Intro to Apache Spark™

“Apache Spark™ is a general purpose, distributed, fast engine for large-scale data processing.”

Spark Core Ideas

- Data Processing Framework
- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)

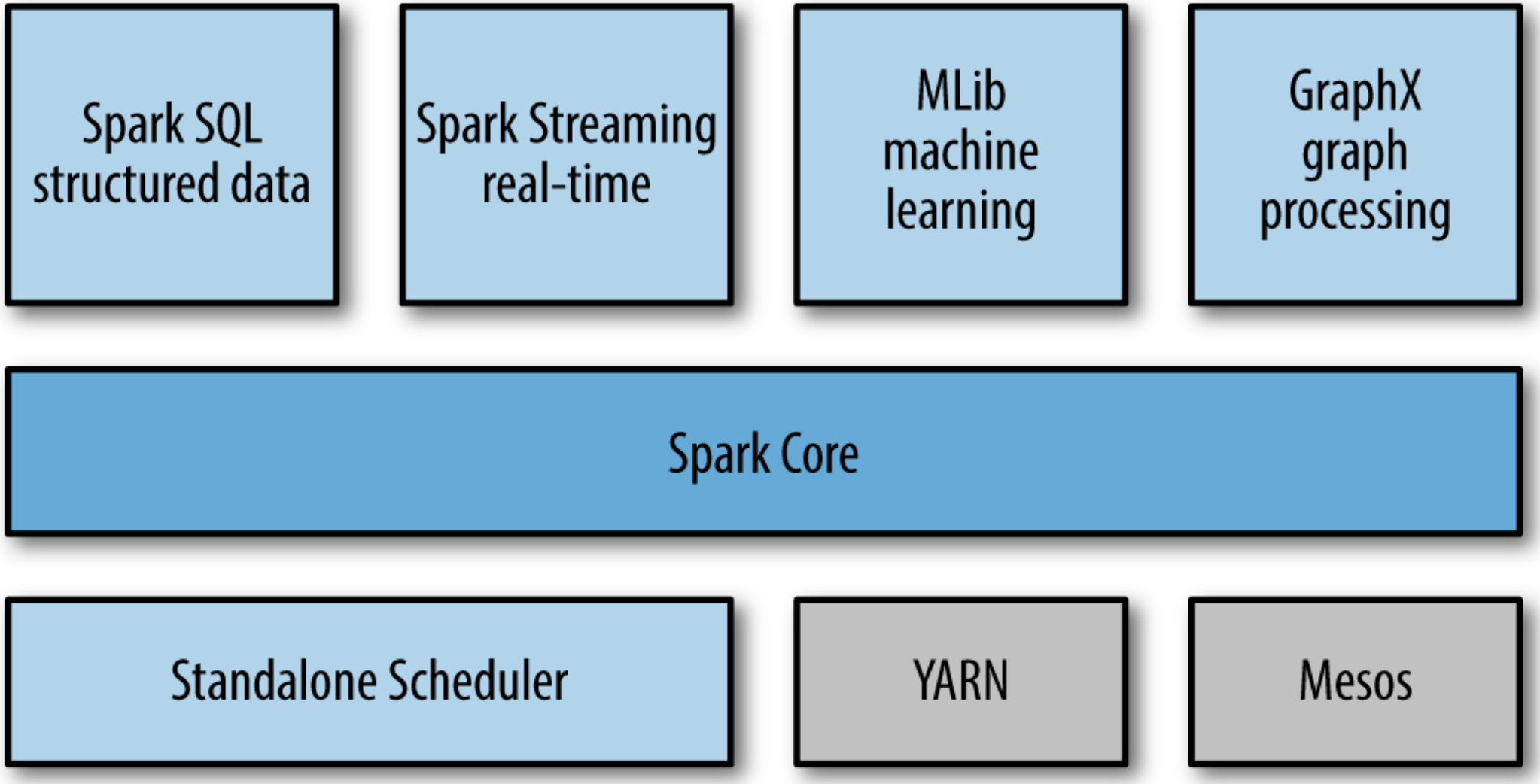
Apache Spark

- Runs on JVM
- Developed using Scala Language
- Faster Adaption into Big Data world
- Support Java/Scala, Python, R, sql language

Spark is more of developer focused than end user query processing engines.

- Developers write code in Scala/Java, Python/R/Spark SQL

Spark Architecture



Apache Spark with Scala Course Summary

- Developer Focused, non-developers may focus on Spark as concept rather than Scala language construct/syntactical parts
- Java and JVM knowledge required
- Apache Spark + Scala course is intended for developers who are familiar with Scala Language and/or proficient in Java Development

Skill Pre-requistees

- Functional Programming Basics
- Scala Fundamentals, Scala Functional aspects
- JVM fundamentals
- Java, threading, processes
- Knowledge on distributed computing
- Knowledge on Map/Reduce model architecture

Challenges of Big Data

Operational Challenges

- Processing and discovery
- Present it to business
- Hardware and network failures

Challenges of Big Data

Data Challenges

- Volume
- Velocity
- Variety of Data

Challenges of Big Data

Computation Challenges

- CPU
- RAM
- Network
- Failures

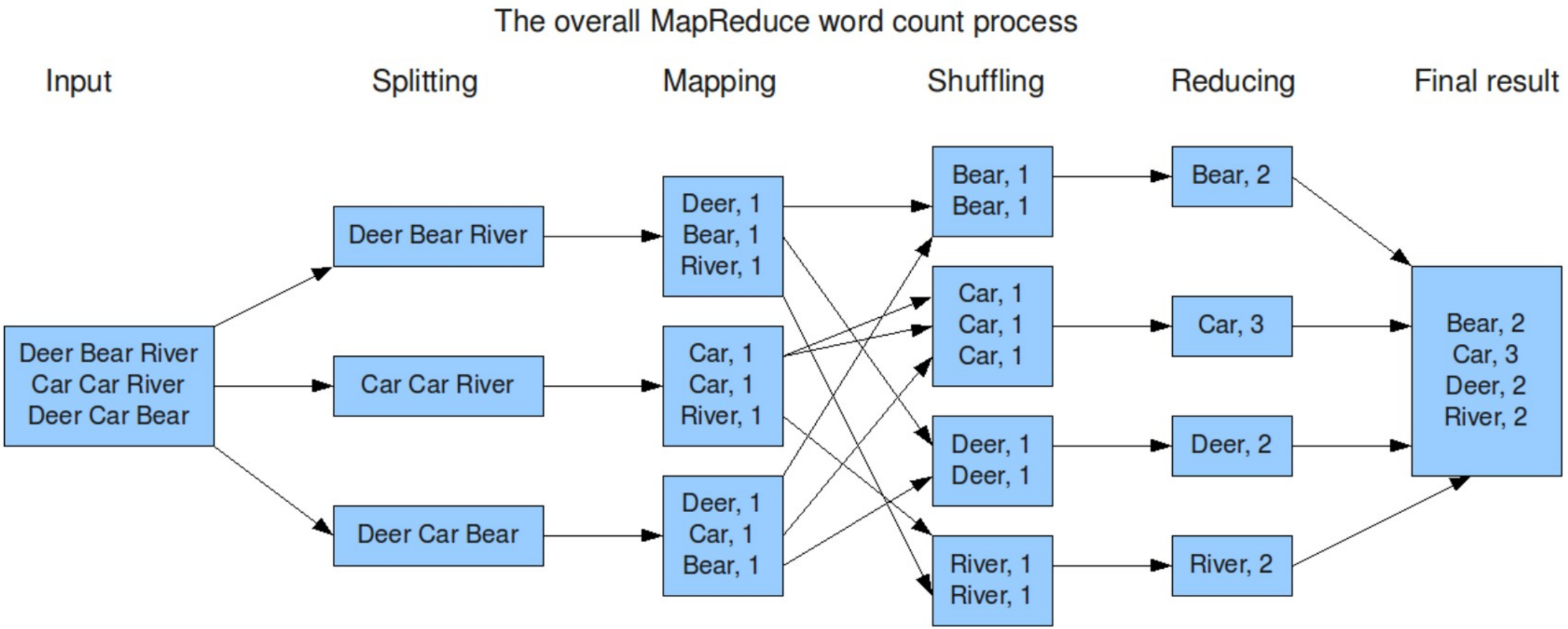
MAP REDUCE

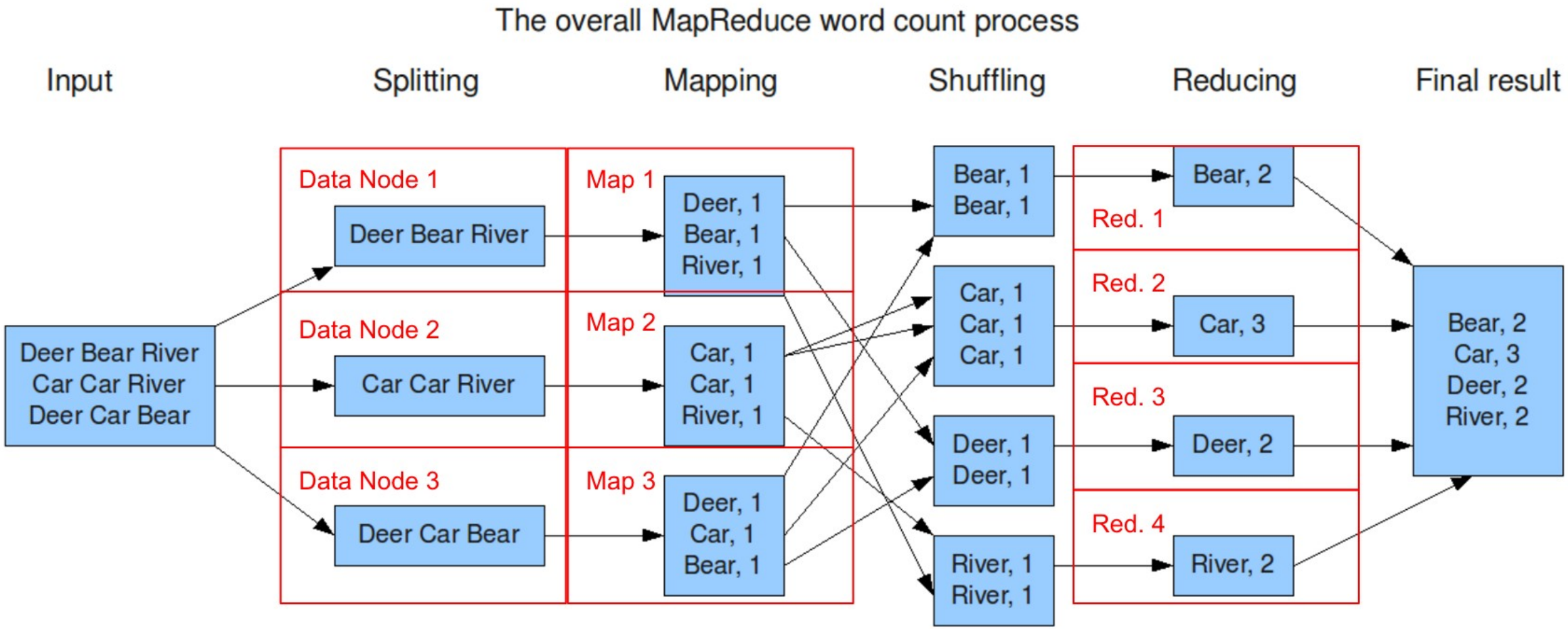
“MapReduce' is a framework for processing parallelizable problems across huge datasets using a cluster, taking into consideration scalability and fault-tolerance”

Map Reduce

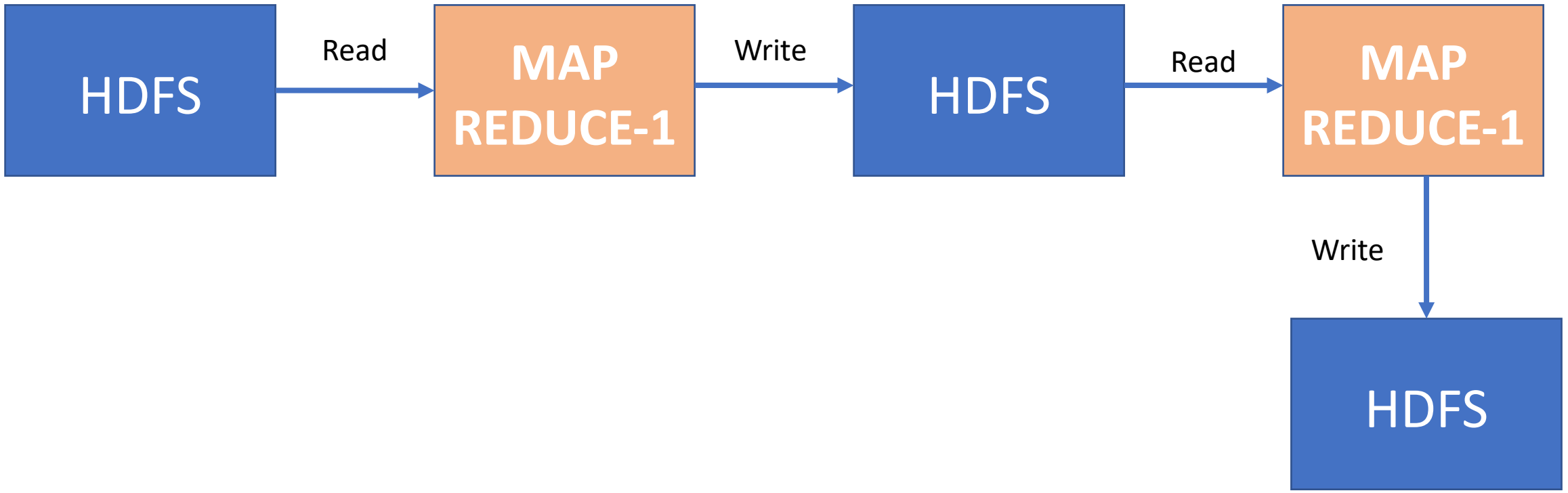
- **MapReduce** is a programming model for processing large data sets with a parallel , distributed algorithm on a cluster
- Basic unit of information, used in MapReduce is a **(Key,value) pair**
- **Map**-function and **Reduce**-function +**Shuffling** in between
- **Mapper** bring **structure to unstructured data**, in word count, makes it (word, 1) pair, where as word is a key, 1 is a value
- **Shuffling**: Take output form Mapping phase, consolidate the output from map, ex same words are clubbed together with respective freq
- **Reducer**: Output values from **Shuffling are aggregated**, like counting, avg, returns a output value, summarize all the output together

MAP REDUCE Models



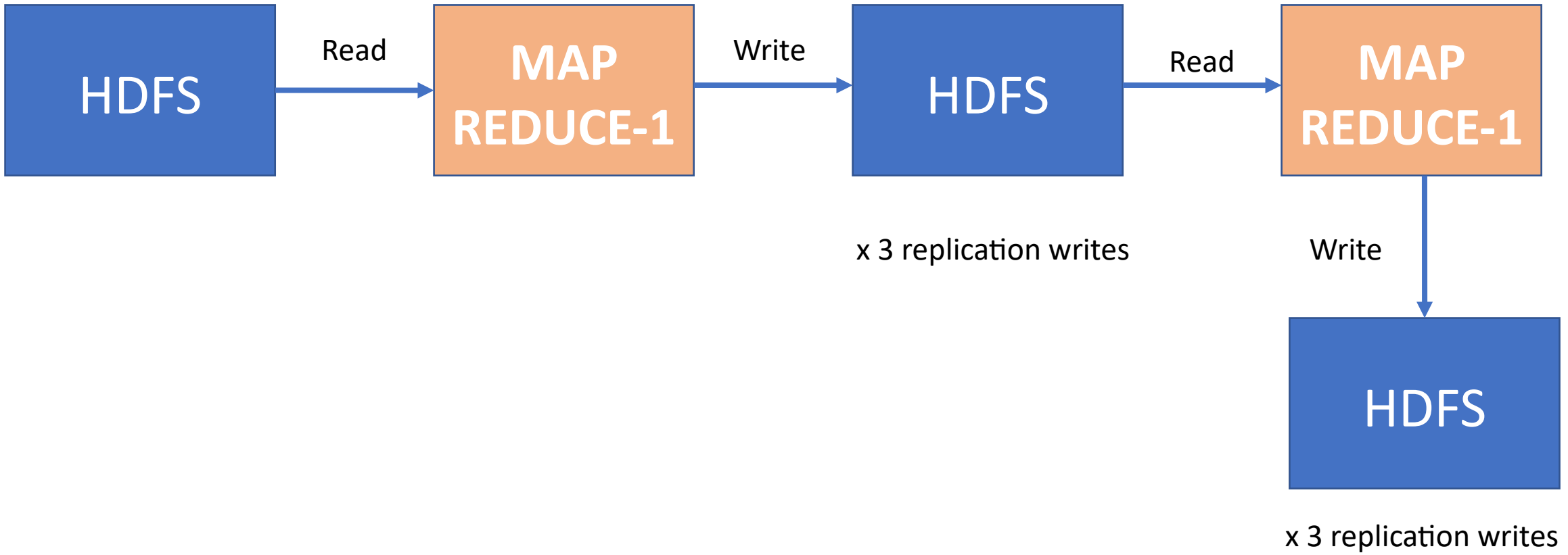

The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Data Node 1
Deer Bear River
Data Node 2
Car Car River
Data Node 3
Deer Car Bear
Map 1
Deer, 1
Bear, 1
River, 1
Map 2
Car, 1
Car, 1
River, 1
Map 3
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Red. 1
Bear, 2
Red. 2
Car, 3
Red. 3
Deer, 2
Red. 4
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

HADOOP MODEL



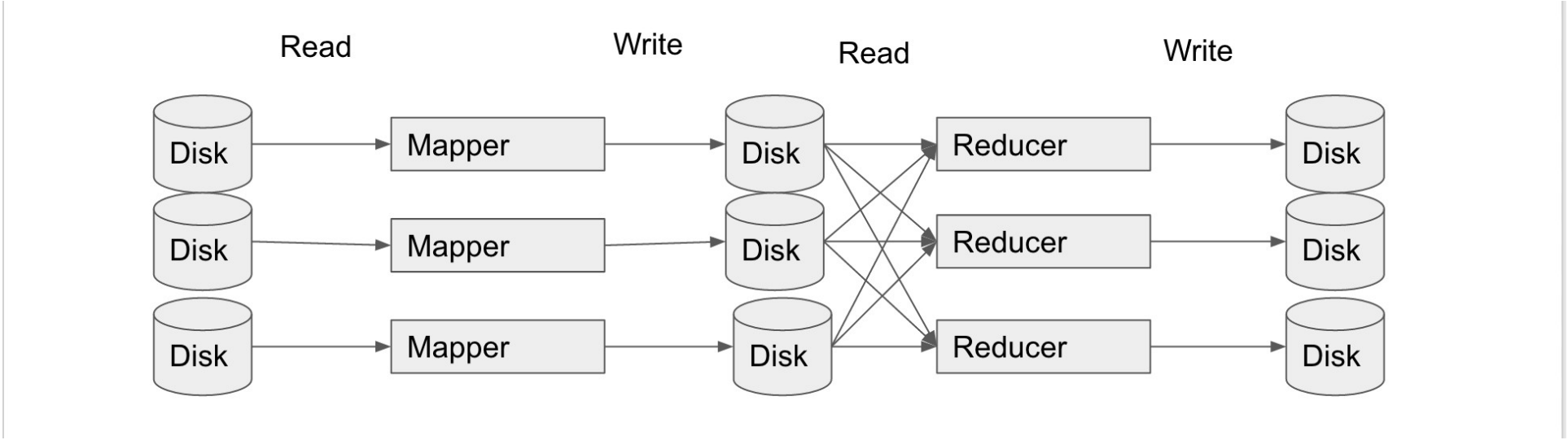
HADOOP MODEL

Do you know HDFS has replication?



MapReduce Performance Bottlenecks

- MapReduce is a very I/O heavy operation
- Map phase needs to read from disk then write back out
- Reduce phase needs to read from disk and then write back out



Problems with MapReduce

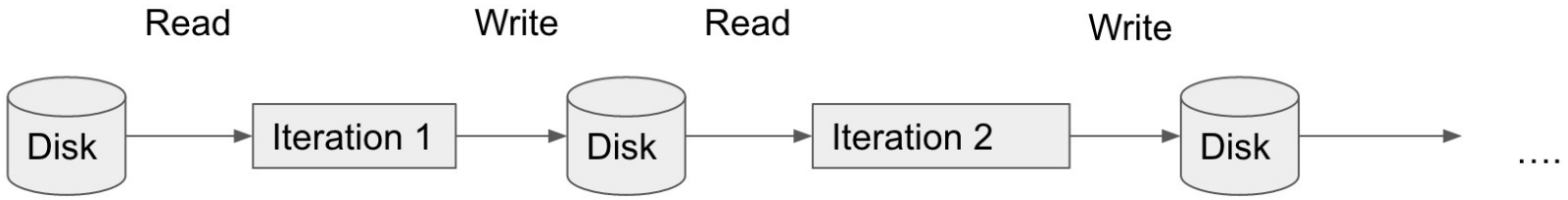
MapReduce use cases showed two major limitations

1. Difficulty of programming directly in MapReduce
2. Performance bottlenecks, or batch not fitting the use cases
    - Trying to be real time

People often create specialized systems to avoid using MapReduce

MapReduce Performance Bottlenecks (Cont.)

We often don't just use one job. Many times we are running jobs back to back.



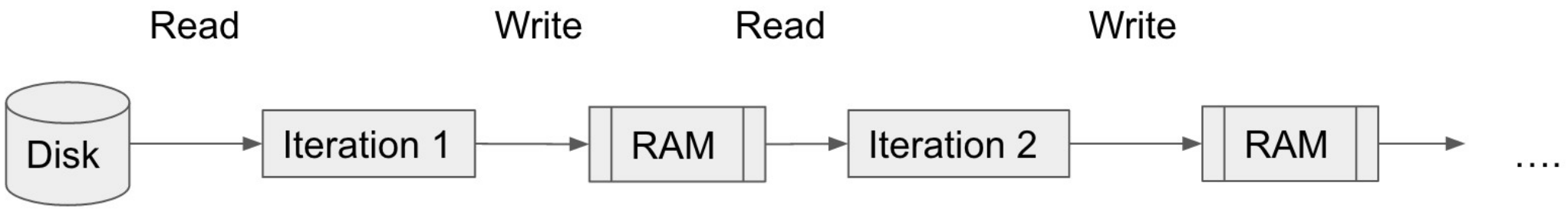
Tech Trends

Compared to where we were at when Hadoop was first around:

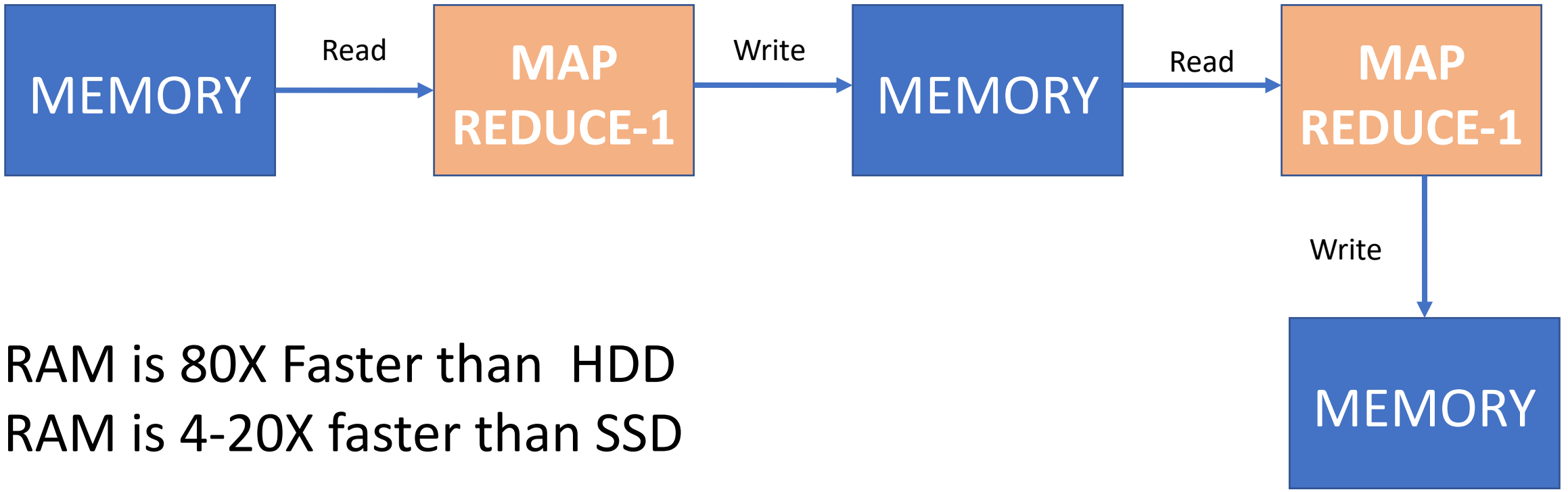
- Now we have SSD's
- Networking has improved (1 gbps, 10 gbps etc)
- RAM is becoming very cheap and abundant, faster

How can MapReduce be improved?

- Use RAM for in-data sharing



SPARK MODEL

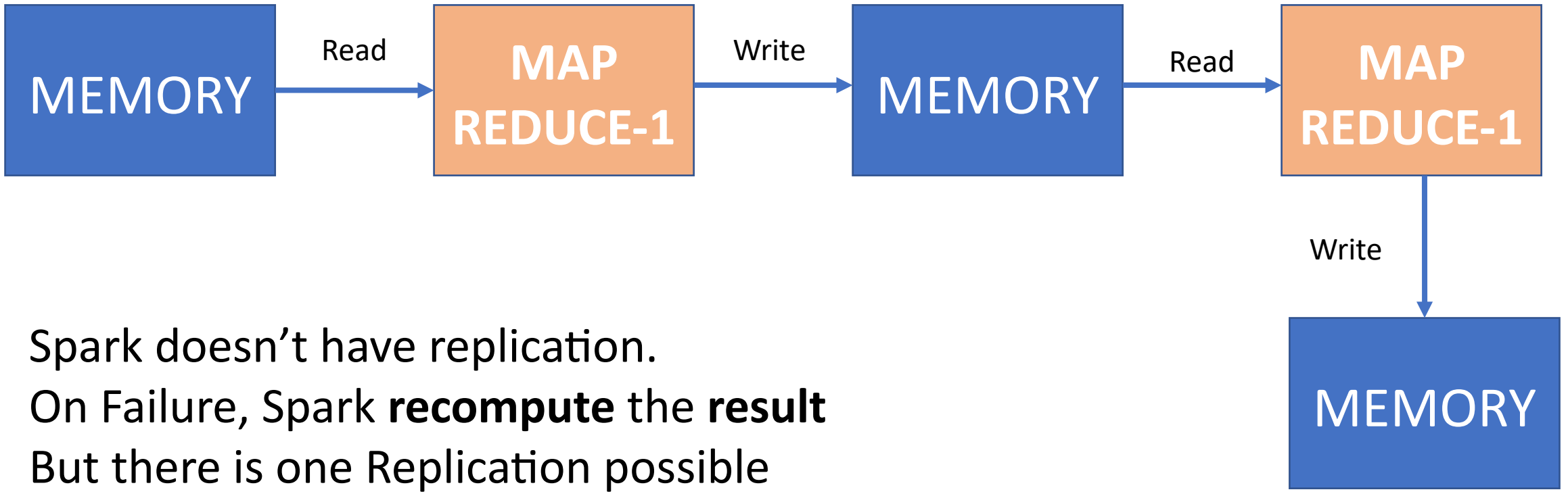


RAM is 80X Faster than HDD

RAM is 4-20X faster than SSD

SPARK MODEL

How about Spark Replication?



Spark doesn't have replication.
On Failure, Spark **recompute** the **result**
But there is one Replication possible
with rdd.persist() * discussed later

MapReduce vs Spark (Performance) (Cont.)

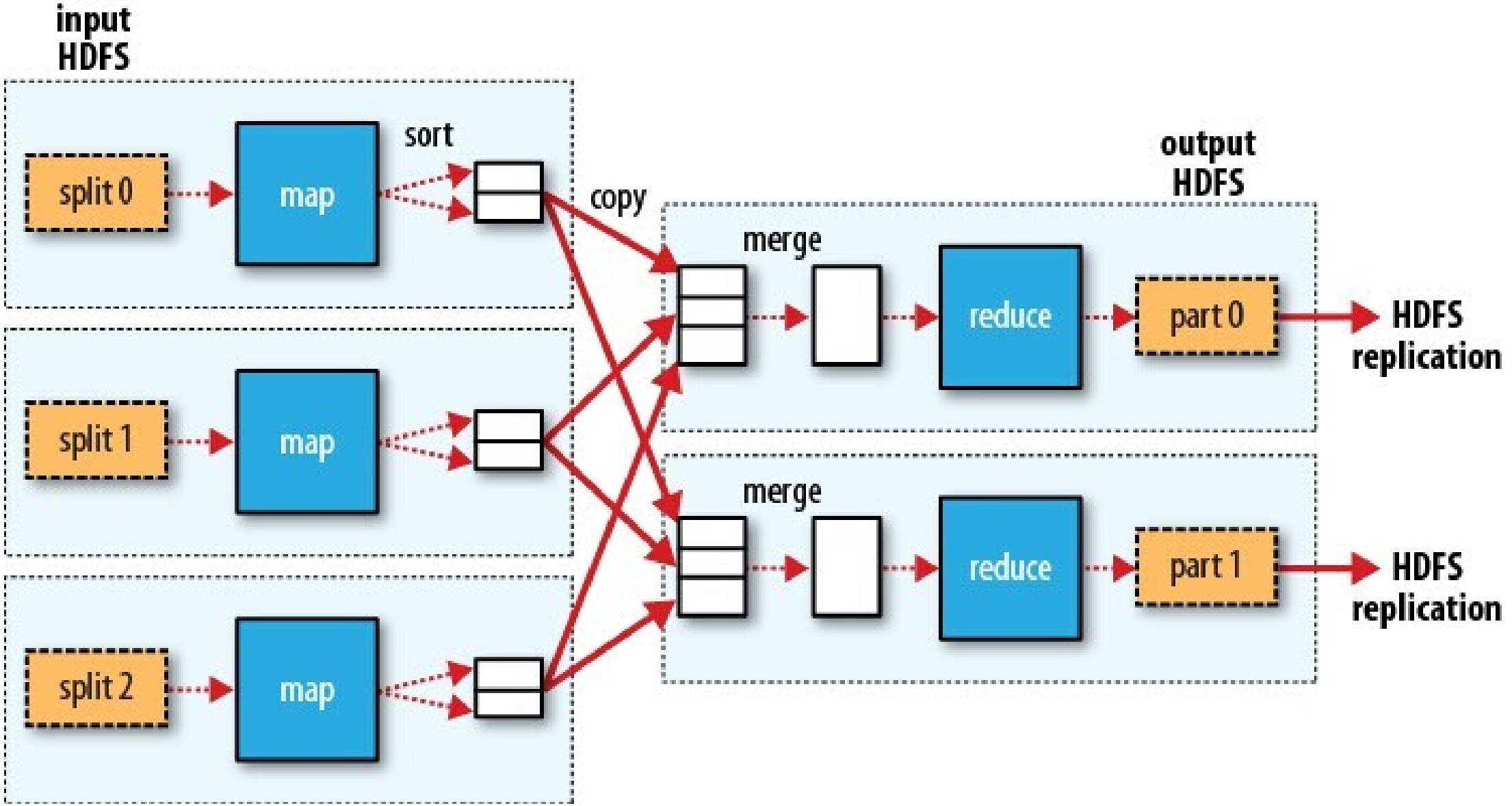
- Dayton Gray 100 TB sorting results
- https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

	MapReduce Record	Spark Record	Spark Record 1PB
Data Size	102.5 TB	100 TB	1000 TB
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Elapsed Time	72 mins	23 mins	234 mins
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

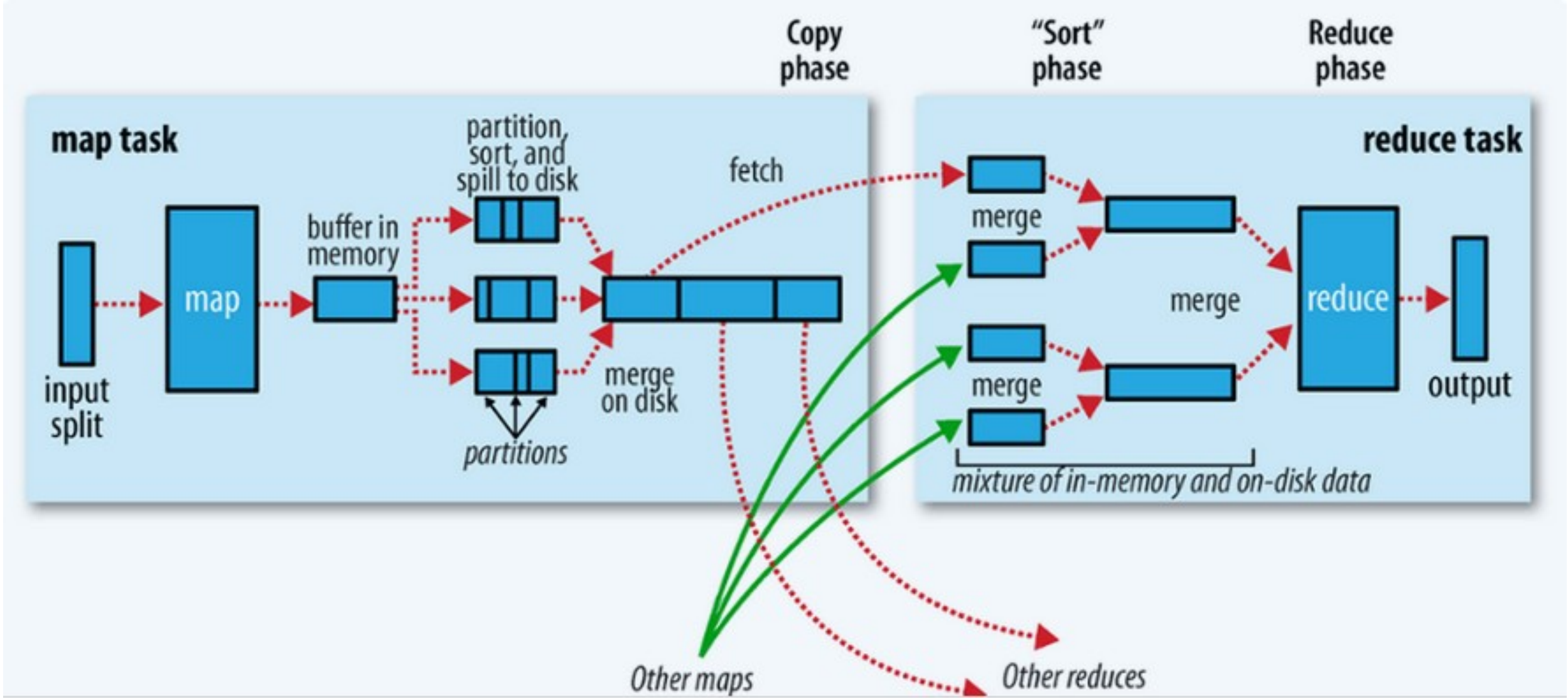
MapReduce vs Spark (Implementation)

	MapReduce	Spark
Storage	Disk Only	In Memory and Disk
Operations	MapReduce	Map, Reduce, Joins, Sample, etc
Execution Model	Batch	Batch, Streaming, Interactive
Programming Environment	Java	Java, Python, Scala

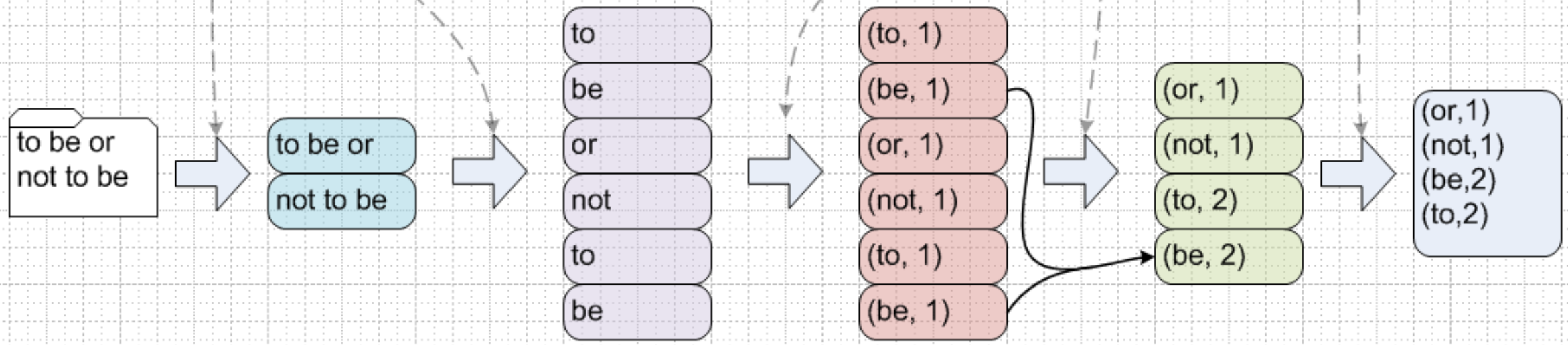
Hadoop Execution

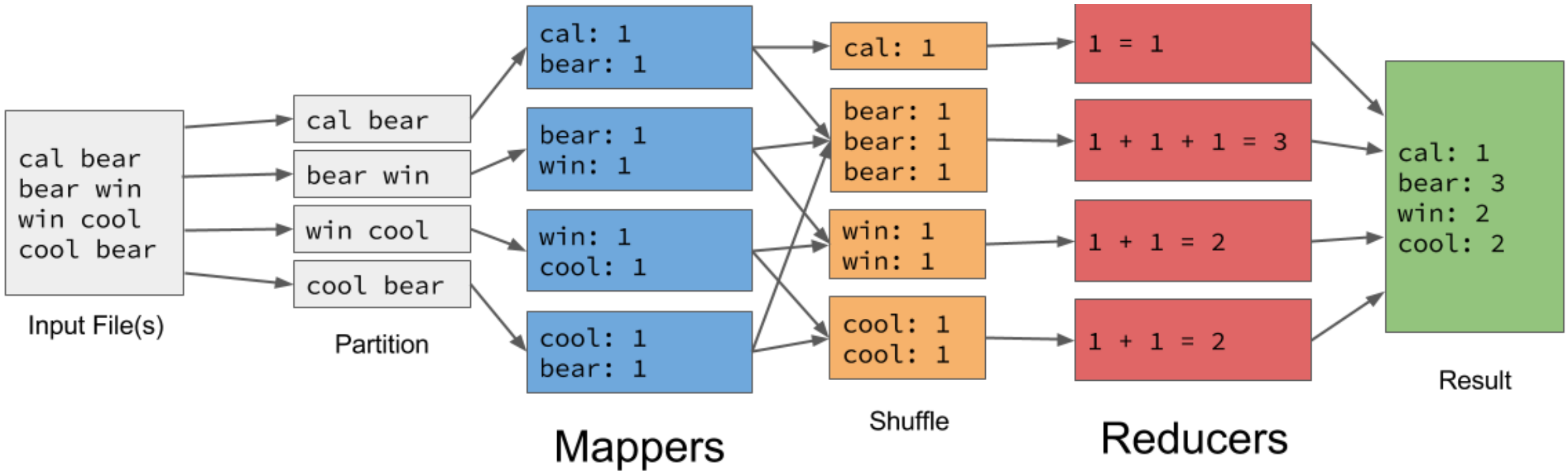


Shuffle and Sort
Slow due to replication, serialization, I/O. Inefficient for Iterative algorithms, interactive queries



```
val lines = sc.textFile("../hamlet.txt")
val wordCounts = lines.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey((a, b) => a + b)
wordCounts.collect().foreach(println)
```





receives part of the input file and emits a list of (key, value) pairs:

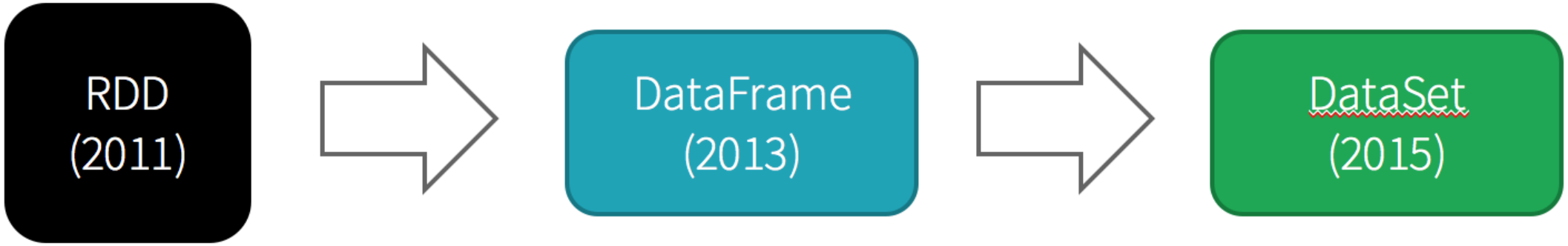
`[(k1, v1), (k2, v2), …]`

receives all the values for the same key and emits a combined value:

`reduce([va, vb, vc, ...], func)`

History of Spark APIs

Resilient Distributed Datasets
JVM Objects



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Python/R, SQL

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

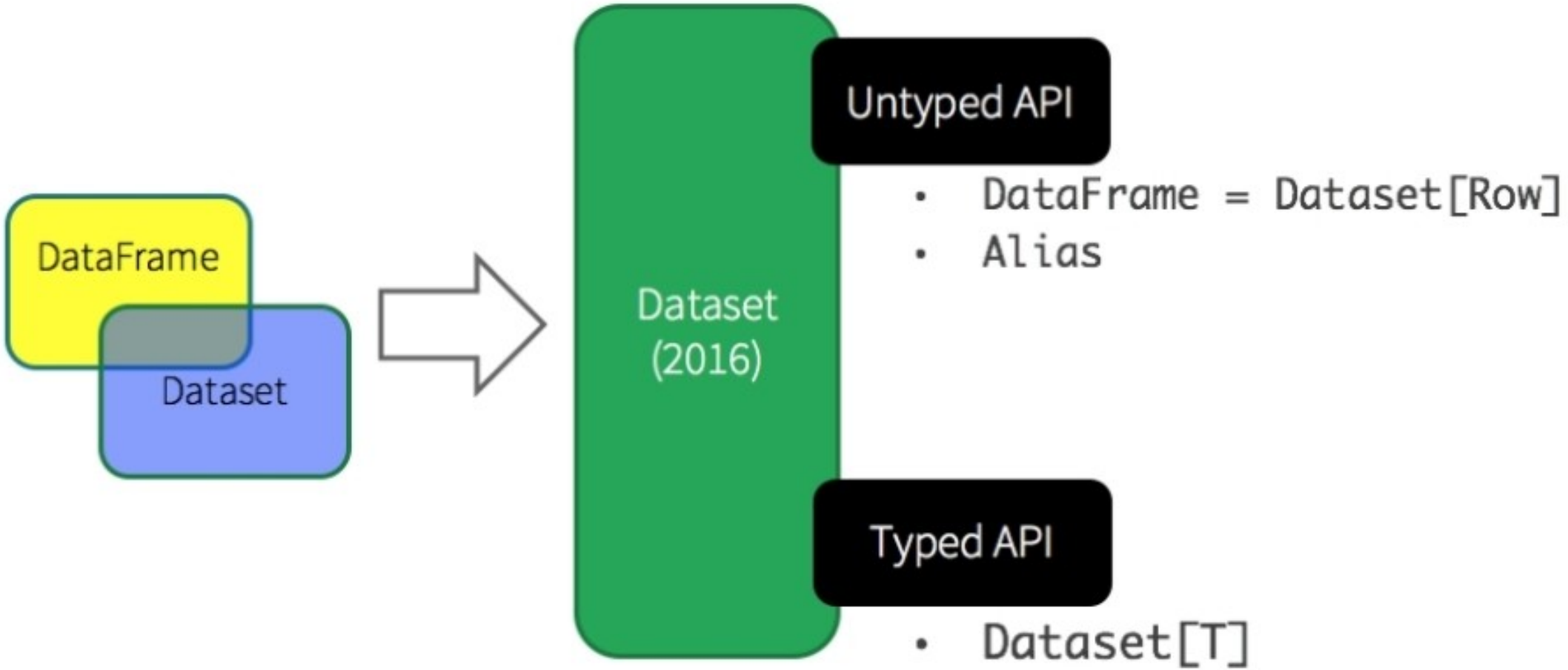
JVM Objects

Internally rows, externally JVM objects

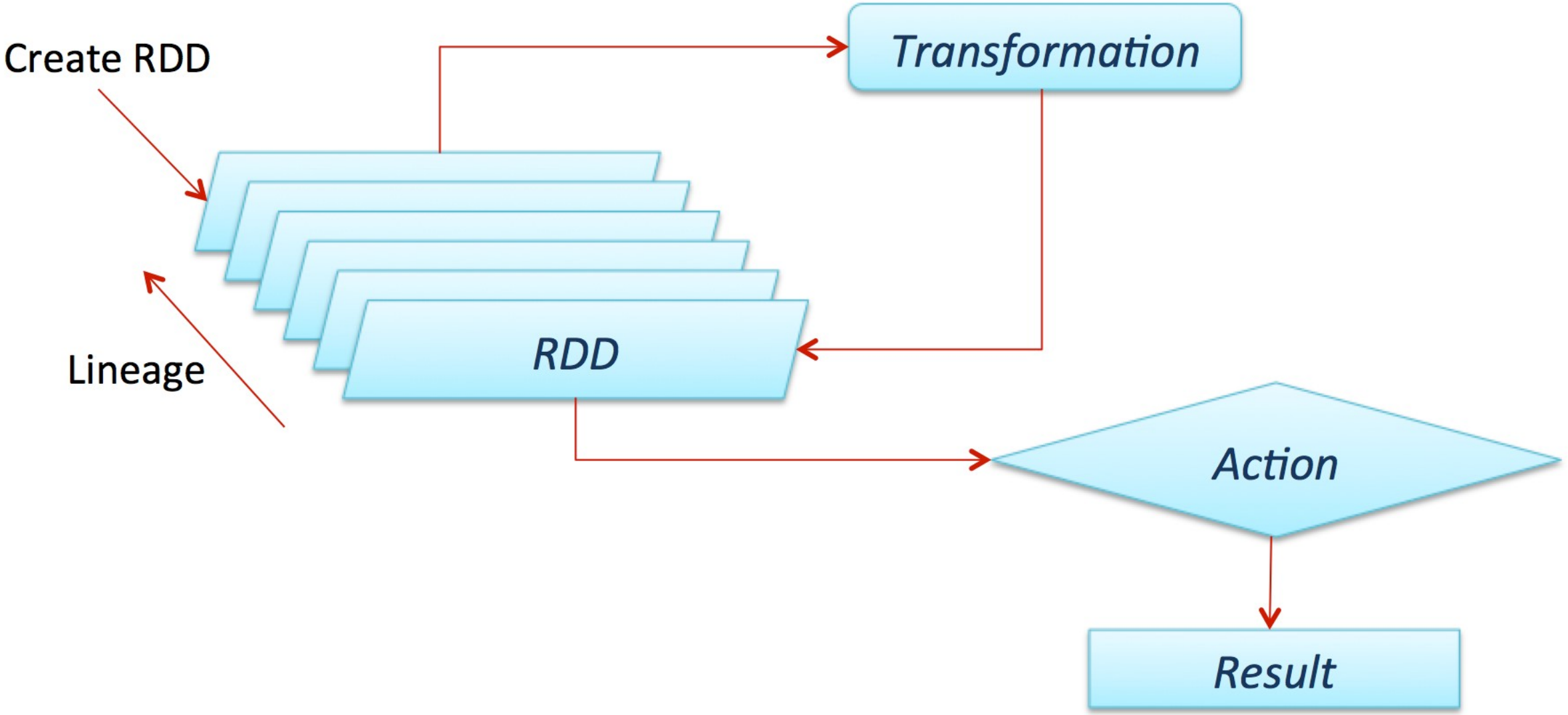
Almost the "Best of both worlds": **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

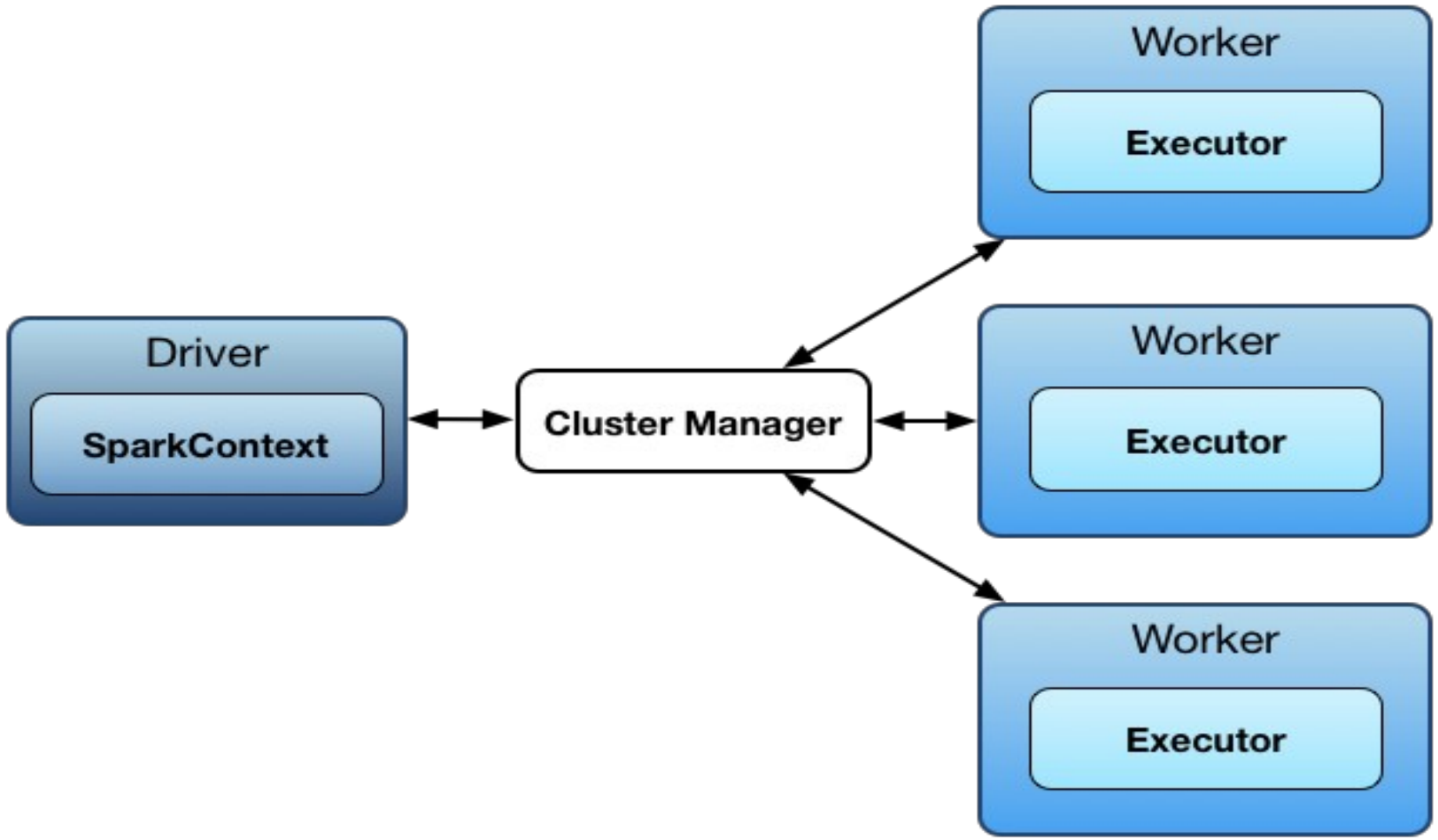
Unified Apache Spark 2.0 API



RDD-Low Level – Core of Spark

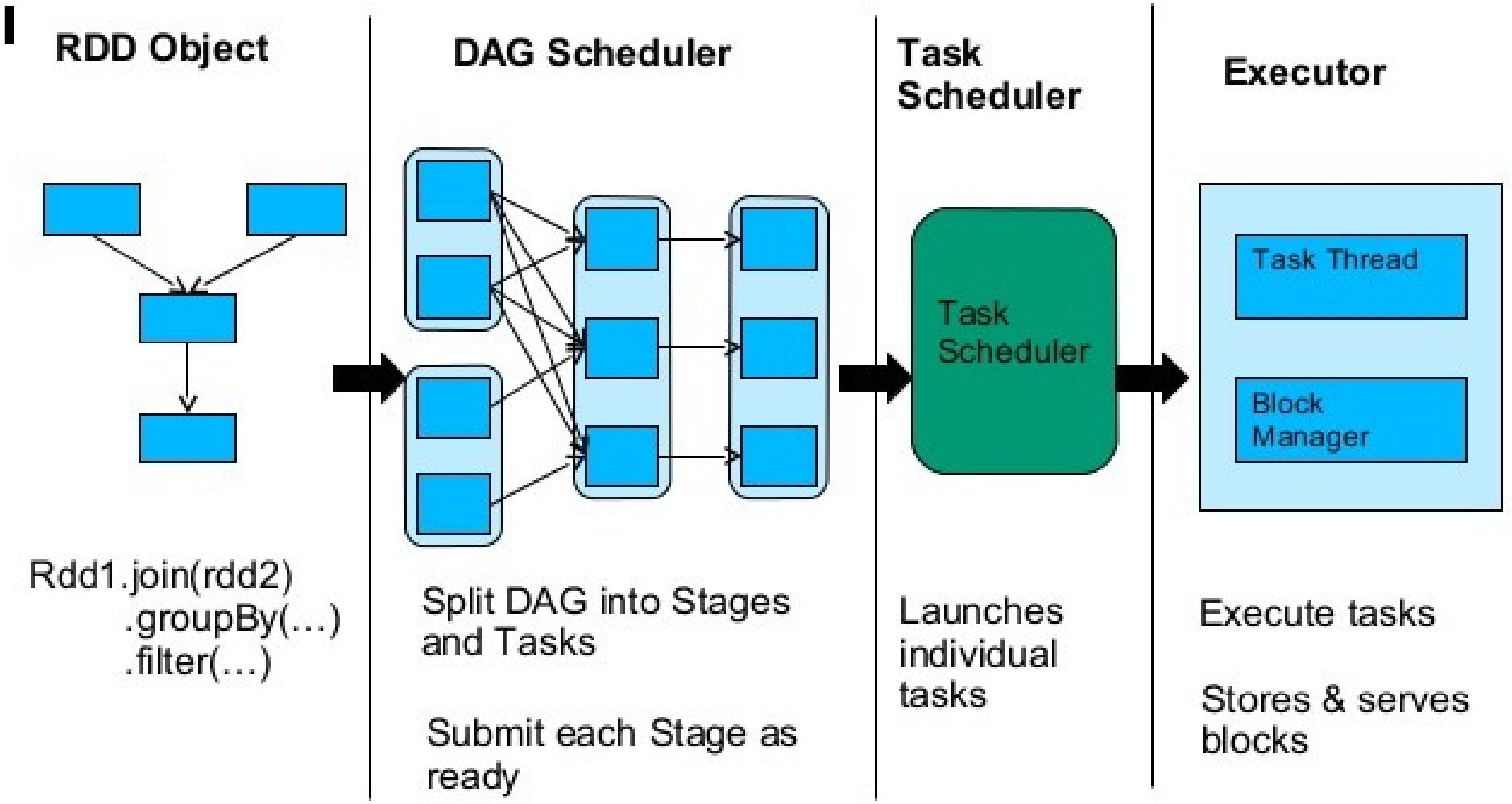


Spark Master Worker Architecture



Lazy Evaluatio

Spark Internal – Job Scheduling



Terminology

Job: The work required to compute an RDD

Stage: A wave of work within a job, corresponding to one or more pipelined RDD's

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

Shuffle: The transfer of data between stages

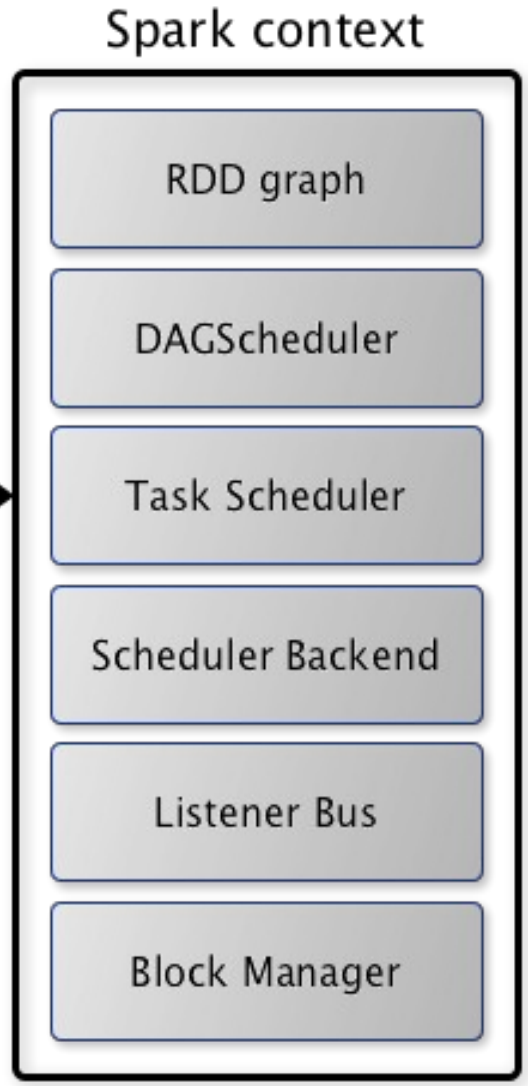
Spark Driver, Executors, Stages & Tasks

- As you submit transformations on RDDs, Spark keeps track of the lineage graph
- **Accumulators** – provide a simple syntax for aggregating values from worker nodes back to the driver program.
- **Broadcast** variables – a second kind of shared variable, allows Spark to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations.

Spark Context

- Entry point to Spark Core
- Spark Context sets up internal services
- Connects to Spark Execution Env
- Used to create RDD,
- Manages accumulator, broadcast variables, run jobs

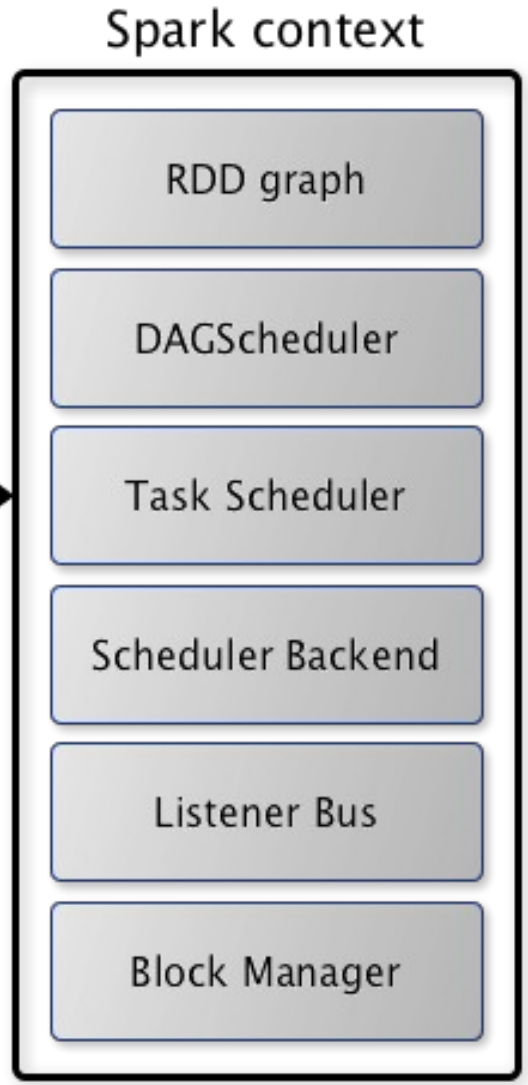
```
val sc = new SparkContext(master="local[*]",
            appName="SparkMe App", new SparkConf)

val lines = sc.textFile(...).cache()

val c = lines.count()
println(s"There are $c lines in $fileName")
```



Spark Context

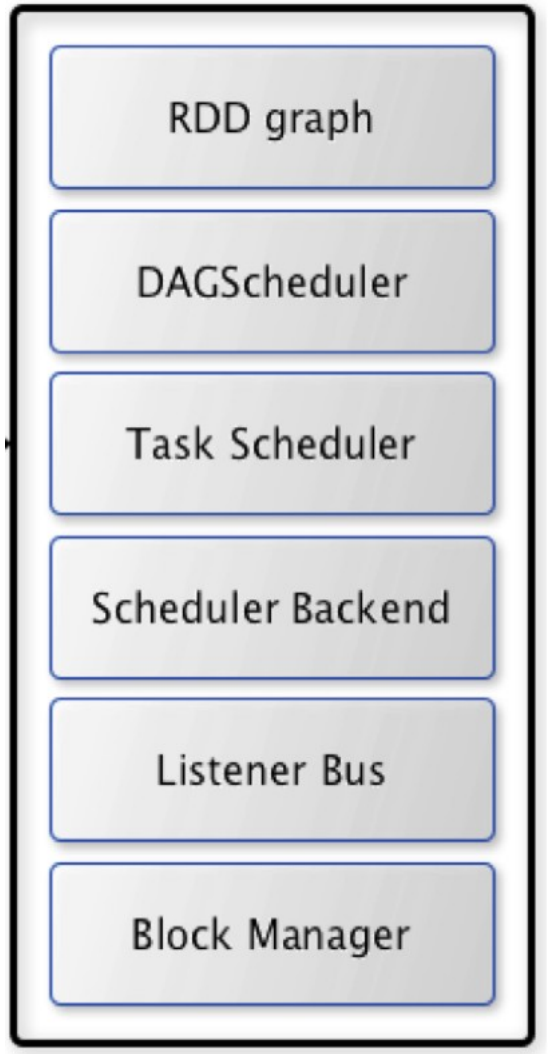
- A Spark program first creates a SparkContext object
  - Spark Shell automatically creates a SparkContext as the **sc** variable
- Tells spark how and where to access a cluster
- Use SparkContext to create RDDs

```
val sc = new SparkContext(master="local[*]",
            appName="SparkMe App", new SparkConf)

val lines = sc.textFile(...).cache()

val c = lines.count()
println(s"There are $c lines in $fileName")
```



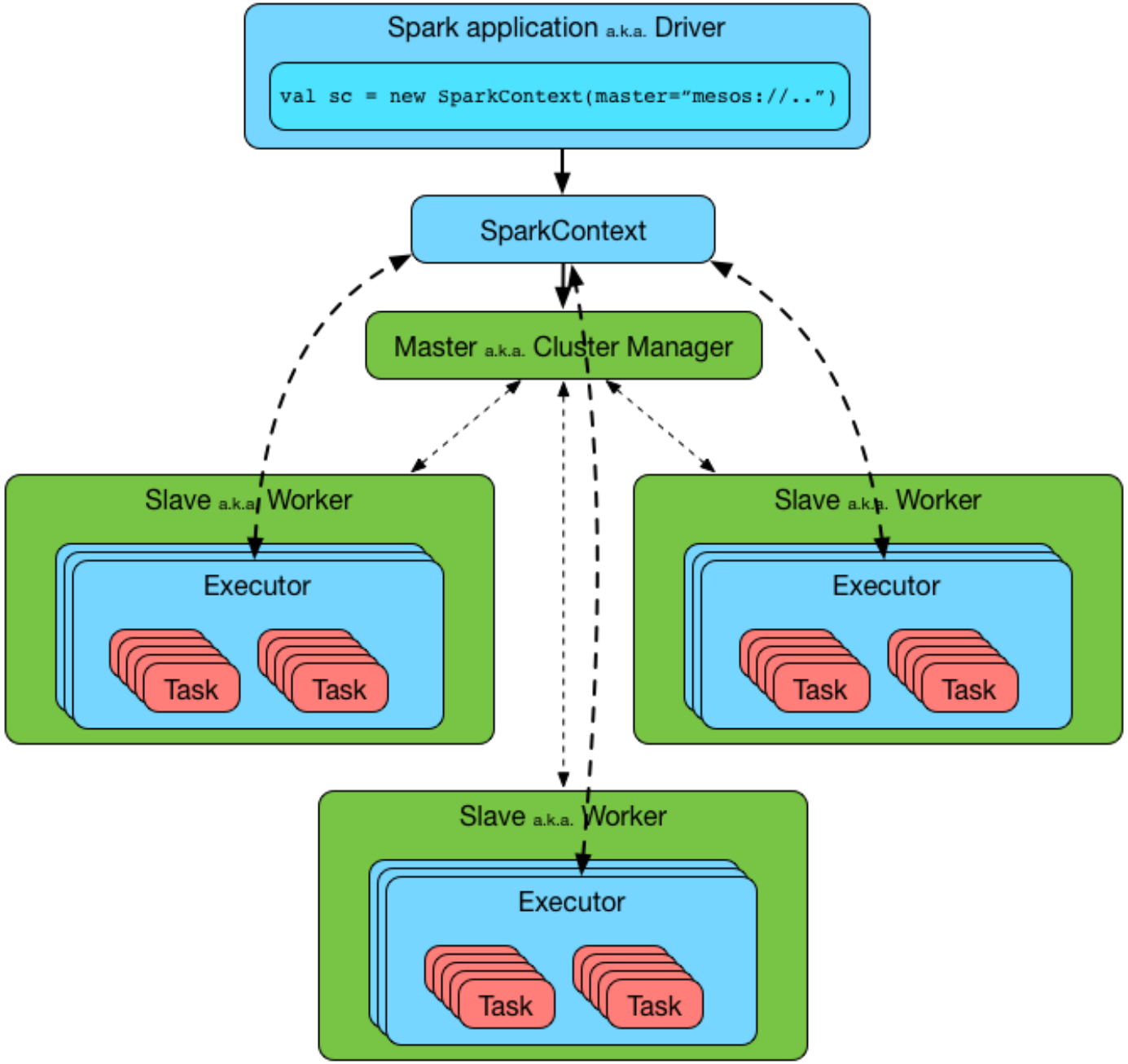
Spark Session

- Older ways - SparkConf, SparkContext, SQLContext, and HiveContext to execute your various Spark queries for configuration, Spark context, SQL context, and Hive context respectively.
- The SparkSession is essentially the combination of these contexts including StreamingContext.
- **The SparkSession is now the entry point for reading data, working with metadata, configuring the session, and managing the cluster resources**



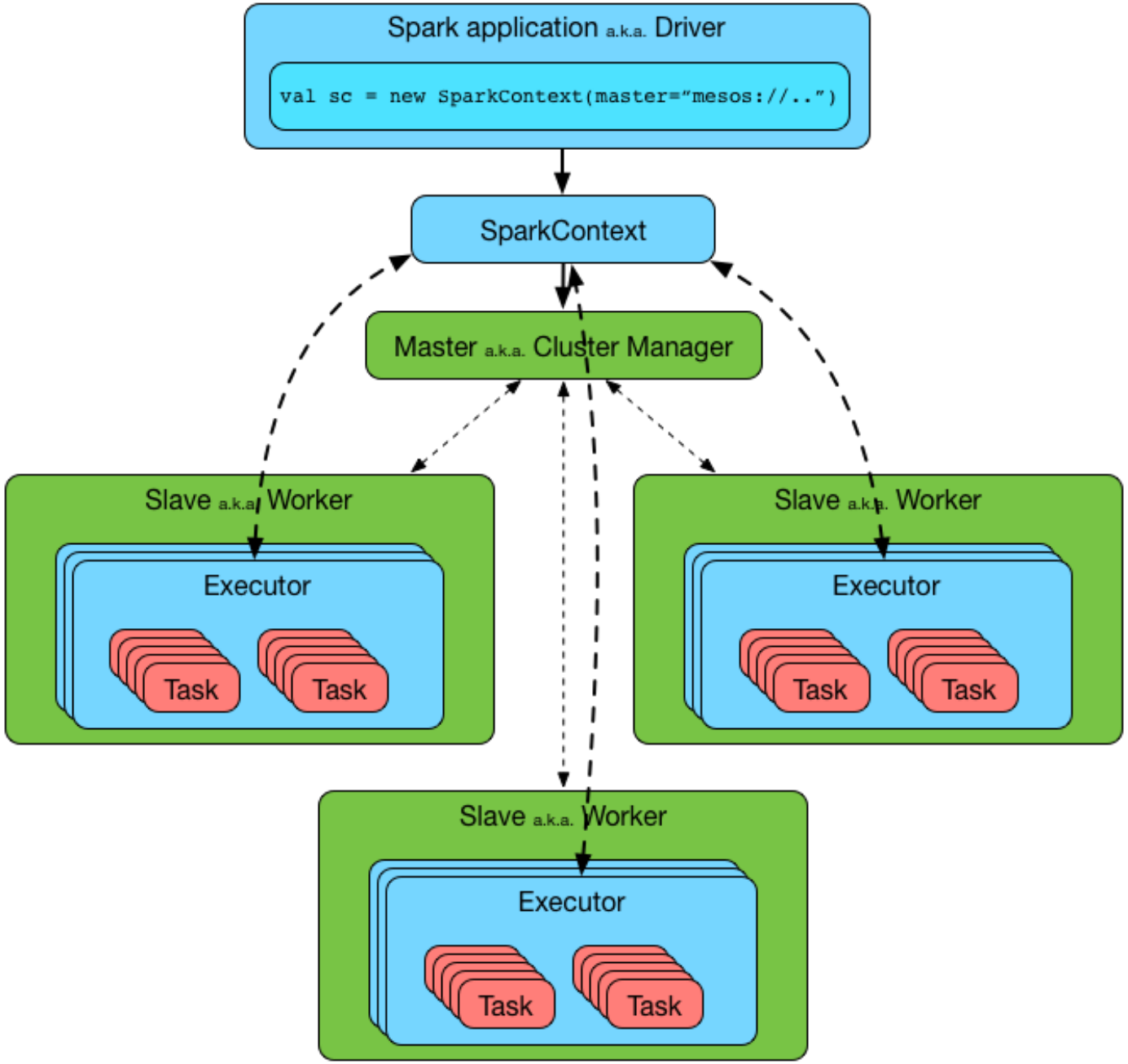
Spark Driver

- **Spark Driver** – runs on the master node and declares transformations and actions on the data.
- **Driver** creates the SparkContext/Spark Session, connected to a given Spark Master.



Executor

- **Executors** – worker processes responsible for running the individual tasks in a given Spark job.
- **Executors** have two roles
  - Run the **tasks** that make up the application and return the results to the driver
  - Provide in-memory storage for RDDs that are cached by user programs.



How it works

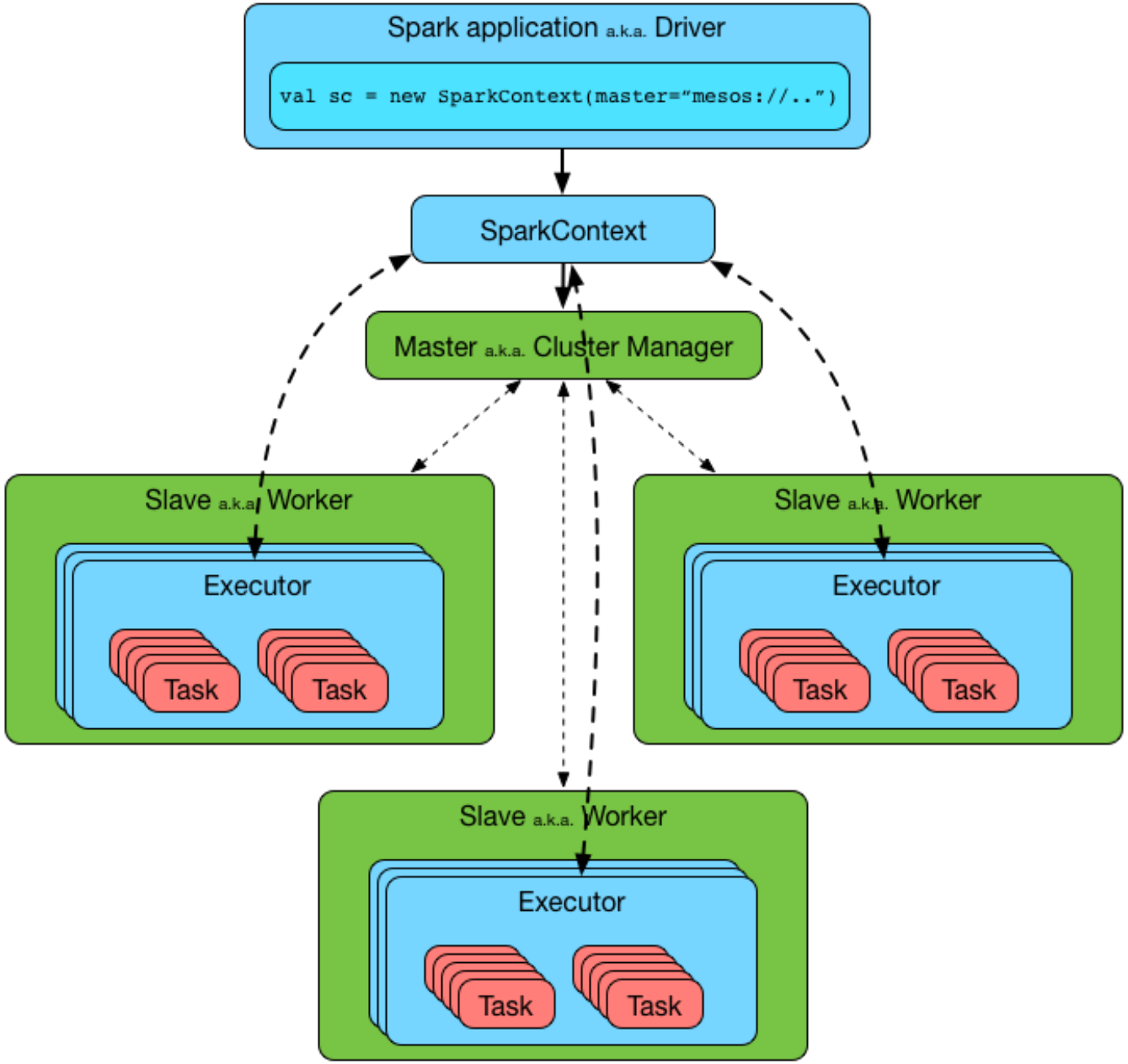
- Spark driver request resources from resource managers [Spark/Yarn/Mesos] for running tasks, parallel processing
- Resource manager should allocate and spawn Spark executor in the cluster and workers

Behind Spark Submit

- Spark driver receives request to run main method of application
- Driver request resource manager for resources to run
- Cluster Manager execute spark job
- Driver runs the spark application, send tasks to executors
- Executors runs the tasks and save the result

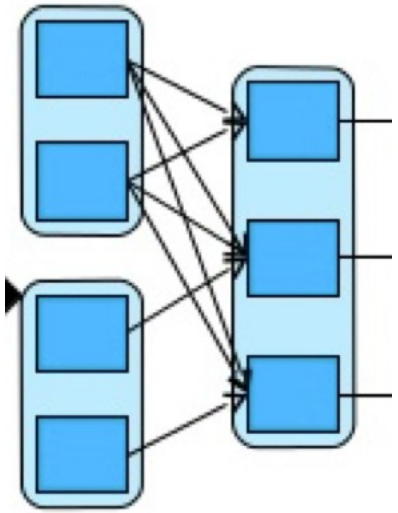
Tasks

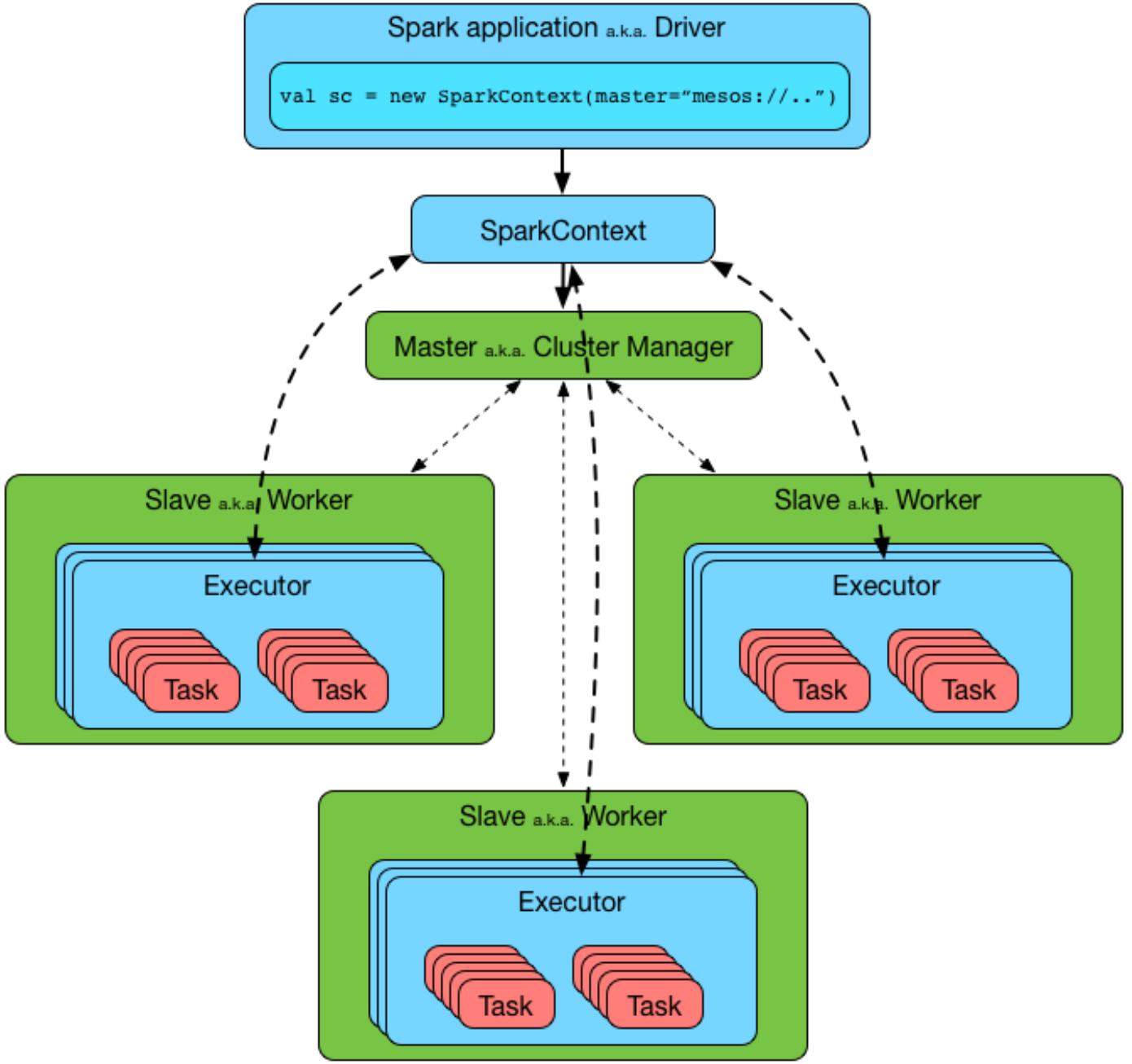
- **Job**: The work required to compute an RDD
- **Stage**: A wave of work within a job, corresponding to one or more pipelined RDD's
- **Tasks**: A unit of work within a stage, corresponding to one RDD partition



Shuffle

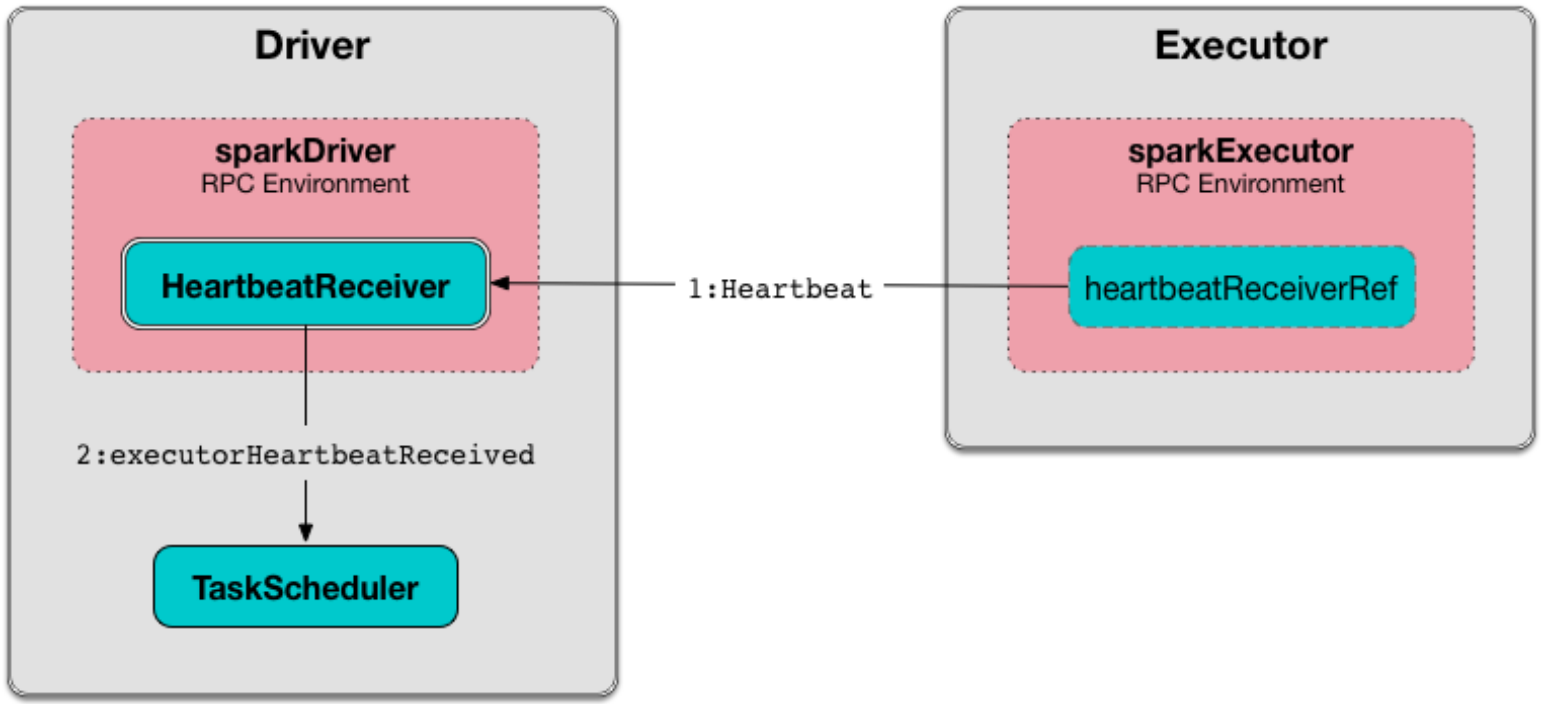
- **Shuffle**: The transfer of data between stages





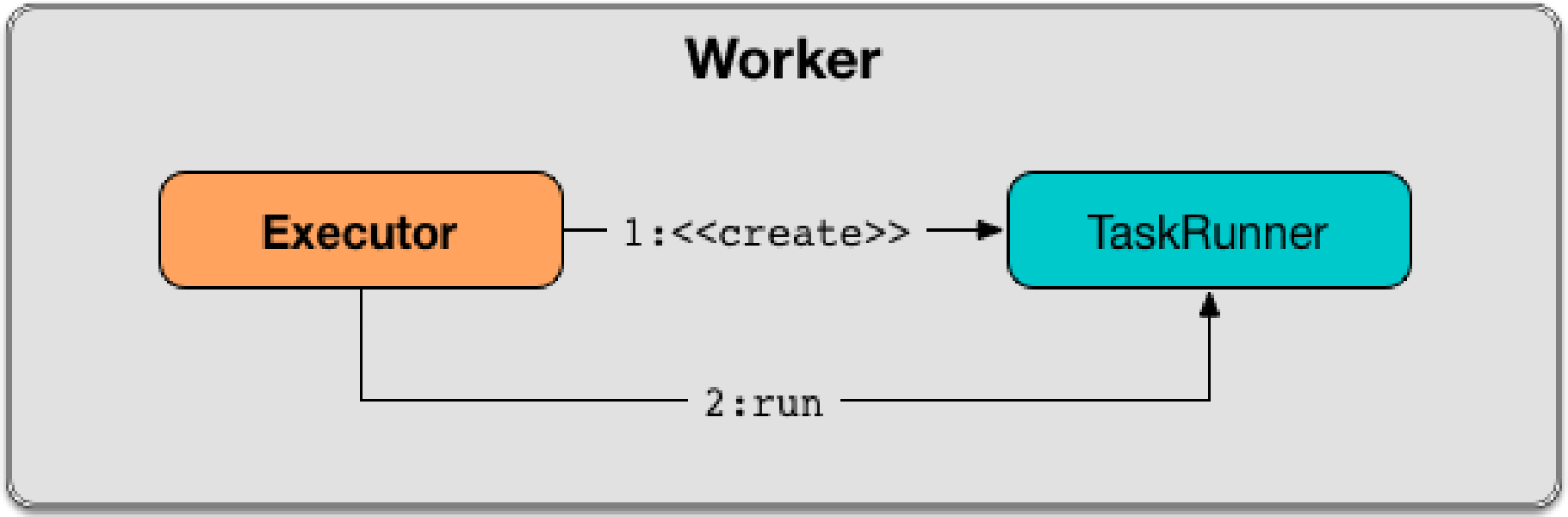
Executor

- Distributed agent execute tasks



TaskRunner

- TaskRunner is a thread of execution that manages a single individual tasks
- TaskRunner is created when executor is requested to launch a job



How it works

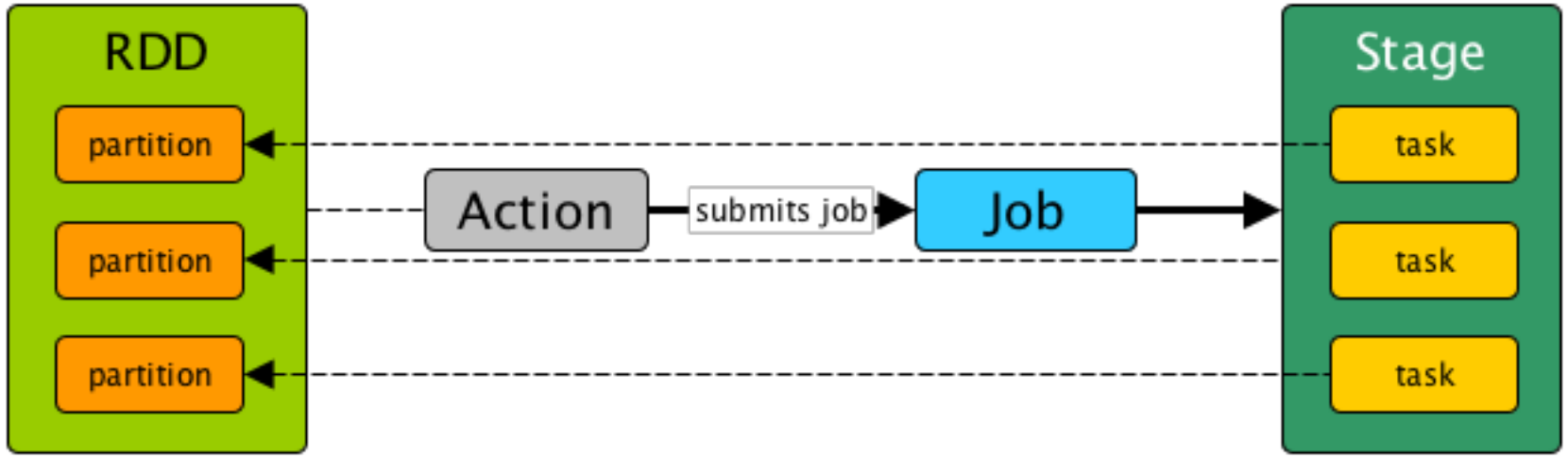
- Spark driver request resources from resource managers [Spark/Yarn/Mesos] for running tasks, parallel processing
- Resource manager should allocate and spawn Spark executor in the cluster and workers

Behind Spark Submit

- Spark driver receives request to run main method of application
- Driver request resource manager for resources to run
- Cluster Manager execute spark job
- Driver runs the spark application, send tasks to executors
- Executors runs the tasks and save the result

Stage

- Stage is a physical unit of execution
- Stage is a set of parallel tasks - one task per partition
- A spark job is a computation, and the computation is sliced into stages



Spark Local

- Also Known as Pseudo Cluster
- Suitable for Development
- All Components ⟹ Driver, Executor, LocalSchedulerBackend and Master runs in single JVM
- Number of threads can be configured

Local Master URL

- local - Single thread only
- local[n] - n is number of thread you specify to run
- local[*] - As many threads on need basis
- local[n, maxFailure] - number of failure allowed before giving up
- Single thread is good for debugging process

Spark Cluster

- Spark Standalone
- Hadoop Yarn
- Apache Mesos

Running Spark on Cluster mode needs Memory and CPU

Spark Cluster Modes

- Cluster Manager provides resources
- Recall from before, for YARN, the resource manager will allocate YARN containers
- Spark's executors run inside these YARN containers
- Executors run computations **and** store data for your application
- Application code (JAR or Python/R scripts to Spark Session) to the executors.
- Driver sends tasks to the executors to run

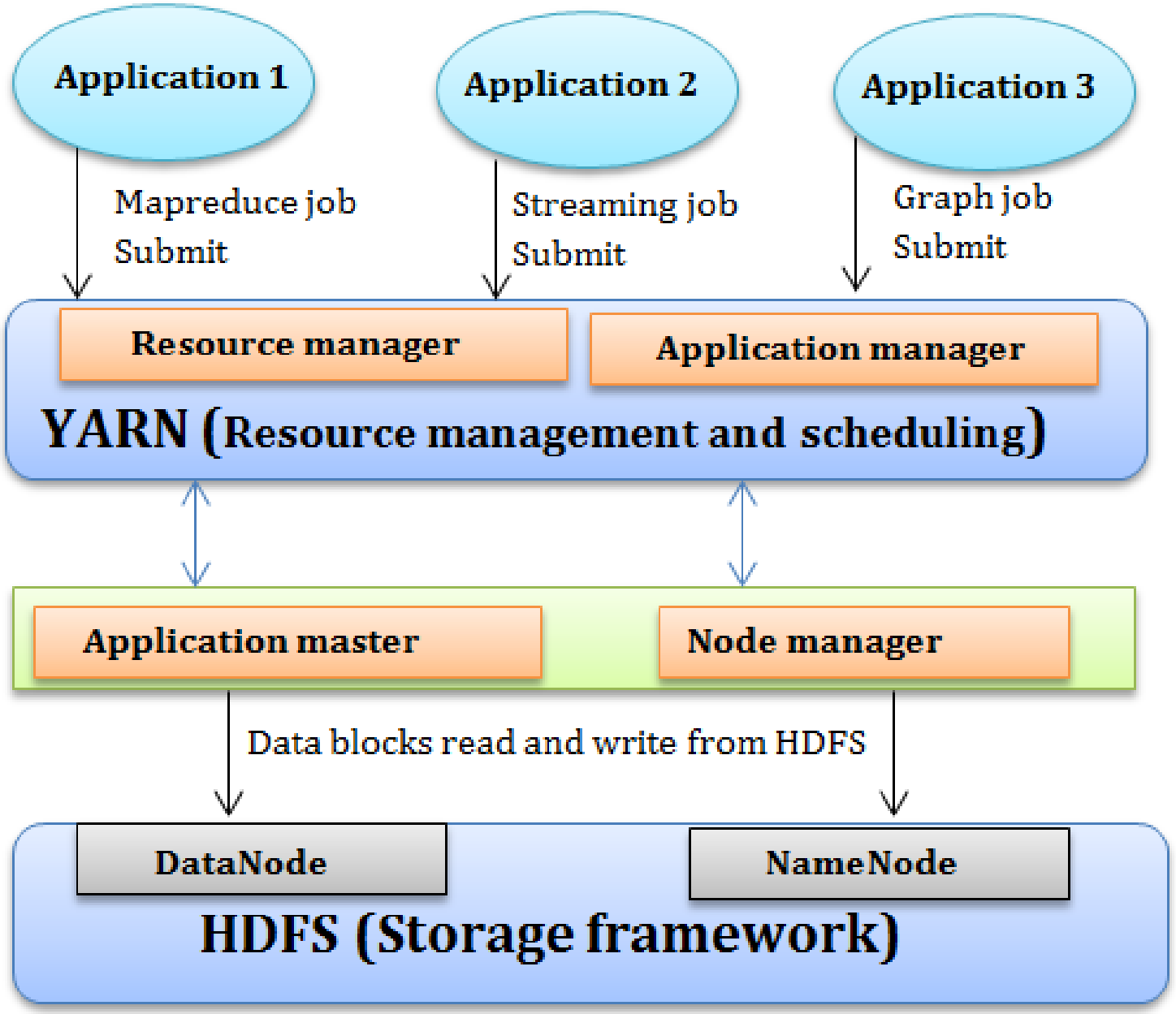
Cluster vs Local

The **master** parameter for the SparkContext determines which type and size of cluster to use.

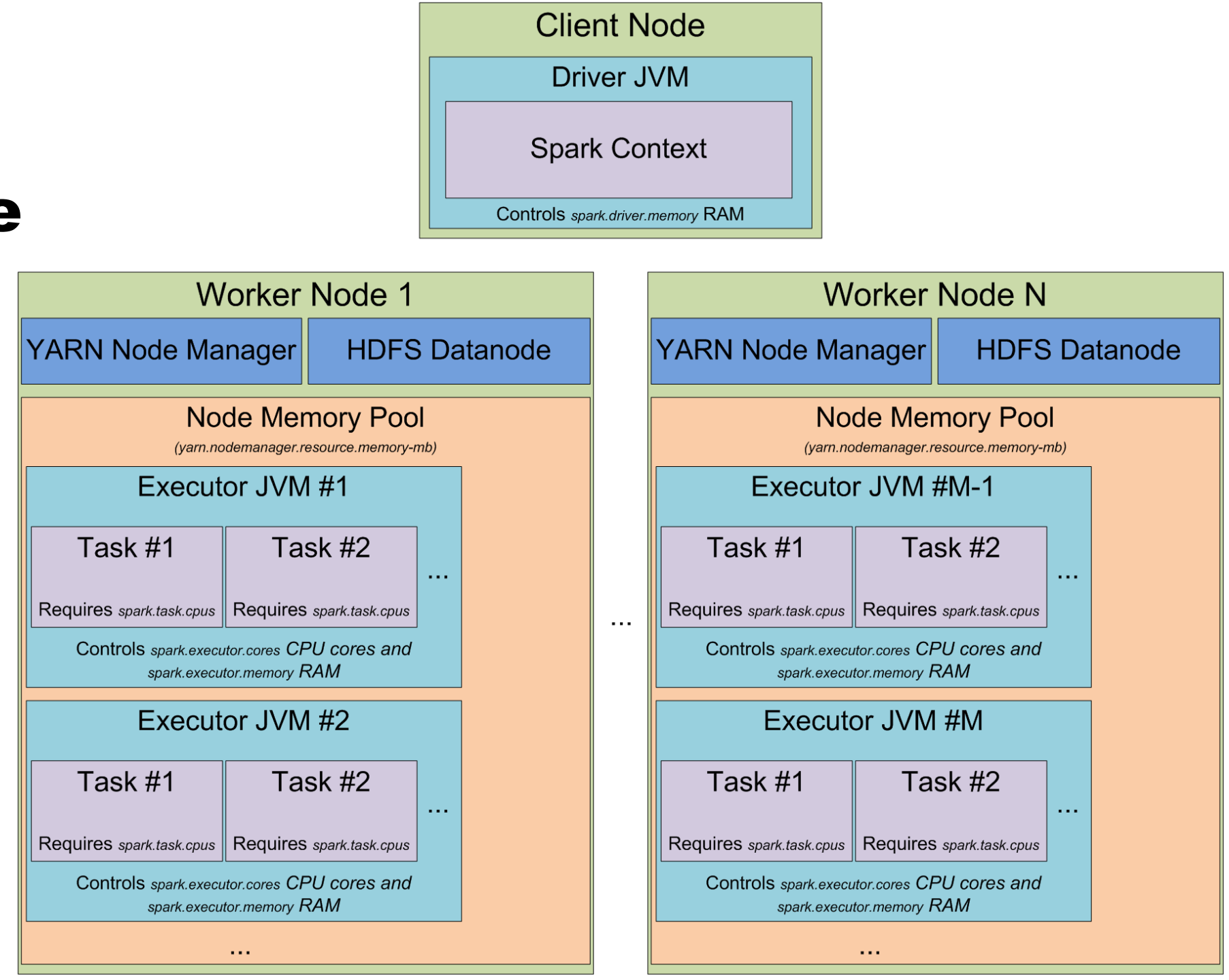
Default: Determined by spark configurations

Value	Description
local	Run locally (one worker)
local[k]	Run locally (k workers)
local[*]	Run locally with as many worker threads as logical cores
spark://{host}:{port}	Connect to spark standalone cluster
mesos://{host}:{port}	Connect to Mesos cluster
yarn-client	Connect to YARN cluster in client mode
yarn-cluster	Connect to YARN cluster in cluster mode

YARN Architecture



YARN Architecture



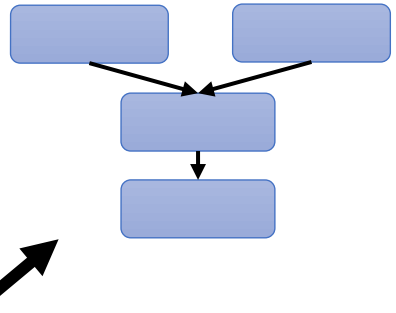
Cluster vs Local (yarn-client vs yarn-cluster)

- In **yarn-client** mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.
- If the local process is killed it kills the job

- In **yarn-cluster** mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster.
- If the local process is kill the job will still run and complete

Job scheduling

RDD Objects

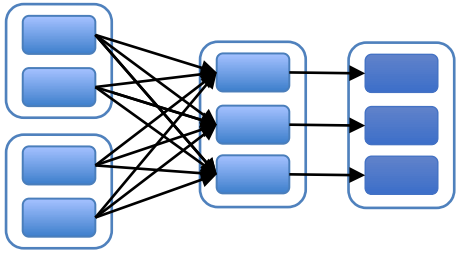


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG

DAGScheduler



split graph into *stages* of tasks
submit each stage as ready

TaskSet

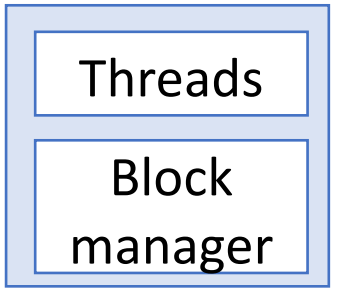
TaskScheduler

Cluster manager

launch tasks via cluster manager
retry failed or straggling tasks

Task

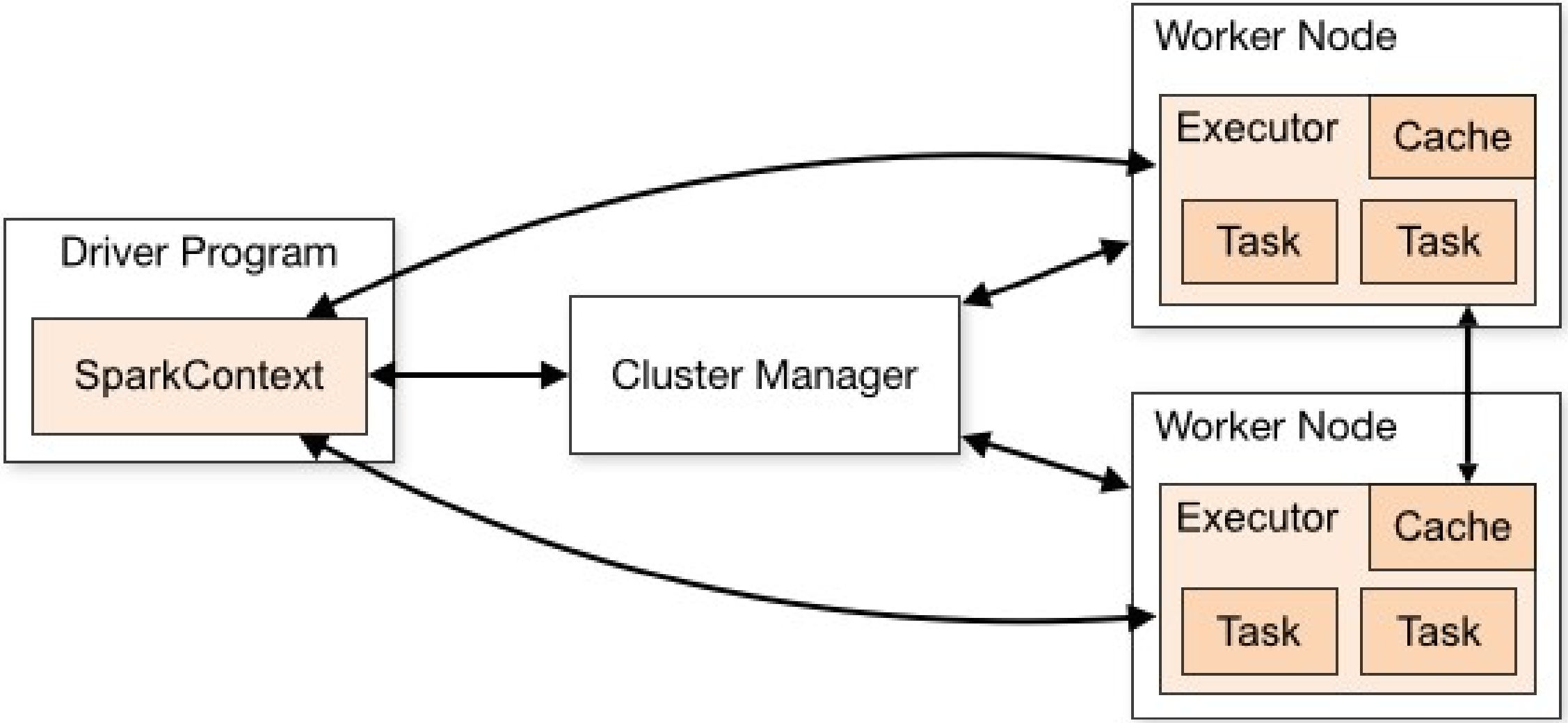
Worker



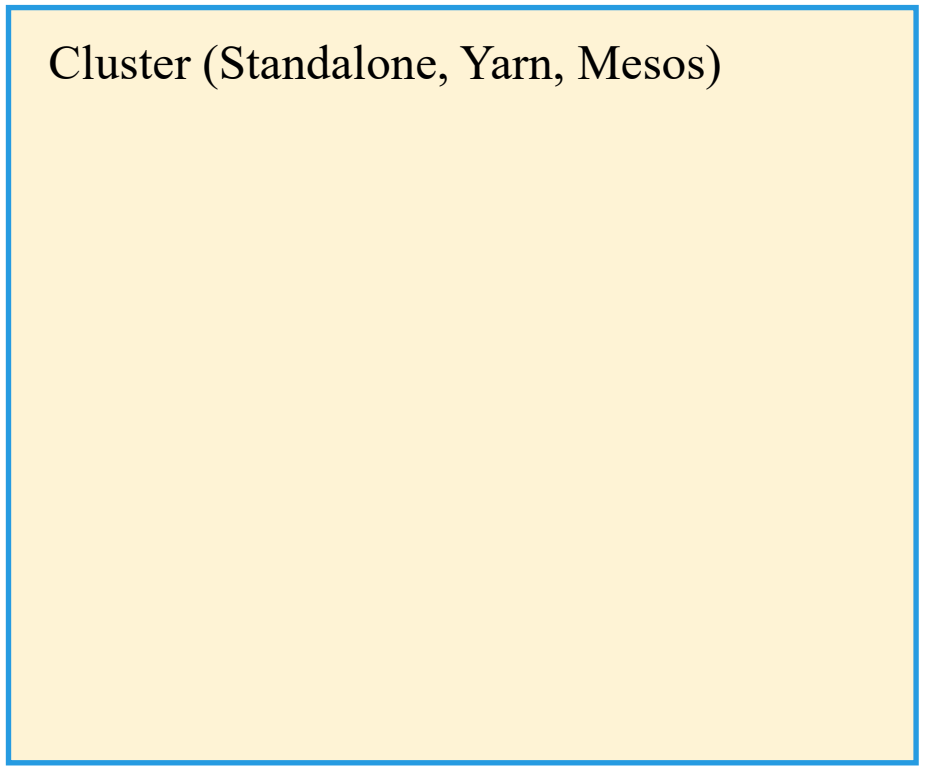
execute tasks

store and serve blocks

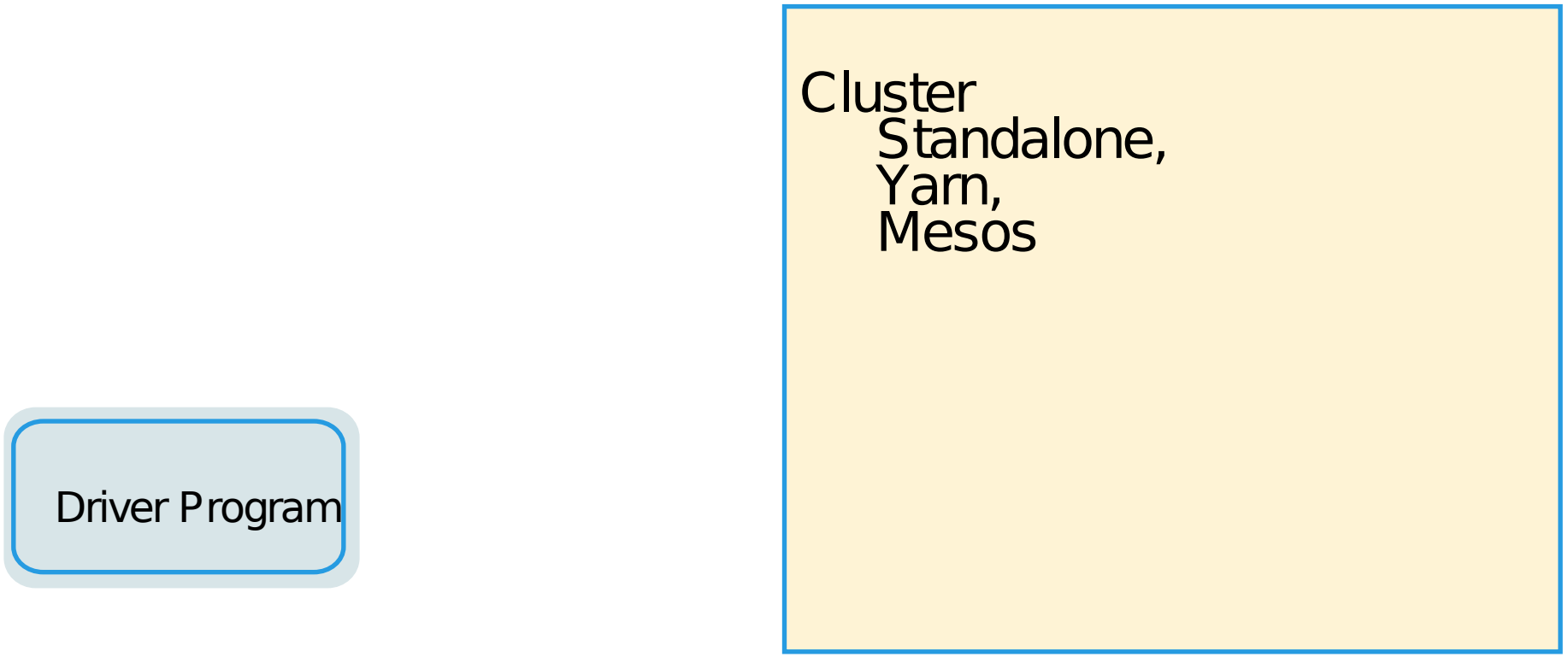
Spark Drivers and Workers



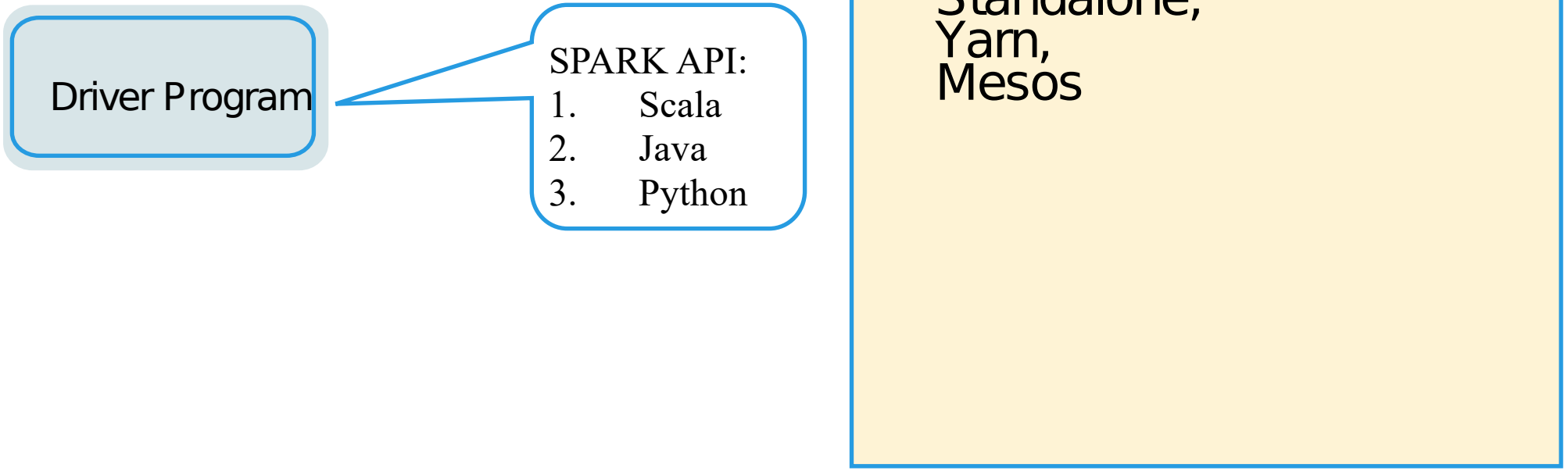
The Big Picture



The Big Picture

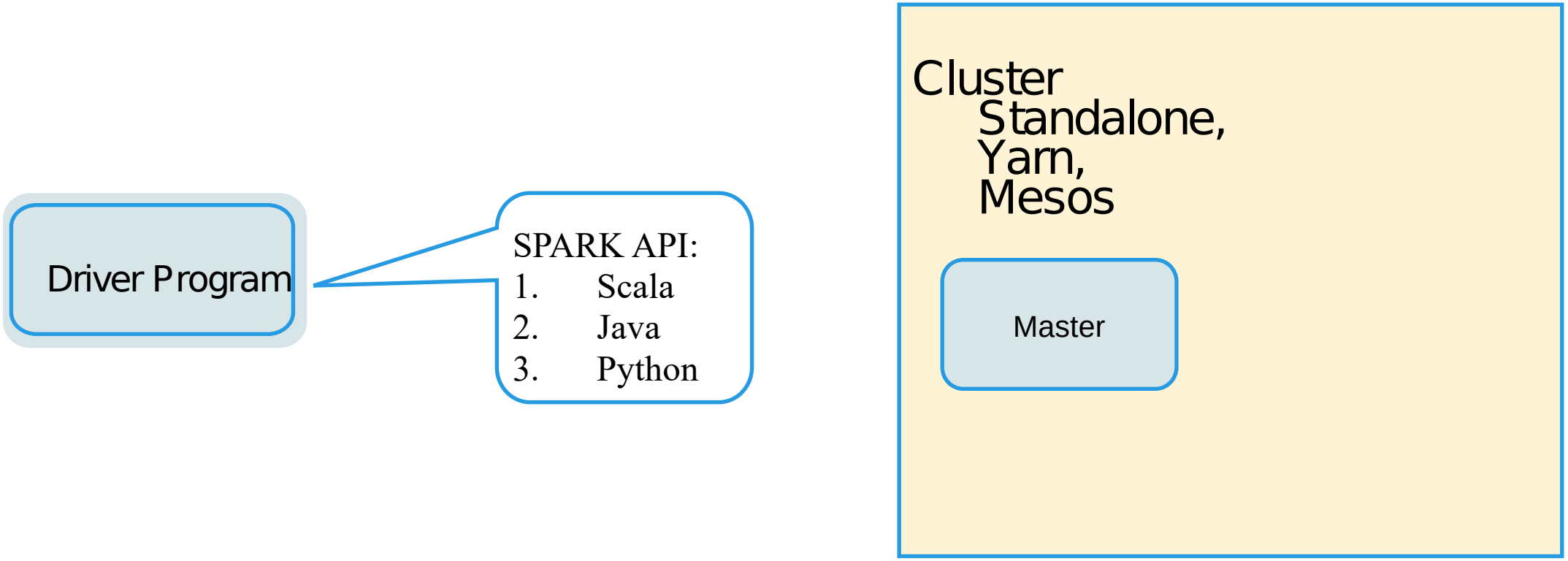


The Big Picture

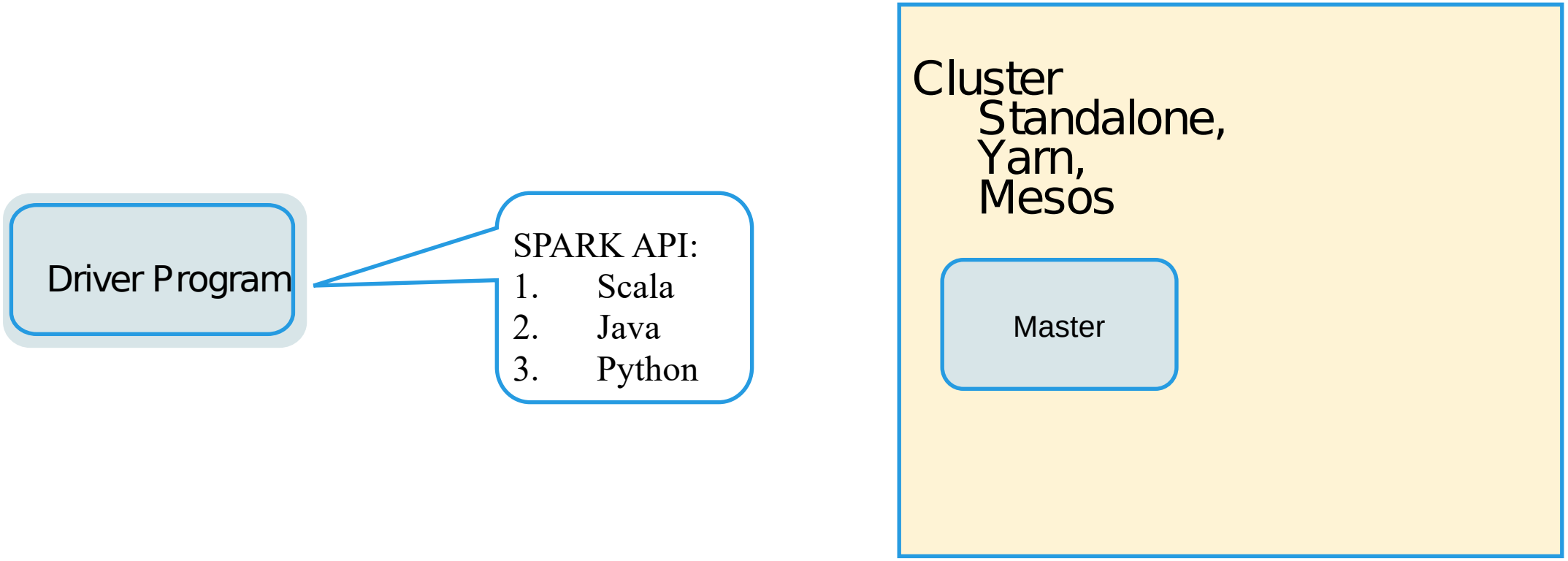


```
val master = "spark://host:pt"
```

The Big Picture

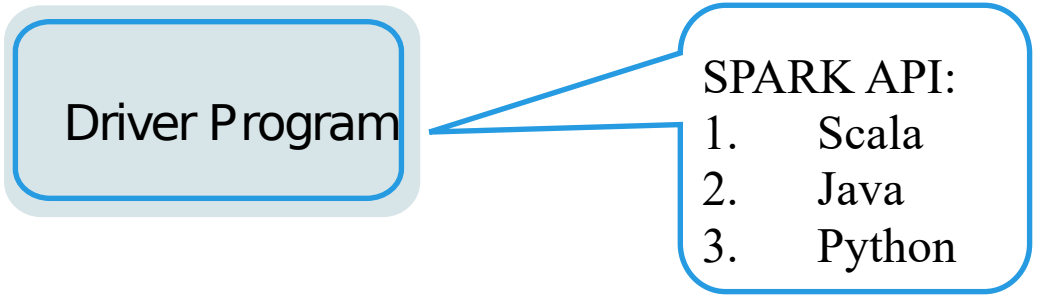


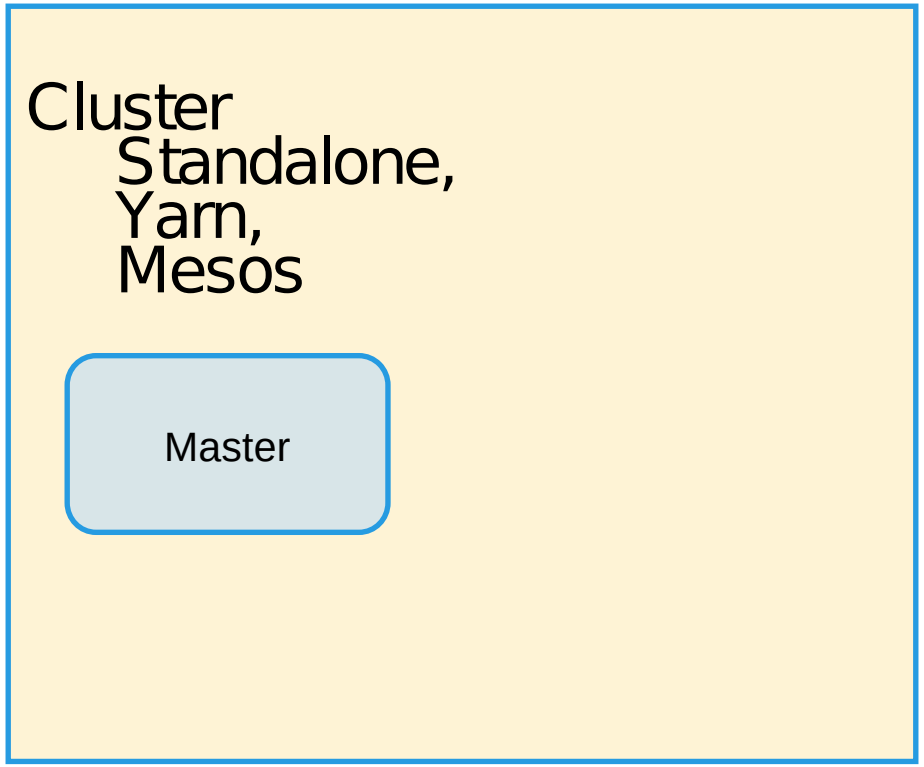
```
val master = "spark://host:pt"
val conf = new SparkConf()
           .setMaster(master)
```



The Big Picture

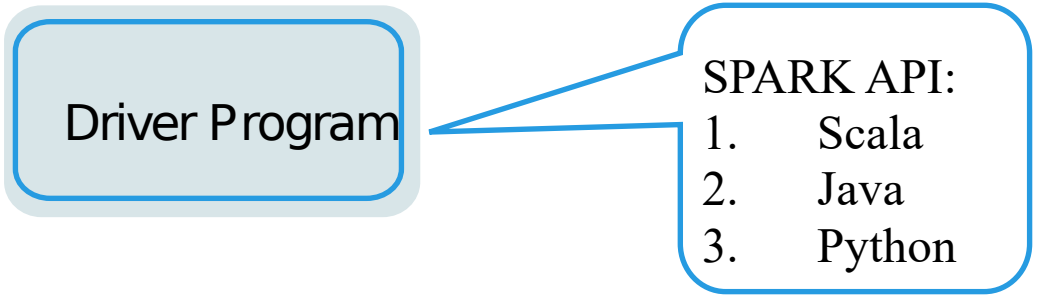
```
val master = "spark://host:pt"
val conf = new SparkConf()
           .setMaster(master)
val sc = new SparkContext(conf)
```

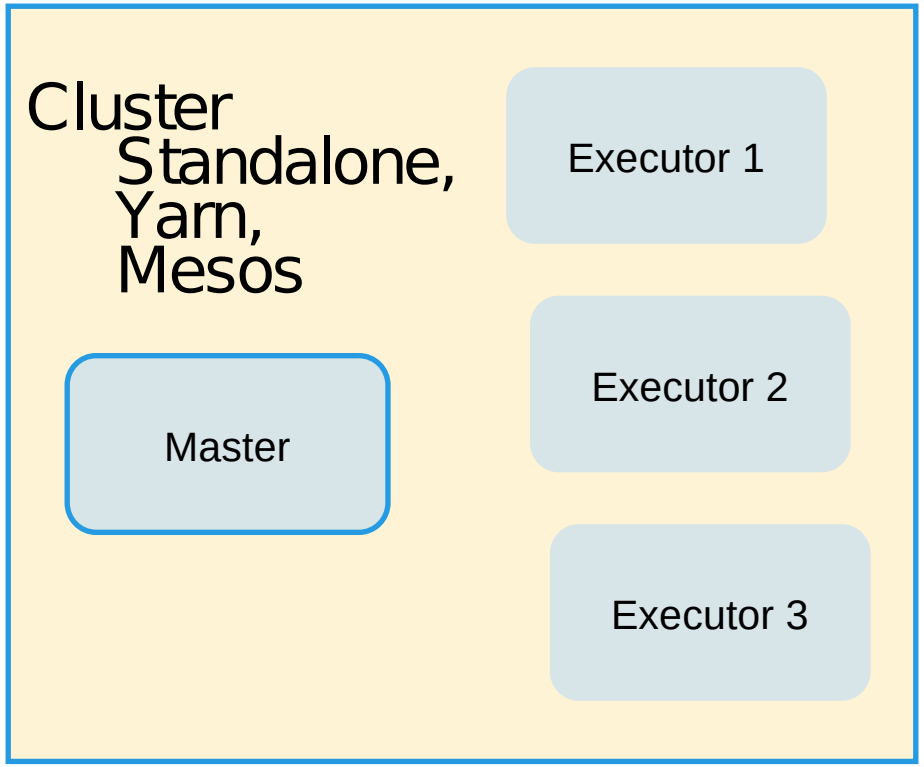




The Big Picture

```
val master = "spark://host:pt"
val conf = new SparkConf()
           .setMaster(master)
val sc = new SparkContext(conf)
```



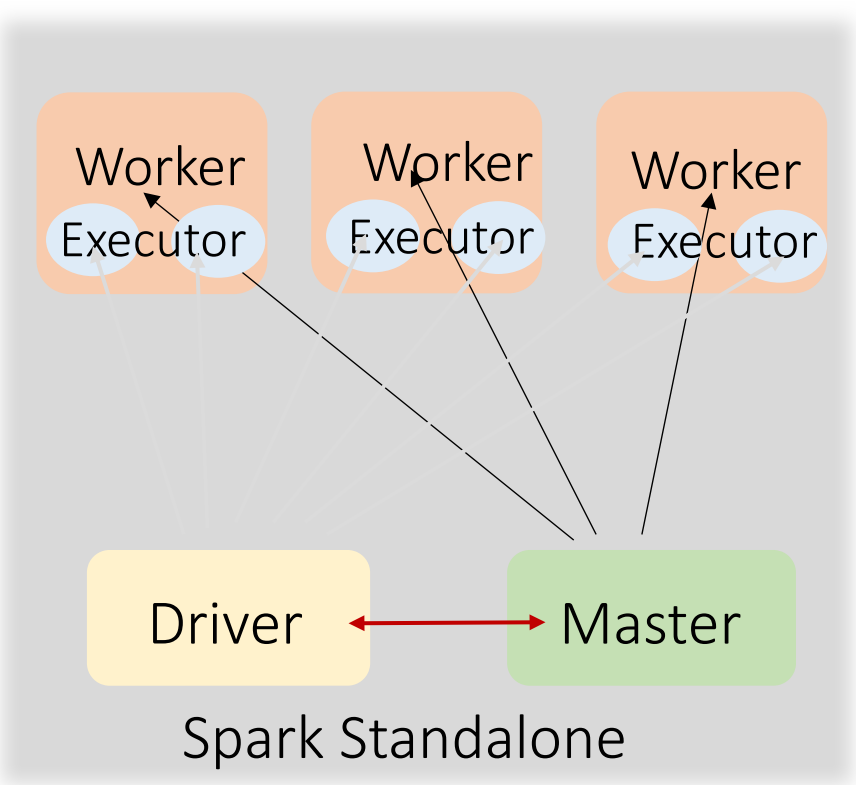


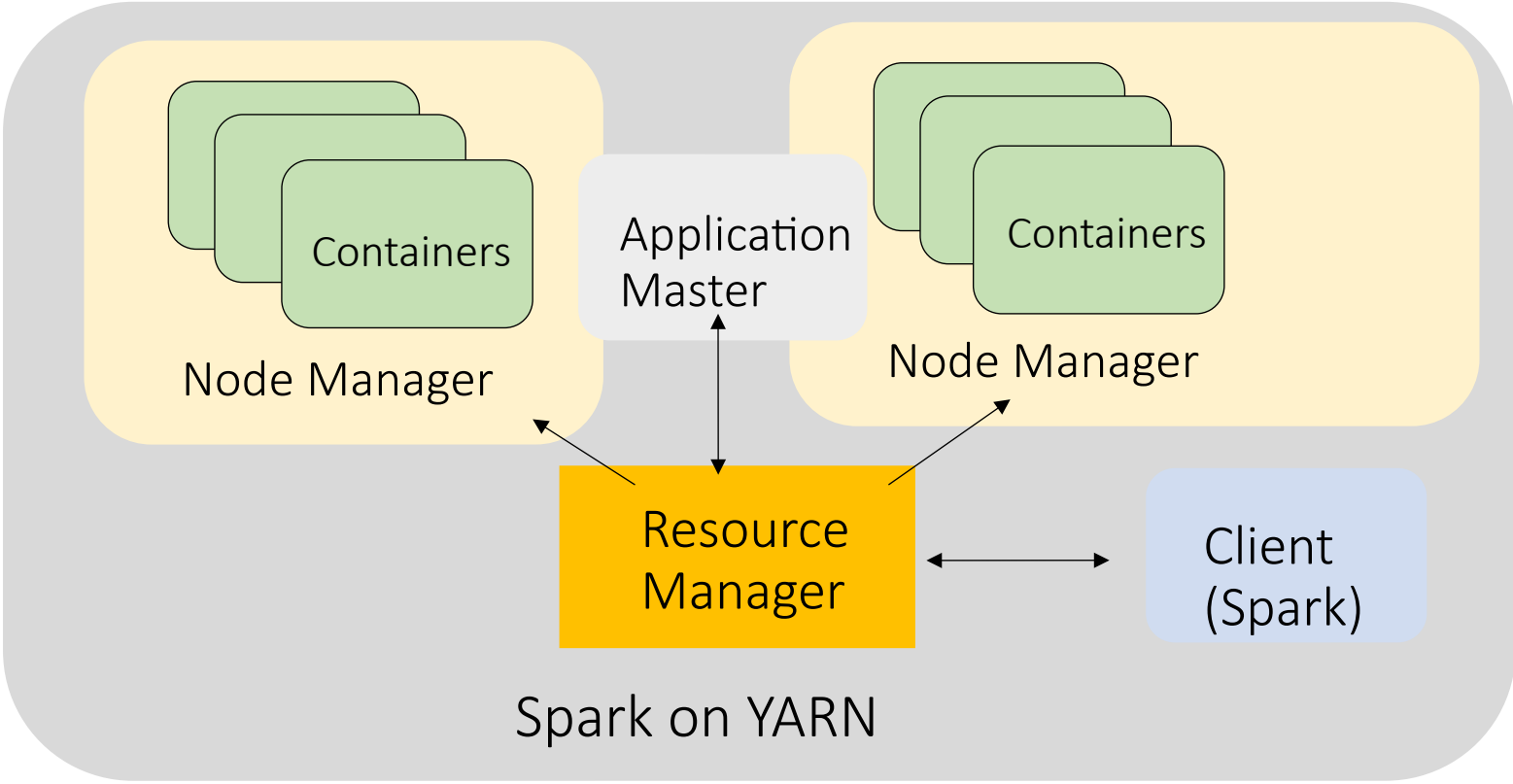
RDD

- Resilient Distributed Datasets
- Scala Collection like APIs
- Lazy evaluation
- Compile Time type-safe
- Based and Similar to Scala Collection APIs

Spark Cluster Modes

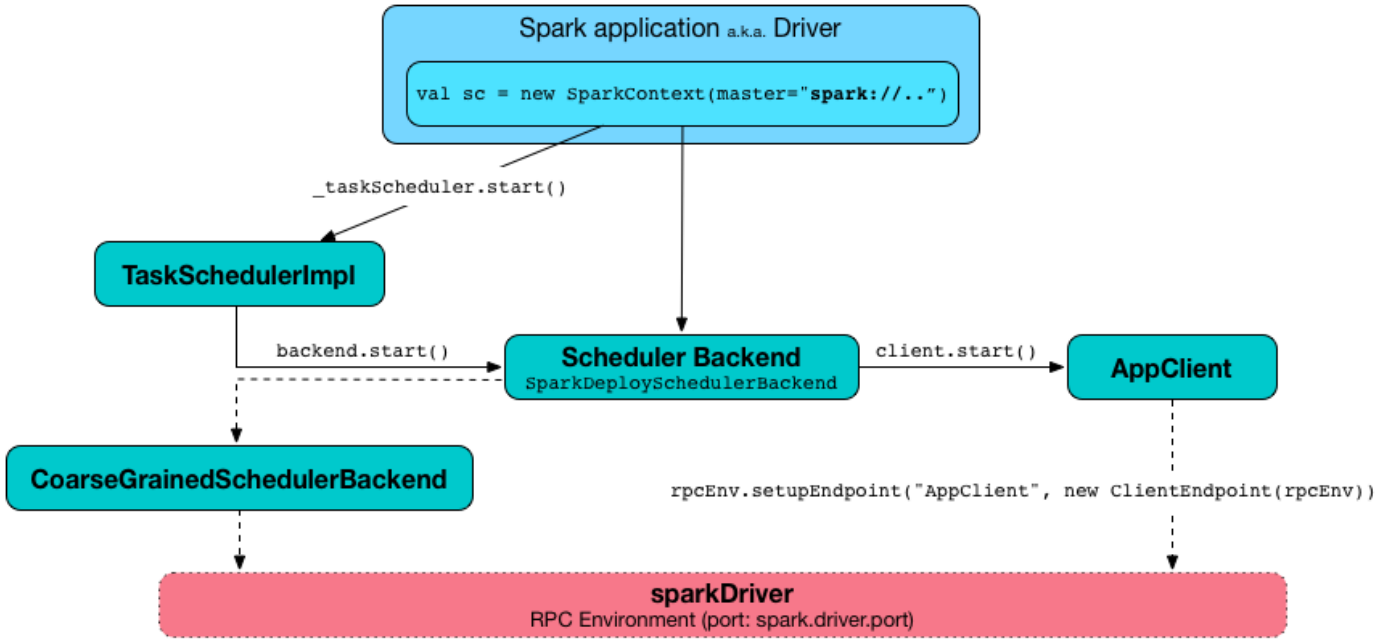
- As mentioned before, Spark users a cluster manager to acquire resources for execution





Launching the Spark In Standalone Mode

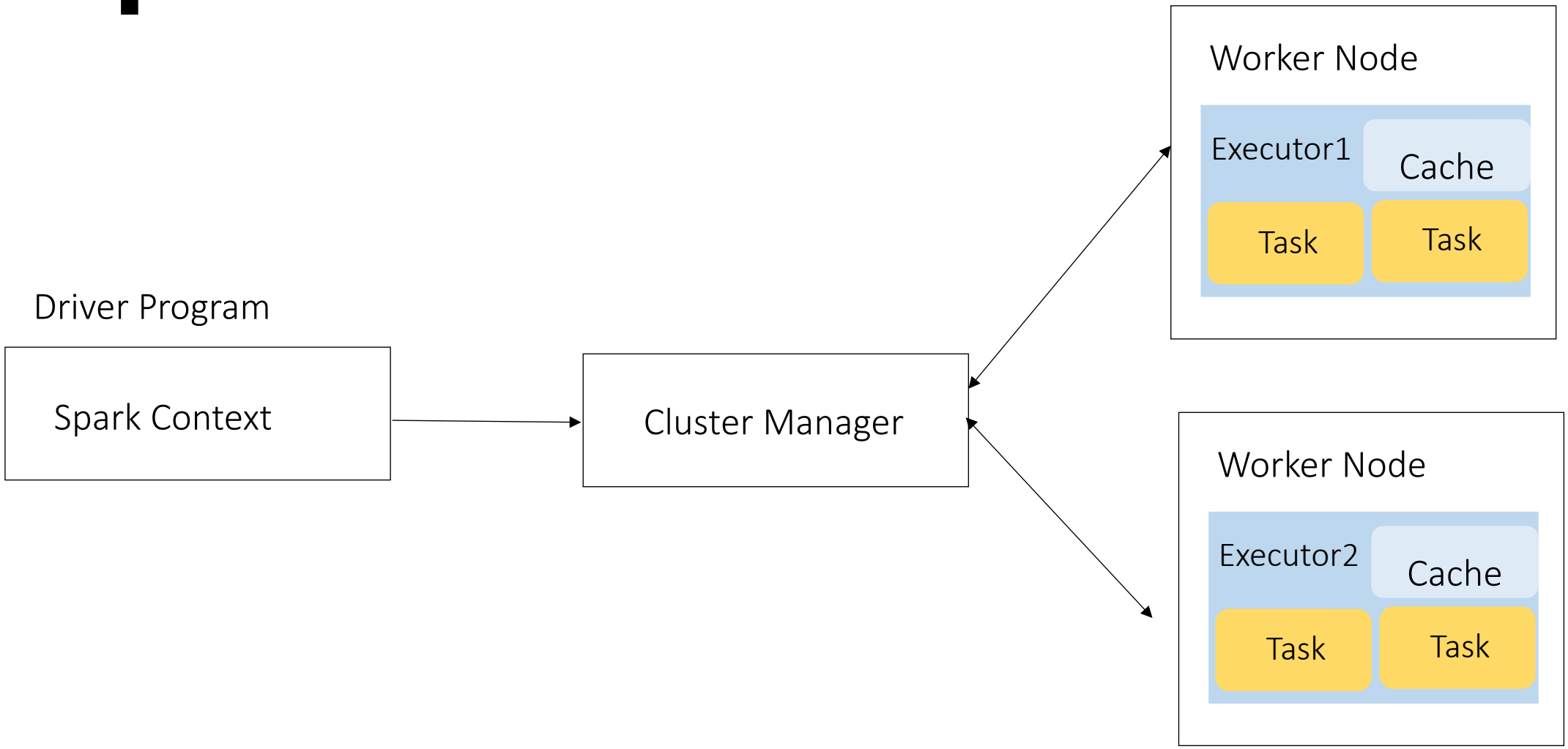
- In standalone mode, Spark uses a Master daemon to coordinate with workers, which run the executors.
- Standalone mode is the default, but cannot be used on secure clusters.
- When you submit an application, you can choose how much memory its executors will use, as well as the total number of cores across all executors.



Spark on YARN Mode

- In YARN mode, the YARN ResourceManager performs the functions of the Spark Master.
- The functions of the workers are performed by the YARN NodeManager daemons, which run the executors.
- YARN mode is a bit more complex than standalone mode, but can be much more secure
- Two modes of running a Spark Application in YARN: client mode, cluster mode
- Client Mode: runs in the foreground (`-deploy-mode cluster`)
- Cluster Mode: runs in the background (`-deploy mode client`)
  - A little more complex than client mode, but allows user to log out without terminating the application

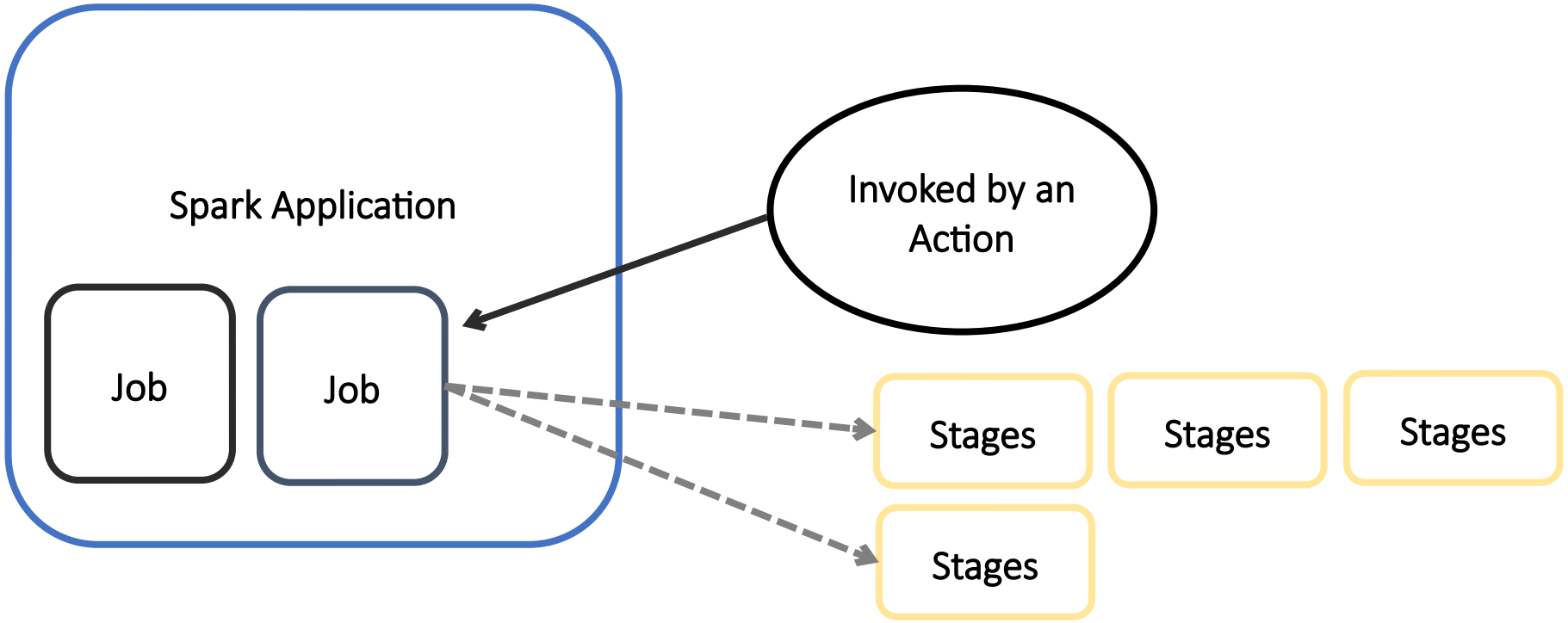
Spark Run Time Architecture



Physical Execution of an Application

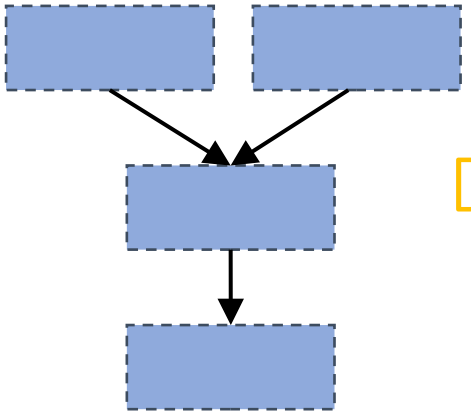
- When you submit a Spark Application, the program gets translated into a physical execution

Application-> Jobs-> Stages-> Tasks



Scheduling Process

RDD Objects

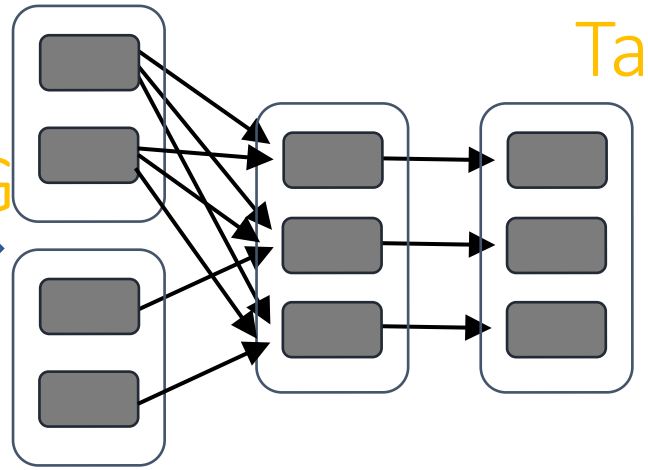


Rdd1.join(rdd2)
.groupBy(…)
.filter(…)

- Build operator DAG

DAG Scheduler

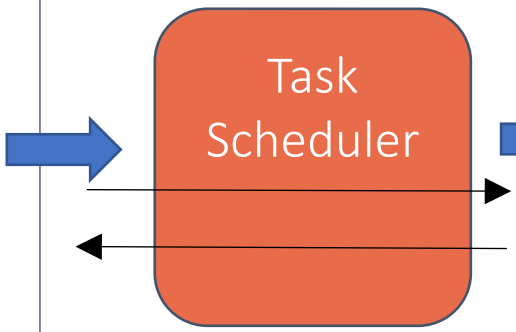
DAG



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

Task Scheduler

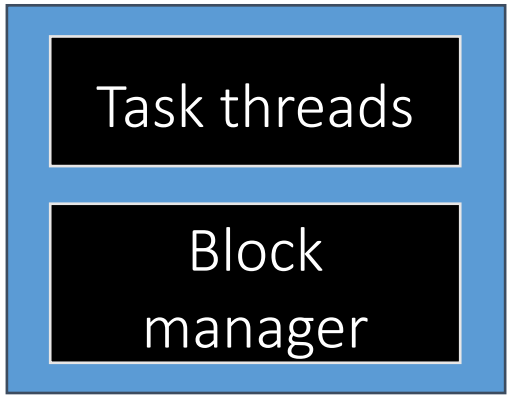
TaskSet



- Launches individual tasks
- Retry failed or straggling tasks

Executor

Task

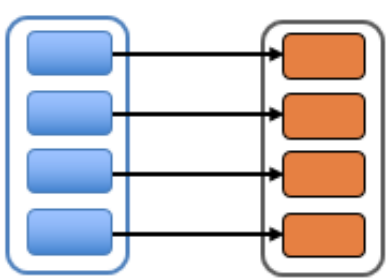


- Execute tasks
- Store and serve blocks

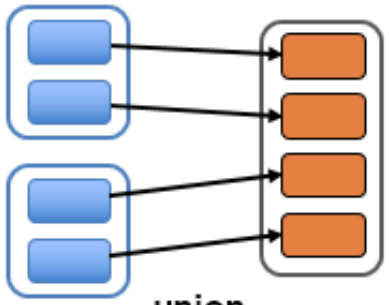
Narrow vs. Wide Dependencies

narrow

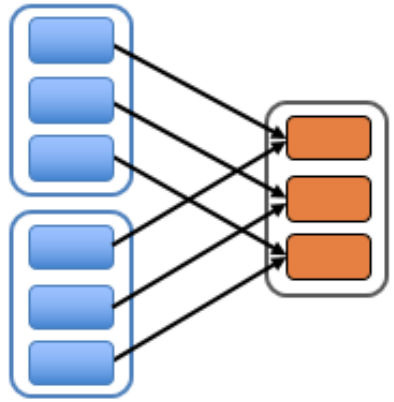
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



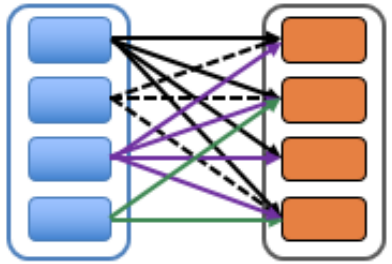
union



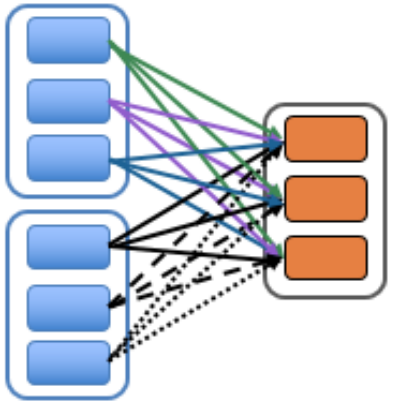
join w/ inputs co-partitioned

wide

multiple child RDD partitions may depend on a single parent RDD partition

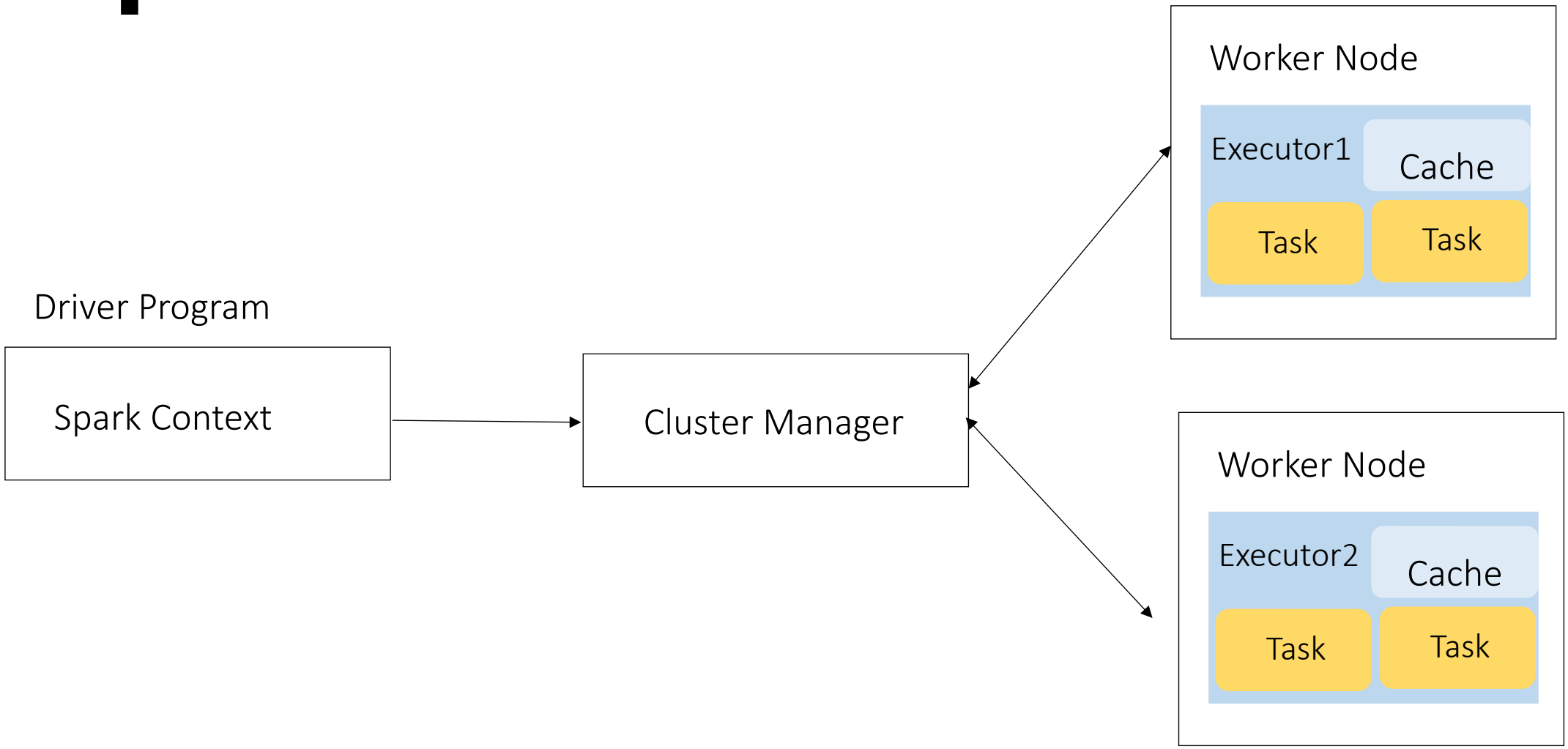


groupByKey



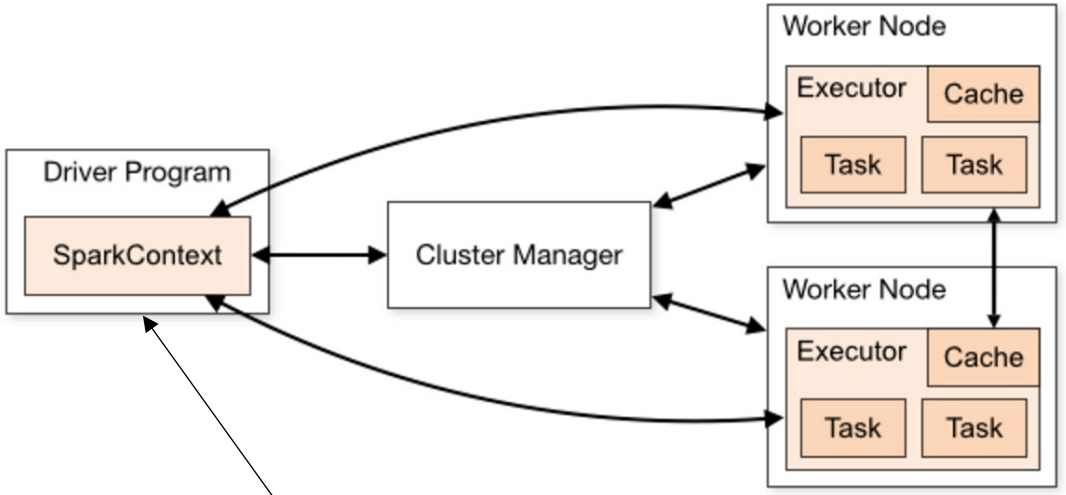
join w/ inputs not co-partitioned

Spark Run Time Architecture



Spark Applications

- Spark Applications can contain a variety of workloads and parameters
- But they always have the following configuration
- **A driver process**
  1. **Maintaining information about the Spark application**
  2. **Responding to a user's program**
  3. **Analyzing, distributing, and scheduling work across the executors**
- A **set of executor** processes
  1. Executing code assigned to it by the driver program
  2. Reporting the state of the computation back to the driver program



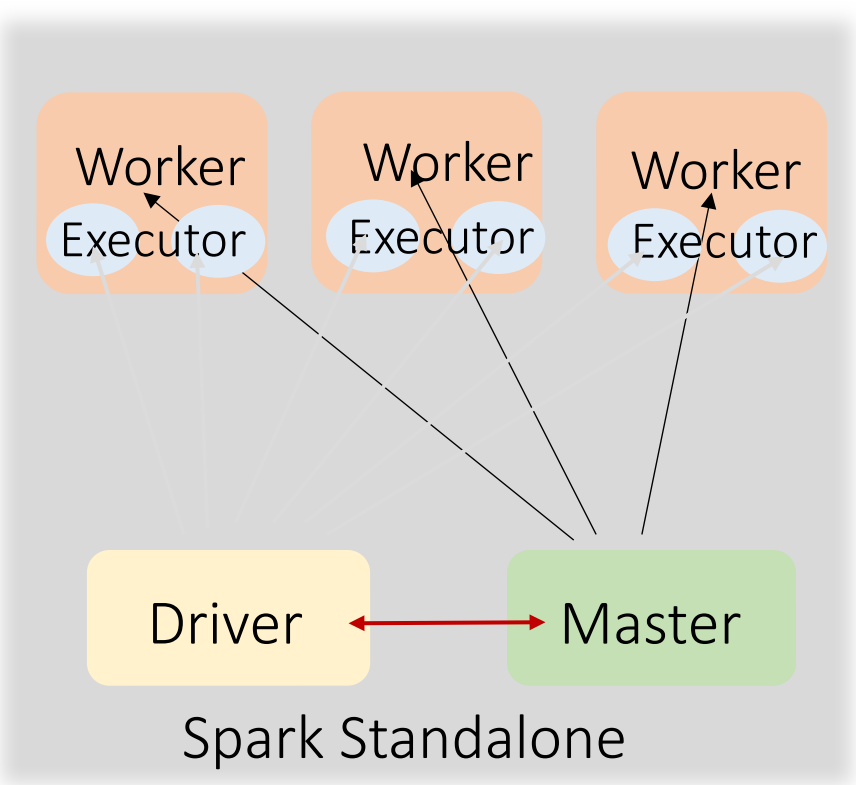
Note:

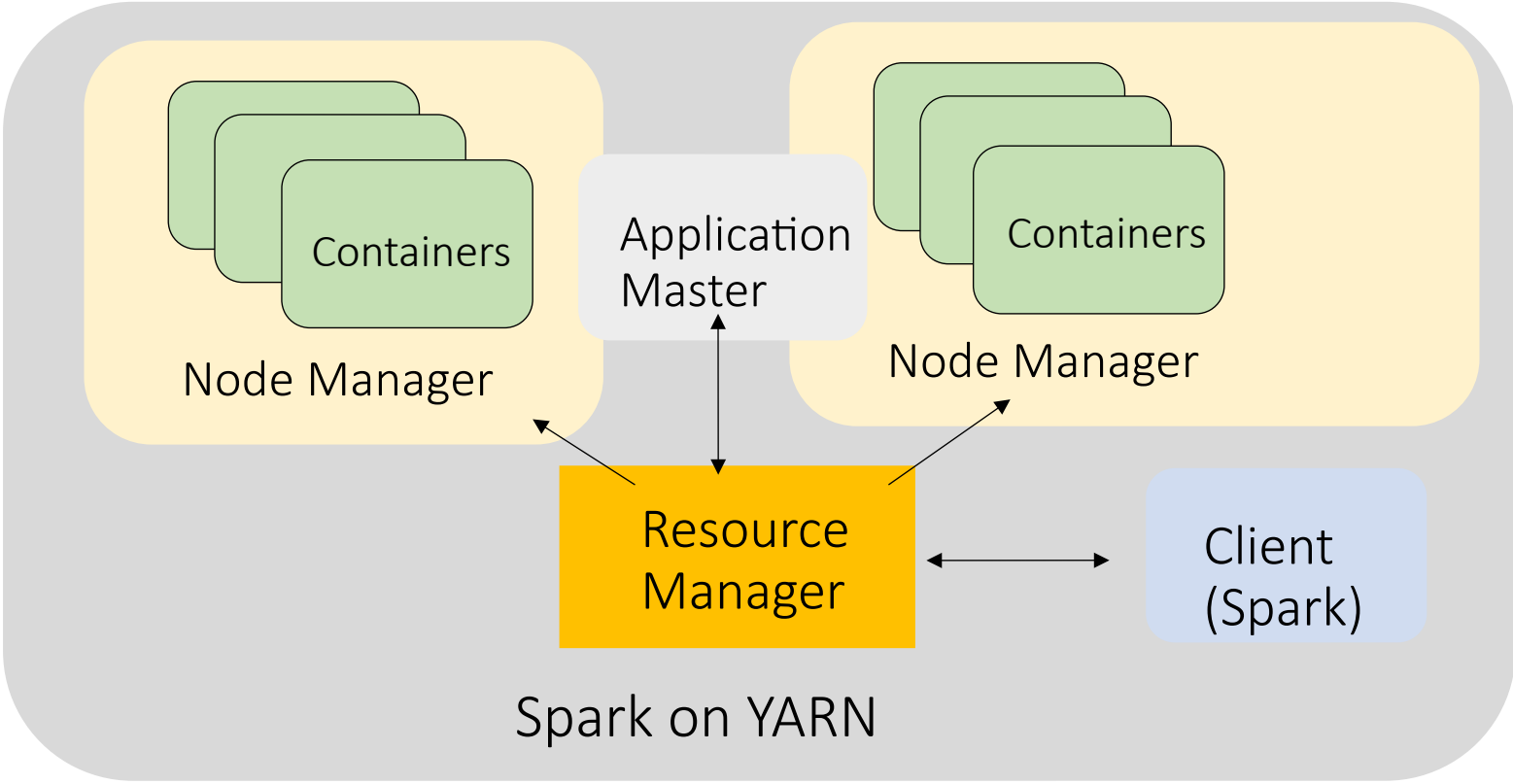
In YARN, you have two configuration modes:

1. *Cluster-mode*: driver runs in application master on one of the nodes
2. *Client-mode*: driver runs in client, AM only needed for YARN requests

Spark Cluster Modes

- As mentioned before, Spark users a cluster manager to acquire resources for execution



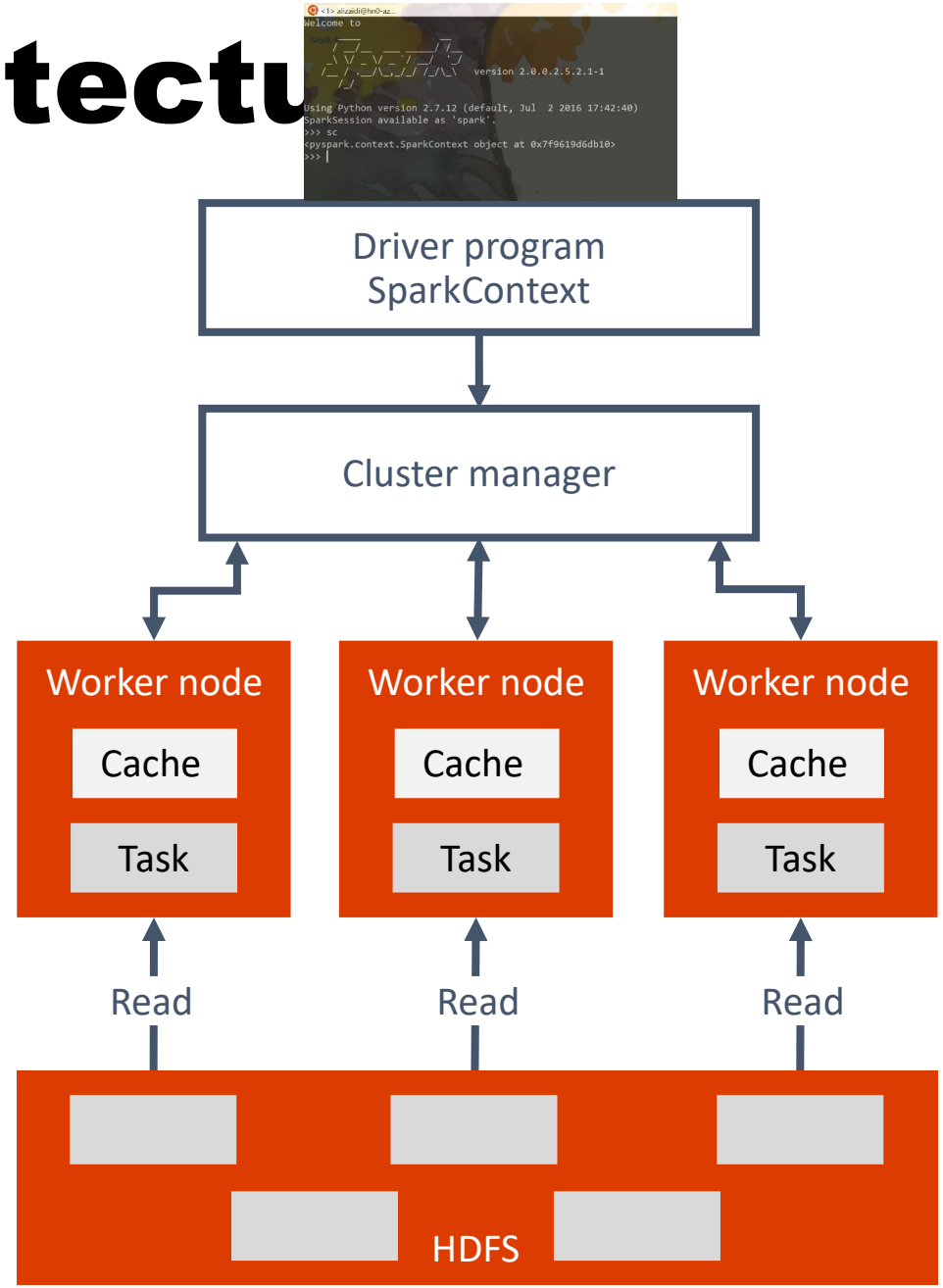


Spark on YARN Mode

- In YARN mode, the YARN ResourceManager performs the functions of the Spark Master.
- The functions of the workers are performed by the YARN NodeManager daemons, which run the executors.
- YARN mode is a bit more complex than standalone mode, but can be much more secure
- Two modes of running a Spark Application in YARN: client mode, cluster mode
- Client Mode: runs in the foreground (`-deploy-mode cluster`)
- Cluster Mode: runs in the background (`-deploy mode client`)
  - A little more complex than client mode, but allows user to log out without terminating the application

Spark cluster architectu

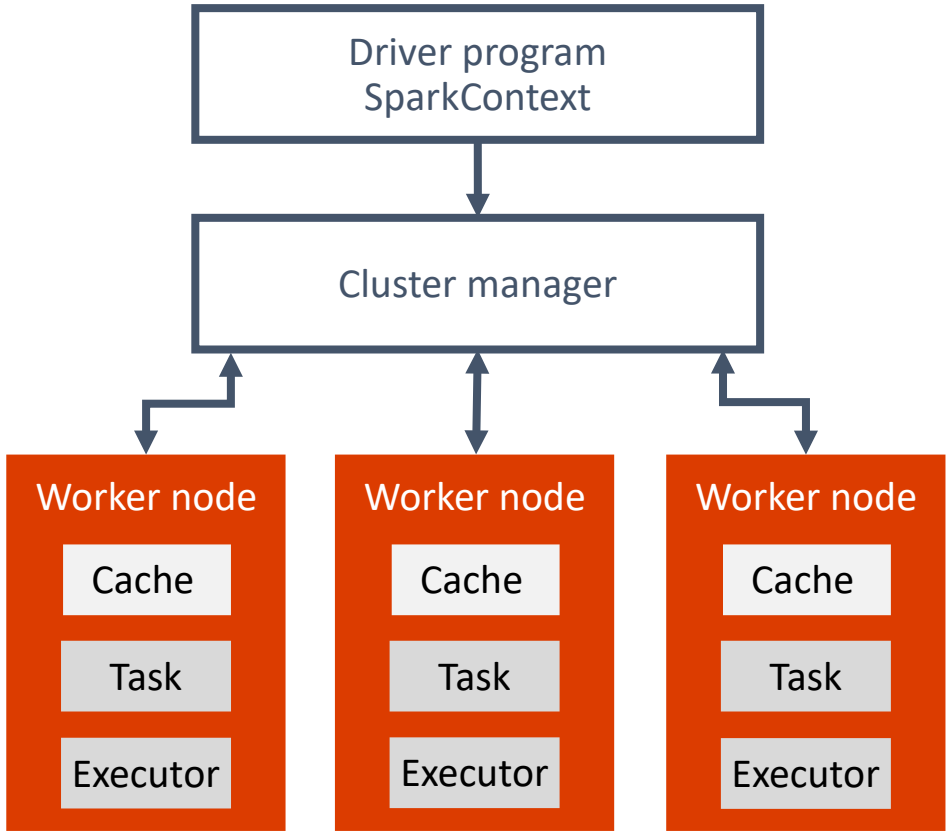
- The driver runs the user's main function and executes the various parallel operations on the worker nodes.
- The driver collects the results of the operations.
- Worker nodes read and write data from/to Hadoop distributed file system (HDFS).
- Worker nodes also cache transformed data in-memory as RDDs.

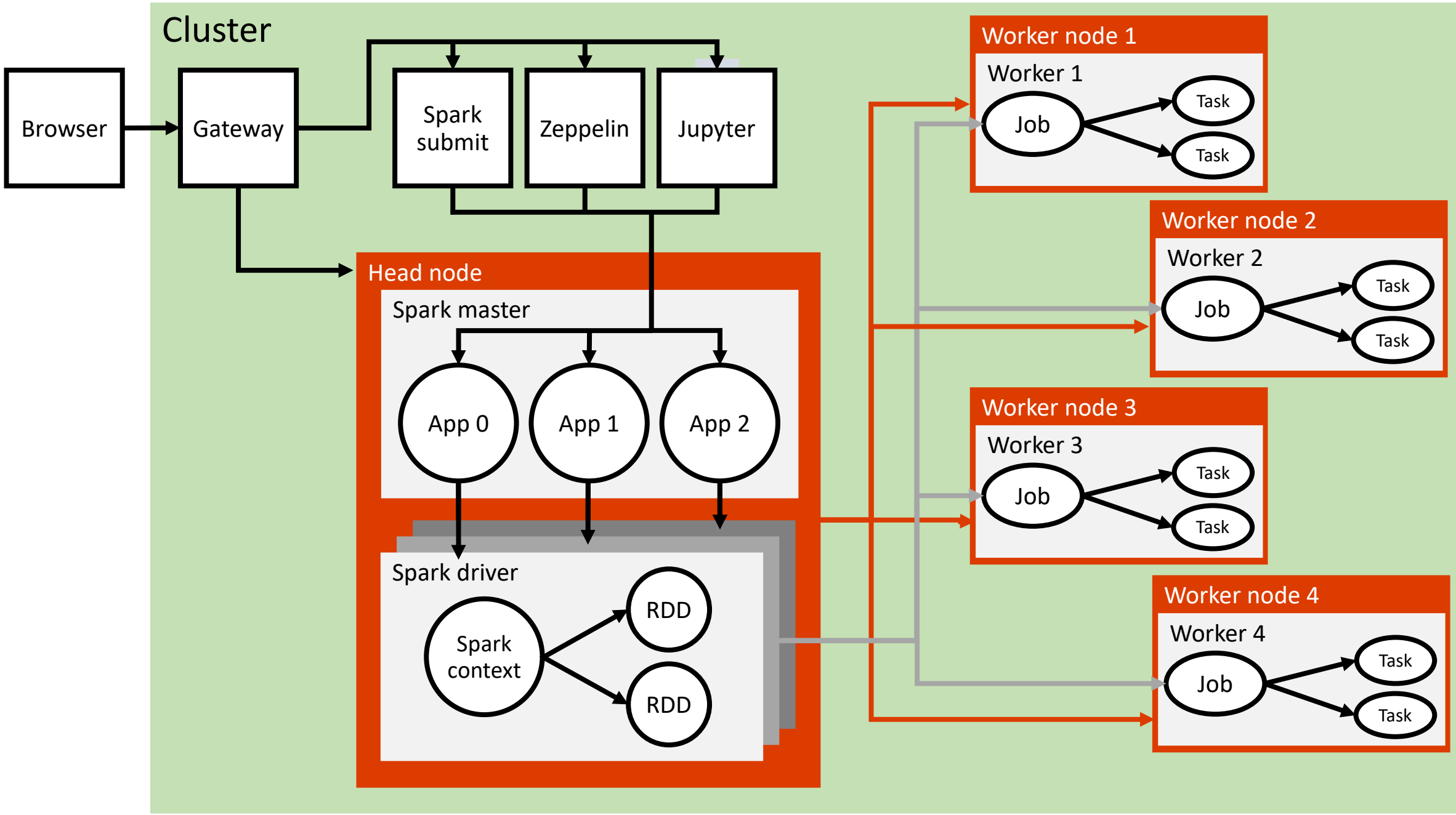


The Spark cluster architecture driver

The driver performs the following:

- Connects to a cluster manager to allocate resources across applications
- Acquires executors on cluster nodes while processes run compute tasks and cache data
- Sends app code to the executors
- Sends tasks for the executors to run




Cluster
Browser
Gateway
Spark submit
Zeppelin
Jupyter
Head node
Spark master
App 0
App 1
App 2
Spark driver
Spark context
RDD
RDD
Worker node 1
Worker 1
Job
Task
Task
Worker node 2
Worker 2
Job
Task
Task
Worker node 3
Worker 3
Job
Task
Task
Worker node 4
Worker 4
Job
Task
Task

Scheduler

- Uses RDD lineage to find efficient execution plan for action
  - Optimizes operations by collapsing down inline narrow dependencies into one task instead of doing individual tasks for each operation
- Schedules tasks based on data locality
- Takes into account which RDDs are in cache
  - If a task needs to process a cached partition, then the task is started on the node where the data is cached
- Tasks are launched to compute missing partitions

Worker Node : Node that run the application program in cluster

- Executor

– Process launched on a worker node, that runs the Tasks

– Keep data in memory or disk storage

- Task : A unit of work that will be sent to executor
- Job

– Consists multiple tasks

– Created based on a Action

- Stage : Each Job is divided into smaller set of tasks called Stages that is sequential
and depend on each other
- SparkContext :

– represents the connection to a Spark cluster, and can be used to create RDDs,
accumulators and broadcast variables on that cluster