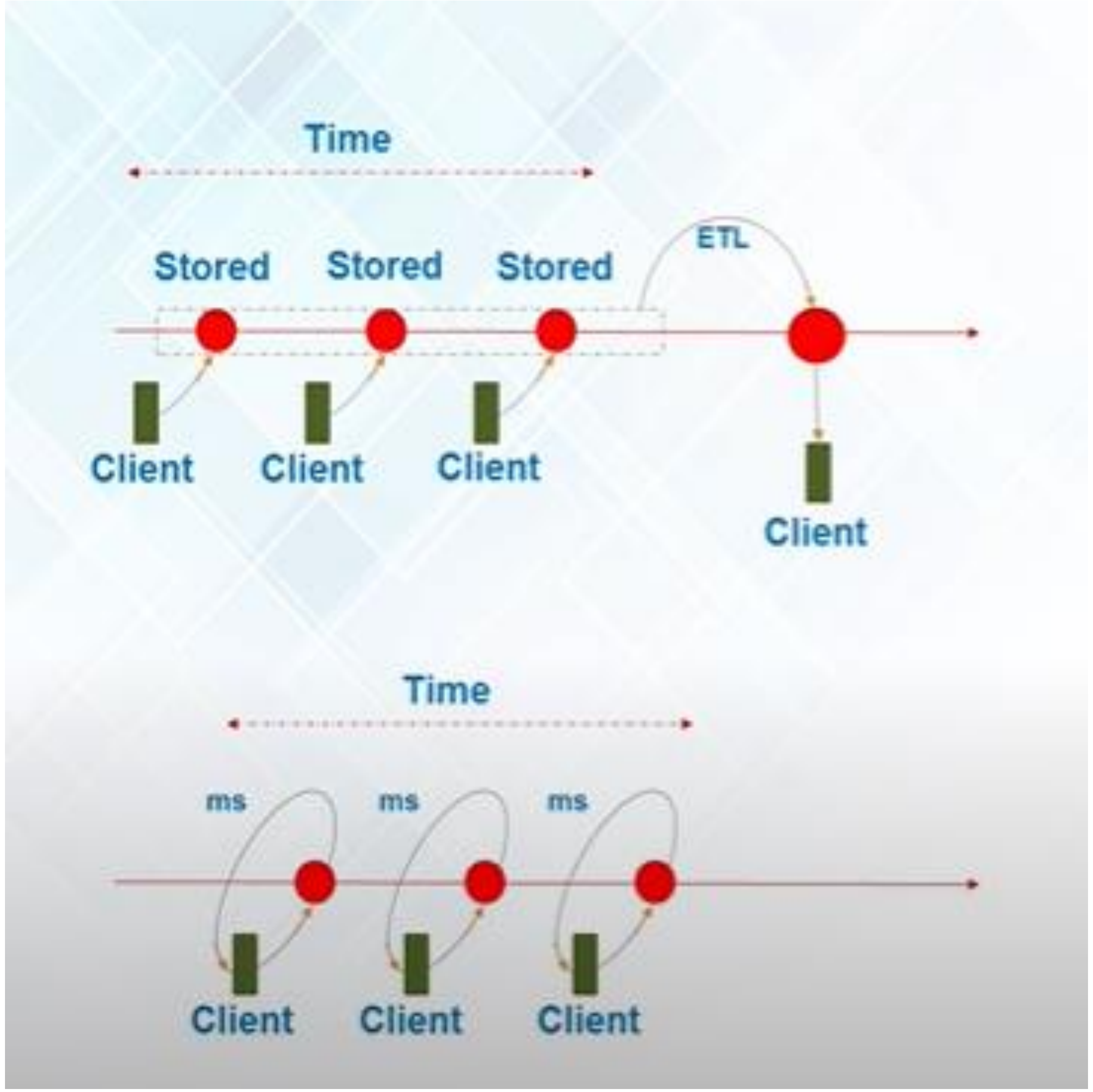
1.Batch Analytics

2.Real Time Analytics

Batch vs Real Time Analytics

- Analytics based on data collected over a period of time is Batch Analytics. e.g. Washing machine (historical data)

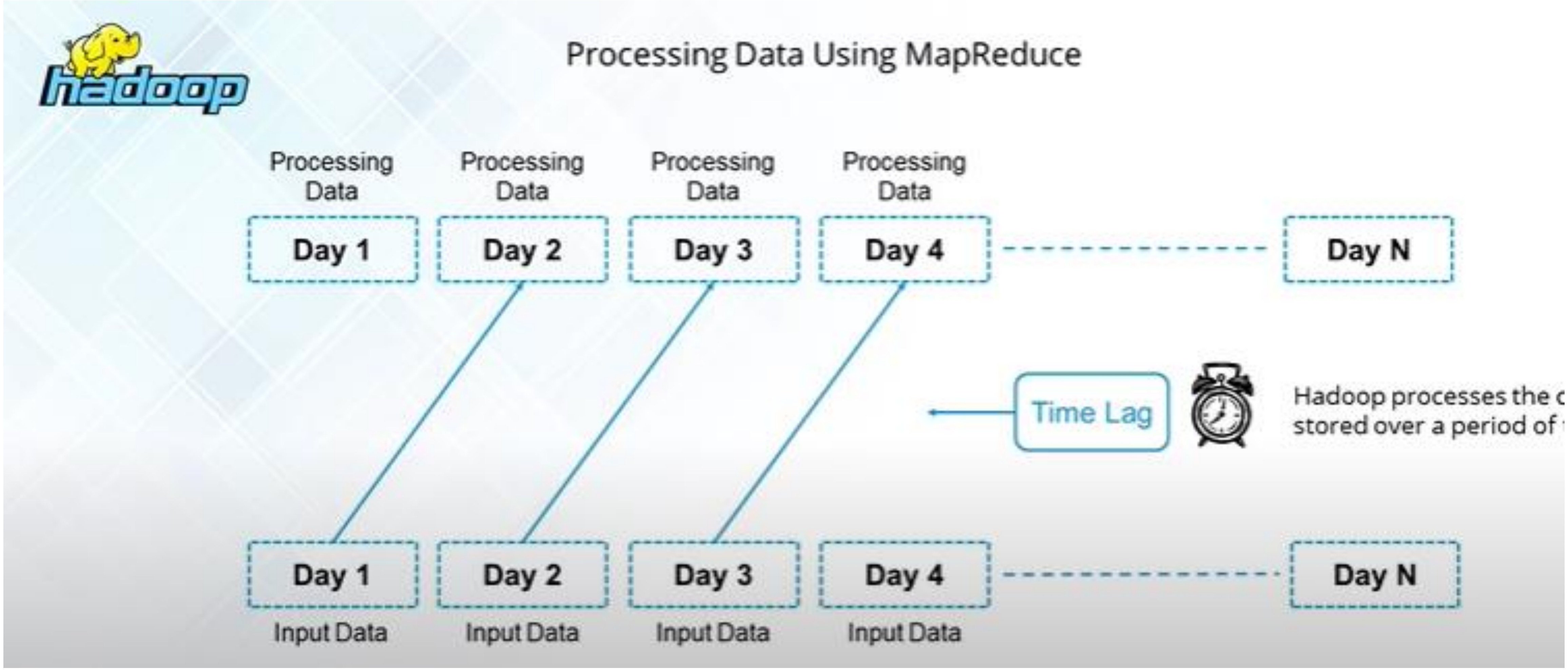
- Analytics based on real time data for instant result is Real Time Analytics/Stream Analytics. E.g. credit card



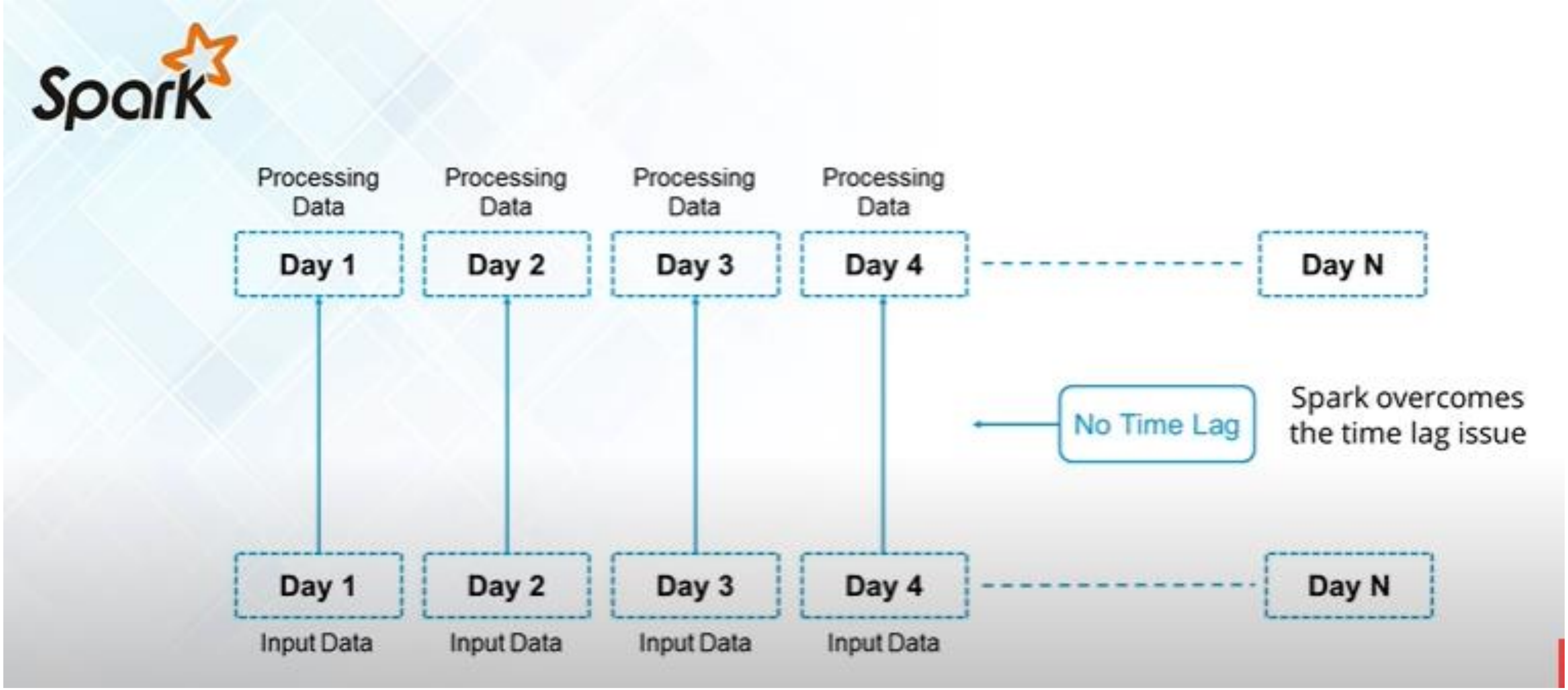
Use cases of Real Time Analytics

Why Spark when Hadoop is already there?

Batch Processing in Hadoop

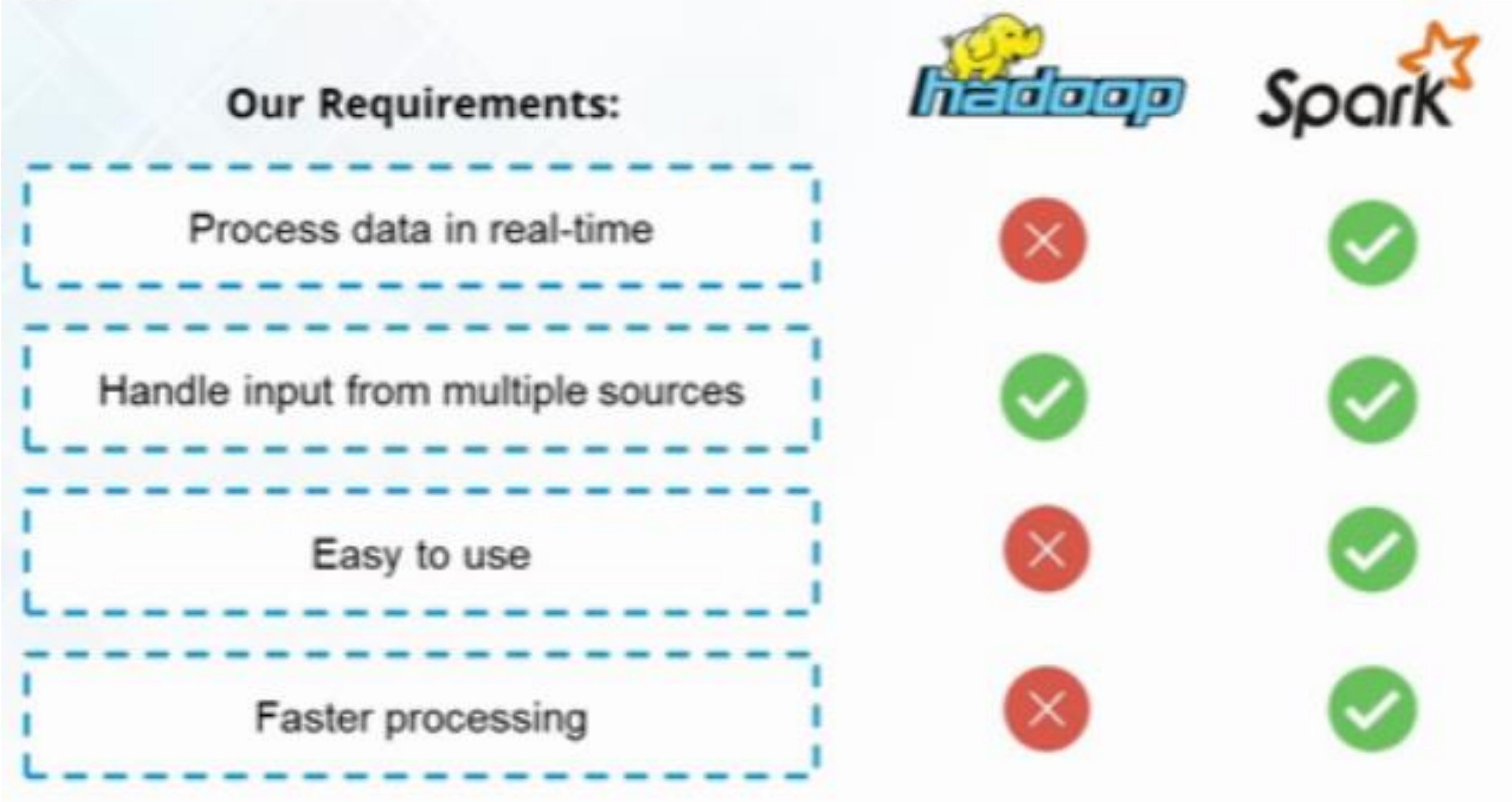


Real Time Processing in Spark



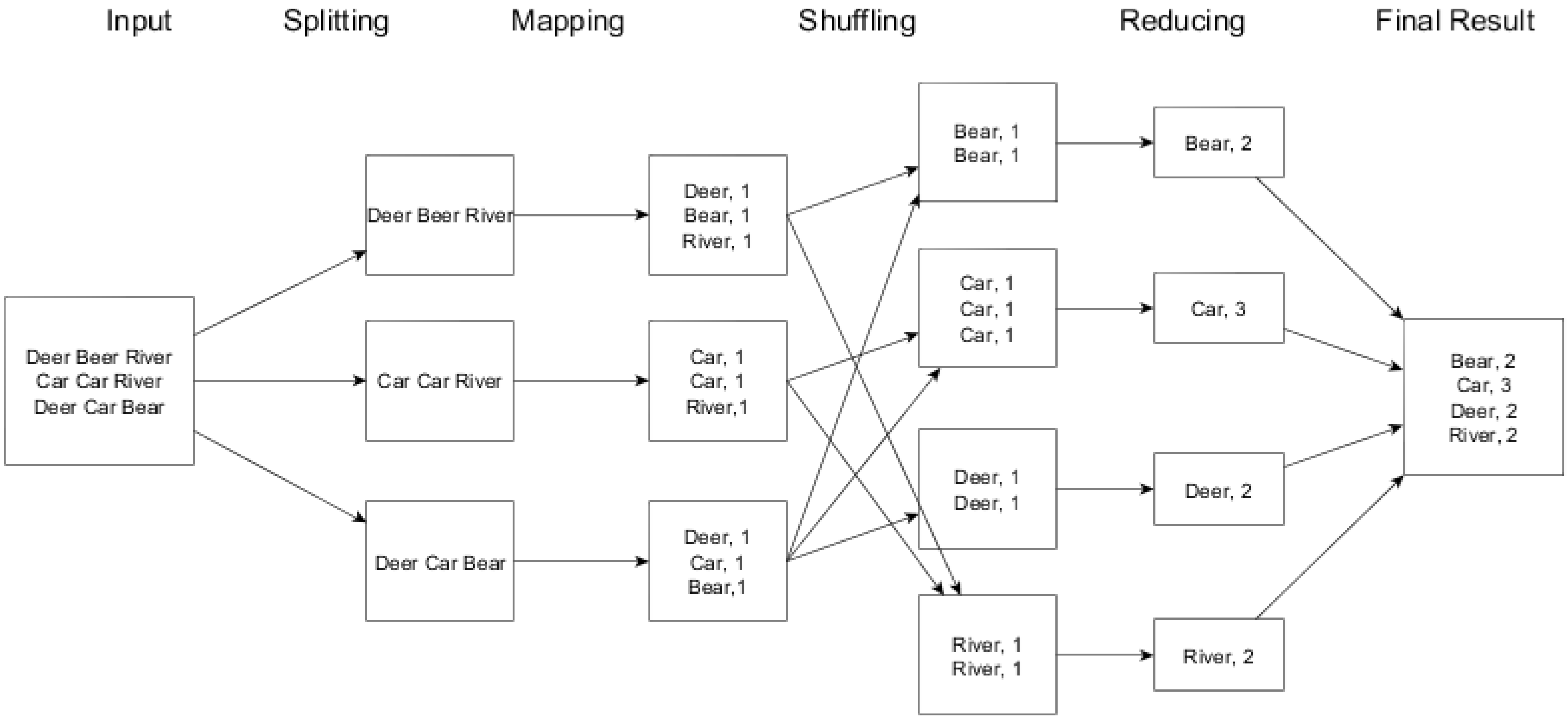
Spark vs Hadoop

- Hadoop implements Batch processing on big data and thus cannot deliver to Real Time use case needs.



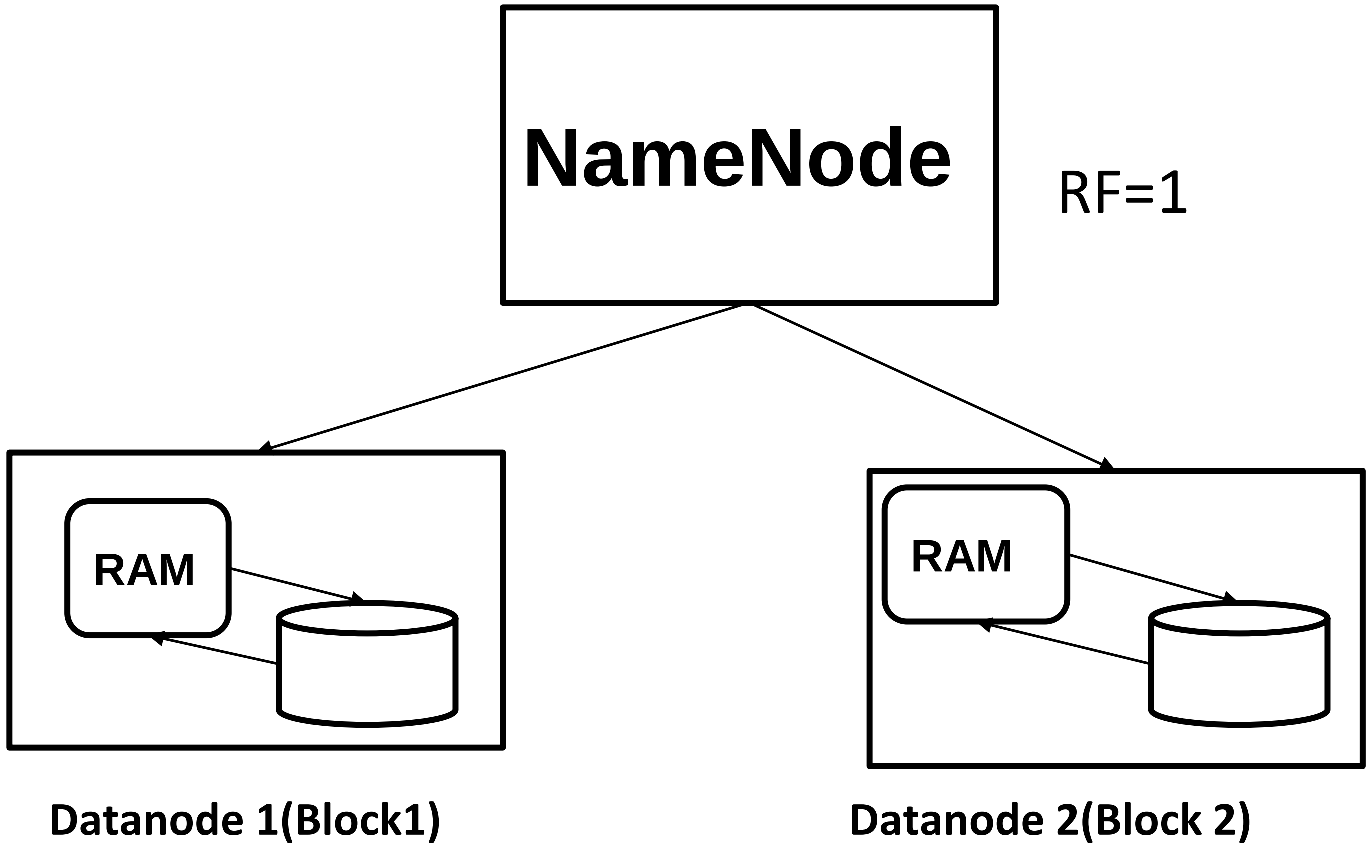
Example

WordCount in Mapreduce



Map reduce operations are slower

- In mapreduce the blocks are stored in disk.
- To perform any operation we need to bring it to Primary memory.
- So lots of I/O operations are required for mapreduce in each phase.

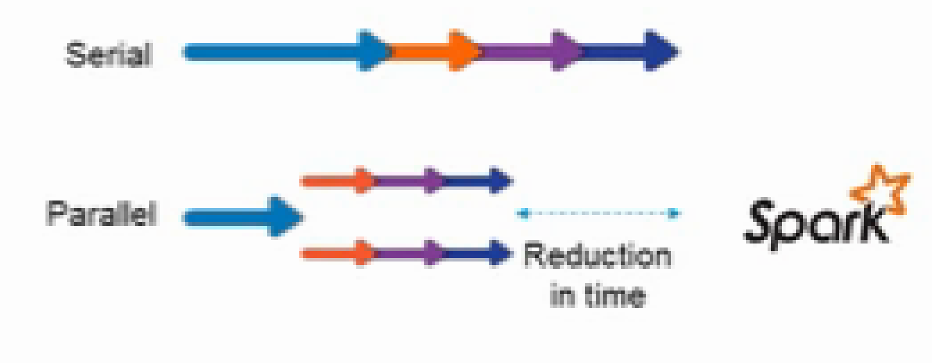


Spark Success Story

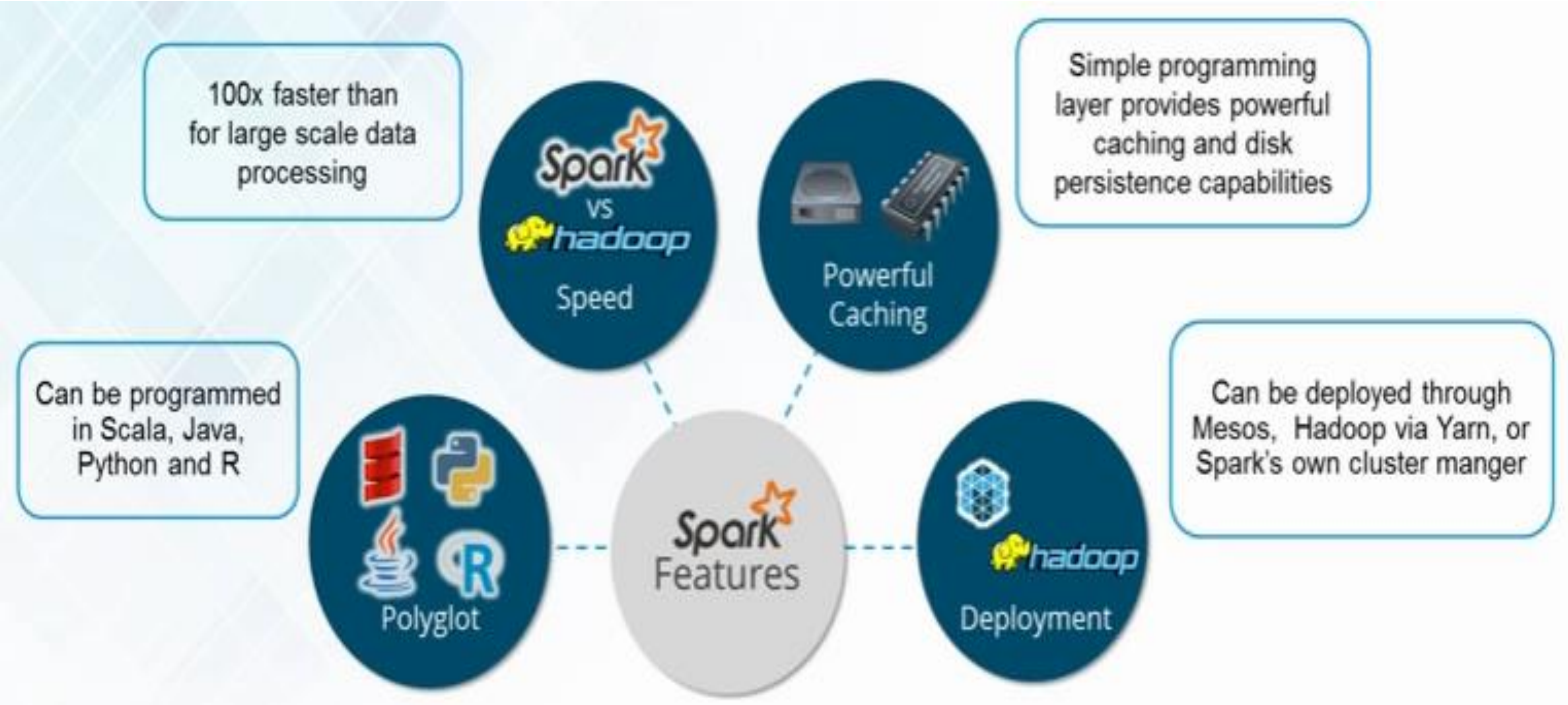
Spark Overview

What is Apache Spark?

- Apache Spark is an open-source cluster-computing framework for real time processing by Apache Software Foundation.
- Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.
- It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations



Why Spark?


100x faster than
for large scale data
processing
Spark
vs
hadoop
Speed
Powerful
Caching
Simple programming
layer provides powerful
caching and disk
persistence capabilities
Can be programmed
in Scala, Java,
Python and R
R
Polyglot
Spark
Features
hadoop
Deployment
Can be deployed through
Mesos, Hadoop via Yarn, or
Spark's own cluster manger

Using Hadoop Through Spark

Spark and Hadoop

Spark and Hadoop

Spark Features

- Speed
- Polyglot
- Advanced Analytics
- In-Memory Computation
- Hadoop Integration
- Machine Learning

Spark Features

1. Spark runs up to 100x times faster than MapReduce.

2. Polyglot: Programming in Scala, Python, Java and R

Spark Features

3. Lazy Evaluation: Delays evaluation till needed.

4. Real time computation & low latency because of in-memory computation

- In Memory Computation -

  Spark stores the data in the RAM of servers which allows quick access and in turn accelerates the speed of analytics.

Spark Features

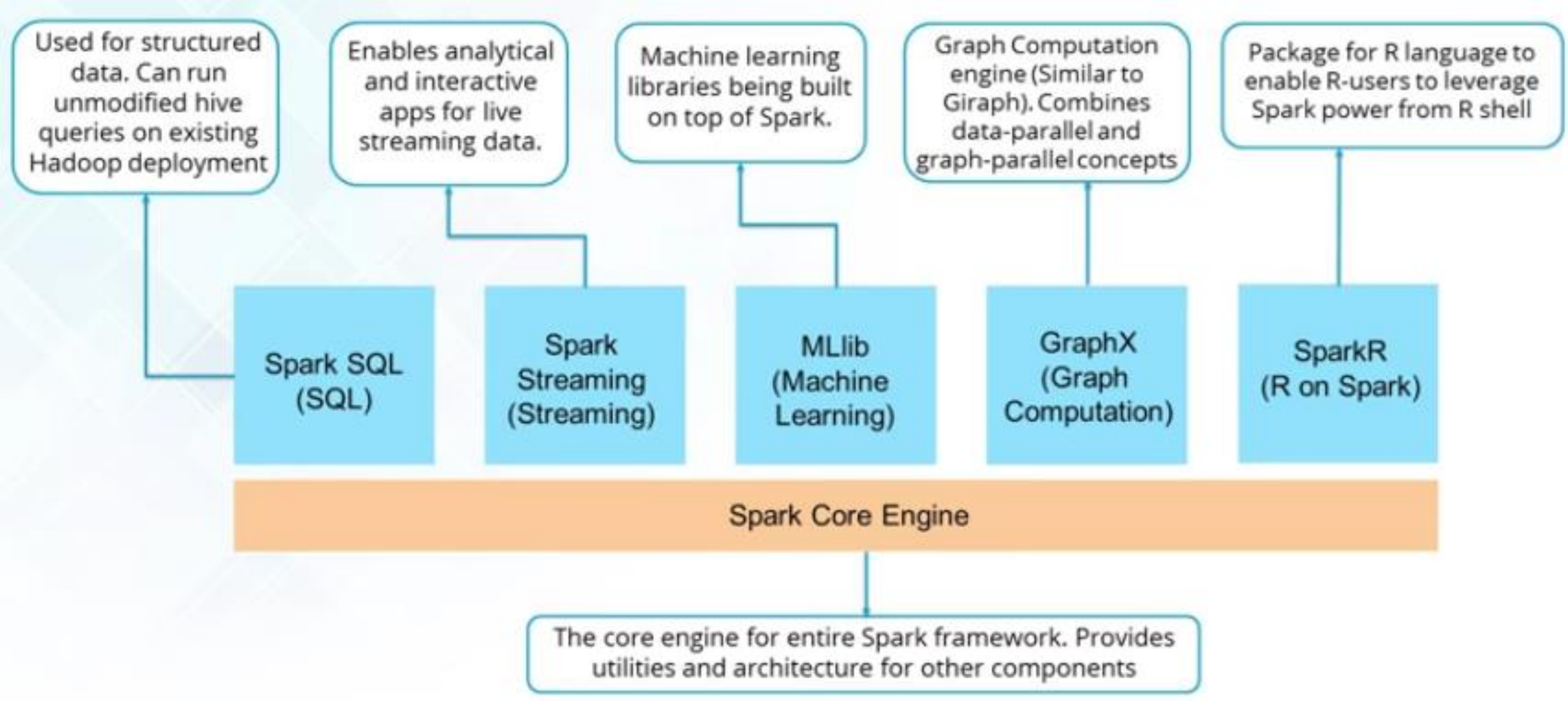
5. Hadoop Integration

6. Machine Learning for iterative tasks

Spark Ecosystem

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

Spark Ecosystem

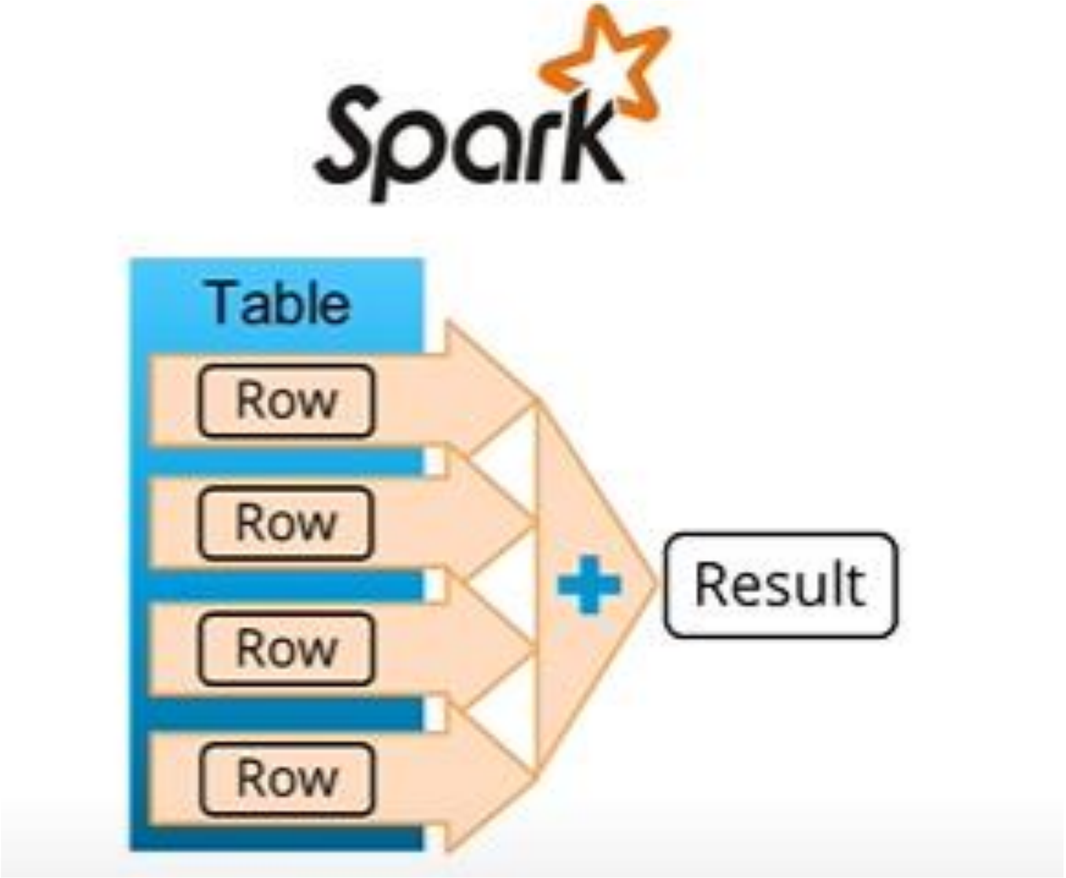


Spark Core

It is the base engine for large scale parallel and distributed data processing.

It is responsible for:

- Memory management and fault recovery
- Scheduling, distributing and monitoring jobs on a cluster.
- Interacting with storage system.



Spark Streaming

- Spark Streaming is used for processing real-time streaming data
- It is useful addition to the core Spark API
- Spark Streaming enables high-throughput and fault-tolerant stream processing of live data streams
- The fundamental stream unit is DStream which is basically a series of RDDs to process the real-time data

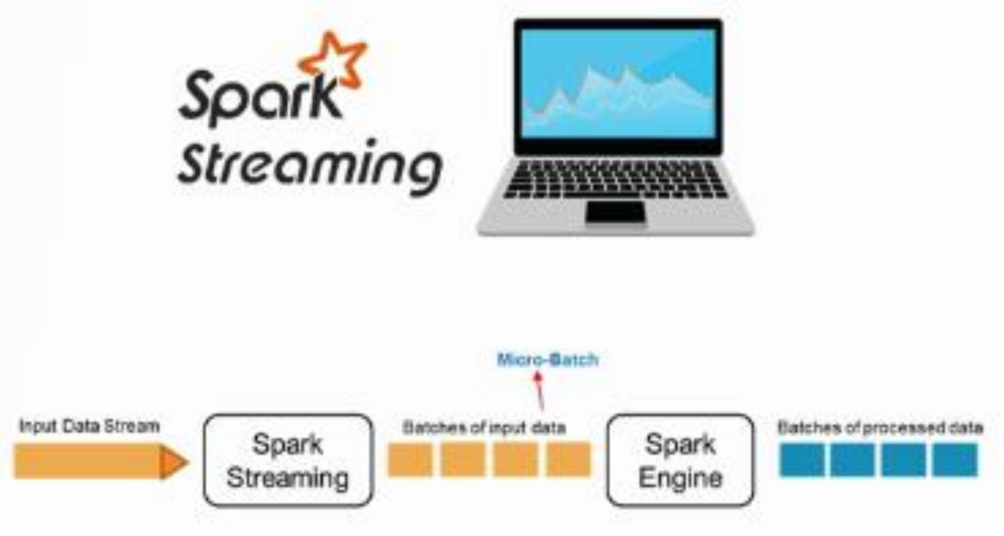


Figure: *Streams In Spark Streaming*

Spark Streaming

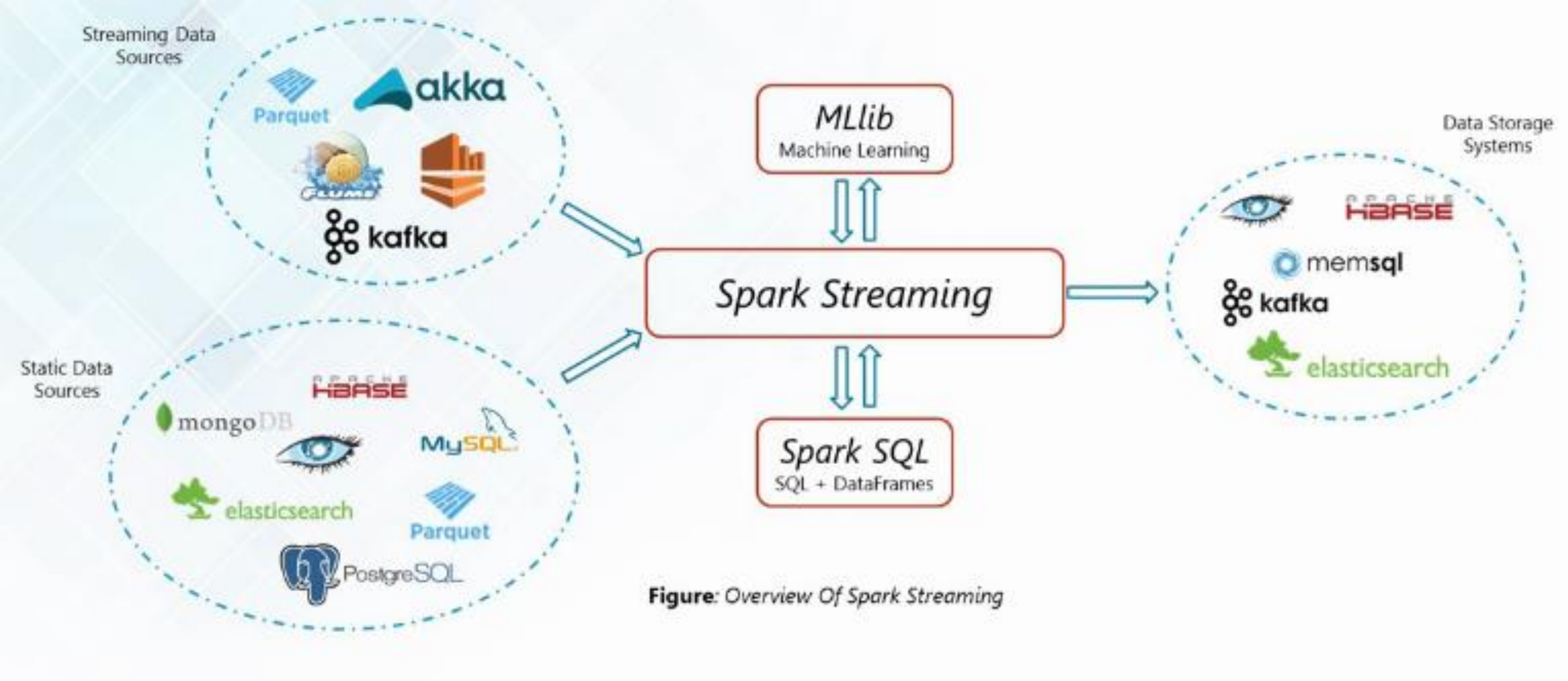
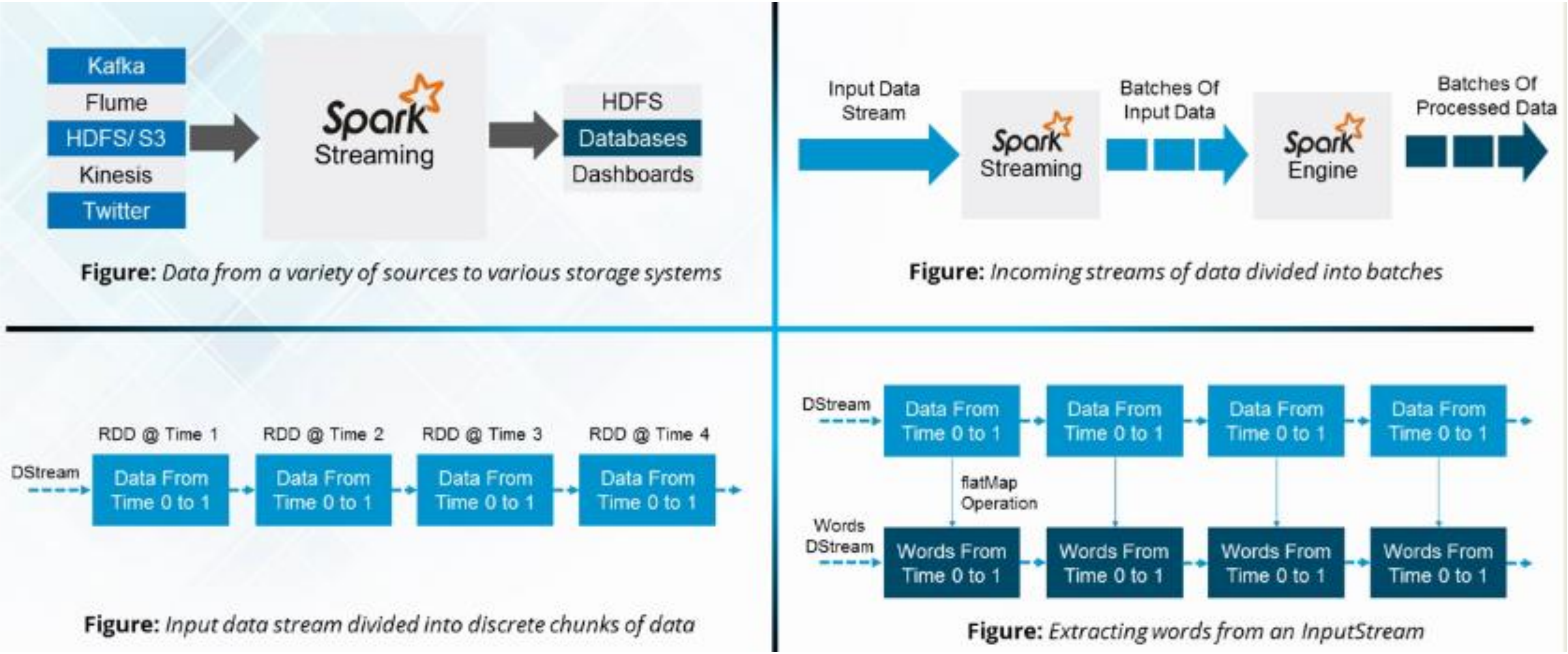


Figure: *Overview Of Spark Streaming*

Spark Streaming



Spark SQL

Spark SQL Features

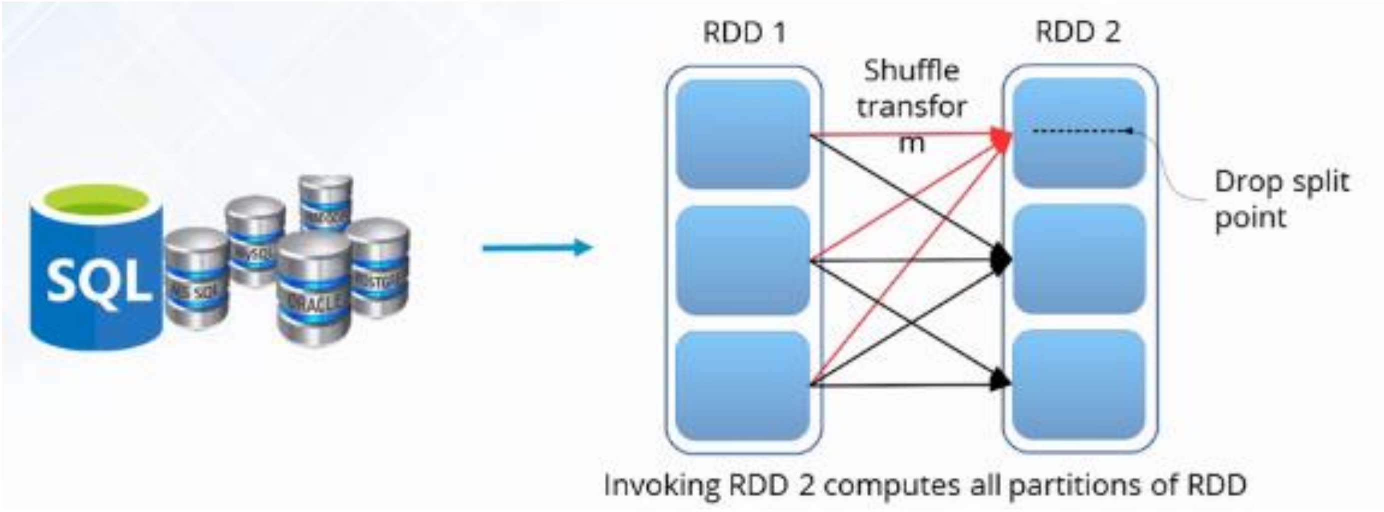
- Spark SQL integrates relational processing with Spark's fundamental programming

- Spark SQL is used for the structured/semi structured data analysis in Spark

Spark SQL Features

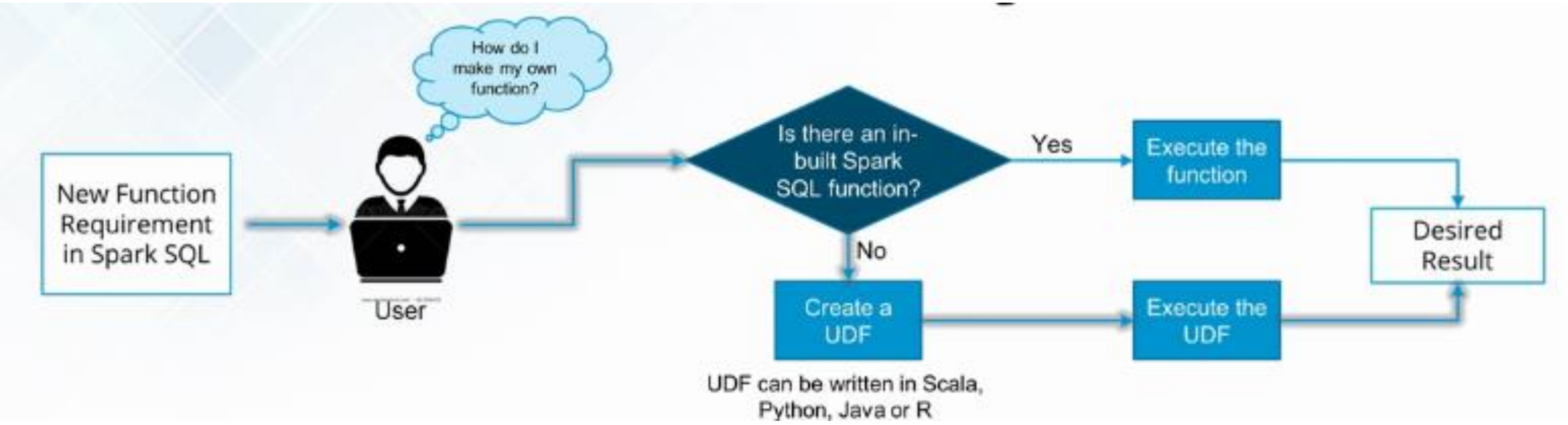
- Support for various data formats

- SQL queries can be converted into RDDs for transformations



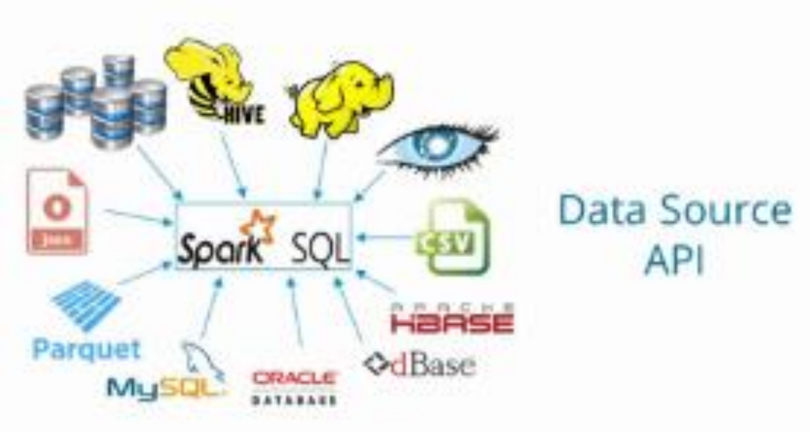
Spark SQL Features

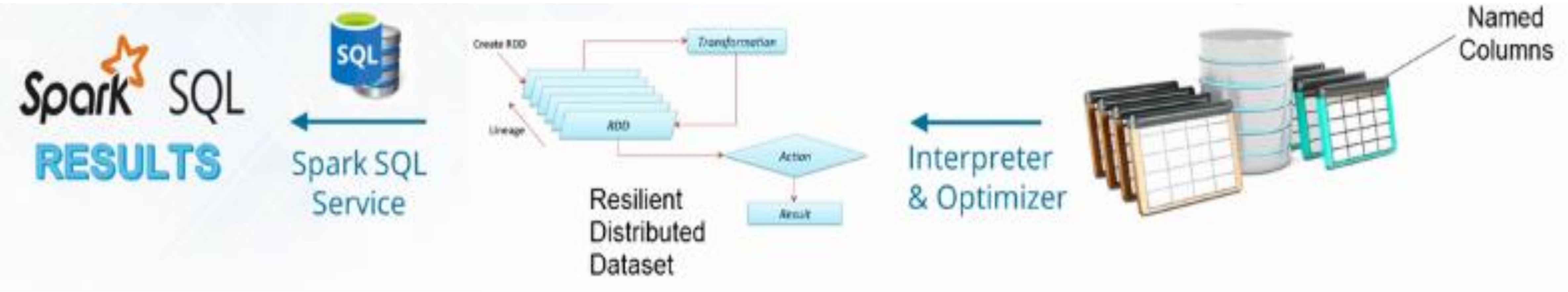
- Standard JDBC/ODBC Connectivity
- User Defined Functions lets user define new Column-based functions to extend the Spark vocabulary



Spark SQL Flow Diagram

- Spark SQL has the following libraries:
  - Data Source API
  - DataFrame API
  - Interpreter & Optimizer
  - SQL Service
- The flow diagram represents a Spark SQL process using all the four libraries in sequence

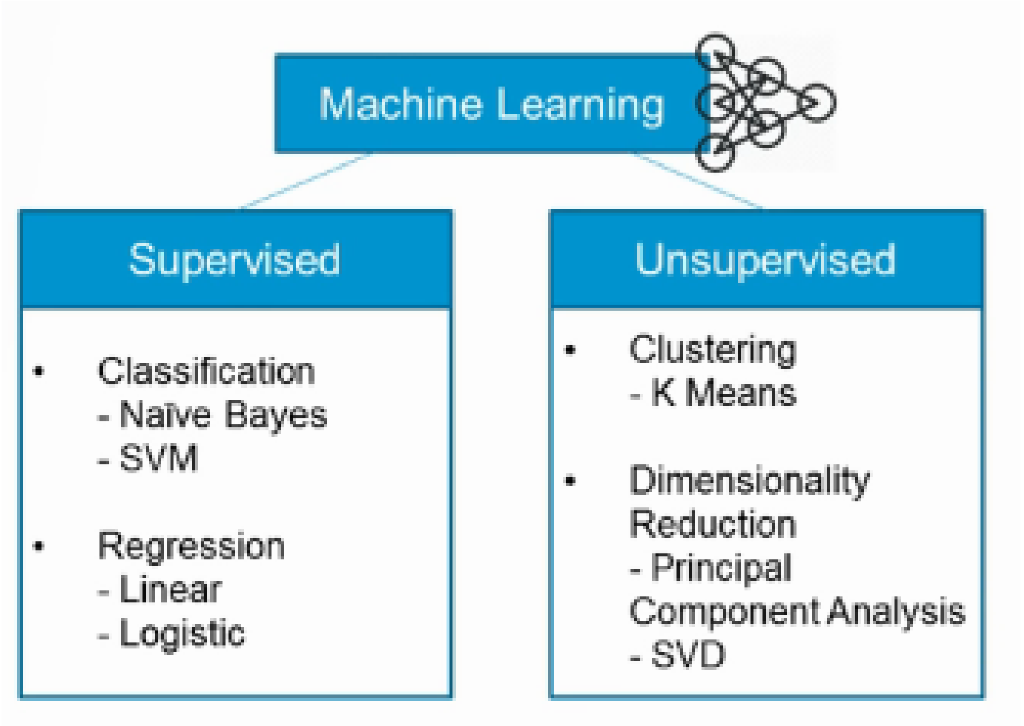




MLlib

- Machine learning can be broken down into two classes of algorithms:

1. Supervised algorithms use labeled data
2. Unsupervised algorithms make sense of the data without labels



MLlib Techniques

- Three common categories of techniques:

1. Classification
2. Clustering
3. Collaborative Filtering

Mllib - Techniques

- Classification: It is family of supervised machine learning algorithms that designate input as belonging to one of several pre-defined classes
  - Some common use cases for classification include: Credit Card fraud detection, Email spam detection
- Clustering: In clustering, an algorithm groups objects into categories by analyzing similarities between input examples

Mllib - Techniques

- Collaborative Filtering: These algorithms recommend items(filtering) based on preference information from many users(collaborative)

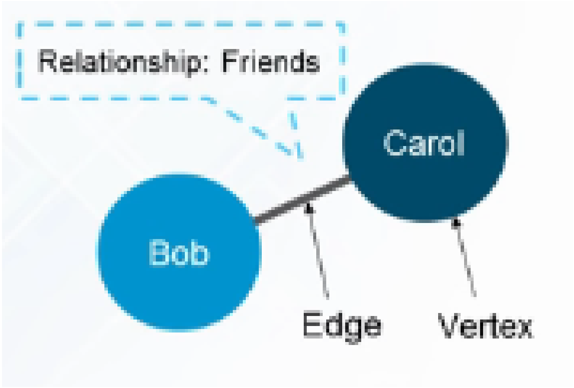
MLlib Features

1. ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
2. Featurization: feature extraction, transformation, dimensionality reduction, and selection
3. Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
4. Persistence: saving and load algorithms, models, and Pipelines
5. Utilities: linear algebra, statistics, data handling, etc.

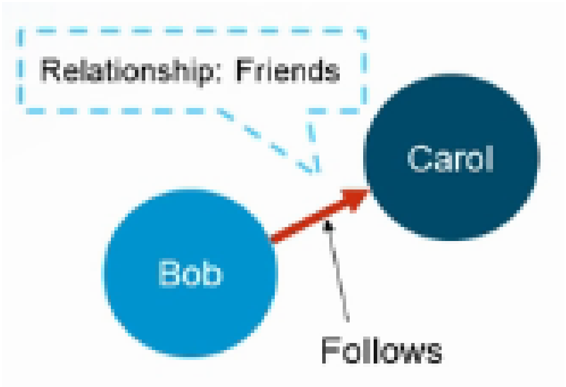
GraphX

GraphX

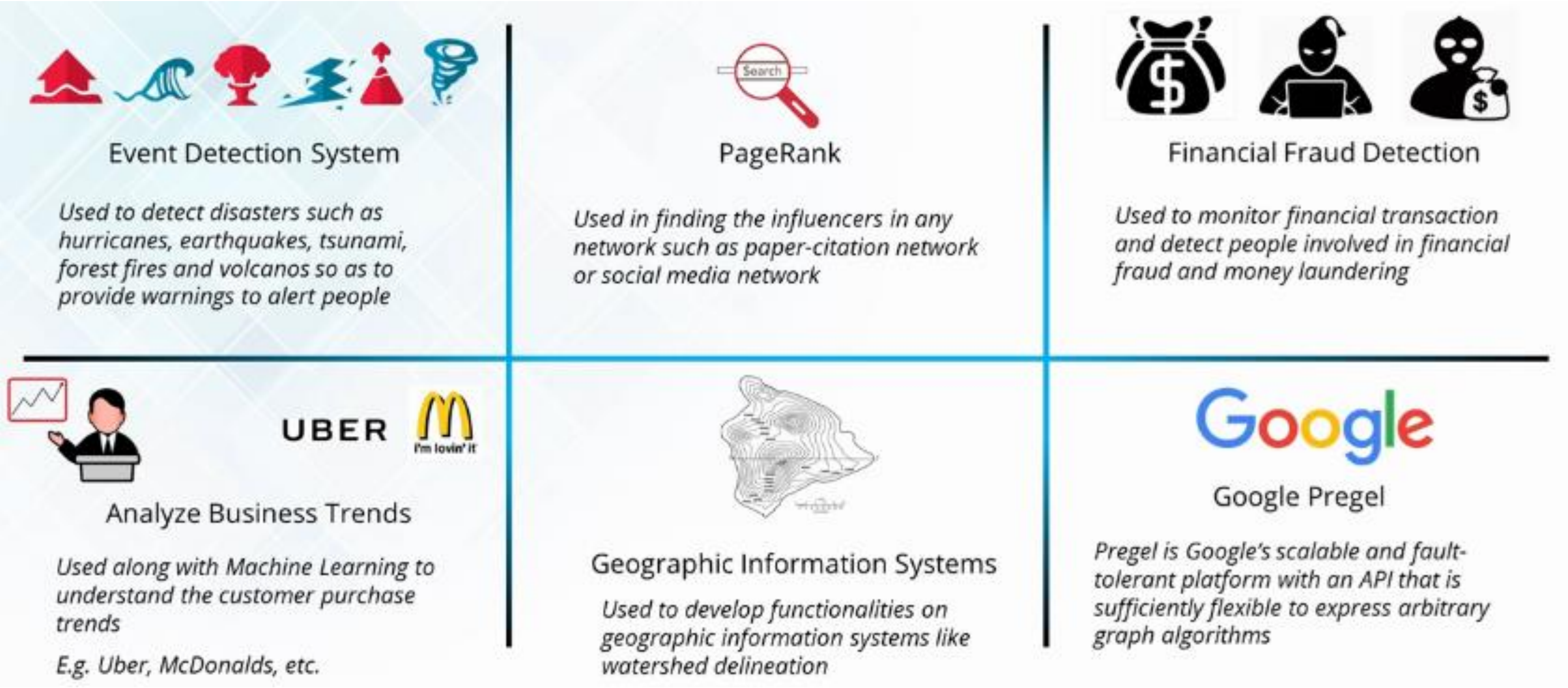
- A graph is a mathematical structure used to model relations between objects. A graph is made up of vertices and edges that connect them. The vertices are the objects and the edges are the relationships between them.



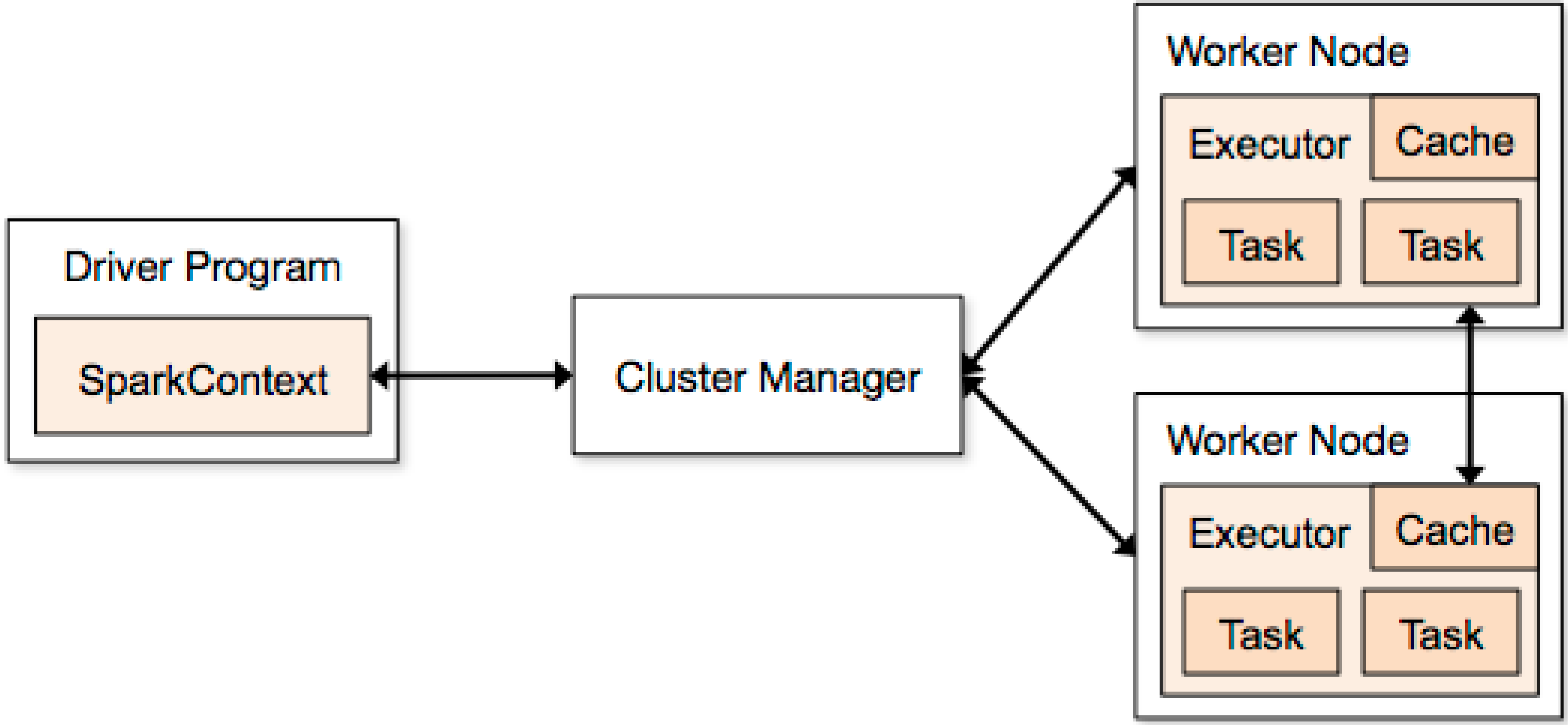
- A directed graph is a graph where the edges have a direction associated with them. E.g. User Bob follows Carol on Facebook.



GraphX Use Cases



Spark Architecture



https://spark.apache.org/docs/1.1.0/cluster-overview.html#:~:text=Once%20connected%2C%20Spark%20acquires%20executors,for%20the%20executors%20to%20run.

Driver Program

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

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

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

Cluster Manager

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

Worker Node

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

Executor

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

Task

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

No of executors(Example)

- 5 node cluster
- 128 GB RAM / Node
- 16 cores per node
- Total cores Available= *16 * 5 = 80*
- Total cores Usable (1 core for OS) = *15 * 5 = 75*
- Total RAM available per node = 128 GB

We leave out 2 GB RAM per Node for OS, leaving us with 126 GB of RAM to work with.

Typical configuration:

- --executor-cores: Based on the 5 cores per executor, we can have a maximum of 15 executors and 3 executors per node (75/5=15)
- --executor-memory: For the RAM allocation, we will allocate 126/3=41 GB RAM per executor
- --num-executors: Since we have to leave out one executor for AM, we will have only 14 total executors (15 - 1 = 14)

MapReduce

- Mappers are always launched on nodes where data is available
- No. of mappers = no. of blocks

Spark

- When executors are launched, there is no guarantee that they will be launched on same machines where data is available as YARN does not know anything about data locality.
- Executors might be launched on same machines or different machines, so initially it might take some time to fetch data in memory, but even then it would be faster than MapReduce.
- Number of executors and its size has to be decided while writing spark program.

MapReduce Block

- One or more blocks will be processed by mappers.
- Size of each block is 128 MB.
- Blocks are stored on disk.

Spark Partition

- One or more partitions will be processed by executors.
- Each block becomes one partition in spark.
- While using other data storage, data has to be divided into partitions first. Normally, spark tries to set the number of partitions automatically based on cluster.
- More partitions results in more parallelism.
- Partitions are stored in RAM

Executor Memory Utilization

- If 10 GB executor is launched, we cannot use full capacity for data storage.
- 10 % of memory is allocated to system calls. E.g. from 10 GB capacity, 1 GB is used for system calls.
- From remaining 90% of memory, we can utilize only 60% of memory.
- Remaining is used by garbage collector, JVM management and all.
- So, we can utilize only 54% of full capacity for data storage. e.g., from 10 GB storage, we can utilize only 5.8 GB.
- Each executor needs at least one processor core. So, on dual core, only two executors can be launched.

Spark Dynamic Memory Utilization

- If set to true, it will go for dynamic memory utilization. It means, if 8 executors are launched, after the work is finished, it will automatically kill idle executors.
- If set to false, it will not go for dynamic memory utilization. It means, if 8 executors are launched, after the work is finished, it will still keep them as it is and will not kill idle executors.
- Executors can be killed automatically after completing their jobs, but driver has to be stopped. Otherwise even after the job is completed, driver will still be there and eat resources.(e.g. default 1 core and all)

RDD

1. **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph **[DAG]** and so able to recompute missing or damaged partitions due to node failures.
2. **Distributed**, since Data resides on multiple nodes.
3. **Dataset** represents records of the data you work with. The user can load the dataset externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

RDD Fundamentals

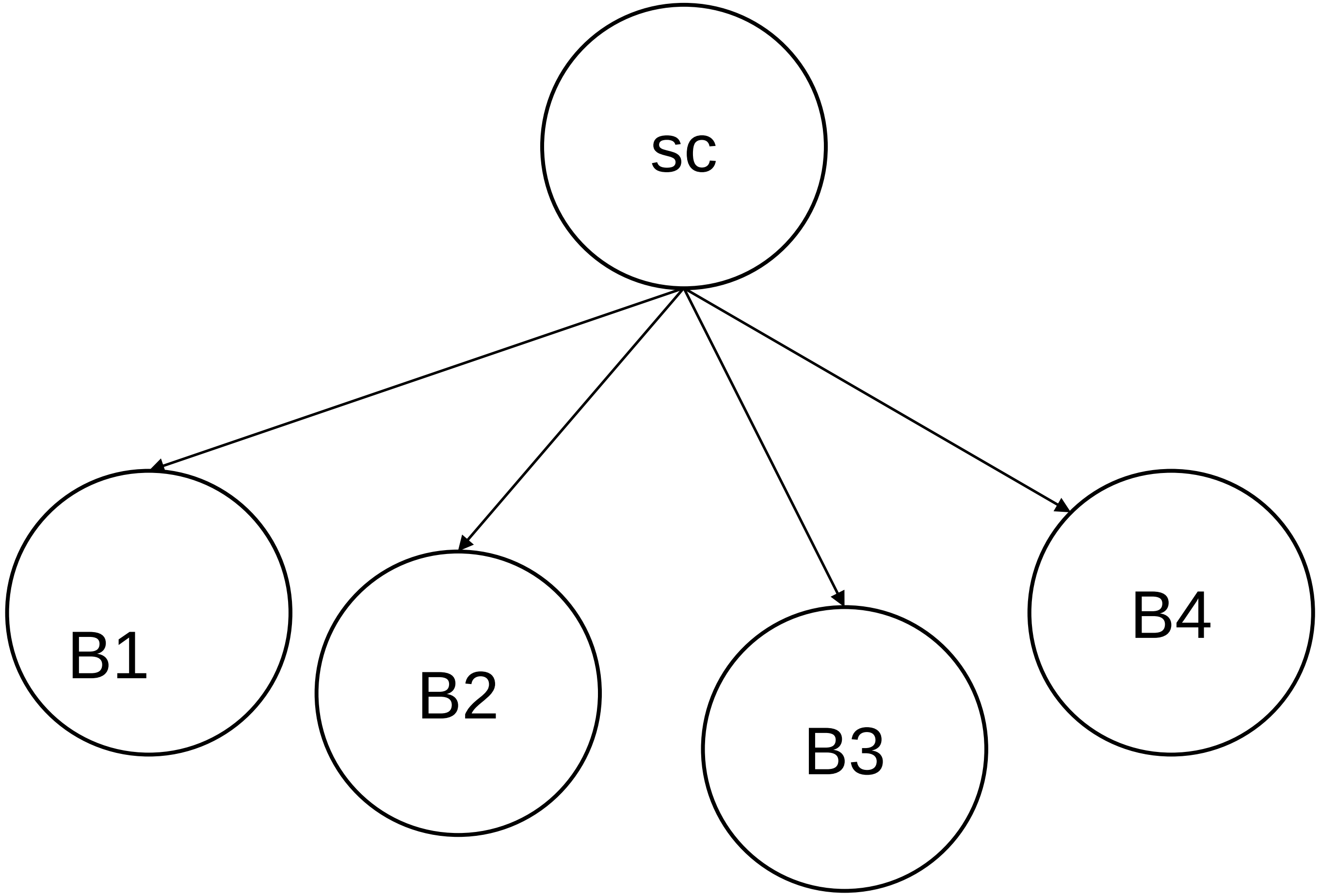
- Spark revolves around the concept of a ***resilient distributed dataset*** (RDD), which is a **fault-tolerant,immutable** collection of elements that can be operated on in parallel.

- There are two ways to create RDDs:
  - *parallelizing* an existing collection in your driver program, or
  - referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.
- RDDs are immutable. If any operation is performed on RDDs, it will create a new RDD.

Example

File1.txt

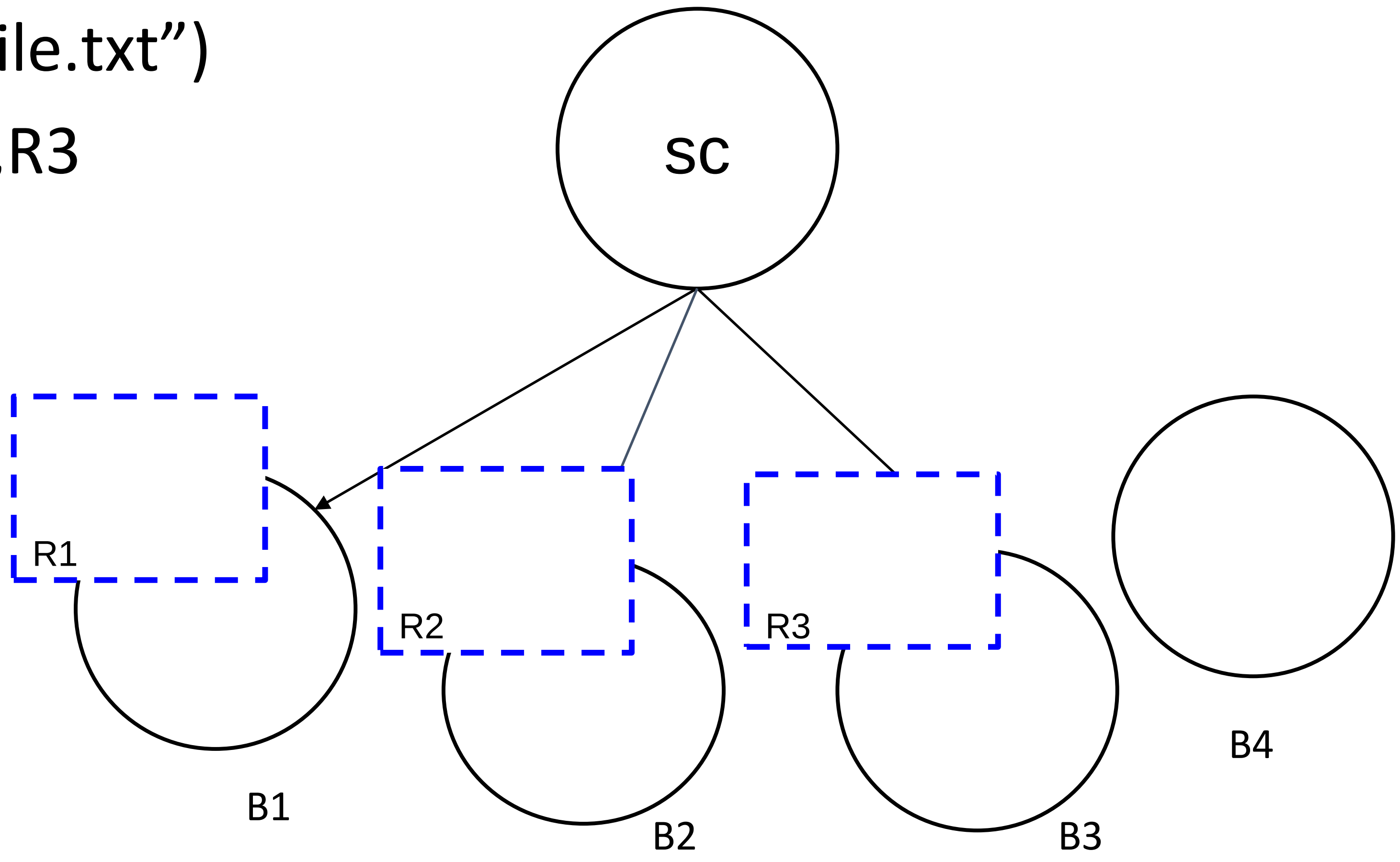
1,3,85,65,99….	B1
2,80,95,6,23….	B2
90,3,4,68,74….	B3



Creating RDD

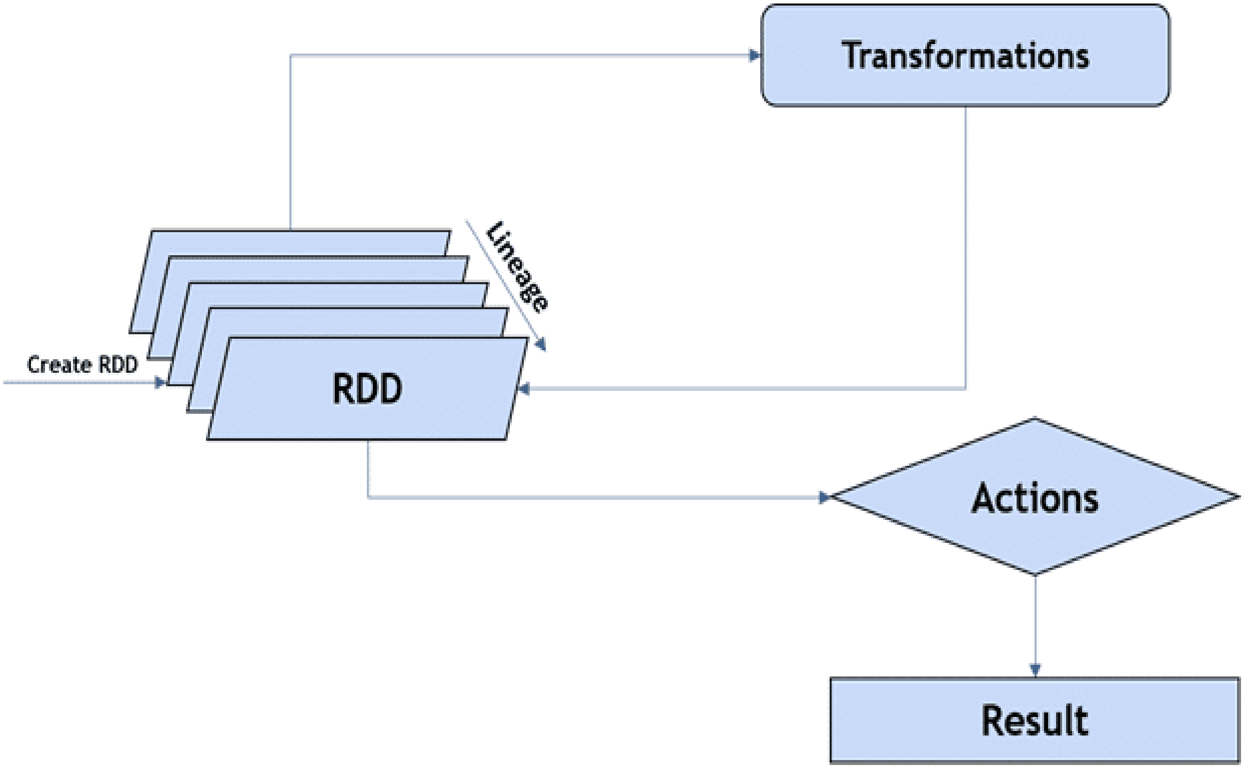
1. RDD number=sc.textfile(“file.txt”)

   // it will create RDD R1,R2,R3



Operations on RDD

1. Transformation
2. Actions



https://intellipaat.com/blog/tutorial/spark-tutorial/programming-with-rdds/

Transformations

These are functions that accept the existing RDDs as input and output one or more RDDs. However, the data in the existing RDD in Spark does not change as it is immutable.

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

RDD OPERATION(Transformation)

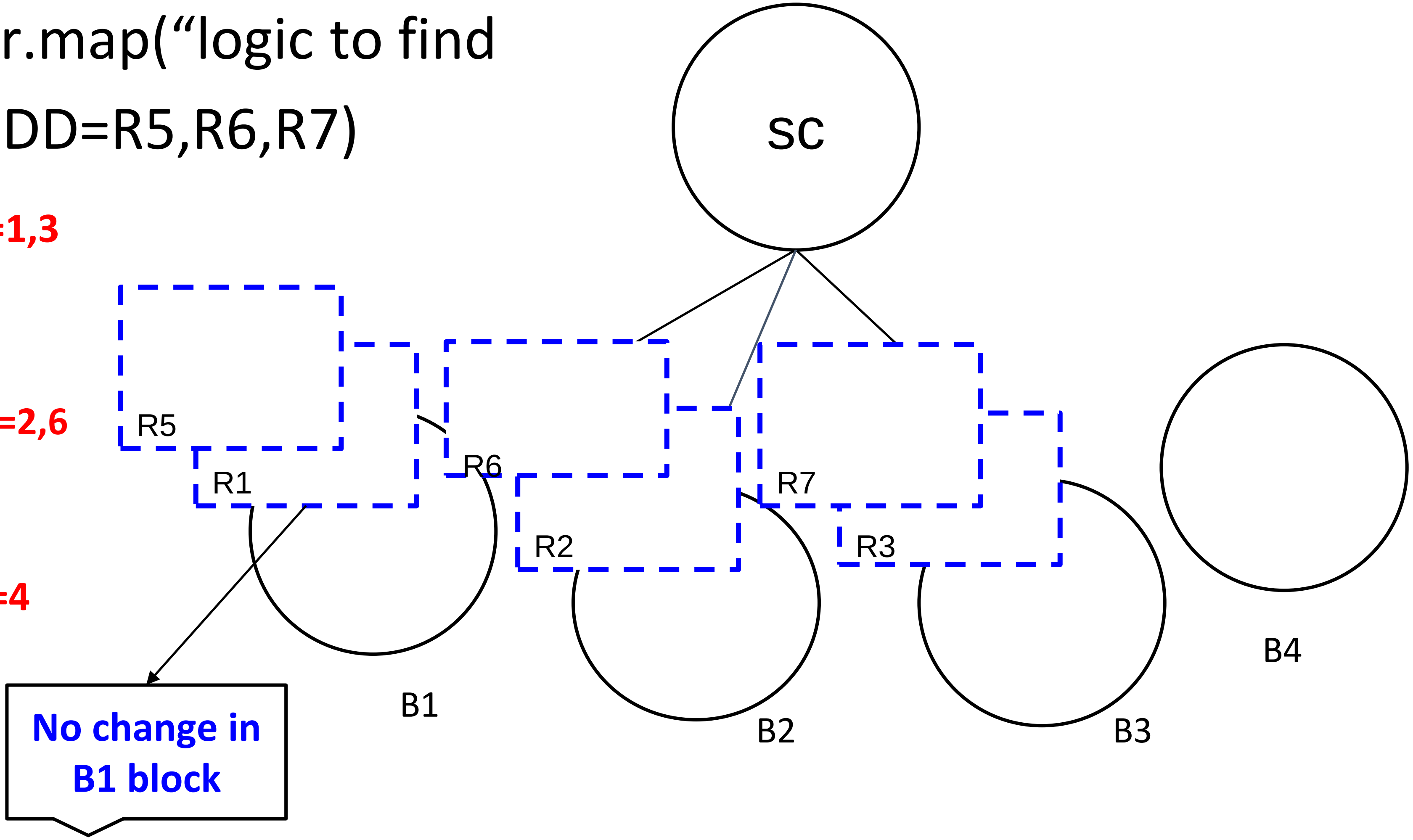
2. RDD filter1=number.map("logic to find The values < 10")(RDD=R5,R6,R7)

1,3,85,65,99….
2,80,95,6,23….
90,33,4,68,74….

R5=1,3

R6=2,6

R7=4

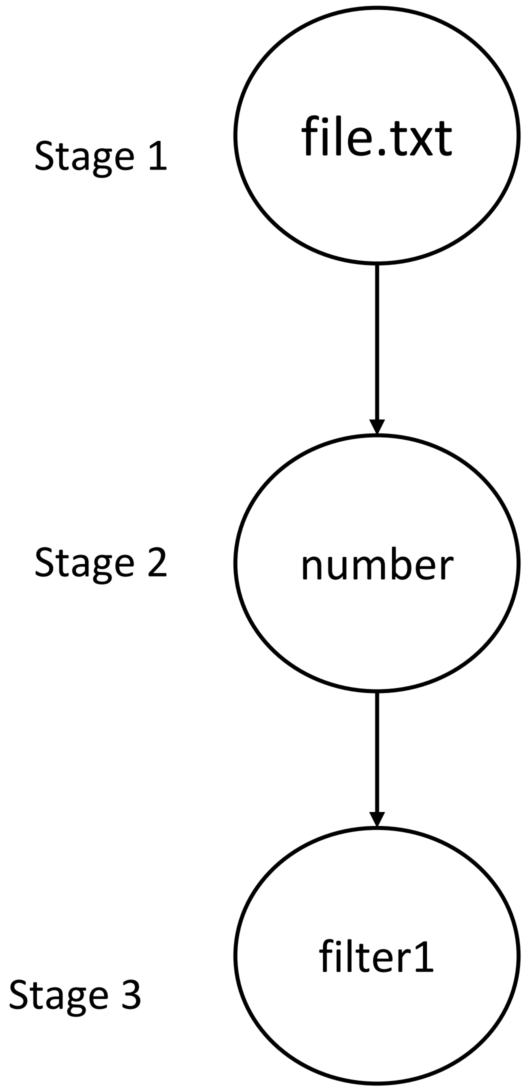


DAG(Directed Acyclic Graph or lineage)

Stage 1 : store file.txt file

Stage 2: create RDD filter1 for stage 1 .

Stage 3: create RDD number for stage 2.



A **stage** represents a ***set of tasks that can be executed together*** in a single wave of computation, resulting in a more efficient execution of the Spark job.

Each stage task is represented as a node in the DAG, and the dependencies between tasks are represented as directed edges.

Actions

Actions in Spark are functions that return the end result of RDD computations. It **uses a lineage graph t**o load data onto the RDD in a particular order. After all of the transformations are done, actions return the final result to the Spark Driver. Actions are operations that provide non-RDD values.

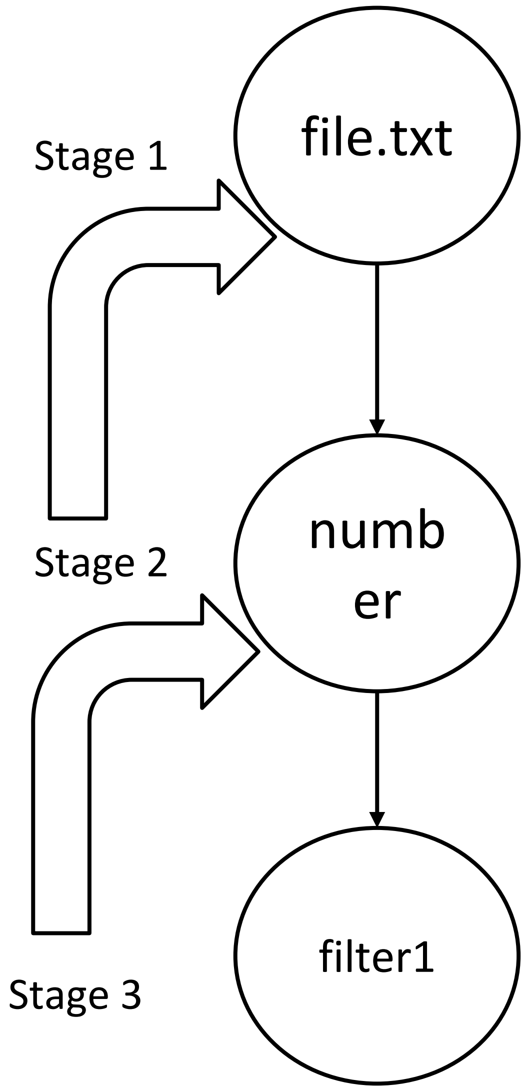
1. **count()**
   Gets the number of data elements in an RDD
1. **collect()**
   Gets all the data elements in an RDD as an array
1. **reduce()**
   Aggregates data elements into an RDD by taking two arguments and returning one
1. **take(n)**
   Fetches the first *n* elements of an RDD
1. **foreach(operation)**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

Use of DAG during action

It **converts a logical execution plan** (which consists of the RDD lineage formed through RDD transformations) **into a physical execution plan.**

3. filter1.collect()

Output :-1,3,2,6,4



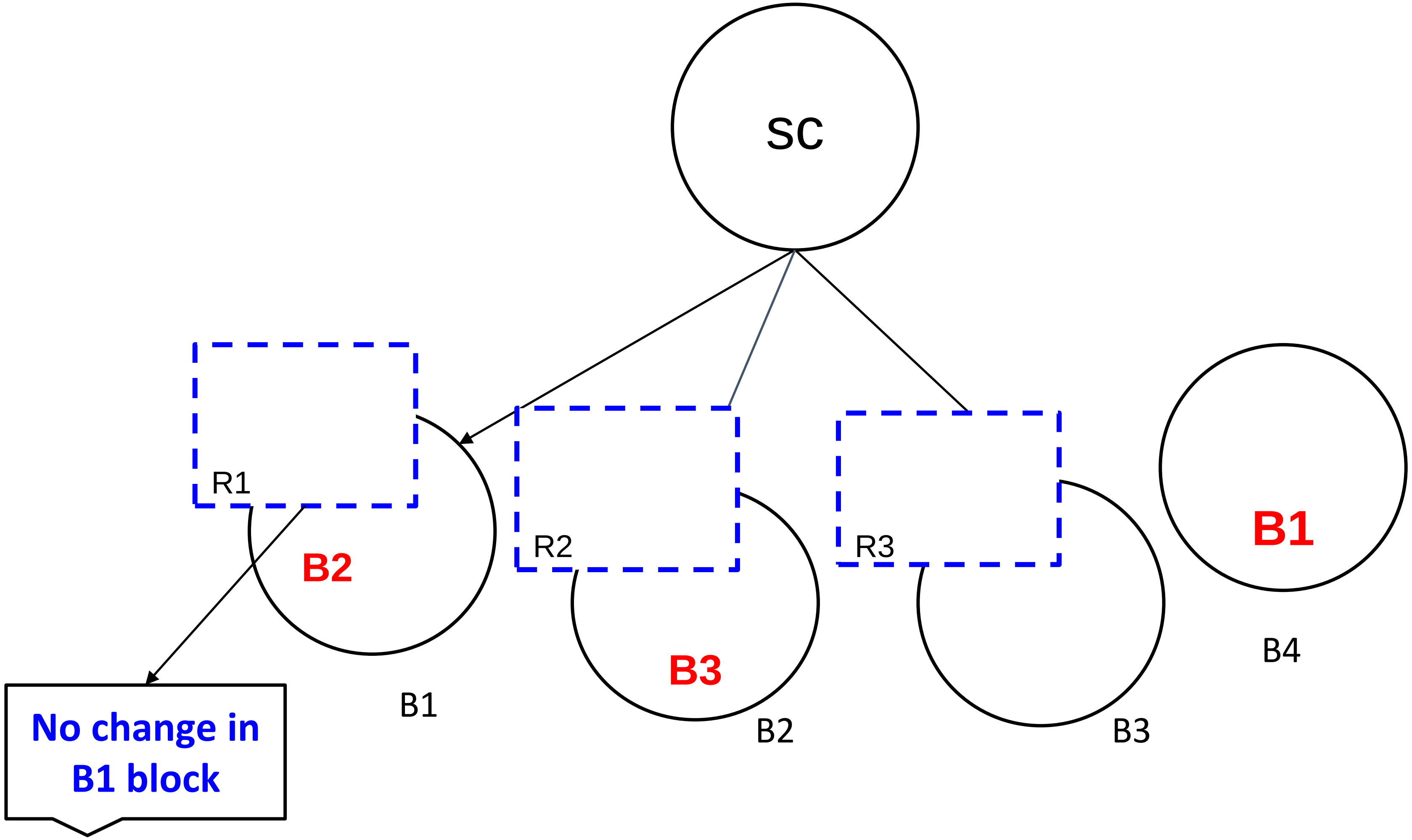
DAG Scheduler

It is the high-level scheduling layer that implements ***stage-oriented scheduling***.

1. It **computes a DAG of stages** for each job, **keeps track** of which RDDs and stage outputs are materialized, and **finds a minimal schedule** to run the job.
2. It then **submits stages** as *TaskSets* to an underlying *TaskScheduler* implementation that runs them on the cluster.
3. It **converts** a logical execution plan (which consists of the RDD lineage formed through RDD transformations) into a physical execution plan.

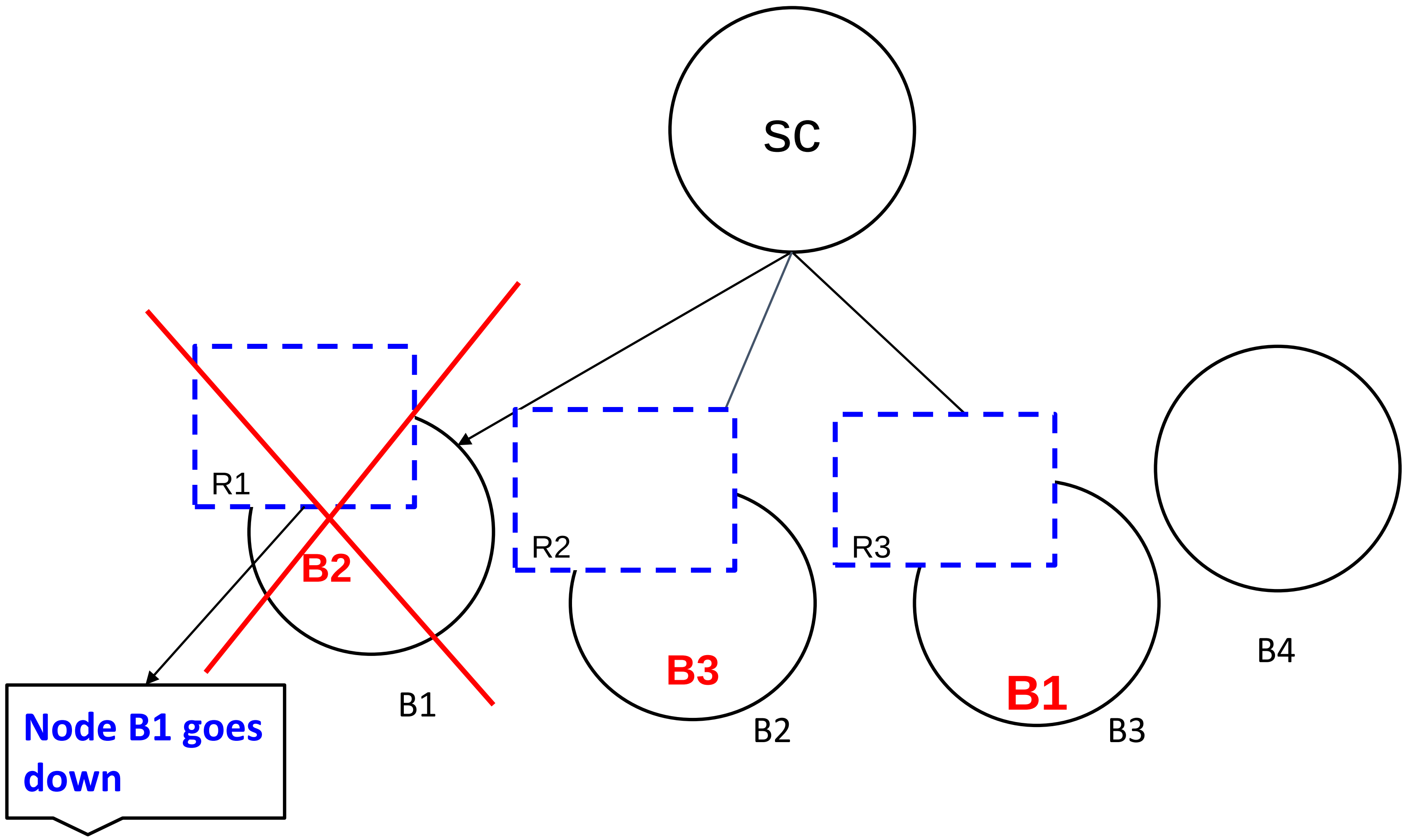
Fault tolerance

RF=2



Fault tolerance

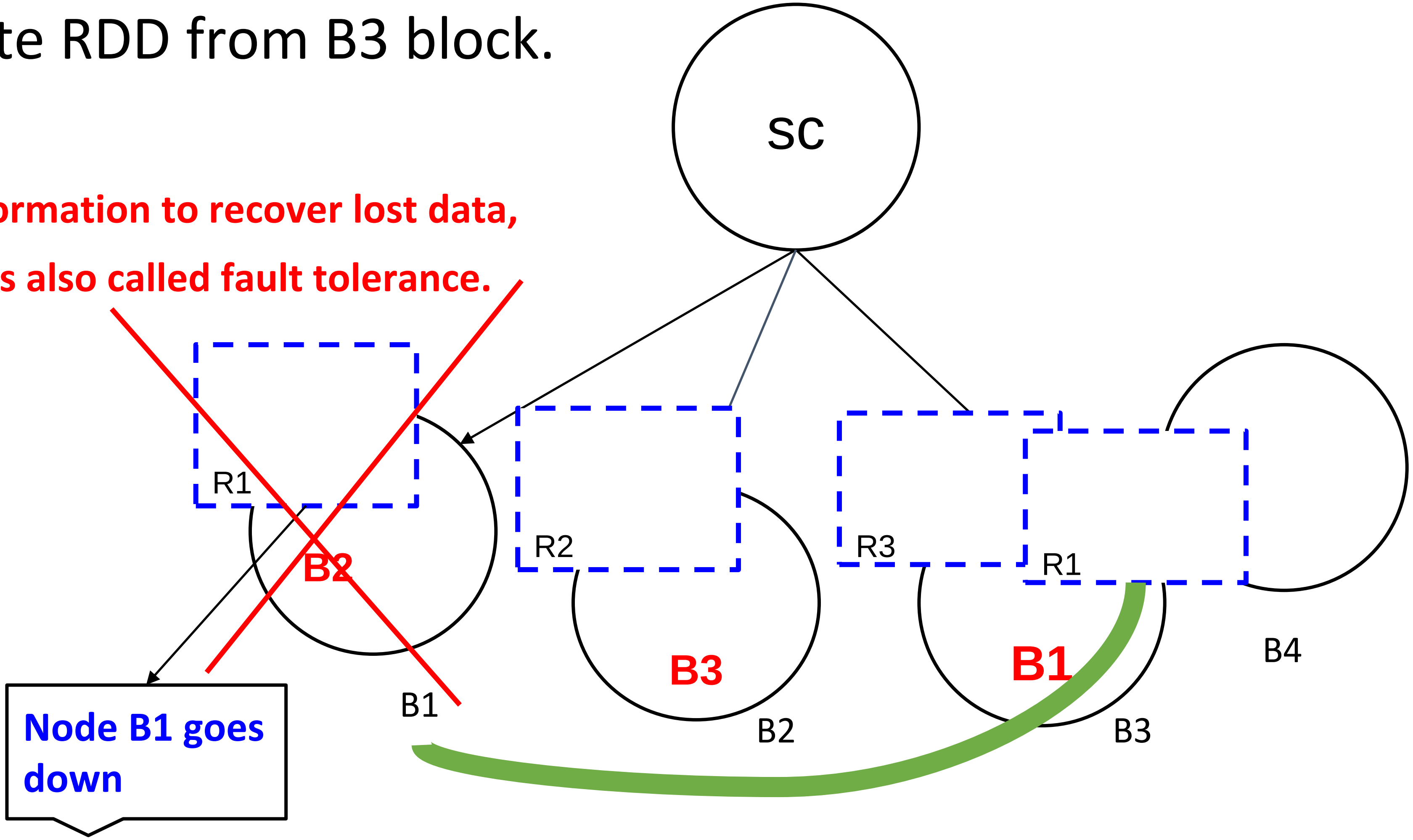
Node failure(B1)



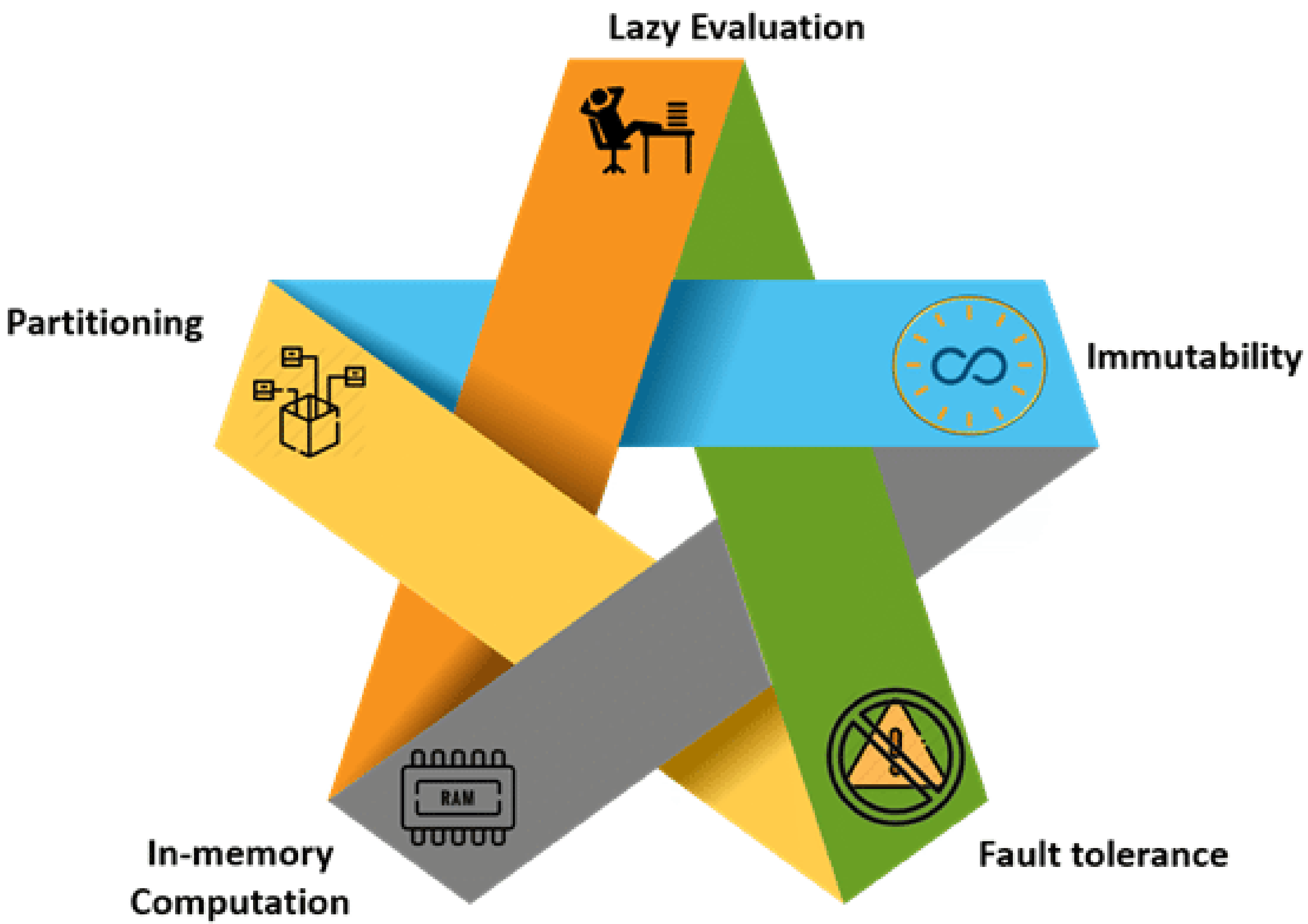
Fault tolerance

Use DAG to compute RDD from B3 block.

RDDs track data lineage information to recover lost data, automatically on failure. It is also called fault tolerance.



Features of RDD in Spark



Features of RDD in Spark

- **Resilience**: RDDs track data lineage information to recover lost data, automatically on failure. It is also called fault tolerance.
- **Distributed**: Data present in an RDD resides on multiple nodes. It is distributed across different nodes of a cluster.
- **Lazy evaluation**: Data does not get loaded in an RDD even if you define it. Transformations are actually computed when you call action, such as count or collect, or save the output to a file system.

Features of RDD in Spark

- **Immutability:** Data stored in an RDD is in the read-only mode—you cannot edit the data which is present in the RDD. But, you can create new RDDs by performing transformations on the existing RDDs.
- **In-memory computation**: An RDD stores any immediate data that is generated in the memory (RAM) than on the disk so that it provides faster access.
- **Partitioning:** Partitions can be done on any existing RDD to create logical parts that are mutable. You can achieve this by applying transformations to the existing partitions.

Scala

Scala programming is a general-purpose computer language that supports both object-oriented and functional styles of programming on a larger scale. Scala is a strong static type of programming language and is influenced by the Java programming language.

val,var ,def in scala

- Val -makes a variable *immutable*
- Var -makes a variable mutable
- Def- used to define a fucntion

```
def add(a: Int, b: Int) = a + b
```

```
scala> val a = 'a'
a: Char = a

scala> a = 'b'
<console>:12: error: reassignment to val
       a = 'b'
         ^
```

```
scala> var a = 'a'
a: Char = a

scala> a = 'b'
a: Char = b
```

How to create RDD?

1. Parallelize method by which already existing collection can be used in the driver program.

1. By referencing a dataset that is present in an external storage system such as HDFS, HBase.

1. New RDDs can be created from an existing RDD.

Creating RDD using parallelize() method

1. **var rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))**

```
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
```

2. **Read Result**

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

scala>
```

3. **Count** :The count action is used to get the total number of elements present in the particular RDD.
   Rdd1.count

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

scala> █
```

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

   rdd1.distinct.collect

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

scala> █
```

5. **Filter** transformation creates a new dataset by selecting the elements according to the given condition.

Syntax of RDD filter()

val filteredRDD = inputRDD.filter(predicate)

Here, inputRDD is the RDD to be filtered and predicate is a function that takes an element from the RDD and returns a boolean value indicating whether the element satisfies the filtering condition. The filteredRDD is the resulting RDD containing only the elements that satisfy the predicate.

rdd1.filter(x => x < 50).collect

```
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.

rdd1.sortBy(x => x, true).collect
rdd1.sortBy(x => x, false).collect

```
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> 
```

Map

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

rdd1.map(x => x + 1).collect

```
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

```
scala> val rdd2 = sc.parallelize(List(25, 73, 97, 78, 27, 82))
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize a
t <console>:27

scala> 
```

```
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> 
```

Reduce Action

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

Syntax:def reduce(f: (T, T) => T): T

rdd1.reduce((x, y) => x + y)

First

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

rdd1.first()

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

scala>
```

Take

Take action returns the first n elements in the RDD.

rdd1.take(5)

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

scala>
```

Wordcount with spark-shell (scala spark shell)

Step 1: Start the spark shell using following command and wait for prompt to appear

```
spark-shell
```

Step 2: Create RDD from a file in HDFS, type the following on spark-shell and press enter:

```
var linesRDD = sc.textFile("/data/mr/wordcount/input/big.txt")
```

Step 3: Convert each record into word

```
var wordsRDD = linesRDD.flatMap(_.split(" "))
```

Wordcount with spark-shell (scala spark shell)

Step 4: Convert each word into key-value pair

```
var wordsKvRdd = wordsRDD.map((_, 1))
```

Step 5: Group By key and perform aggregation on each key:

```
var wordCounts = wordsKvRdd.reduceByKey(_ + _ )
```

Step 6: Save the results into HDFS:

```
wordCounts.saveAsTextFile("my_spark_shell_wc_output")
```

Spark Wordcount Output

/home/hadoop/output$ cat my_spark_shell_wc_output.txt

(branches,1)
(sent,1)
(mining,1)
(tasks,4)

▸ Event Timeline

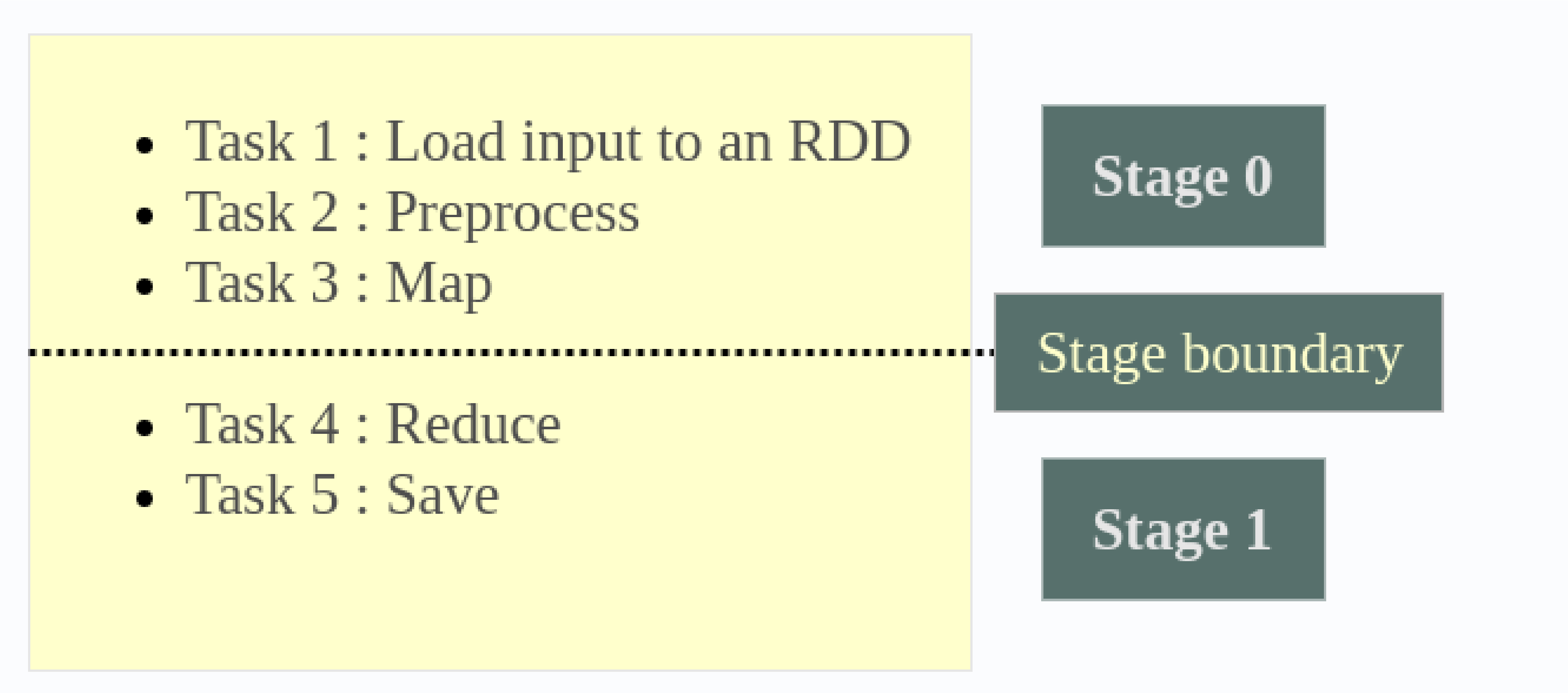
Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	saveAsTextFile at <console>:29	2017/11/12 14:59:20	1 s	2/2	4/4

Operations performed for program

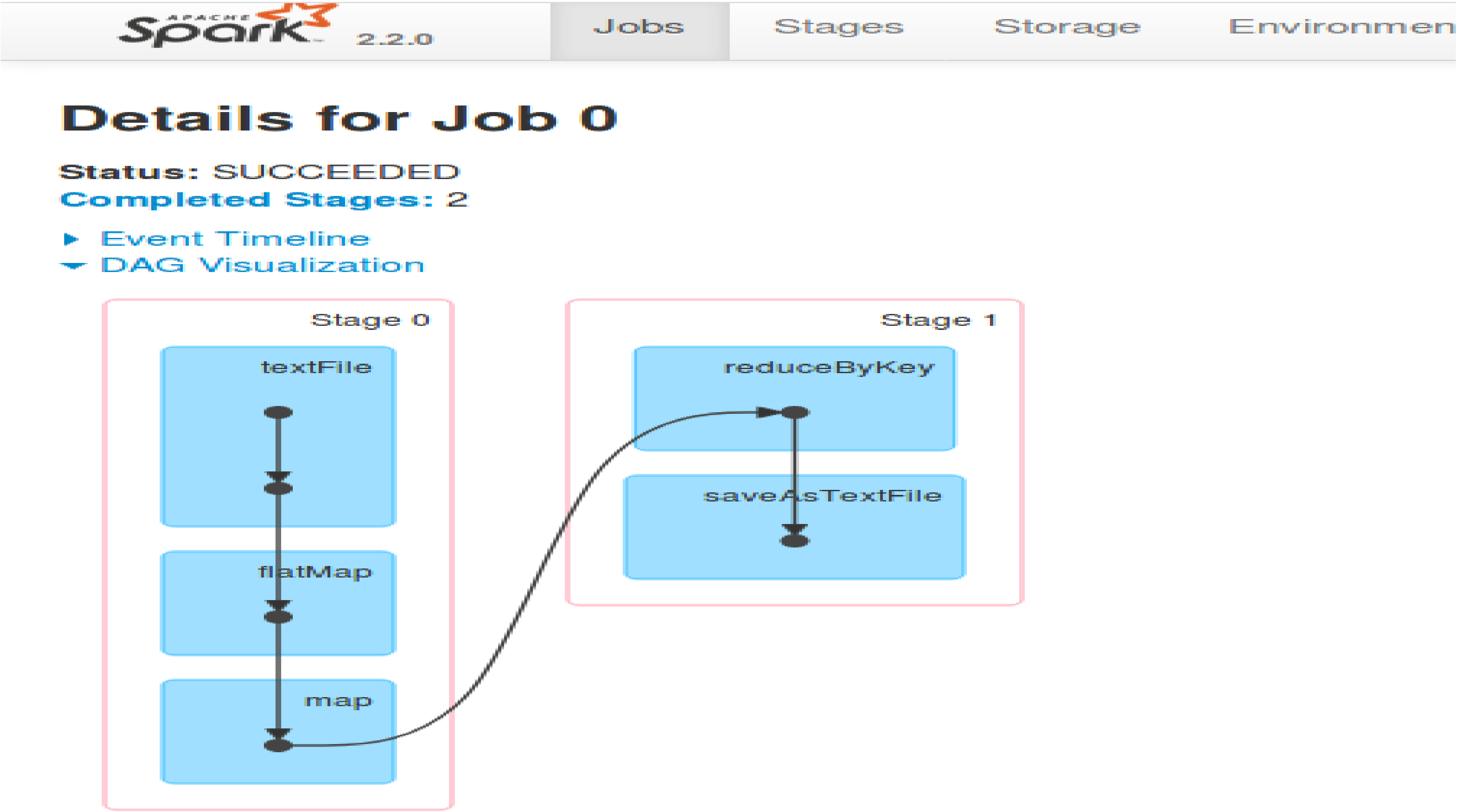
- Task 1 : Load input to an RDD
- Task 2 : Preprocess
- Task 3 : Map
- Task 4 : Reduce
- Task 5 : Save

Stages in DAG visualization



DAG Visualization in spark UI

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


Spark 2.2.0
Jobs
Stages
Storage
Environmen
Details for Job 0
Status: SUCCEEDED
Completed Stages: 2
Event Timeline
DAG Visualization
Stage 0
textFile
flatMap
map
Stage 1
reduceByKey
saveAsTextFile

Wordcount Program in Spark using scala

```
/** map */
var map = sc.textFile("/path/to/text/file").
                                            flatMap(line => line.split(" ")).
                                            map(word => (word,1));

/** reduce */
var counts = map.reduceByKey(_ + _);

/** save the output to file */
counts.saveAsTextFile("/path/to/output/")
```

Deploy mode of Spark

1. Local Mode
2. Standalone mode
3. YARN mode

Demonstration of apache spark wordcount program using all modes

Types of Transformation in RDD

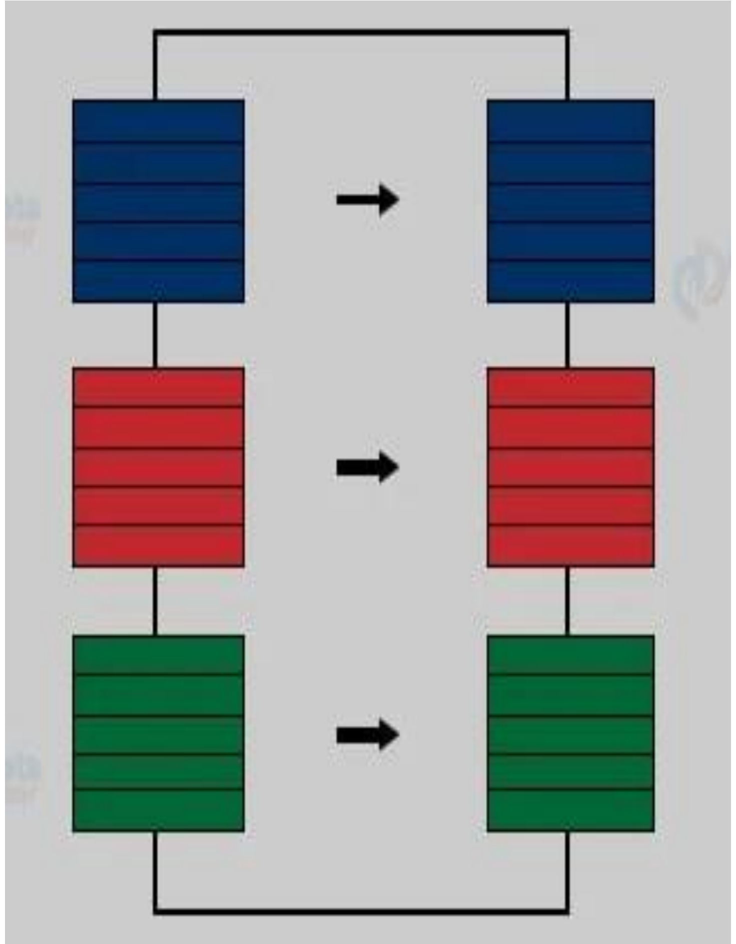
1. Narrow Transformation
2. Wide Transformation

Narrow Transformation(Pipelining)

- In *Narrow transformation*, all the elements that are required to compute the records in **single** partition live in the single partition of parent RDD.
- A limited subset of partition is used to calculate the result.
- No shuffling of data across the nodes in the cluster.

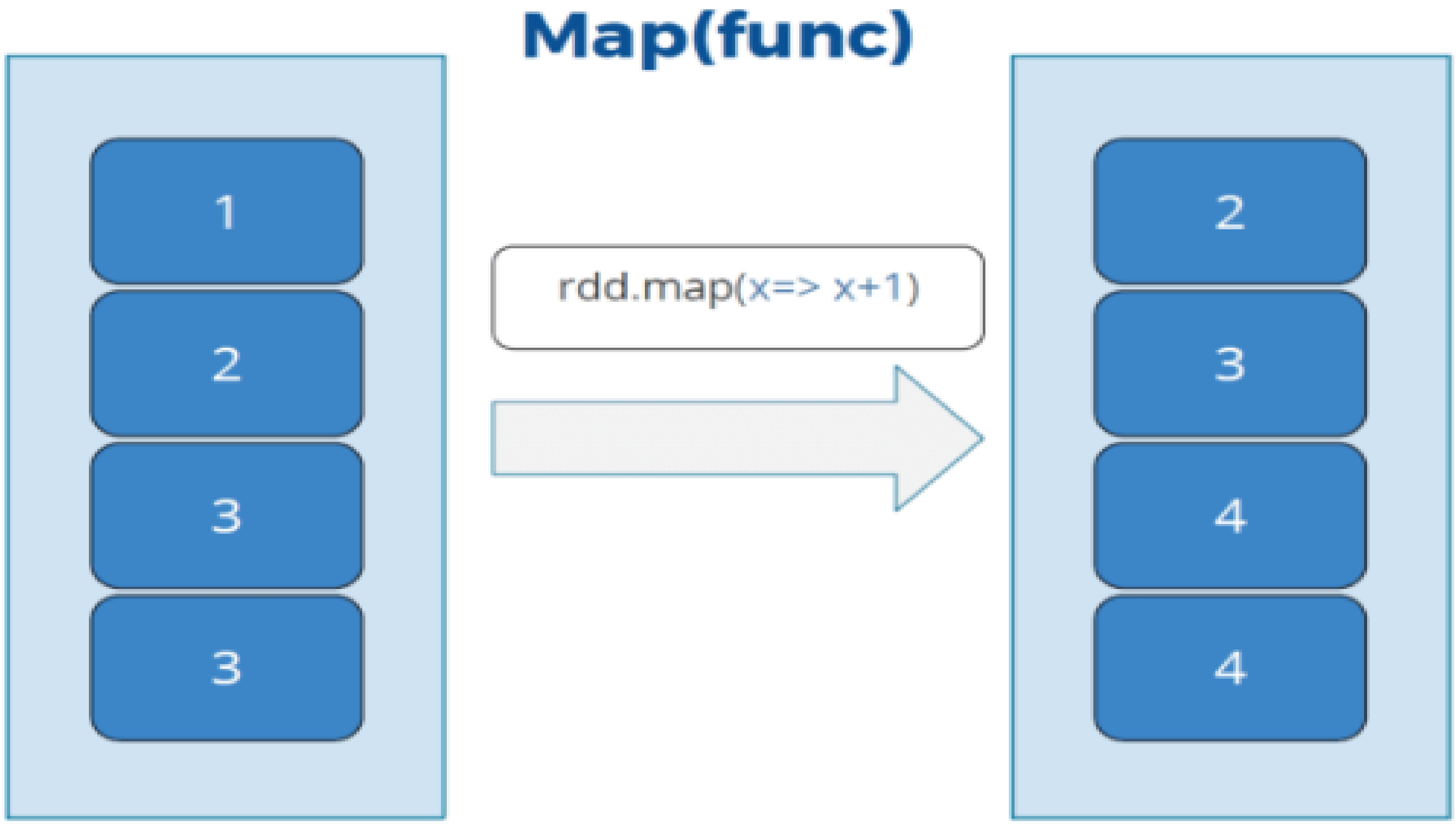
- **Example:**

Map
filter
FlatMap
sample
MapPartition
union



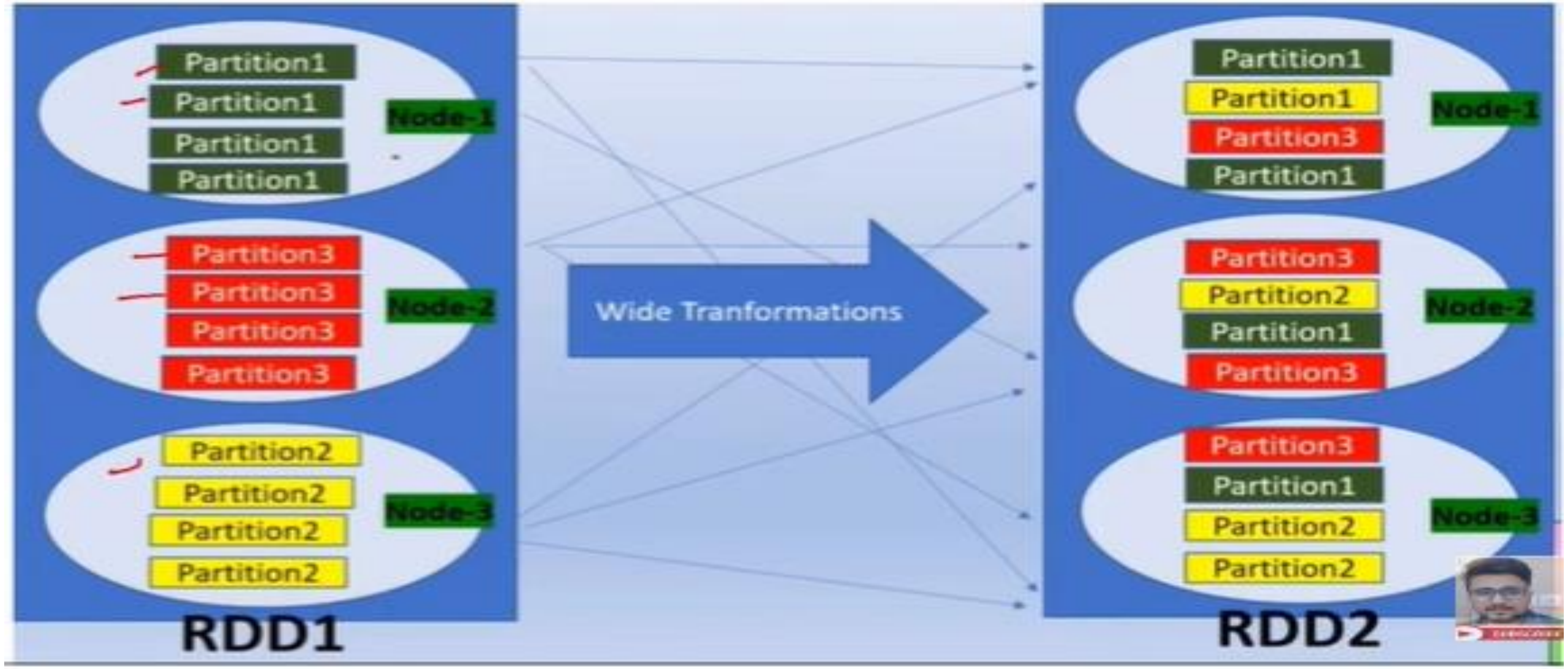
https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Example of Narrow Transformation(Map)



Wider Transformation(Shuffling)

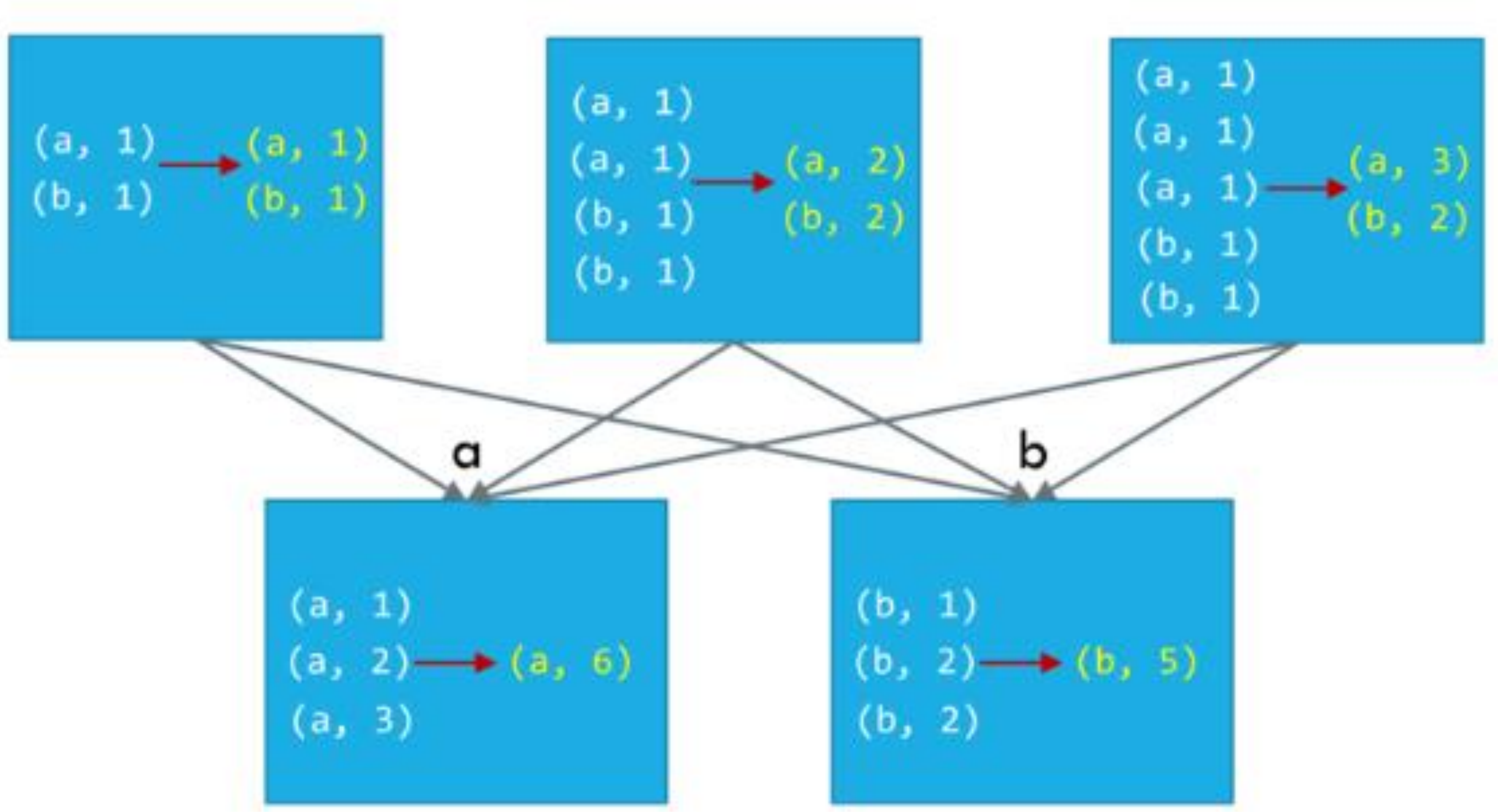
In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD.Wide transformations require data to be **"shuffled"** across partitions. They can involve data exchanges between partitions and over the network of the Spark cluster, which can be expensive in terms of both time and resources.



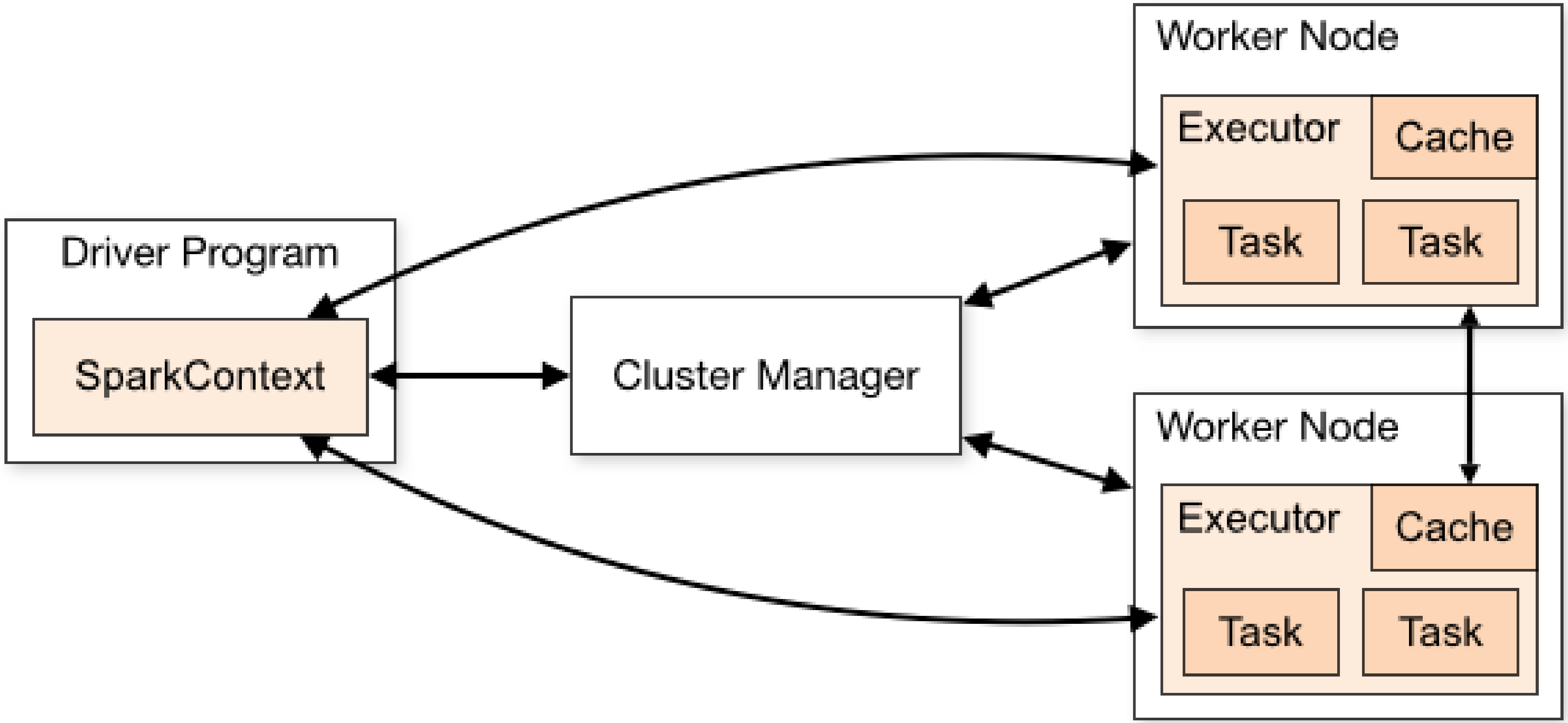
Example of Wider transformations

- ReduceByKey
- GroupByKey
- Join
- Cartesian
- Intersection
- Distinct
- Repartition
- coalesce

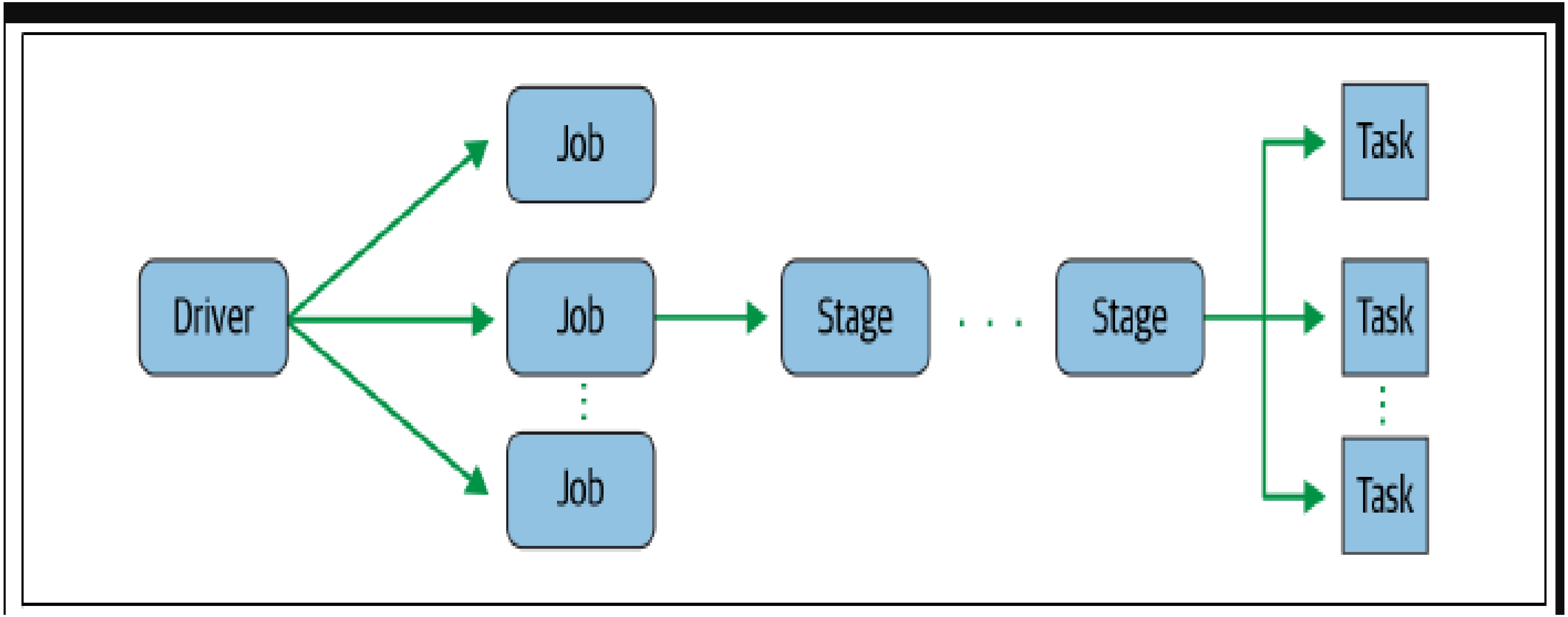
Example(ReduceByKey)



Spark Architecture



The Spark Application tree



The spark application tree

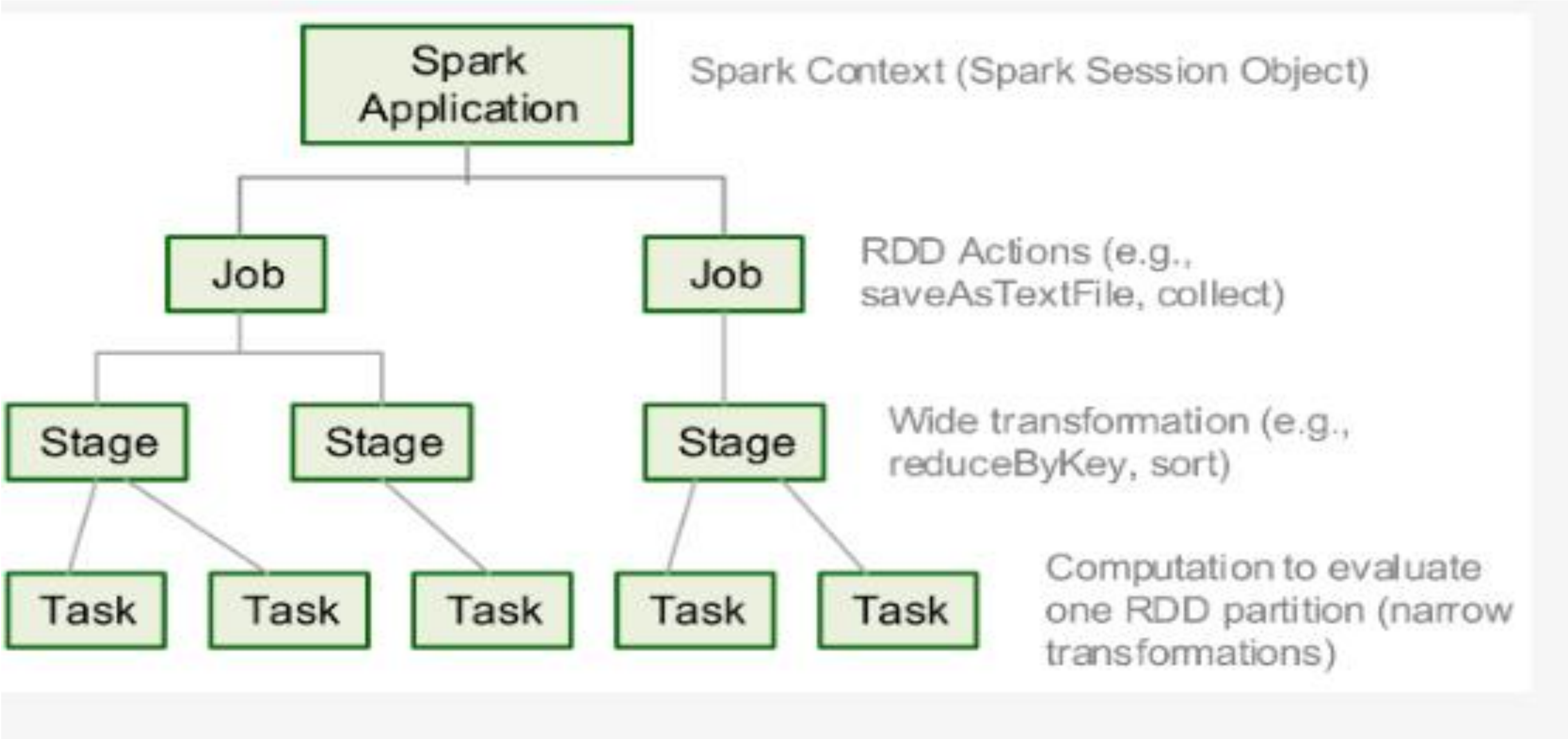
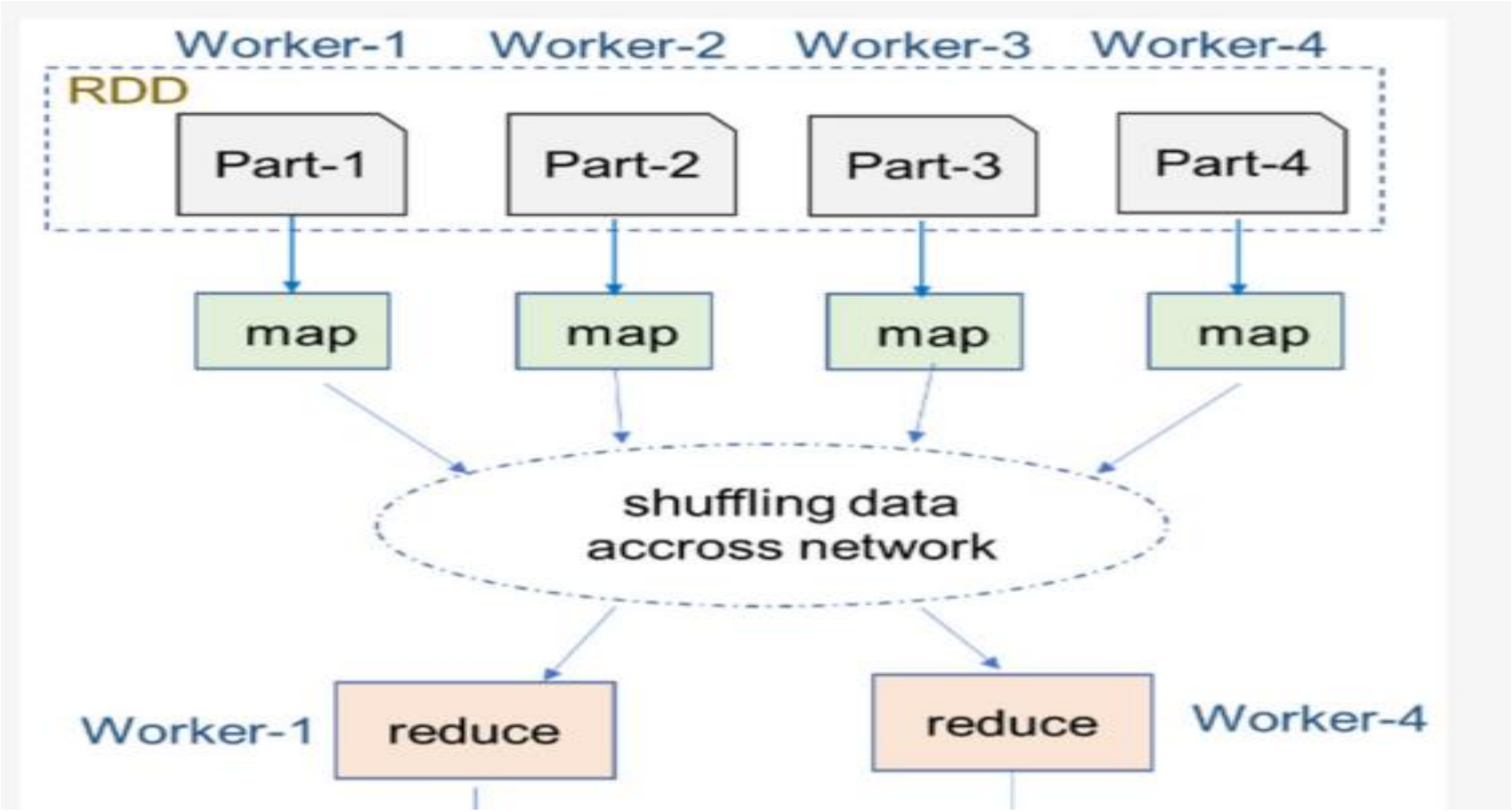


Illustration of shuffling phase for a stage



Terms related to Processing application in spark

Application - A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

Job - A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()). During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs. It then transforms each job into a DAG. This, in essence, is Spark’s execution plan, where each node within a DAG could be a single or multiple Spark stages.

Stage - Each job gets divided into smaller sets of tasks called stages that depend on each other. As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel. Not all Spark operations can happen in a single stage, so they may be divided into multiple stages. Often stages are delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors.

Task - A single unit of work or execution that will be sent to a Spark executor. Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data. As such, an executor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark's tasks exceedingly parallel!

Spark Memory Management

1. **--num-executors**

This argument only works on YARN only. The value indicates the ***number of executors to launch.*** By default, the value is 2. If dynamic allocation is enabled (spark.dynamicAllocation.enabled = true), this number will become the minimum initial number of executors.

2. **--executor-cores**

This argument only works on Spark standalone, YARN and Kubernetes only. The value indicates the ***number of cores*** used by each executor. The default is 1 in YARN

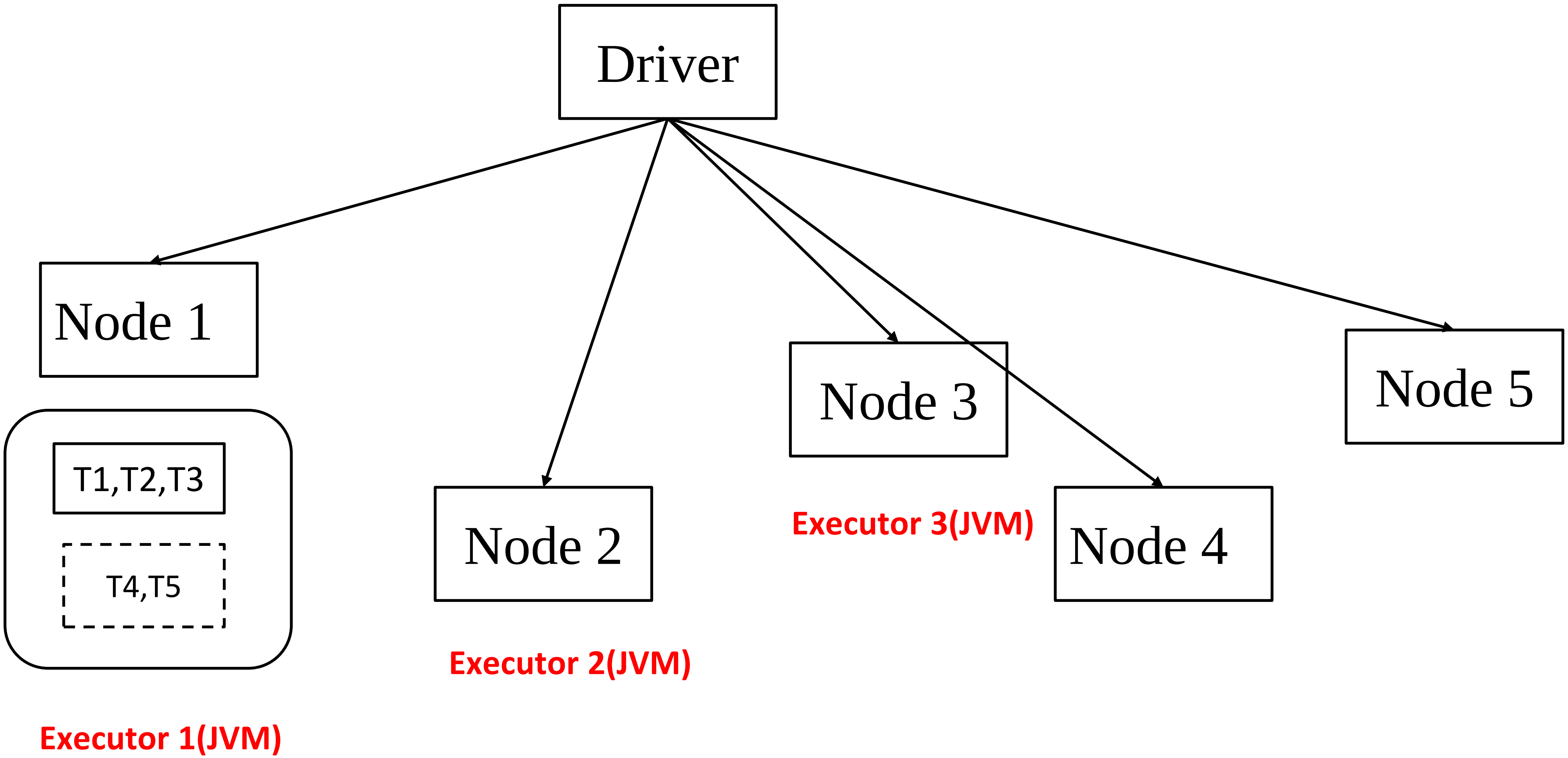
3. **--executor-memory**

This argument represents the ***memory per executor*** (e.g. 1000M, 2G, 3T). The default value is 1G.Executor memory is the amount of memory allocated to each executor process to store intermediate data, cached data, and execution buffers.

4. **–driver-memory**

The driver memory refers to the ***memory assigned to the driver process***, which is responsible for tasks such as creating execution plans, tracking data, and gathering results.

Spark Memory Management (Example)



Spark Memory Management

1. **--num-executors - 4**

2. **--executor-cores -3**

3. **--executor-memory - 1g**

4. **–driver-memory- 1g**

Distribution of Executors,Cores and memory for a spark application

spark-submit

--class <CLASS_NAME>

--num-executors ?

--executor-cores ?

--executor-memory ? ....

Example

Cluster Config:

10 Nodes

16 cores per Node

64GB RAM per Node

First Approach: Tiny executors [One Executor per core]:

- `--num-executors` = `In this approach, we'll assign one executor per core`
  = `total-cores-in-cluster`
  = `num-cores-per-node * total-nodes-in-cluster`
  = 16 x 10 = 160
- `--executor-cores` = 1 (one executor per core)
- `--executor-memory` = `amount of memory per executor`
  = `mem-per-node/num-executors-per-node`
  = 64GB/16 = 4GB

Second Approach: Fat executors (One Executor per node):

- `--num-executors` = `In this approach, we'll assign one executor per node`

  = `total-nodes-in-cluster`

  = 10

- `--executor-cores` = `one executor per node means all the cores of the node are assigned to one executor`

  = `total-cores-in-a-node`

  = 16

- `--executor-memory` = `amount of memory per executor`

  = `mem-per-node/num-executors-per-node`

  = 64GB/1 = 64GB

Third Approach: Balance between Fat (vs) Tiny

- **Let’s assign 5 core per executors** => --executor-cores = 5 (for good HDFS throughput)
- **Leave 1 core per node for Hadoop/Yarn daemons** => Num cores available per node = 16-1 = 15
- So, **Total available of cores in cluster** = 15 x 10 = 150
- **Number of available executors** = (total cores/num-cores-per-executor) = 150/5 = 30
- **Leaving 1 executor for ApplicationManager or Driver** => --num-executors = 29
- **Number of executors per node** = 30/10 = 3
- **Memory per executor** = 64GB/3 = 21GB
- **Counting off heap overhead** = 7% of 21GB = 3GB. So, actual
- **--executor-memory** = 21 - 3 = **18GB**

Size of Executor

- We cannot ask for **one executor** of **100 GB RAM and 100 cores**!
- Also, asking for **one executor** of **20 GB RAM and 8 cores i**s not good. It is similar to executing in single local machine.
- Appropriate size of executors should be demanded in code. Generally, this decision is taken after discussing with admin. E.g. **4 executors of 5 GB RAM and 2 cores each.**
- This would increase parallelism.

Phases of Massive Parallel Processing

1. Parallelism(Map)
2. Aggregation(Reduce)

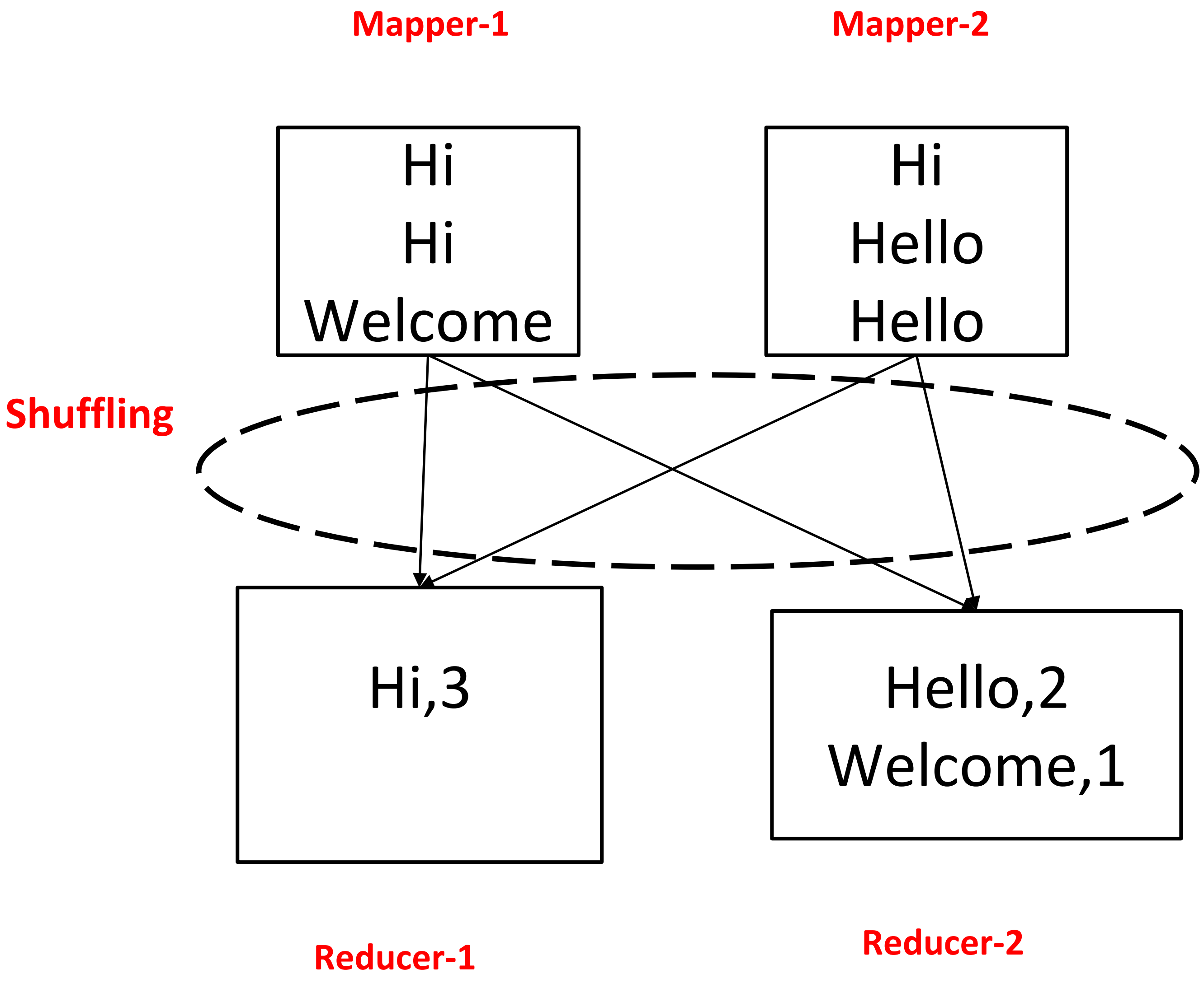
Input and output tasks

No of input task?

No of blocks= no of tasks

No of Output task ?

No of input task =no of output task



Partitioning

it means dividing the data into small parts and storing it in distributed systems for parallel computing.

Why Partitioning is required:-

Shuffling operations like repartition, groupBy, groupByKey, joins and many more needs transfer/shuffling of same sort of data in the same partitions to perform faster computation. These operations also need lots of I/O operations. Therefore, partitioning becomes imperative, when the data is key-value oriented. Since the range of keys or similar keys is in the same partition that minimizes shuffling. Hence, processing becomes substantially fast.As a result, by applying partitioning we can reduce the number of I/O operations rapidly. Thus, it speeds up the data processing. As spark works on data locality principle. So, partitioner tells which record goes to which partition.

Types of Partitioning

1. Hash partitioning
2. Range partitioning
3. Custom partitioning

Hash Partitioning

It spreads the data based on hash function. It means to spread the data evenly across various partitions, on the basis of a key. To determine the partition in Spark we use **Object.hashCode** method

partition = key.hashCode () % numPartitions.

GroupByKey ,ReduceByKey — by default this operation uses Hash Partitioning with default parameters.

rdd.getnumPartition(gives no of partitions)

Output: -2(default why ?)

Example of Hash Partitioning in wordcount program

Hash value of the key	mod(%)	No of output task	Output task no(reducer)
Hi(12171)	%	2	1
Hello(278)	%	2	0
Welcome(1088)	%	2	0

Range Partitioning

It spreads the data based on range. Example if we have id from 1 to 100, and wanted to store in 3 partitions then 1 to 33 will store in p -0 , 34 to 66 to p-1 and 67 to 100 to p-2 .

Note : — Partitioning is only possible in pair RDDs.

SortByKey *— uses Range partitioning to shuffle and store the data in partitions*

```
val pairRDD = data.map(x =>(x.key,x.value));

val partitionedRDD = pairRDD.partitionBy(new RangePartitioner(8,pairRDD));
```

Custom Partitioner

Sometimes we can see there are less number of partitions when we compare with specified number of partitions.

Wordcount Program with Custom Partitioning

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.{SparkConf, SparkContext}
object WC {
  def main(args : Array[String])  {
    var conf = new SparkConf().setAppName("Read Text File in Spark").setMaster("spark://celab3:7077")
    var map = sc.textFile(args(0)).flatMap(line => line.split(" ")).map(word => (word,1));
            var counts = map.reduceByKey(_ + _);
Val partitiondata=count.partitionBy(new Mycustompartitioner())
            counts.saveAsTextFile("file:///home/hadoop/Desktop/scalademo/output")
  }
}
```

```
class Mycustompartitioner(numParts:Int) extends partitioner
{
override def numPartitions=numParts;
override def getPartition(key : any):Int
{
        if(key.toString().equalsignoreCase("welcome"))
                return 0;
        else
                return 1;
}
}
```

Output

P0000:- welcome,1

p0001:- (hi,3),(hello,2)

RDD repartition()

Spark RDD repartition() method is used to **increase** or decrease the partitions. The below example decreases the partitions from 10 to 4 by moving data from all partitions.

```
val rdd2 = rdd1.repartition(4)

println("Repartition size : "+rdd2.partitions.size)

rdd2.saveAsTextFile("/tmp/re-partition")
```

output:-

Partition 1 : 1 6 10 15 19

Partition 2 : 2 3 7 11 16

Partition 3 : 4 8 12 13 17

Partition 4 : 0 5 9 14 18

RDD coalesce()

Spark RDD coalesce() is used o**nly to reduce the number of partitions**. This is optimized or improved version of repartition() where the movement of the data across the partitions is lower using coalesce.

```
val rdd2= rdd1.coalesce(3)
println("Repartition size : "+rdd3.partitions.size)
rdd3.saveAsTextFile("/tmp/coalesce")
```

Partition 1 : 1 6 10 15 19
Partition 3 : 4 8 12 13 172 3 7
Partition 4 : 0 5 9 14 1811 16

Difference between coalesce and repartition

- Repartition shuffles full data to reduce partitions, whereas coalesce is more intelligent and does less data movement.
- Repartition can also be used to increase number of partitions.
- Coalesce cannot be used to increase number of partitions.

MapReduce

- Mappers are always launched on nodes where data is available
- No. of mappers = no. of blocks

Spark

- When executors are launched, there is no guarantee that they will be launched on same machines where data is available as YARN does not know anything about data locality.
- Executors might be launched on same machines or different machines, so initially it might take some time to fetch data in memory, but even then it would be faster than MapReduce.
- Number of executors and its size has to be decided while writing spark program.

MapReduce Block

- One or more blocks will be processed by mappers.
- Size of each block is 128 MB.
- Blocks are stored on disk.

Spark Partition

- One or more partitions will be processed by executors.
- Each block becomes one partition in spark.
- While using other data storage, data has to be divided into partitions first. Normally, spark tries to set the number of partitions automatically based on cluster.
- More partitions results in more parallelism.
- Partitions are stored in RAM

Executor Memory Utilization

- If 10 GB executor is launched, we cannot use full capacity for data storage.
- **10 %** of memory is allocated to system calls. E.g. from 10 GB capacity, 1 GB is used for system calls.
- From remaining 90% of memory, we can utilize only 60% of memory.
- Remaining is used by garbage collector, JVM management and all.
- So, we can utilize only **54%** of full capacity for data storage. e.g., from 10 GB storage, we can utilize only 5.8 GB.
- Each executor needs at least one processor core. So, on dual core, only two executors can be launched.

Resource Allocation in spark

1. Static Resource Allocation
2. Dynamic Resource allocation

Static Resource Allocation

- In static resource allocation, the resources are **pre-allocated** to the Spark application before it starts running. The amount of resources is fixed and cannot be changed during runtime. It means that if the Spark application requires more resources than what was allocated, it will result in longer execution times or even failure of the job.
- Static resource allocation is suitable for scenarios where the resource requirements of the Spark application are known in advance and the workload is consistent throughout the job.
- **Disadvantages** :
    - Inefficient Resource Utilization
    - Limited Flexibility

https://community.cloudera.com/t5/Community-Articles/Dynamic-Allocation-in-Apache-Spark/ta-p/368095

Dynamic Resource Allocation

- **Dynamic allocation** is a feature in Apache Spark that allows for **automatic adjustment** of the **number of executors** allocated to an application. This feature is particularly useful for applications that have varying workloads and need to scale up or down depending on the amount of data being processed. It can help **optimize** the use of **cluster resources** and **improve application performance**.

- When dynamic allocation is enabled, Spark can dynamically allocate and deallocate executor nodes based on the application workload. If the workload increases, Spark can automatically allocate additional executor nodes to handle the additional load. Similarly, if the workload decreases, Spark can deallocate executor nodes to free up resources.

Advantages of Dynamic Allocation

- Resource efficiency
- Scalability
- Cost savings
- Fairness

Disadvantages of Dynamic Allocation

- Overhead
- Latency
- Configuration complexity
- Unpredictability
- Increased network traffic
- Spark Shuffle Service Overload

Spark Dynamic Memory Utilization

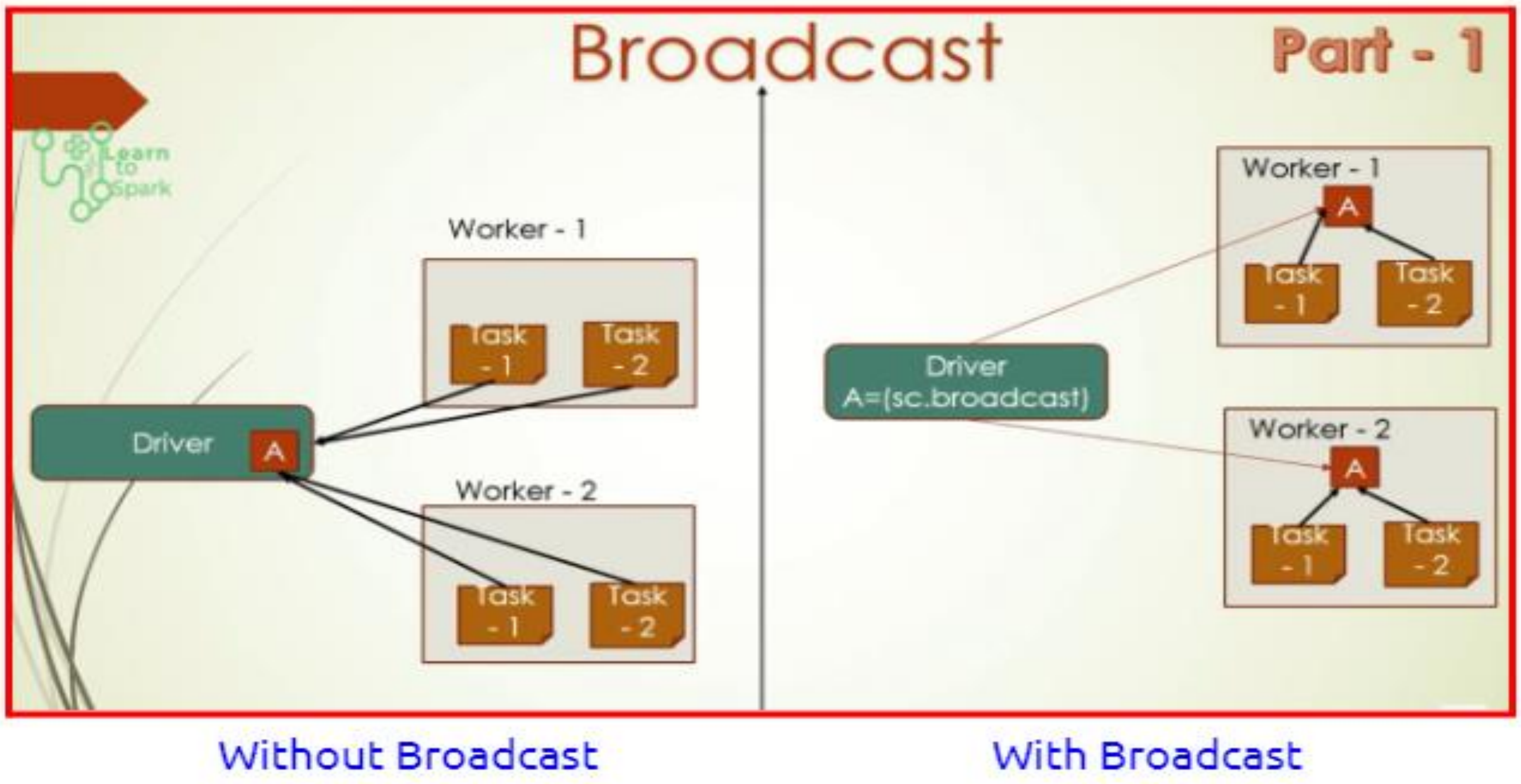
- If set to **true**, it will go for dynamic memory utilization. It means, if 8 executors are launched, after the work is finished, it will automatically kill idle executors.
- If set to **false**, it will not go for dynamic memory utilization. It means, if 8 executors are launched, after the work is finished, it will still keep them as it is and will not kill idle executors.

- Executors can be killed automatically after completing their jobs, but driver has to be stopped. Otherwise even after the job is completed, driver will still be there and eat resources.(e.g. default 1 core and all)

Configuration properties for Dynamic Resource allocation

Property Name	Default Value	Description
spark.shuffle.service.enabled	false	Enables the external shuffle service.
spark.dynamicAllocation.enabled	false	Set this to true to enable dynamic allocation.
spark.dynamicAllocation.minExecutors	0	Set this to the minimum number of executors that should be allocated to the application.
spark.dynamicAllocation.initialExecutors	spark.dynamicAllocation.minExecutors	The initial number of executors to run if dynamic allocation is enabled. If \`--num-executors\` (or \`spark.executor.instances\`) is set and larger than this value, it will be used as the initial number of executors.
spark.dynamicAllocation.maxExecutors	infinity	Set this to the maximum number of executors that should be allocated to the application.

Need of Shared variable in spark



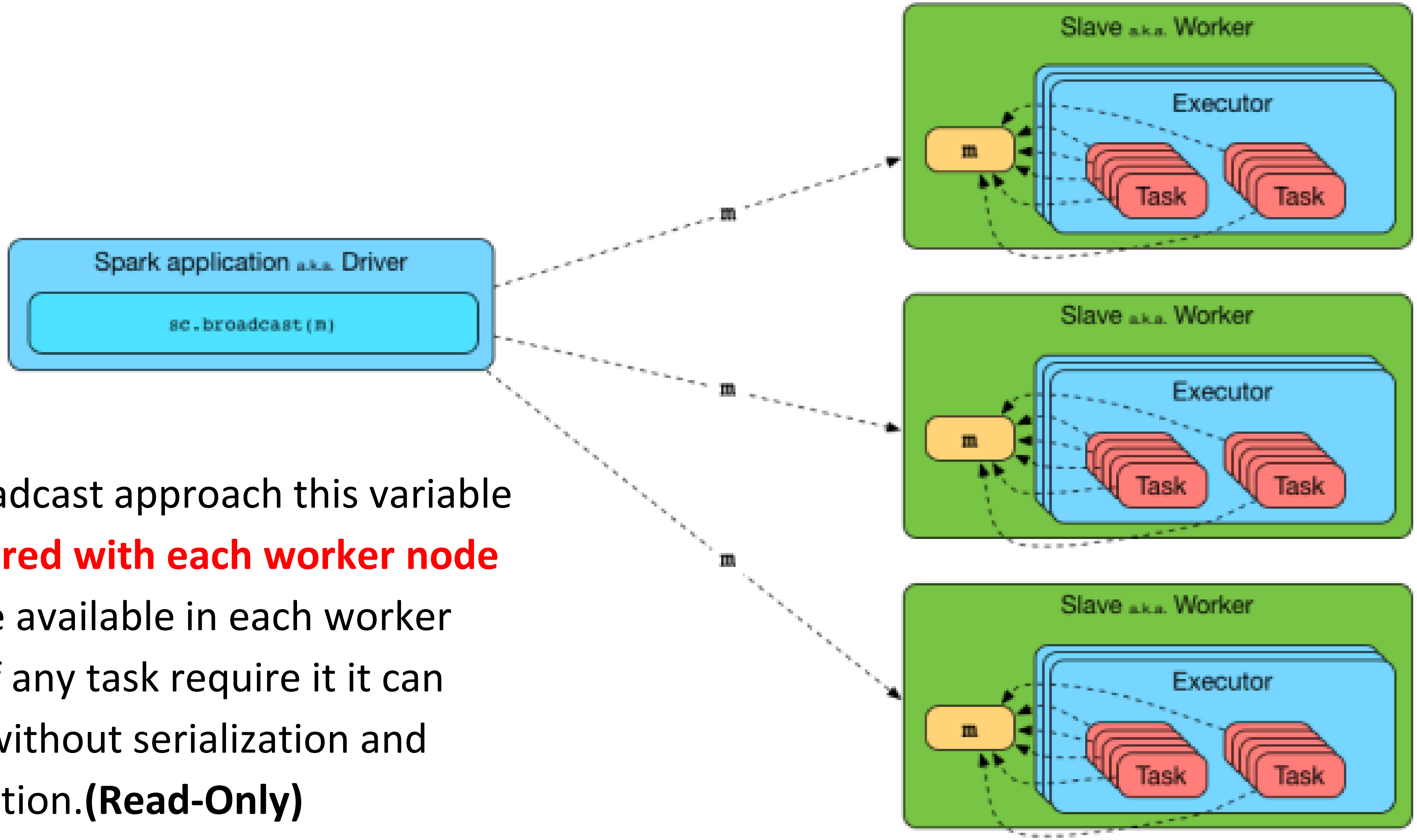
Spark Shared Variable

By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program.

Spark supports two types of shared variables:

1. ***broadcast variables*****:** which can be used to cache a value in memory on all nodes
2. ***accumulators*****:** which are variables that are only "added" to, such as counters and sums.

Broadcast variable



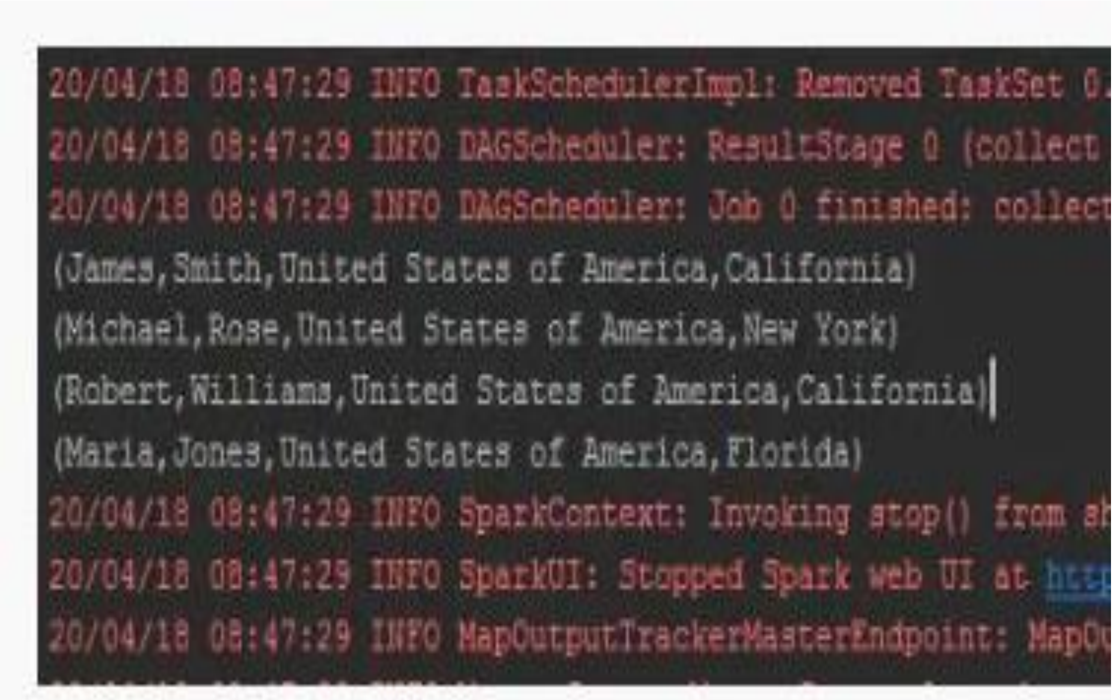
In the broadcast approach this variable will be **shared with each worker node** and will be available in each worker node. So if any task require it it can accessed without serialization and deserialization.**(Read-Only)**

USe case of broadcast variable

```
import
org.apache.spark.sql.SparkSession
object RDDBroadcast extends App {
  val spark = SparkSession.builder()
    appName("SparkByExamples.com")
    .master("local")
    .getOrCreate()
  val states = Map(("NY","New York"),("CA","California"),("FL","Florida"))
  val countries = Map(("USA","United States of America"),("IN","India"))
  val broadcastStates =sc.broadcast(states)
  val broadcastCountries = sc.broadcast(countries)
  val data = Seq(("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  )

val rdd = sc.parallelize(data)
```

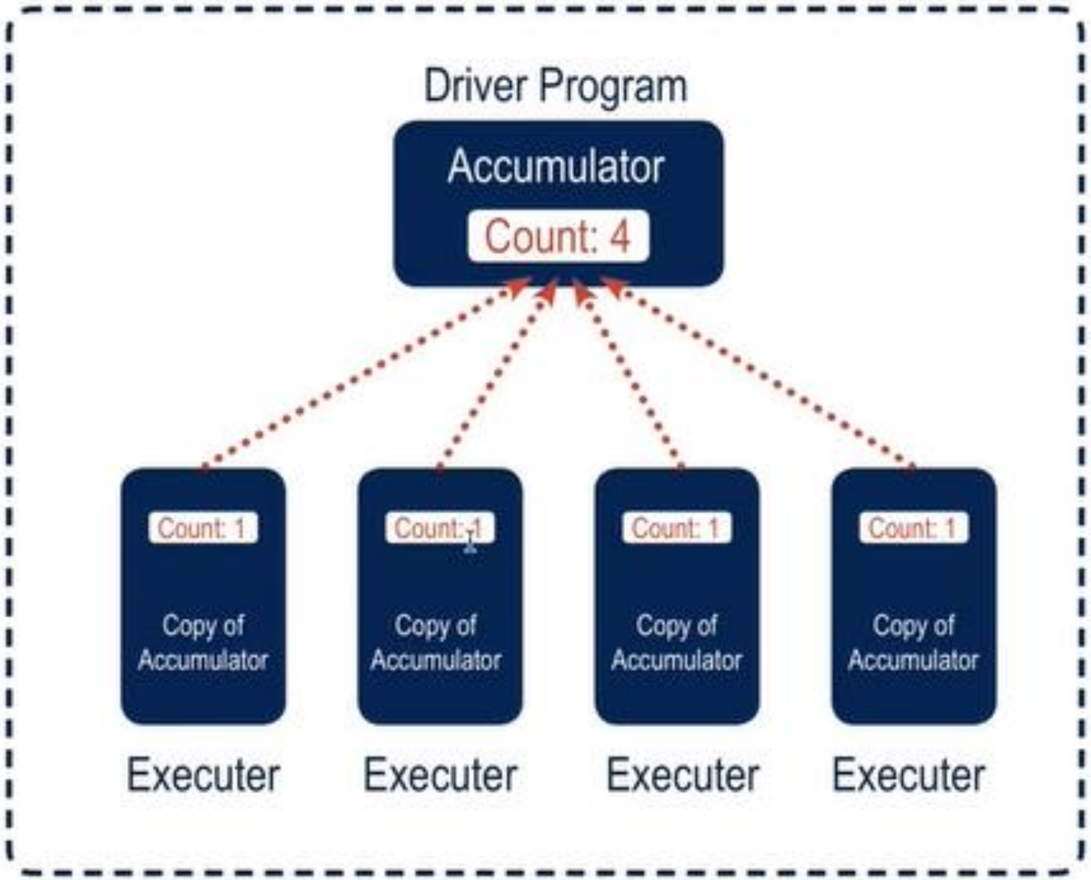
USe case of broadcast variable

```
val rdd2 = rdd.map(f=>{
    val country = f._3
    val state = f._4
    val fullCountry = broadcastCountries.value.get(country).get
    val fullState = broadcastStates.value.get(state).get
    (f._1,f._2,fullCountry,fullState)
  })

  println(rdd2.collect().mkString("\n"))

}
```



Accumulator

- Accumulators are the variables which are used to aggregate information from multiple executors.
- This is a shared variable which everyone wants to update.
- We can not read the value of accumulator, we can only add the value to the accumulator.



Use case(find the number of blank lines in the text file)

```
scala> sc.textFile("/Users/trendytech/samplefile.txt")
res8: org.apache.spark.rdd.RDD[String] = /Users/trendytech/samplefile.txt MapPartiti
onsRDD[9] at textFile at <console>:25

scala> val myrdd = sc.textFile("/Users/trendytech/samplefile.txt")
myrdd: org.apache.spark.rdd.RDD[String] = /Users/trendytech/samplefile.txt MapPartit
ionsRDD[11] at textFile at <console>:24

scala> val myaccum = sc.longAccumulator("blank lines accumulator")
myaccum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 79, name: Some(
blank lines accumulator), value: 0)

scala> myrdd.foreach(x => if (x=="") myaccum.add(1))

scala> myaccum.value
res10: Long = 8

scala>
```

Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Err
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

References

- https://spark.apache.org
- https://www.youtube.com/watch?v=9mELEARcxJo
- https://www.edureka.co/community/41392/what-are-the-spark-job-and-spark-task-and-spark-staging
- https://stackoverflow.com/questions/28973112/what-is-spark-job
- https://data-flair.training/blogs/learn-apache-spark-sparkcontext/
- https://www.mdpi.com/2504-2289/5/4/46
- https://community.cloudera.com/t5/Community-Articles/Dynamic-Allocation-in-Apache-Spark/ta-p/368095