# Spark Performance Tuning

By: Shakti Garg and Nisha Kumari

- general purpose execution engine
  - batch/ iterative/ interactive jobs
- fast, in-memory processing
- distributed/ scalable
- fault-tolerant
- rich, optimized API support

# What are we covering?

## Coding Optimization

1. Data structures
2. Data format and compression
3. API optimization
4. Serialization

## Cluster Optimization

1. Resource Allocation (executors/memory)
2. Dynamic Allocation
3. Tuning parallelism
4. Spark speculative execution
5. YARN optimization

# 1. Slim down data structures!

- Why?
- How?
  - Numeric or Enum keys over Strings
  - Java fastUtil over Java Collections
  - Avoid wrapper objects, nested data structures, pointers etc
  - JVM flag XX:+UseCompressedOops to set pointers of 4 byte size than 8

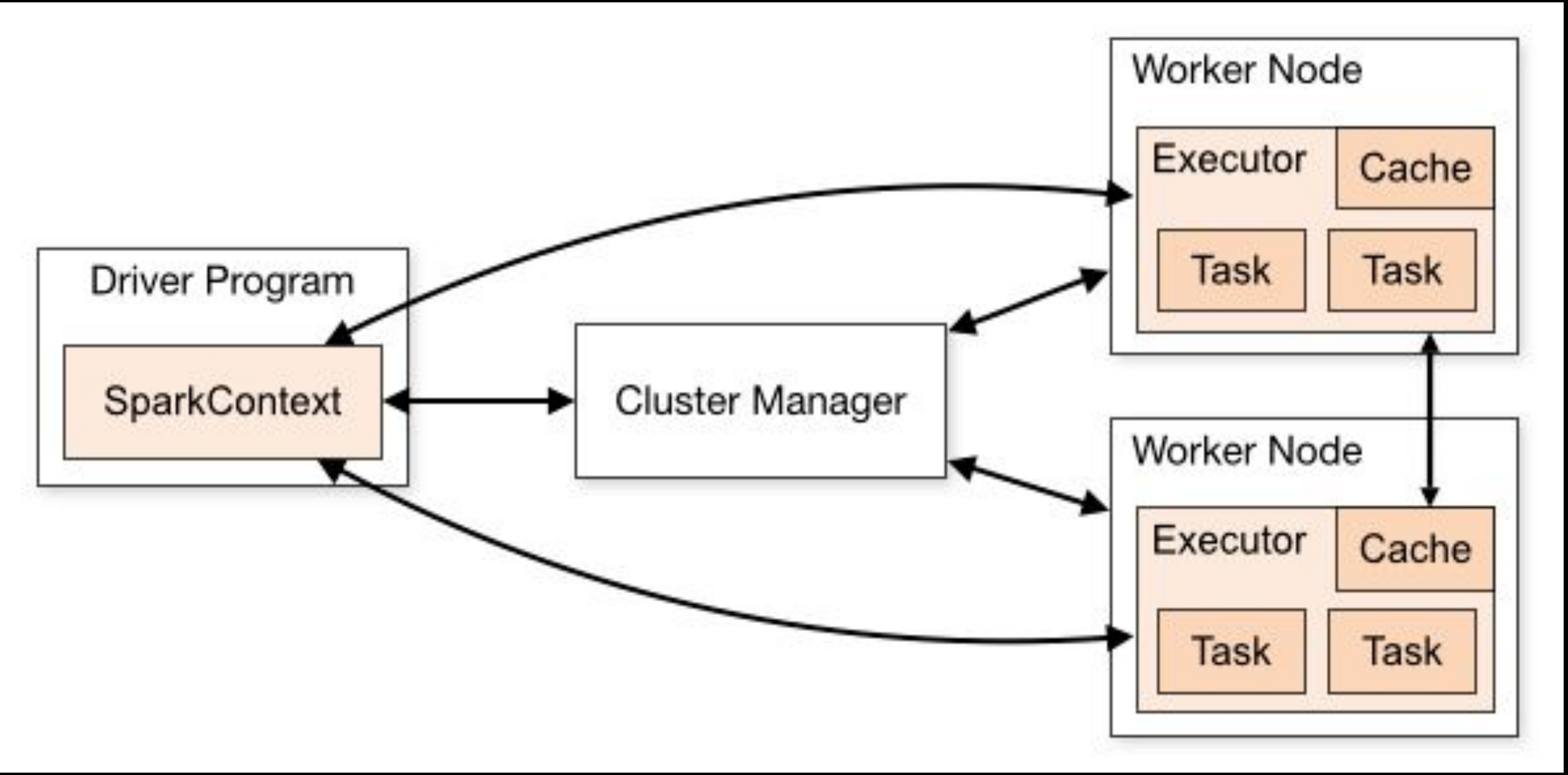
# 2. Data format and compression

- Data Serialization
  - Avro
  - Protobuf
- Data RC formats
  - Parquet
  - ORC
- Data Compression (In Rest & In Transit)

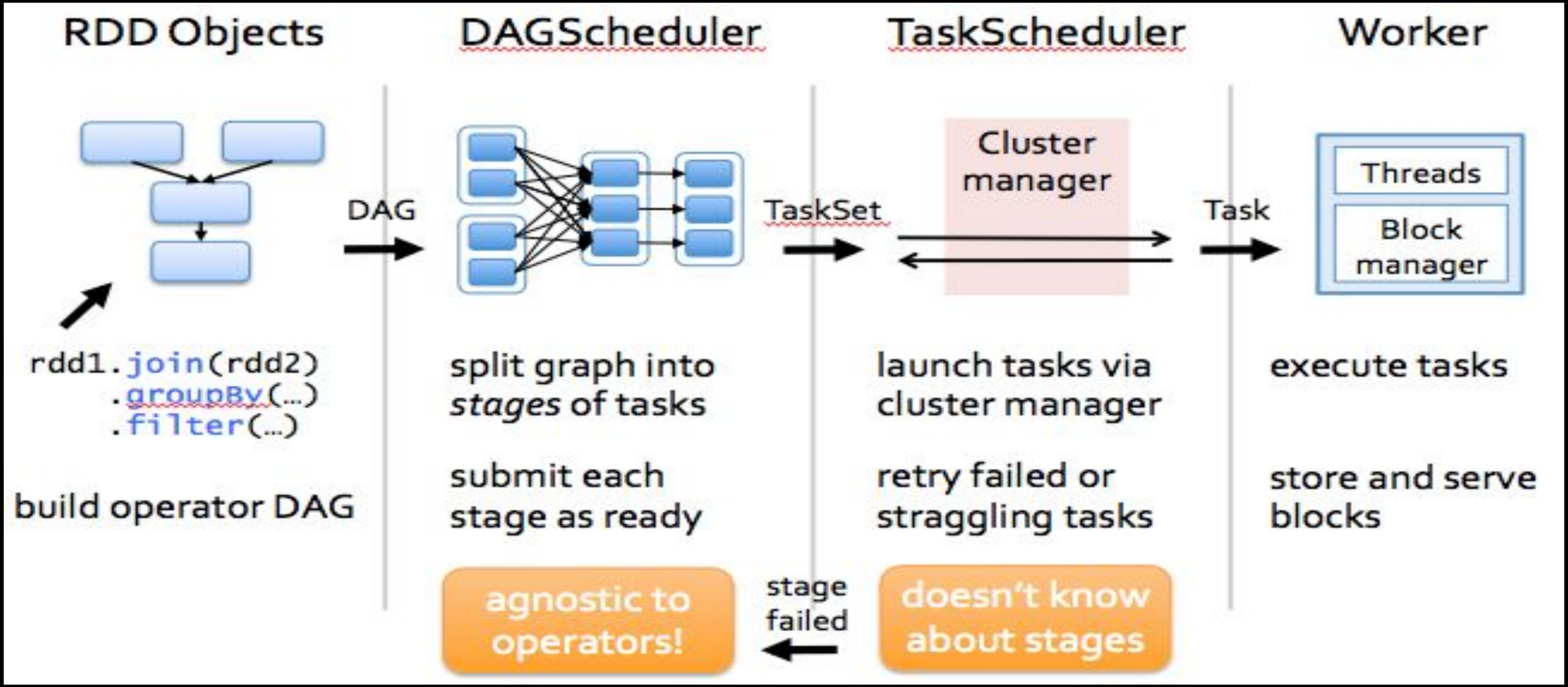
```
--conf spark.hadoop.mapred.output.compress=true
--conf spark.hadoop.mapred.output.compression.codec=snappy
```

```
--conf spark.io.compression.codec=snappy
```

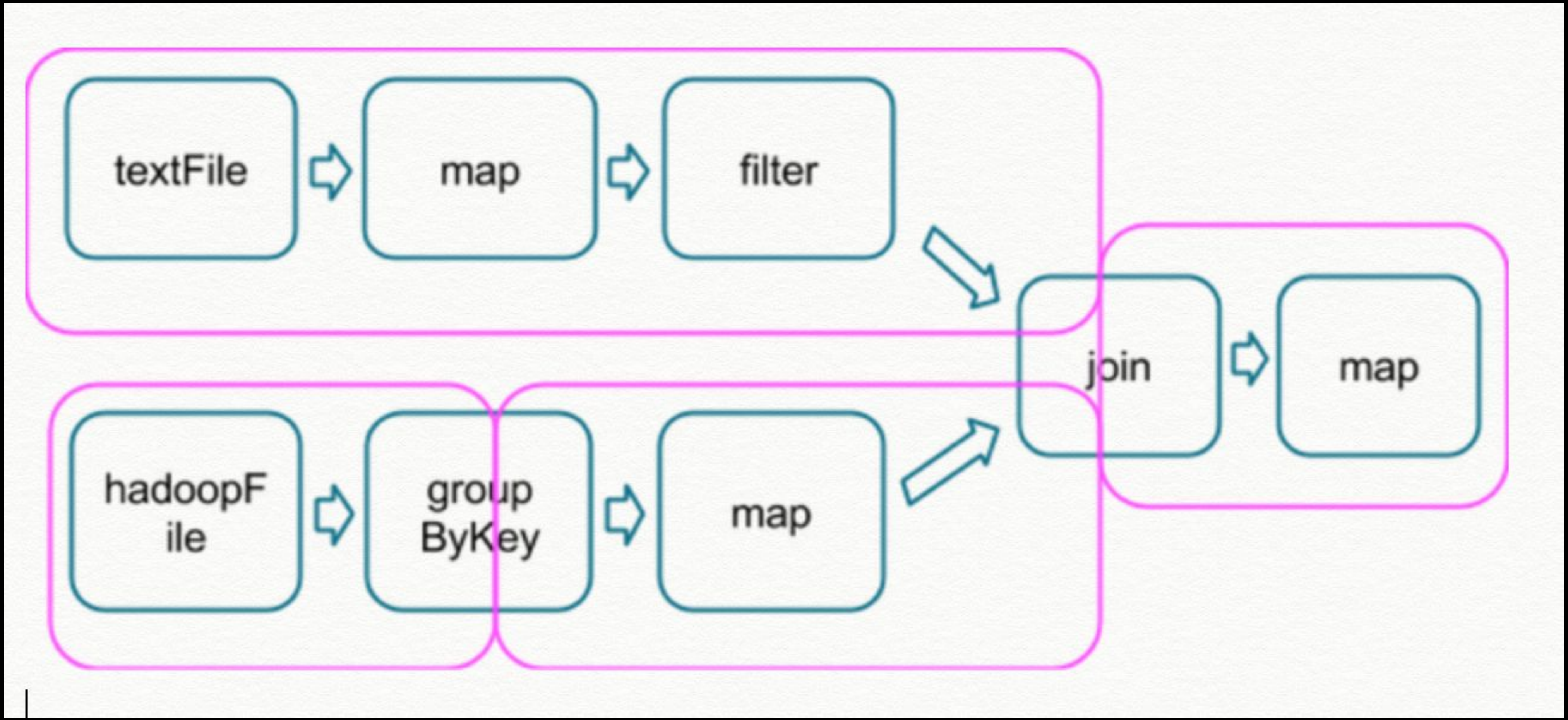
# Spark at work: Overview



# Spark at work: Detailed



# Execution: Job-Stages-Tasks



# 3. API Optimization

*"Understand execution plan of your Spark application!"*

- Prefer higher level Spark abstractions, DataFrame/Dataset over RDDs
- Still we need RDD's flexibility!
  - Don't collect large RDDs

```
rdd.take   /     rdd.takeSample
```

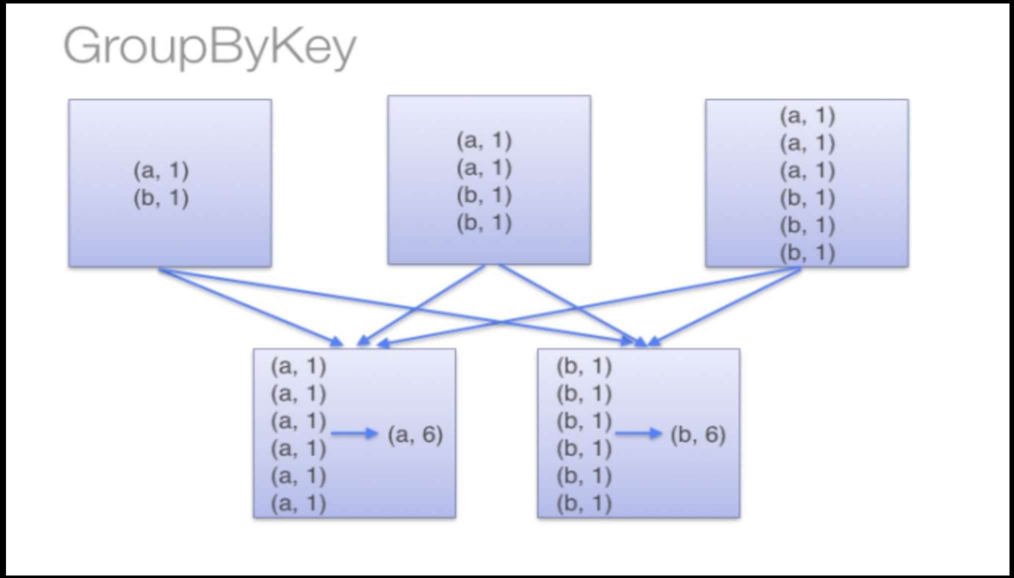
  - Avoid using count()

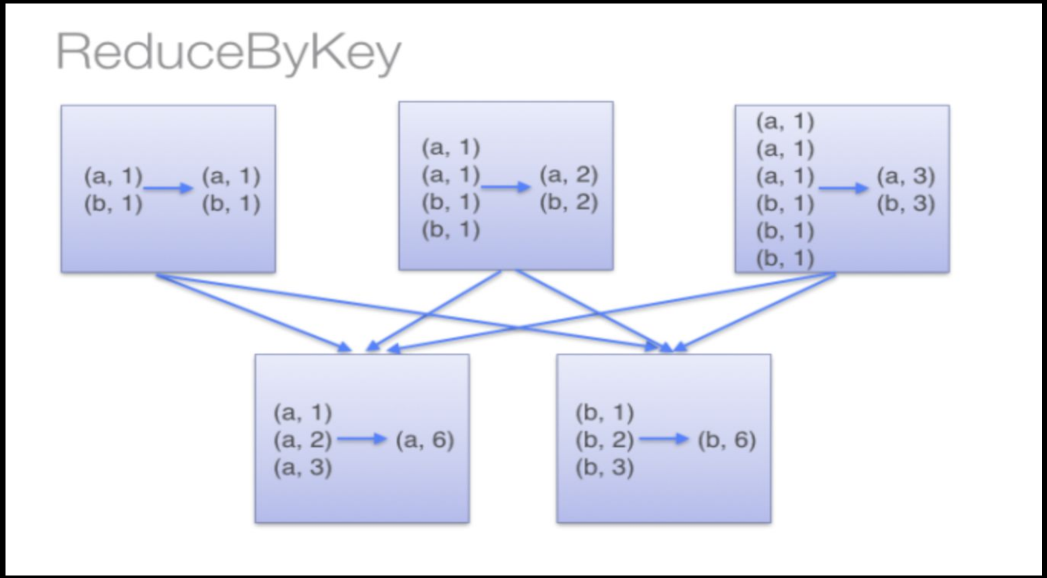
```
~~if (dataFrame.count() == 0) {}~~

if (dataFrame.take(1).length == 0) {}

rdd.isEmpty
```

# 3. API Optimization ..

- avoid groupBykey() for associative reductive operation

```
~~rdd.groupByKey().mapValues(_.sum)~~

rdd.reduceByKey(_ + _)
```



# 3. API Optimization ..

- avoid reduceBykey() when input and output value types are different

~~rdd.map(kv => (kv._1, new Set[String]() + kv._2)) .reduceByKey(_ ++ _)~~

rdd.aggregateByKey(new Set[String]())( (set, v) => set += v, (set1, set2) => set1 ++= set2 )

- avoid flatmap()-join()-groupBy() pattern

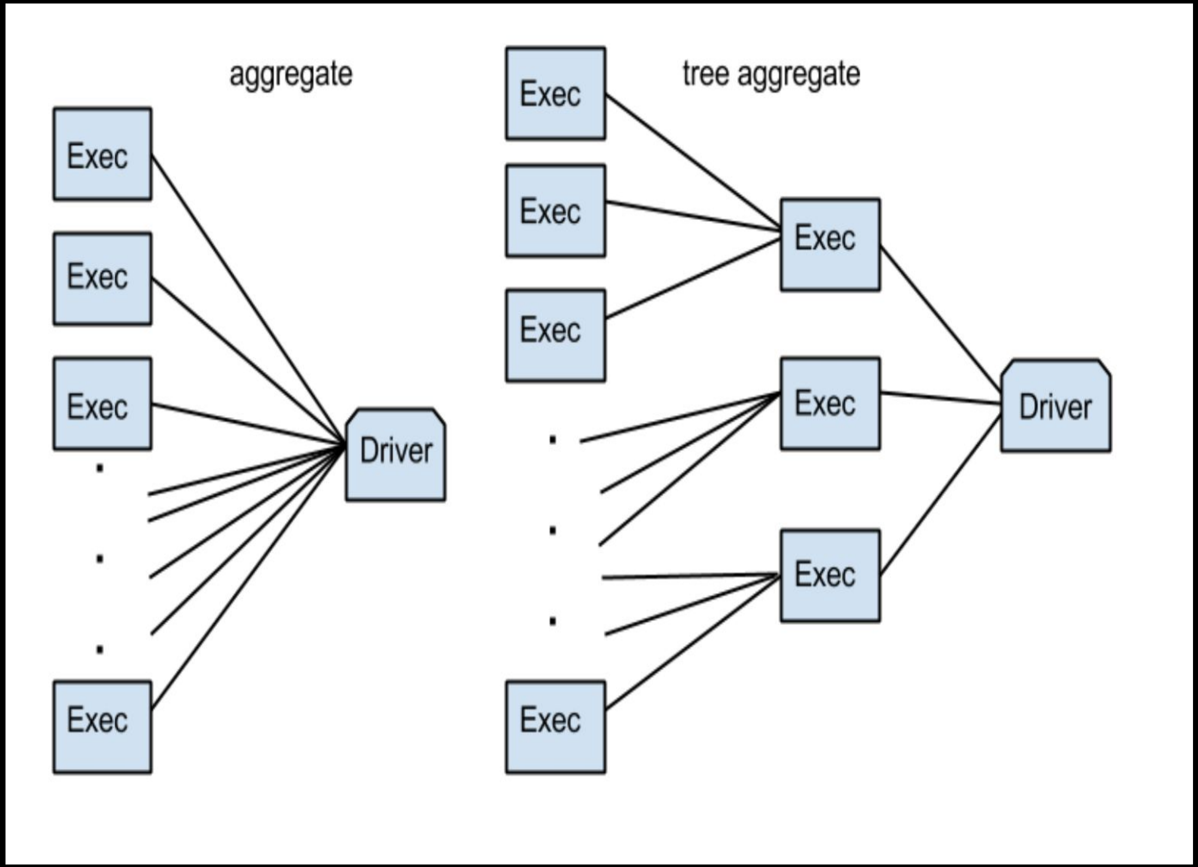
rdd1 = inputRDD1.groupByKey
rdd2 = inputRDD2.groupByKey

~~joinRDD = rdd1.join(rdd2)~~ => (K, (V1,V2))

joinRDD = rdd1.cogroup(rdd2) => (K, (Iterable<V1>, Iterable<V2>))

# 3. API Optimization ..

- Use TreeReduce/TreeAggregate over reduce/aggregate
  - ➔ treeReduce uses function, reduceByKey to do reduction in parallel



# 3. API Optimization ..

- Broadcast variables for efficient joins
  - immutable shared variable, cached on each executor

➔ Join between a large and a small RDD

```
val lookupTable = sc.broadcast(smallRDD.collect.toMap)

largeRDD.flatMap( { case(key, value) =>
  lookupTable.value.get(key).map { otherValue =>
    (key, (value, otherValue))
  }
}
)
```

# 3. API Optimization ..

➔ Join between a large and a medium RDD

```
val keys = sc.broadcast(mediumRDD.map(_._1).collect.toSet)

val reducedRDD = largeRDD.filter{ case(key, value) => keys.value.contains(key) }
reducedRDD.join(mediumRDD)
```

- coalesce() over repartition()

  - After filtering down a large dataset, if you want to decrease number of partitions
  - coalesce() do optmized local shuffling, repartition() do random shuffling

*“ Shuffle less, Shuffle efficiently”*

# How much tasks?

- No. of tasks processing Input RDD = number of partitions explicitly mentioned (default is number of HDFS blocks)
- No. of tasks processing Join/CoGroup RDD = max(number of tasks of each input RDD)

# Optimum # of tasks

There should be sufficient number of tasks

- More Tasks = More parallelization = Better performance as each task gets less data
- More Tasks = Prevents shuffle spill. (Shuffle spill can be very costly to a stage and if it is in your SparkUI's stage metrics, you needs to increase tasks or reduce data per task)
- More Tasks = More containers needed = More Resources and container warmup time

# 5. Data Serialization

1. Storage - RDD persistence level
    - ❑ MEMORY_ONLY (default)
    - ❑ prefer MEMORY_AND_DISK_SER over MEMORY_ONLY

2. Each stage persists its shuffle output on disk in order to prevent recomputation in case of failure.

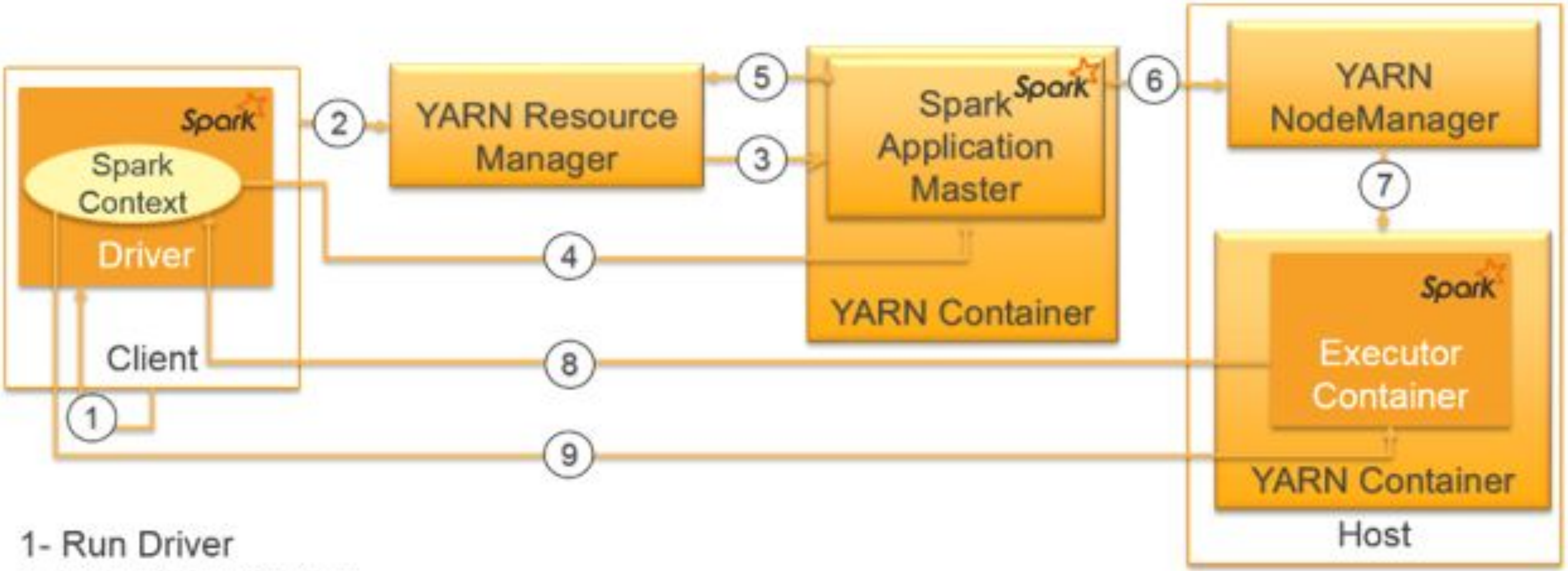
As a result, if a stage is input to multiple stages, it is computed for one and skipped for others.

3. Persist only costly & reusable RDDs

4. Network transfer
    - ❑ Shuffle data between nodes
    - ❑ I/O tasks

Recommended: ***Kryo Serializer*** over ***Java Serializer***

# Cluster Optimization: Spark + YARN




1- Run Driver
2- Submit Application
3- Launch Application Master
4,5- Request Resources
6- Launch Containers via YARN NodeManager
7- Launch Spark Executors
8- Register with the Driver
9- Launch Tasks

# 1. Resource Allocation - Executors

Number of executors

- # executors != # of cores in cluster
- other applications on cluster
- cores taken by application master & driver

Cores per executor

- Avoid Single core executor
- single core can easily handle 3-5 tasks in parallel

# continued…

## Given

10 data node cluster with 64 cores each i.e. 640 cores
and your pool has 40% resource allocation
and you need to support on average 3 jobs in parallel

## Assumption

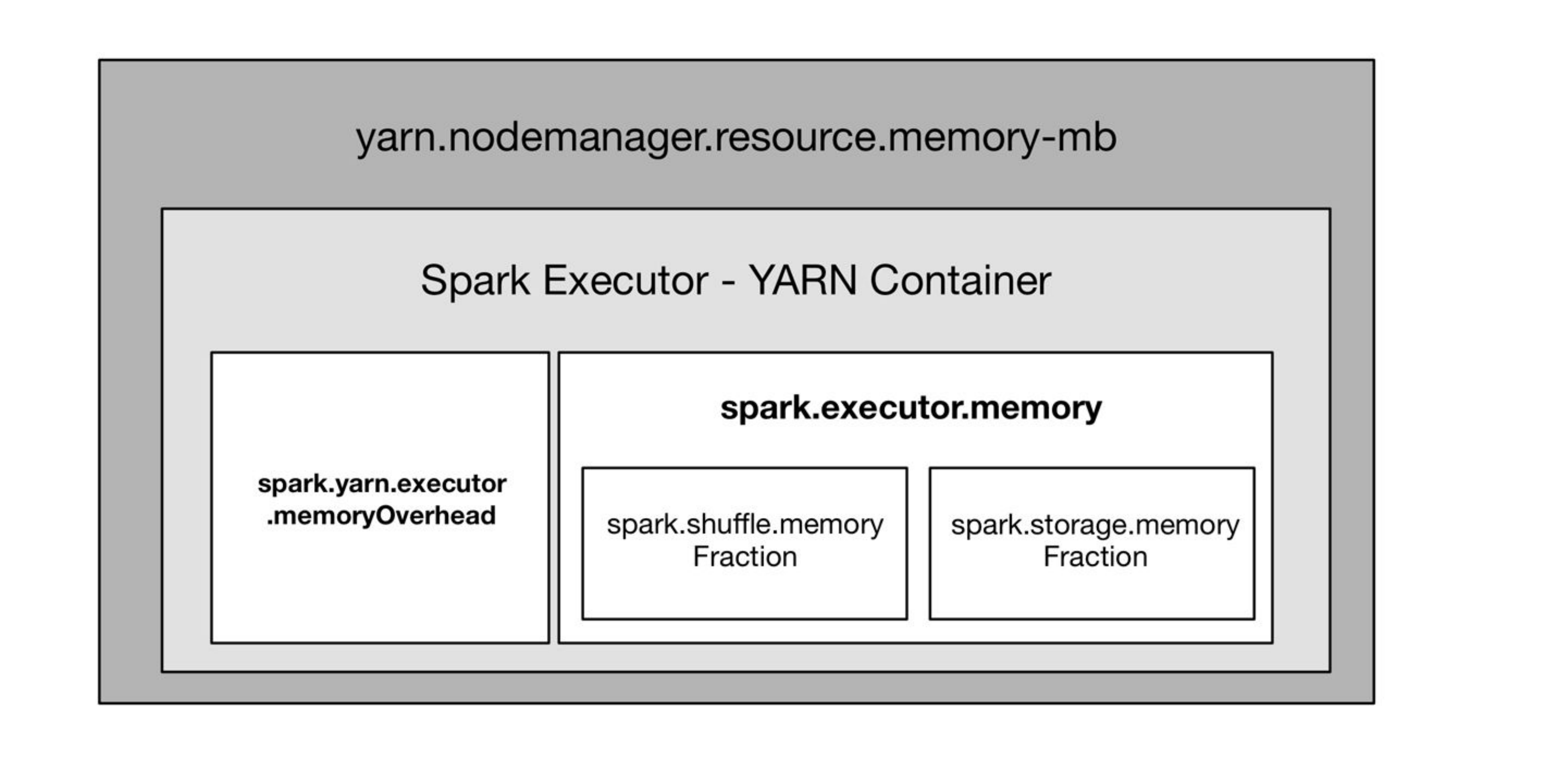
1 core per node is reserved for OS and hadoop overhead
Each job's driver is taking 1 core
Each executor is allocated 4 cores

## Solution

Number of cores per job should be ~ (10 * 64 * .4 – 1 * 10 – 3) / 3 = 81

Number of executors ~( 81 / 4 ) = 20

# 2. Resource Allocation - Memory



# continued…

## Given

10 data node cluster with each having 64 GB of RAM
And # of executor = 20 i.e. 2 executor per node
And memory reserved for OS and hadoop processes = 4 GB

## Solution

Memory available for each YARN container ~(64 - 4) / 2 = 30

Memory available for each Spark Executor ~(30 - 0.1 * 30) = 27

*“Running executors with too much memory often results in excessive garbage-collection delays.”*

# 3. Dynamic allocation

- Assign executors to a job based on workload
- Scale up policy
- Scale down policy
- External shuffle service should be enabled (to safely persist task output in case executor is scaled down)

```
spark-submit ..
    --conf spark.shuffle.service.enabled=”true”

    --conf “spark.dynamicAllocation.enabled”=”true”
    --conf “spark.dynamicAllocation.minExecutors”=”1”
    --conf “spark.dynamicAllocation.maxExecutors”=”5”
    --conf “spark.dynamicAllocation.executorIdleTimeout”=”30”
    --conf “spark.dynamicAllocation.cachedExecutorIdleTimeout”=”30”
```

# 3. Dynamic allocation…

- MaxExecutors can be skipped if we are not sure on upper limit of data. (If no of task > (maxExecutor * tasks per executor), performance worsens as it tries to launch more tasks per executor and hence more failure retries)
- cachedExecutorIdleTimeout is infinity by default. In case of cached RDD, tune it to scope time of cached RDD. If not, it will result in recomputation.

# 4. Spark Speculative Execution

*Turn on to prevent Straggler Tasks...*

```
spark-submit ..
      --conf “spark.speculation”=”true”
      --conf “spark.speculation.interval”=”5000”
      --conf “spark.speculation.multiplier”=”5”
     --conf “spark.speculation.quantile”=”0.90”
```

# 5. YARN Optimization

```
YARN container size
    = spark.executor.memory + spark.yarn.executor.memoryOverhead
```

where

default value of

spark.yarn.executor.memoryOverhead

    = max(384 MB, 0.10*spark.executor.memory)

or can be explicitly set in configuration

# Thank You!

# QUESTIONS?