# How Spark does it internally?

**Ramandeep Kaur**
**Software Consultant**
**Knoldus Inc.**

# Agenda

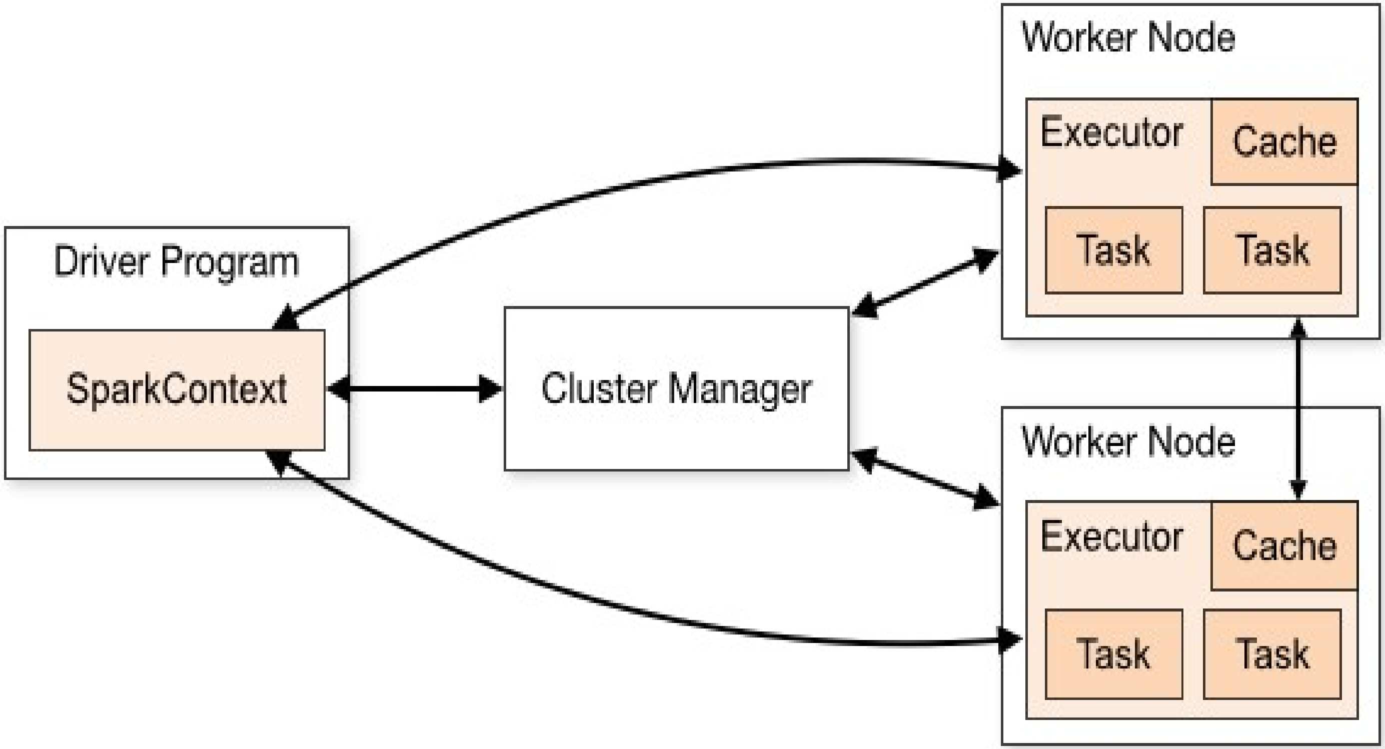
- Introduction to Apache Spark
- Architecture of Spark cluster
- Spark API: RDD
- Jobs, Stages & Tasks
- DAG
- DAGScheduler
- Shuffling in Spark
- Execution Workflow
- Demo

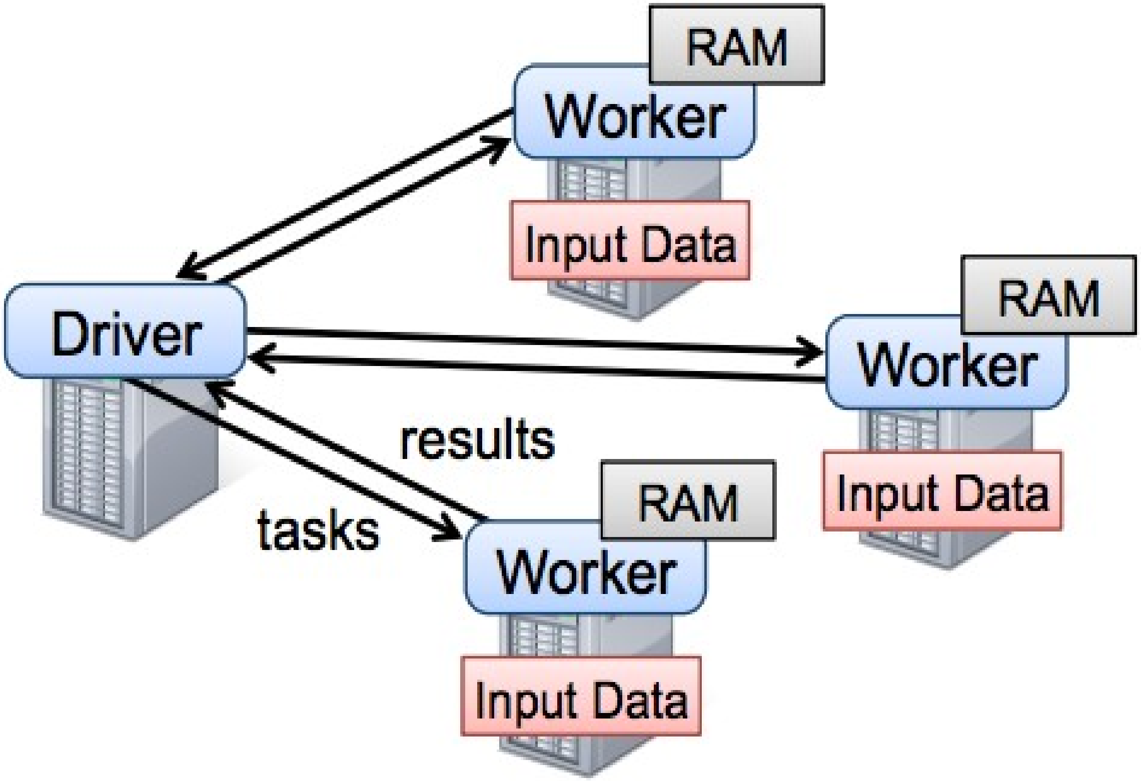
# Apache Spark

- Apache Spark is a fast and general-purpose cluster computing system.

- A lightning-fast cluster computing technology, which is faster than Hadoop MapReduce.

# Spark Architecture

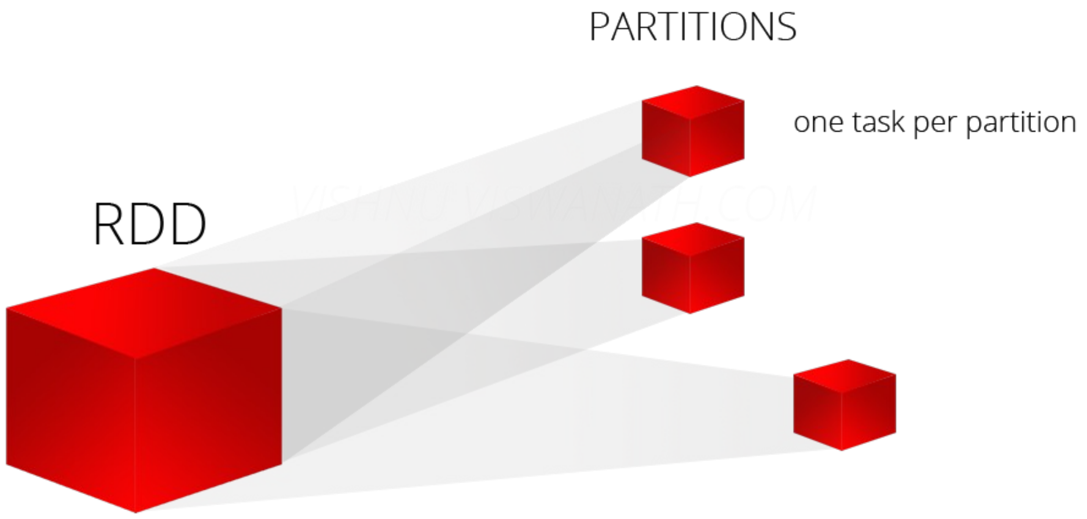



RAM
Worker
Input Data
RAM
Driver
Worker
Input Data
results
RAM
tasks
Worker
Input Data

# SPARK API: RDD

***"Fundamental data structure of Apache Spark"***

- Resilient
- Distributed
- Dataset
- Lazy evaluated
- Immutable



# Operations in Spark

- There are two types of operations:

~ **Transformations**:

- produces new RDD from the existing RDDs
- **lazily evaluated**
- takes RDD as input and produces one or more RDD as output.

~ **Actions**:

- RDD operations that give non-RDD values.
- way of sending data from Executer to the driver

## TRANSFORMATIONS

map(func)
flatMap(func)
filter()
mapPartitions
mapPartitionWithIndex
union(dataset)
intersection(other-dataset)
distinct()
groupByKey()
reduceByKey(func, [numTasks])
sortByKey()
join()
coalesce()

## ACTIONS

count()
collect()
take(n)
top()
countByValue()
reduce()
fold()
aggregate()
foreach()

# Jobs, Stages & Tasks

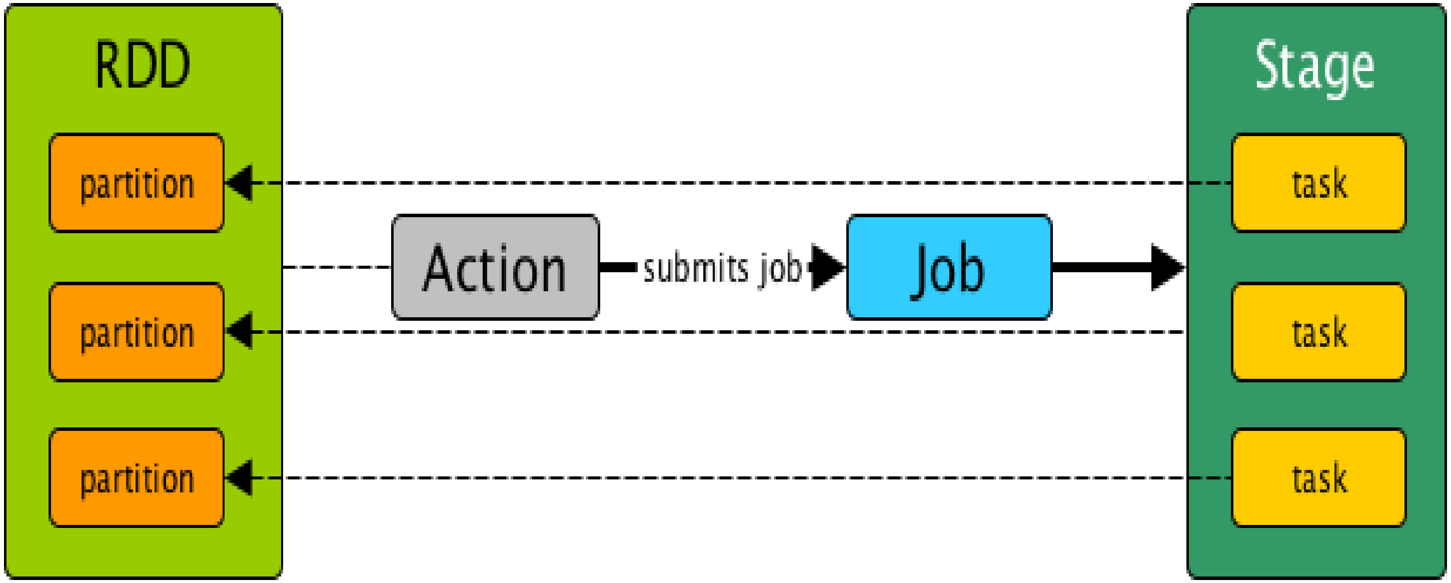
- **Jobs**
  A top-level work item (computation).
  When an action is called the processing gets started and a Job is created which is then submitted to DAGScheduler to be computed.

- **Stages**

  Sets of tasks that compute intermediate results in jobs, where each task computes the same function on partitions of the same RDD.

- **Tasks**

  A unit of work within a stage, corresponding to one RDD partition.


RDD
partition
partition
partition
Action
submits job
Job
Stage
task
task
task

# Transformations

- **Narrow Transformation:** doesn’t require data to be shuffled across the partitions.
  Eg. Map, filter etc.

- **Wide Transformation:** requires the data to be shuffled among the partitions.
  Eg. GroupByKey, reduceByKey etc.

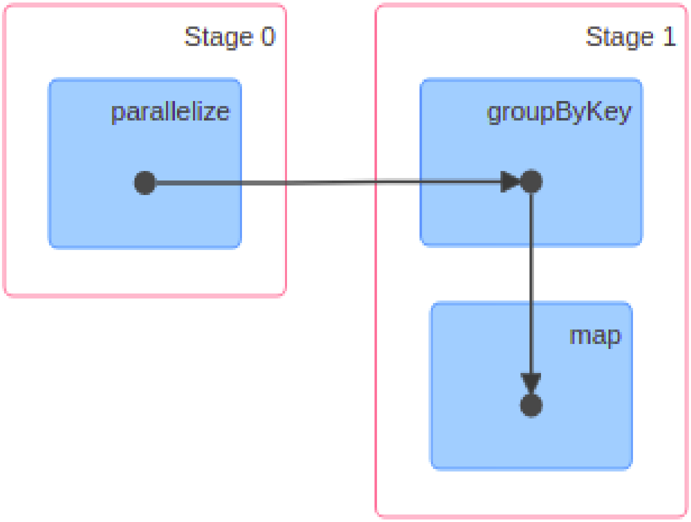
# DAG

- **Directed Acyclic Graph**

- Set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD.

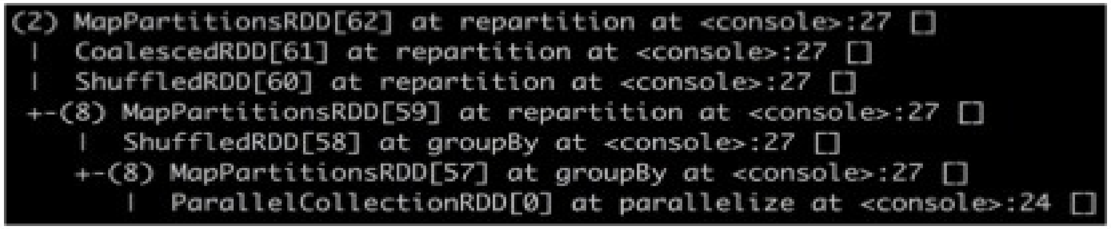
# DAG

- DAG is a graph denoting the sequence of operations that are being performed on the target RDD.

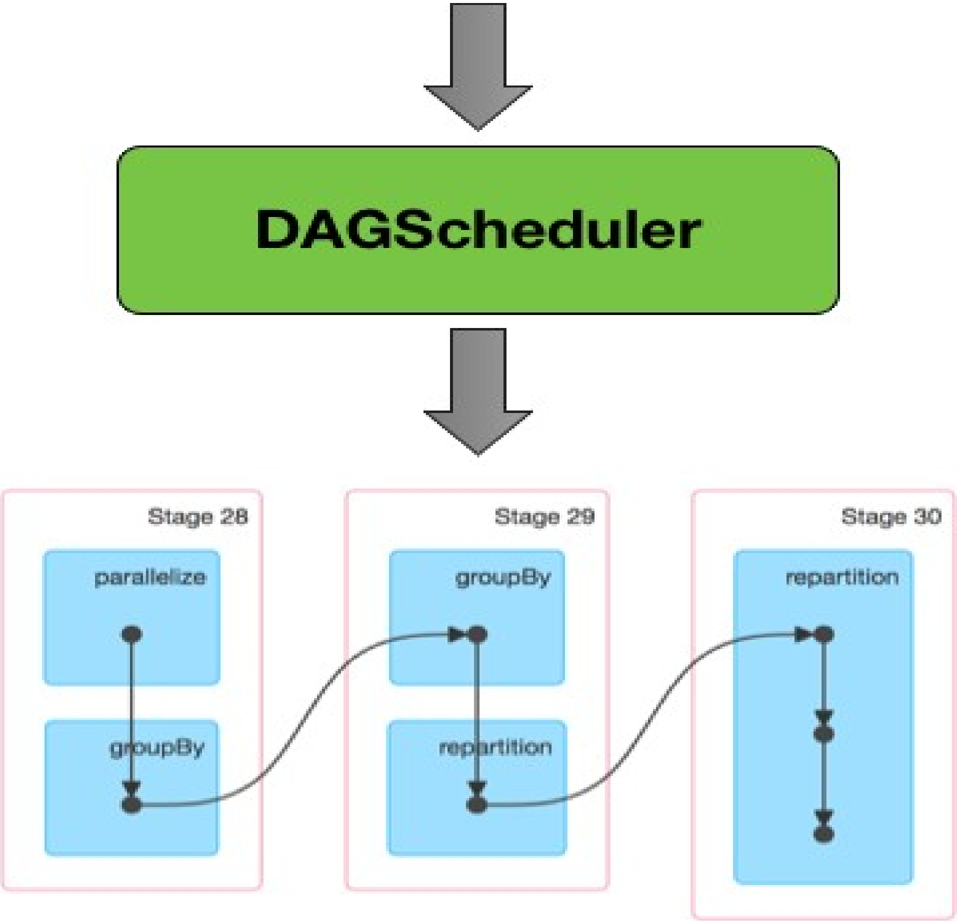


# DAG

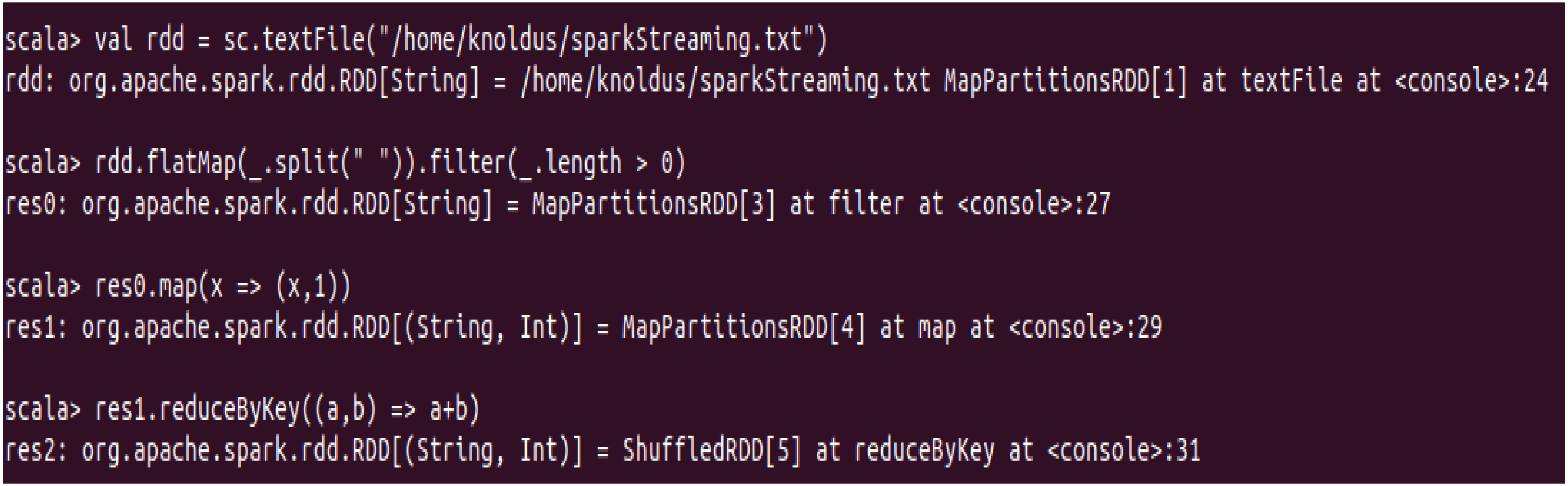
- After an action has been called, **SparkContext** hands over a logical plan to DAGScheduler that it in turn translates to a set of stages that are submitted as TaskSets for execution.



```
(2) MapPartitionsRDD[62] at repartition at <console>:27 □
 |  CoalescedRDD[61] at repartition at <console>:27 □
 |  ShuffledRDD[60] at repartition at <console>:27 □
 +-(8) MapPartitionsRDD[59] at repartition at <console>:27 □
    |  ShuffledRDD[58] at groupBy at <console>:27 □
    +-(8) MapPartitionsRDD[57] at groupBy at <console>:27 □
       |  ParallelCollectionRDD[0] at parallelize at <console>:24 □
```



DAGScheduler

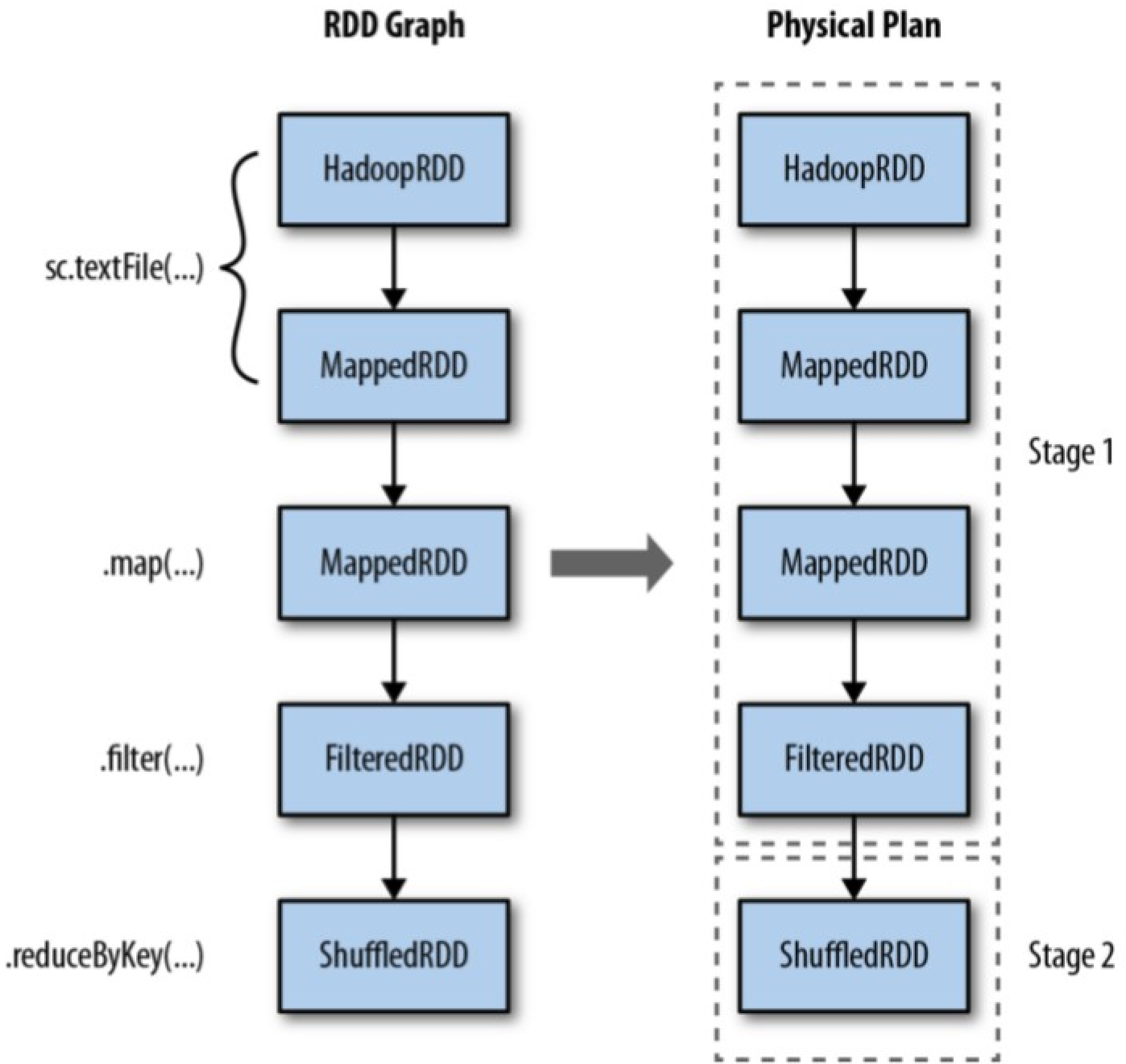


```
scala> val rdd = sc.textFile("/home/knoldus/sparkStreaming.txt")
rdd: org.apache.spark.rdd.RDD[String] = /home/knoldus/sparkStreaming.txt MapPartitionsRDD[1] at textFile at <console>:24

scala> rdd.flatMap(_.split(" ")).filter(_.length > 0)
res0: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at filter at <console>:27

scala> res0.map(x => (x,1))
res1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[4] at map at <console>:29

scala> res1.reduceByKey((a,b) => a+b)
res2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[5] at reduceByKey at <console>:31
```

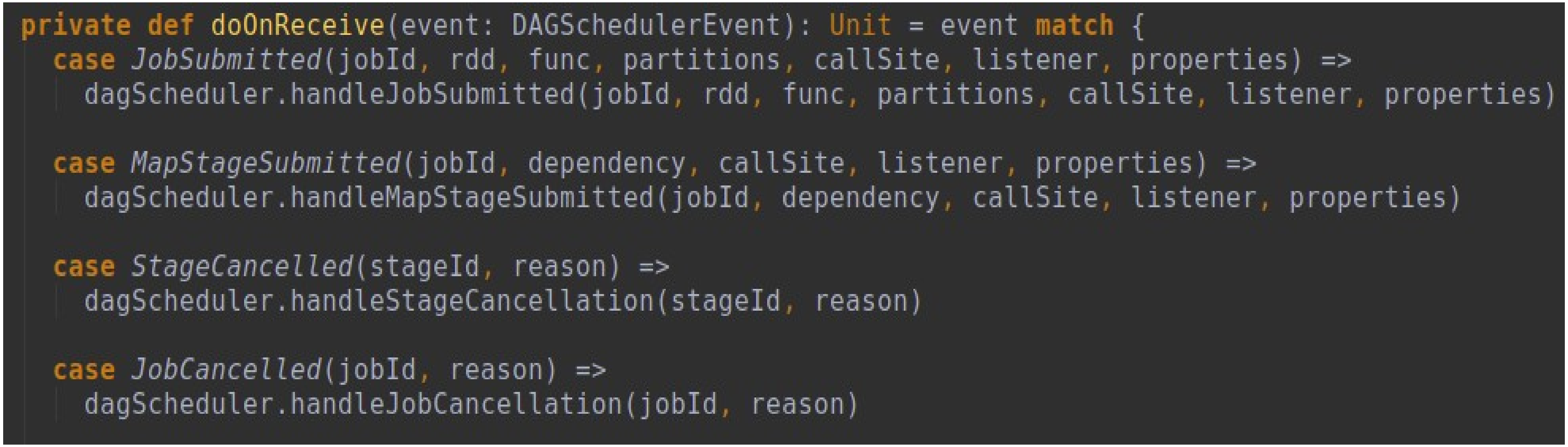

RDD Graph
Physical Plan
sc.textFile(...)
HadoopRDD
MappedRDD
.map(...)
MappedRDD
.filter(...)
FilteredRDD
.reduceByKey(...)
ShuffledRDD
HadoopRDD
MappedRDD
MappedRDD
FilteredRDD
Stage 1
ShuffledRDD
Stage 2

# DAGScheduler

- It computes a DAG of stages for each job, keeps track of which RDDs and stage outputs are completed, and finds a minimal schedule to run the job.

- It then submits stages as TaskSets to an underlying TaskScheduler implementation that runs them on the cluster.

- DAGScheduler uses an event queue architecture in which a thread can post **DAGSchedulerEvent** events, e.g. a new job or stage being submitted, that DAGScheduler reads and executes sequentially.



```
private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {
  case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>
    dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)

  case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>
    dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)

  case StageCancelled(stageId, reason) =>
    dagScheduler.handleStageCancellation(stageId, reason)

  case JobCancelled(jobId, reason) =>
    dagScheduler.handleJobCancellation(jobId, reason)
```

# STAGES

There are two types of Stages:

- **ResultStage**

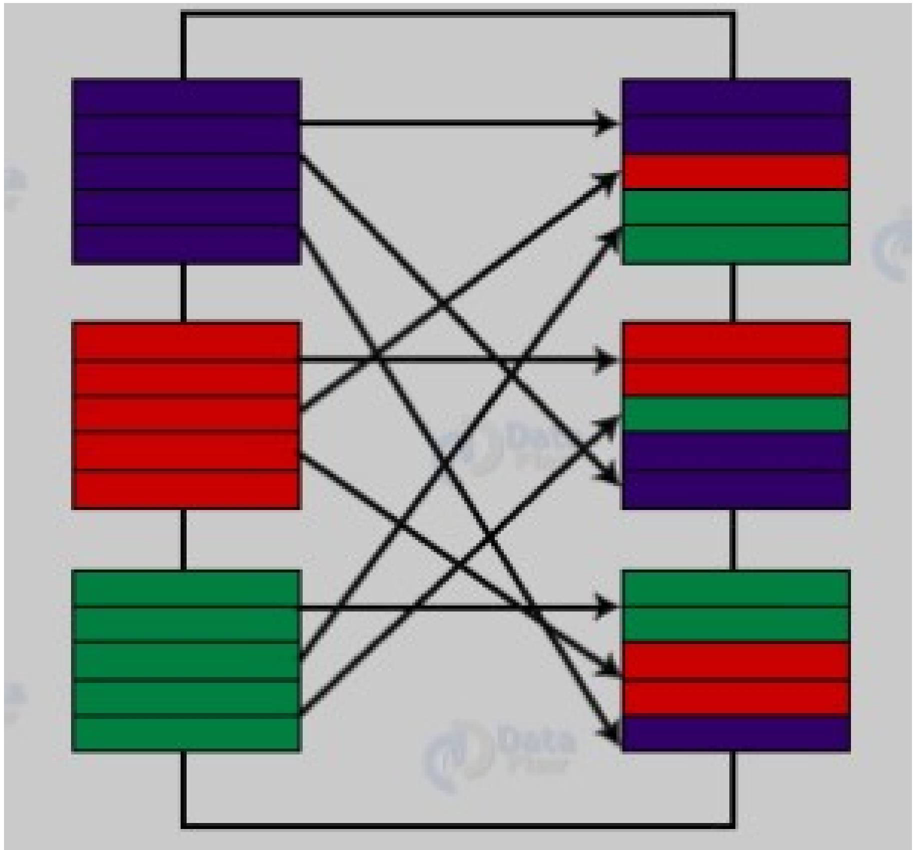
  The final stage in a job that applies a function to one or many partitions of the target RDD to compute the result of an action.

- **ShuffleMapStage**

  An intermediate stage in the physical execution DAG that corresponds to a ShuffleDependency.

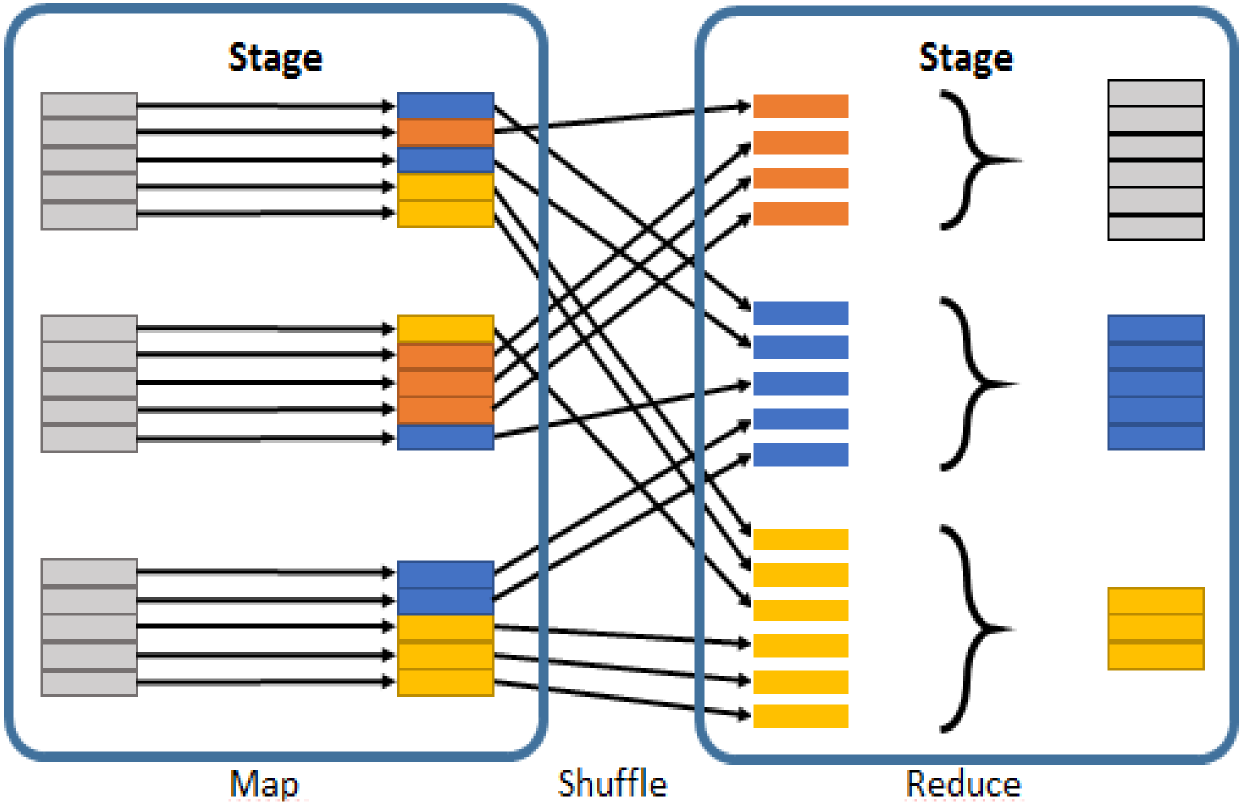
  A ShuffleMapStage may contain multiple pipelined operations, e.g. map and filter, before shuffle operation.

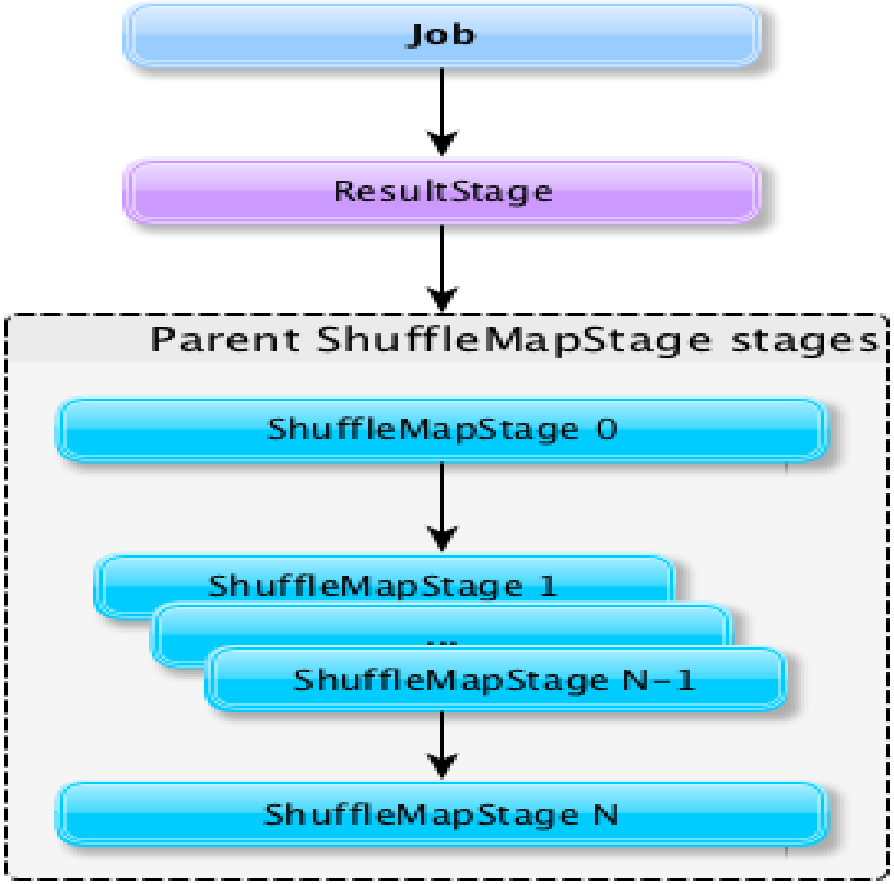
# Shuffle Operation



***Process of redistribution of data across partitions***

# Shuffle Map Stage?





There will ALWAYS be only **ONE** ResultStage and can be multiple ShuffleMap stages

# Why multiple Stages?

***"FAULT TOLERANCE"***

- After every shuffle operation, a new stage is created so that whenever data is lost due to shuffle(network I/O) only the previous stage will be calculated for fault tolerance.

# Fault Tolerance: HOW?

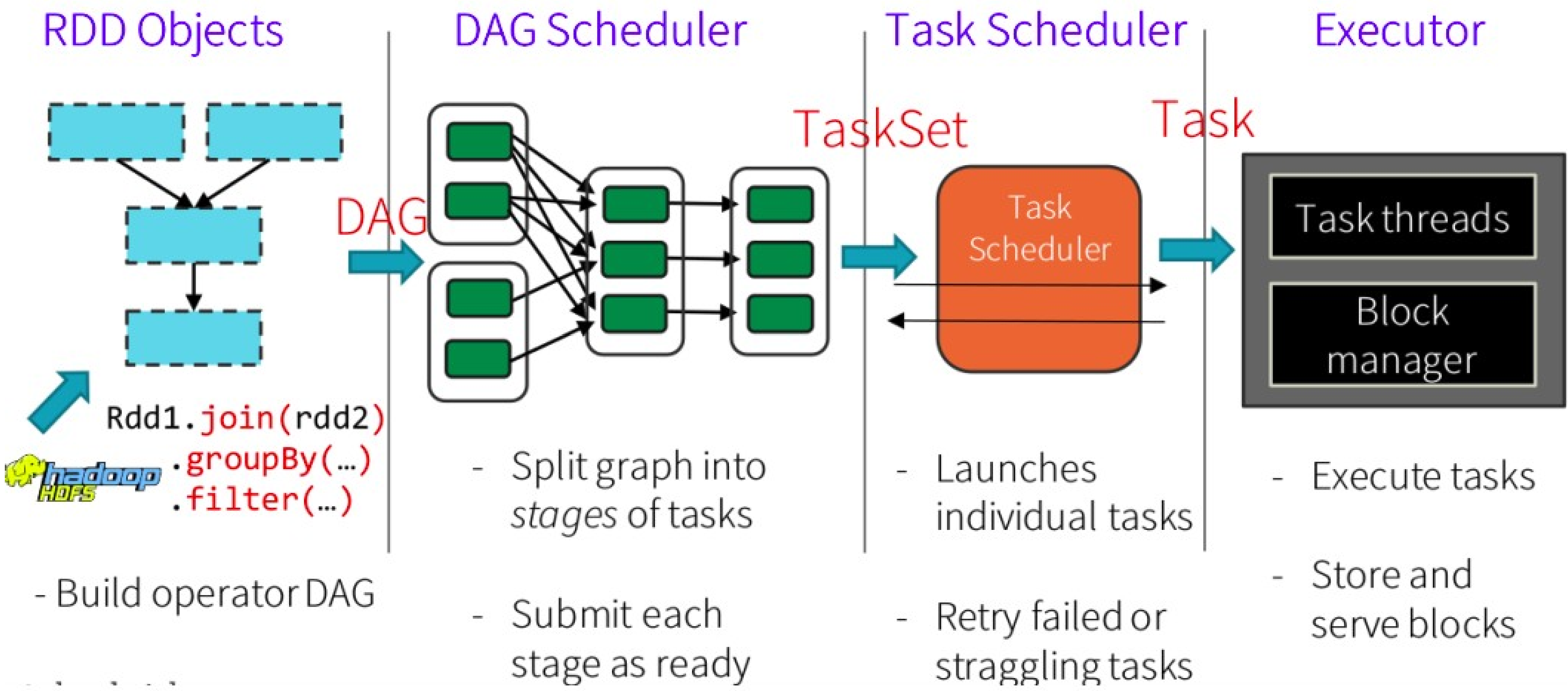
- To recover from failures, the same stage might need to run multiple times, which are called "attempts".

- If the TaskScheduler reports that a task failed because a map output file from a previous stage was lost, the DAGScheduler resubmits that lost stage.

- This is detected through a **CompletionEvent** with **FetchFailed**, or an **ExecutorLost** event.

- The DAGScheduler will wait a small amount of time to see whether other nodes or tasks fail, then resubmit TaskSets for any lost stage(s) that compute the missing tasks.

# Execution Workflow



- Build operator DAG

- Split graph into *stages* of tasks
- Submit each stage as ready

- Launches individual tasks
- Retry failed or straggling tasks

- Execute tasks
- Store and serve blocks

# References

- https://github.com/apache/spark/blob/master/core/s

- https://jaceklaskowski.gitbooks.io/mastering-apach

# Thank you