Week 5-1

Spark

Big Data

Prof. Hwanjo Yu

Spark

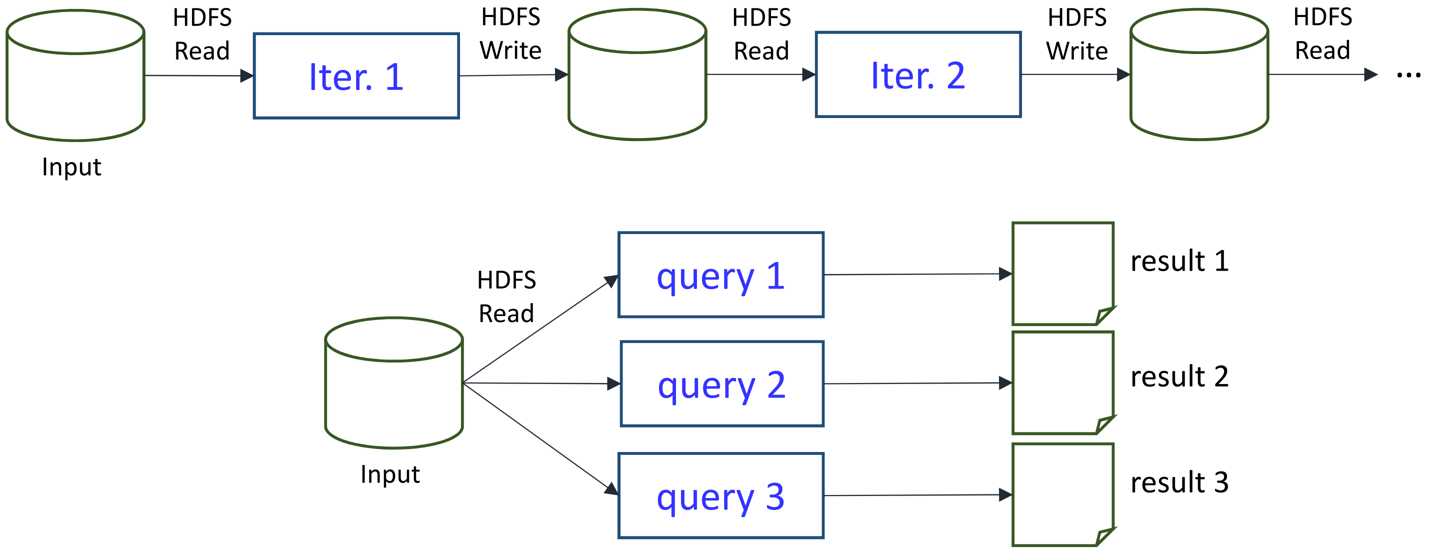
- Spark is a *fast* and *general* engine for large-scale data processing
  - In-memory data processing
  - Highly efficient distributed operations
- Open source cluster computing framework
- Originally developed in the AMPLab at Berkley from 2009
  - Apache Project

Motivation

- MapReduce
  - Force your data analysis workflow into Map and Reduce steps
    - Other work flows : filter, map-reduce-map, …
  - Read data from disk for each MapReduce Job
    - Iterative algorithm : machine learning, graph
  - Only native JAVA programming interface
    - Other languages
    - Interactivity
  - We need *general*, *iterative* and *interactive* framework.

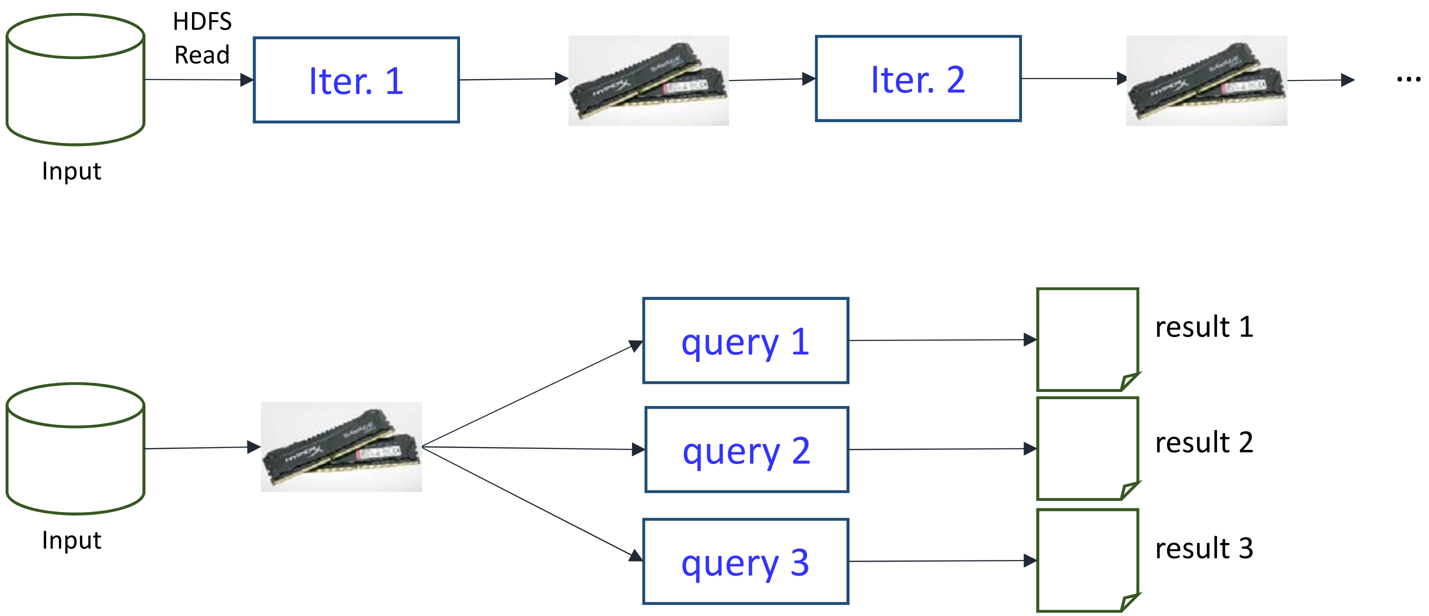
Why MapReduce is not suitable for iteration?

- In MapReduce, the only way to share data across jobs is stable storage



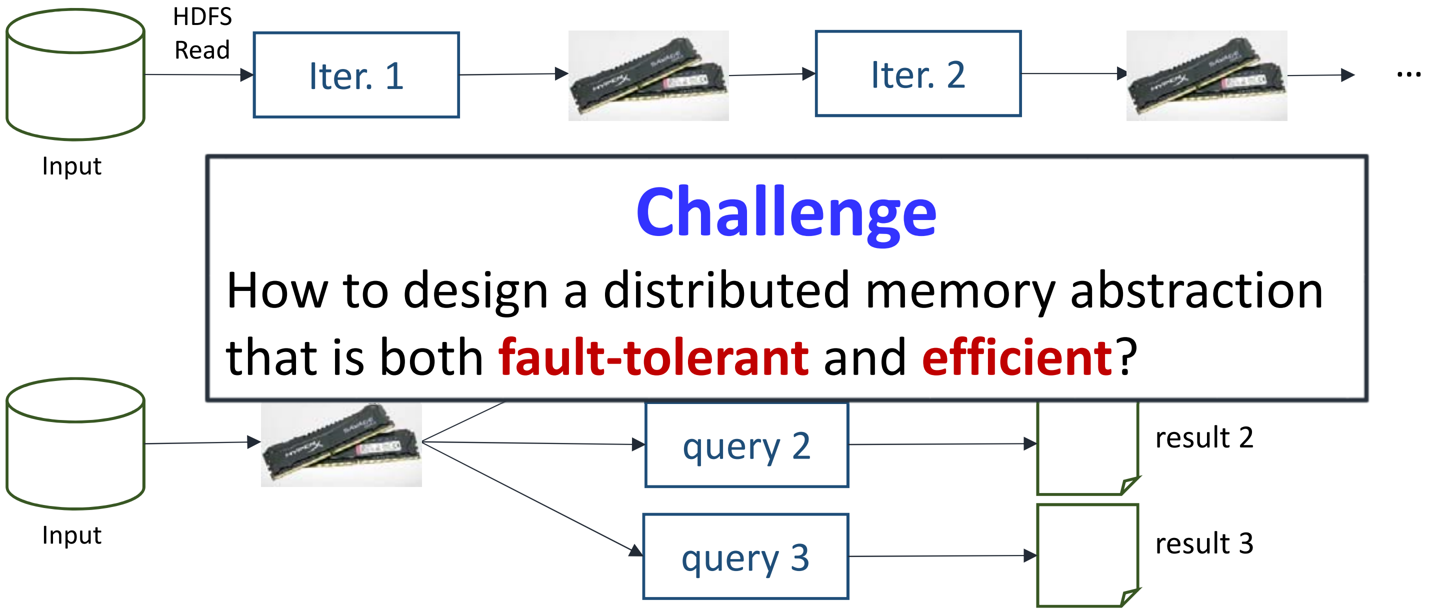
Slow due to replication and disk I/O, but necessary for fault tolerance

Goal : In-Memory Data sharing



10-100x faster than network/disk, but how to get ***fault tolerance***?

Goal : In-Memory Data sharing



Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

10-100x faster than network/disk, but how to get ***fault tolerance***?

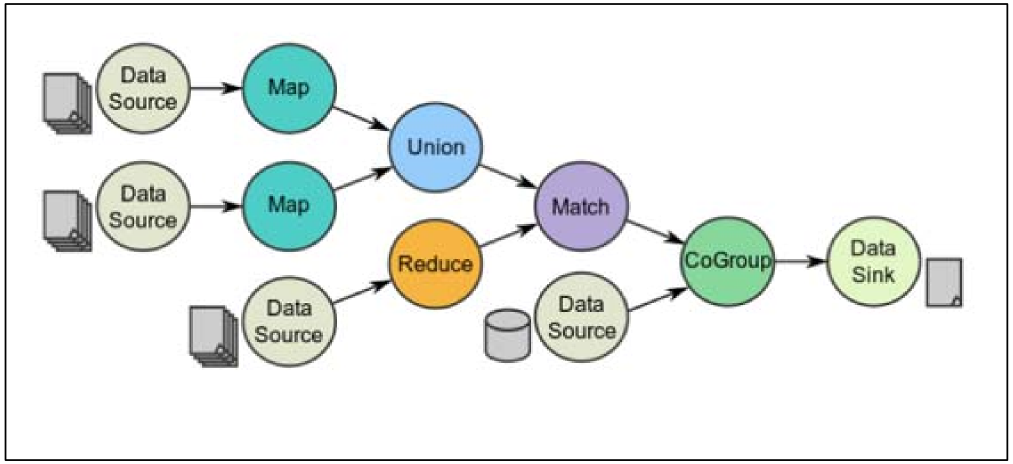
Challenge

- Existing abstractions for in-memory storage on clusters offer an interface based on ***fine-grained*** updates to mutable state (e.g., updating cells in a table)
  - RamCloud, Databases, distributed memory …
- Requires replicating data across nodes or frequent logging including checkpoint for ***fault tolerance***
  - Costly for data-intensive applications

Solution : Resilient Distributed Dataset(RDD)

- Restricted form of *distributed shared memory*
  - **Immutable(read-only),** collections of objects spread across a cluster, stored in RAM
  - Can only be built through ***coarse-grained*** deterministic transformations (operations)
    - From stored data in stable storage (HDFS, S3…) or other RDDs
    - ***Coarse-grained*** : same operation on the whole dataset
    - Example : map, filter
- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

sequence of transformations that created the RDD



Solution : Resilient Distributed Dataset (RDD)

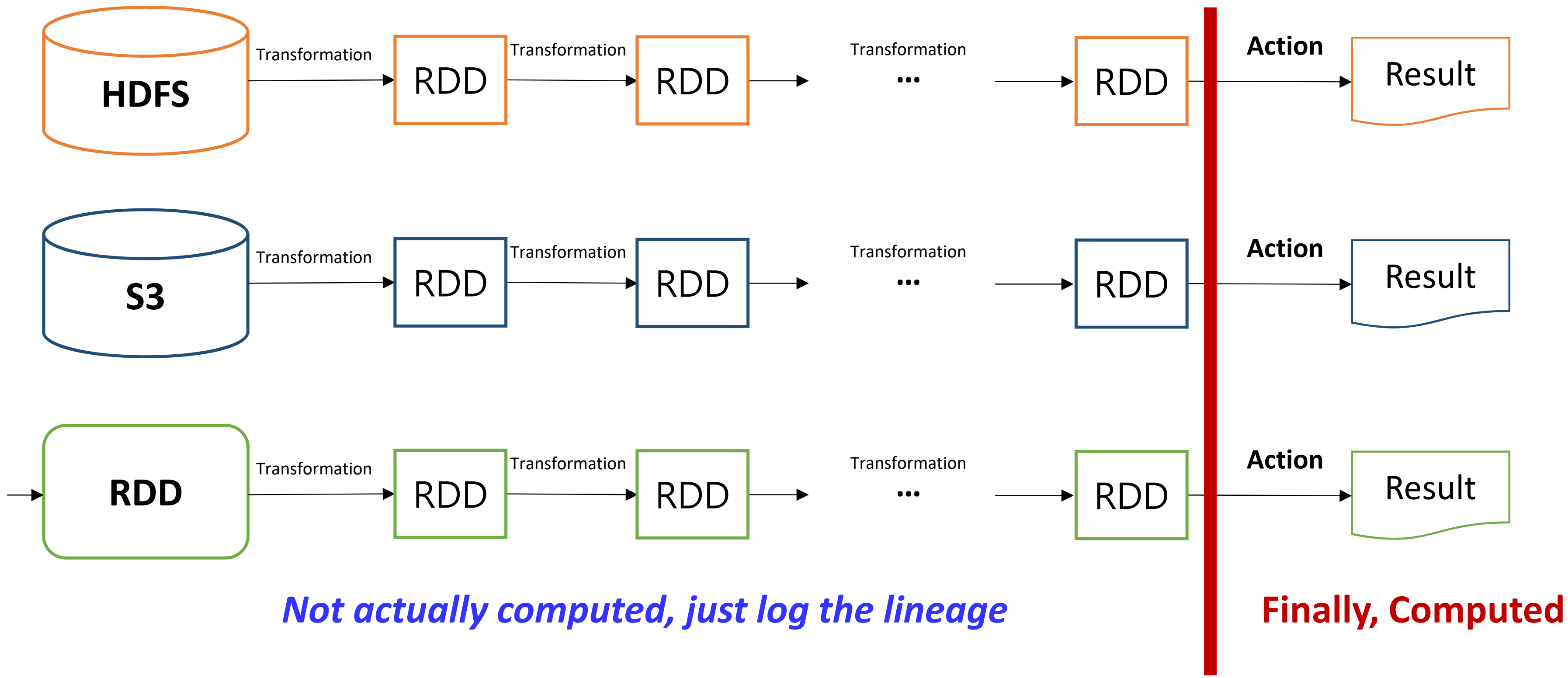
- *Lazy computation* (why?)
  - RDDs are computed only when the first action(operation) is invoked.
- Whenever a user run an *action* on an RDD, Spark optimize the computing process with scheduler

RDD is fault tolerant(by lineage) and efficient(by lazy computation) distributed memory abstraction

Two type of operations on RDD

- Transformation
  - Never modify RDD in place
  - Transform RDD to another RDD
- Action
  - Final stage of workflow
  - Returns a value after computing on the dataset

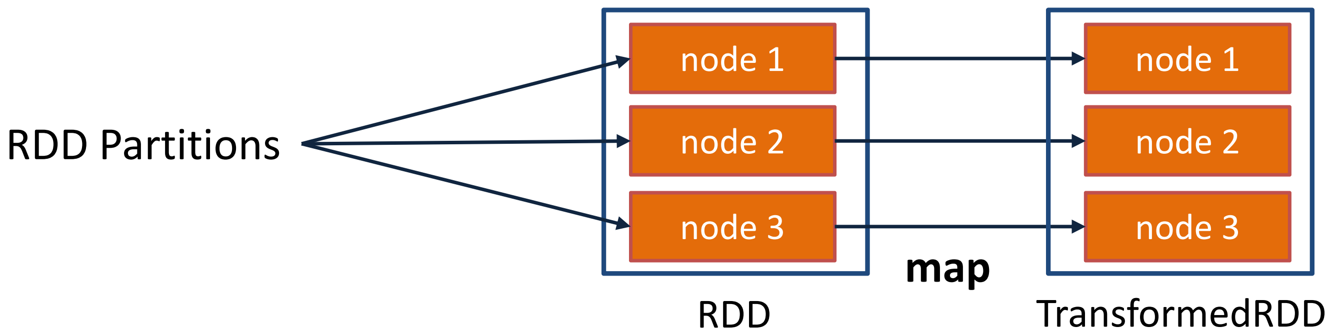
Two type of operations on RDD



RDDs are created from stored data in stable storage (HDFS, S3…) or other RDDs

Spark Programming Interface

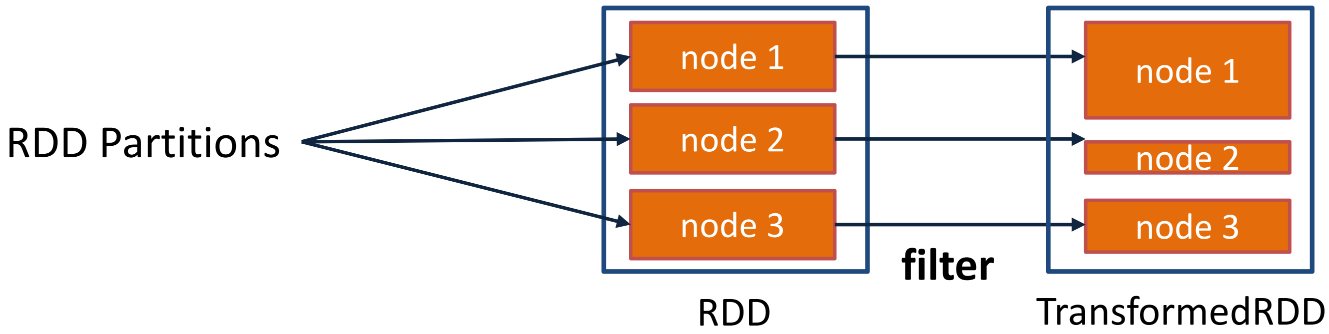
- Transformations
  - map(function) : apply function to each element of RDD



Each box is an RDD, with partitions shown as shaded rectangles

Spark Programming Interface

- Transformations
  - filter(function) : keep only elements where function is true



Each box is an RDD, with partitions shown as shaded rectangles

Spark Programming Interface

- Transformations
  - flatMap(function) : map then flatten output
  - filter(function) : keep only elements where function is true
  - …
  - Check http://spark.apache.org/docs/latest/programming-guide.html

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

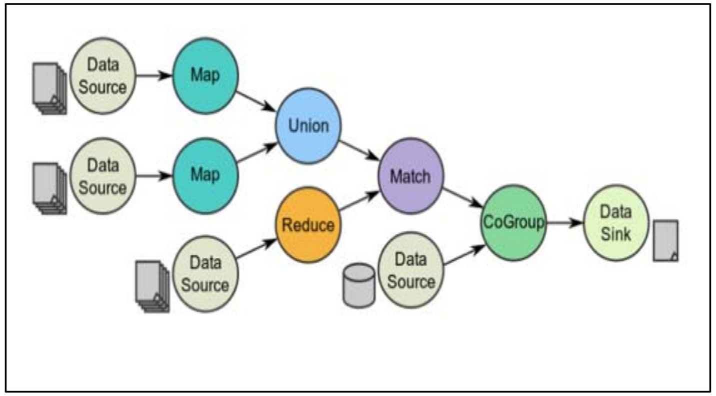
Spark Programming Interface

- Action
  - Final stage of workflow
  - Triggers **execution** of the DAG (lineage)
  - Returns a value after computing on the dataset to the Driver or writes to HDFS
  - Check http://spark.apache.org/docs/latest/programming-guide.html

Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Representing RDDs (Lineage)

- Simple Graph based representation : Directed Acyclic Graphs (DAG)
- Track lineage across a wide range of transformations
- Each RDD is represented by five factors.
- Five factors
  - A set of partitions, which are atomic pieces of the dataset
  - ***A set of dependencies on parent RDDs***
  - A function for computing the dataset based on its parents
  - Metadata about its partitioning scheme
  - Metadata about its data placement scheme

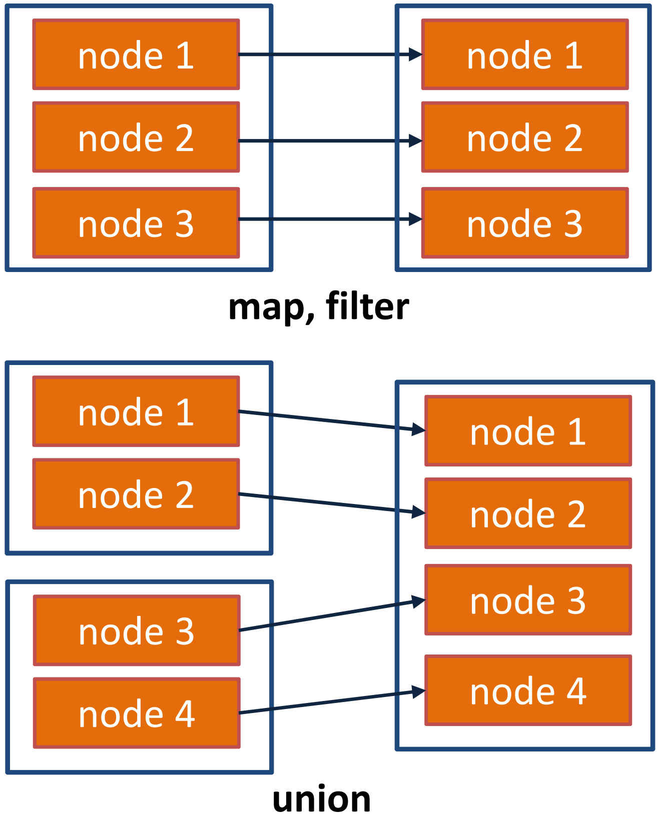


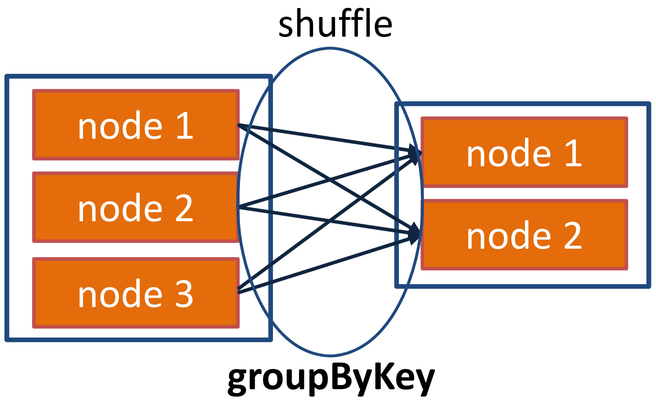
Narrow & Wide Dependencies of RDDs

- **Narrow dependencies**
  - Each partition of the parent RDD is used by at most one partition of the child
  - Allow for pipelined execution on one cluster node
  - Low cost in failure
- **Wide dependencies**
  - Multiple child partitions may depend on it
  - Require data from all parent partitions to be available and to be ***shuffled*** across the nodes
  - High cost in failure

> - Global redistribution of data (Network I/O)
> - High impact on performance
> - Know shuffle, avoid it

Example of Narrow & Wide Dependencies





Each box is an RDD, with partitions shown as shaded rectangles

Job Scheduling (Optimizing computing process)

- Whenever a user runs an action on an RDD,
  - the scheduler examines that RDD's lineage graph to build a DAG of stages to execute.
- Job : Spark action (e.g. save, action)

Run in parallel

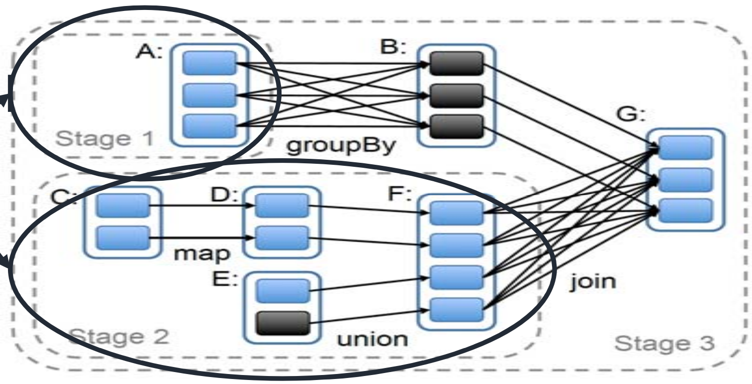
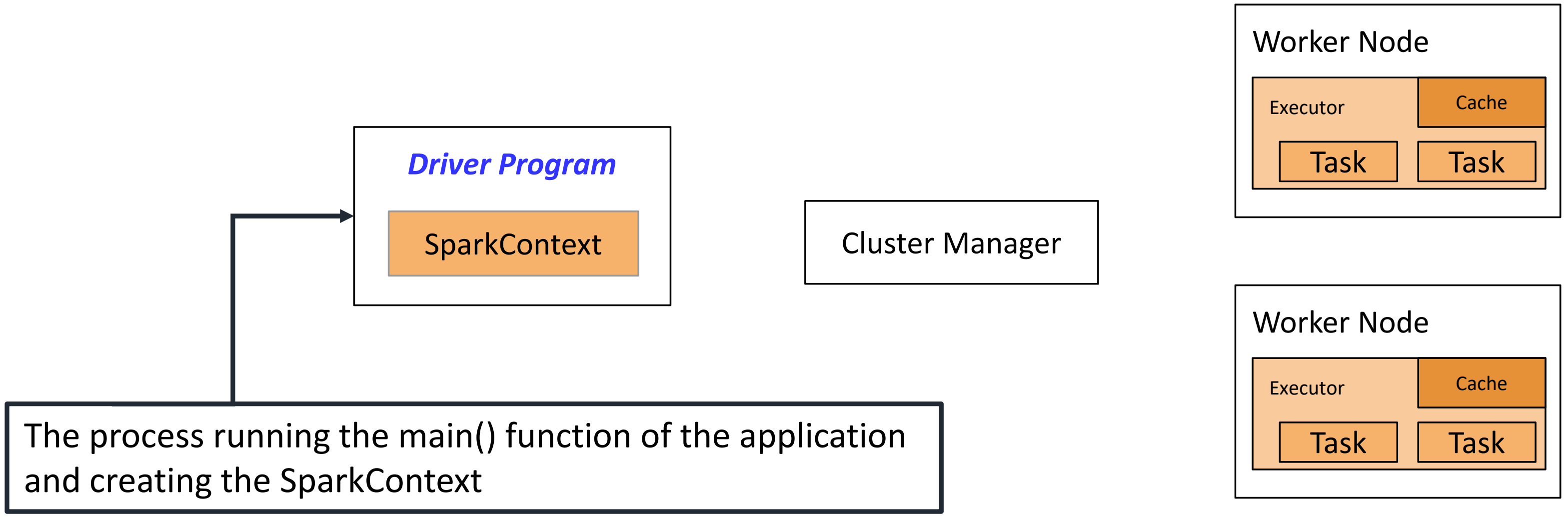


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

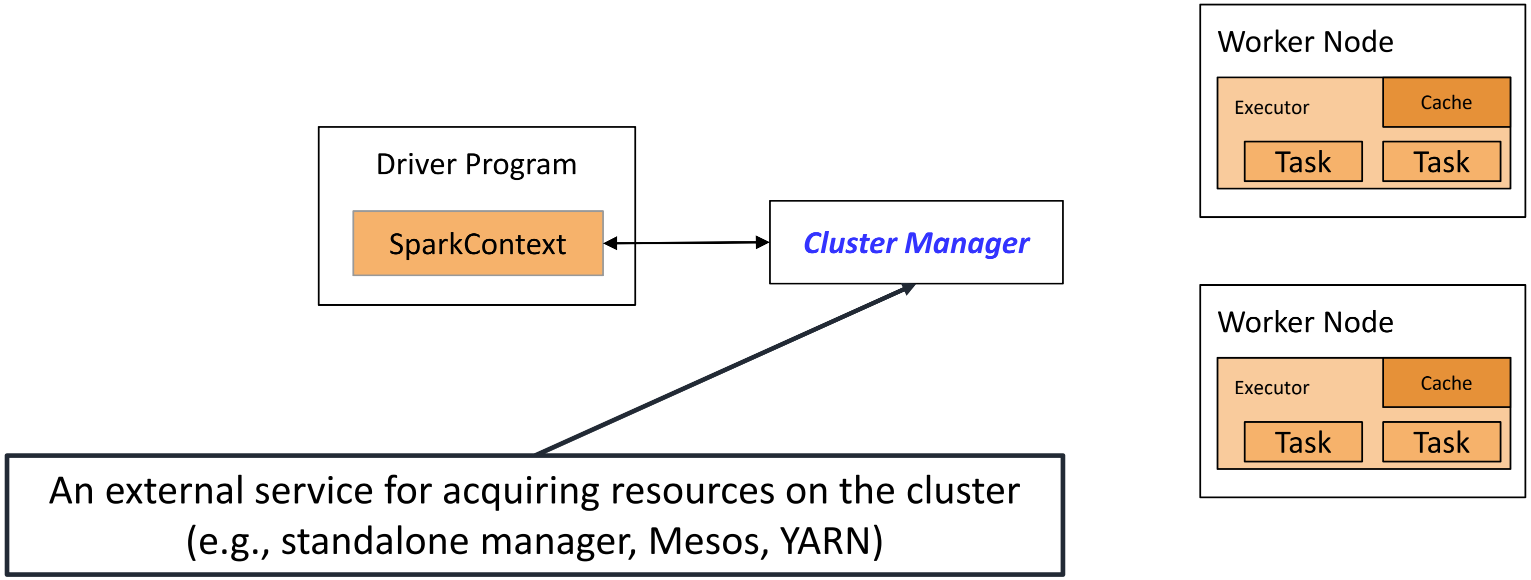
Spark Cluster Architecture

- Spark application run as independent sets of processes on a cluster, coordinated by the **SparkContext** object in your main program(called ***Driver program***)



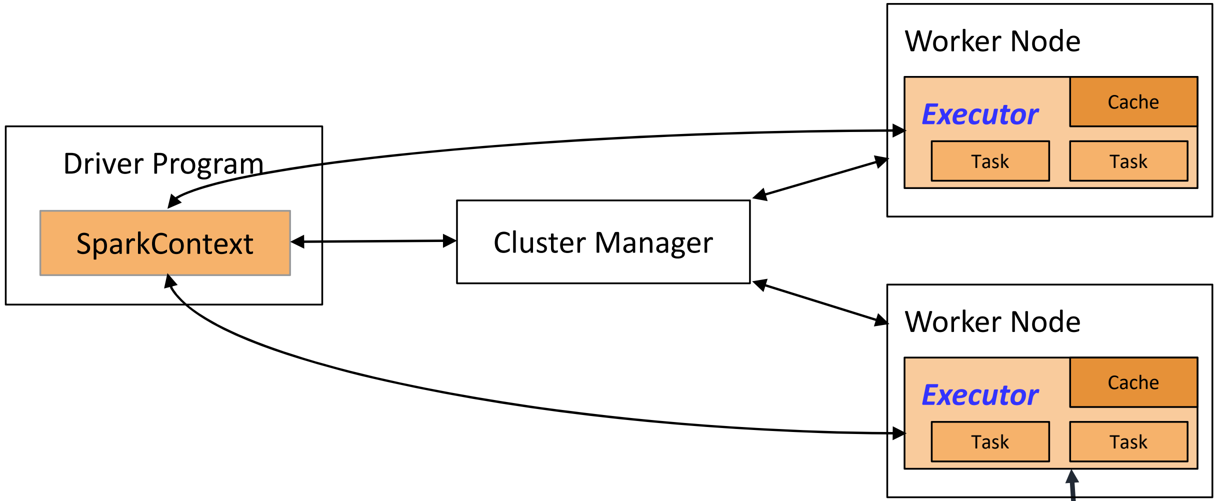
Spark Cluster Architecture

- To run on a cluster the SparkContext can connect to several types of ***cluster managers***, which allocate resources across applications.



Spark Cluster Architecture

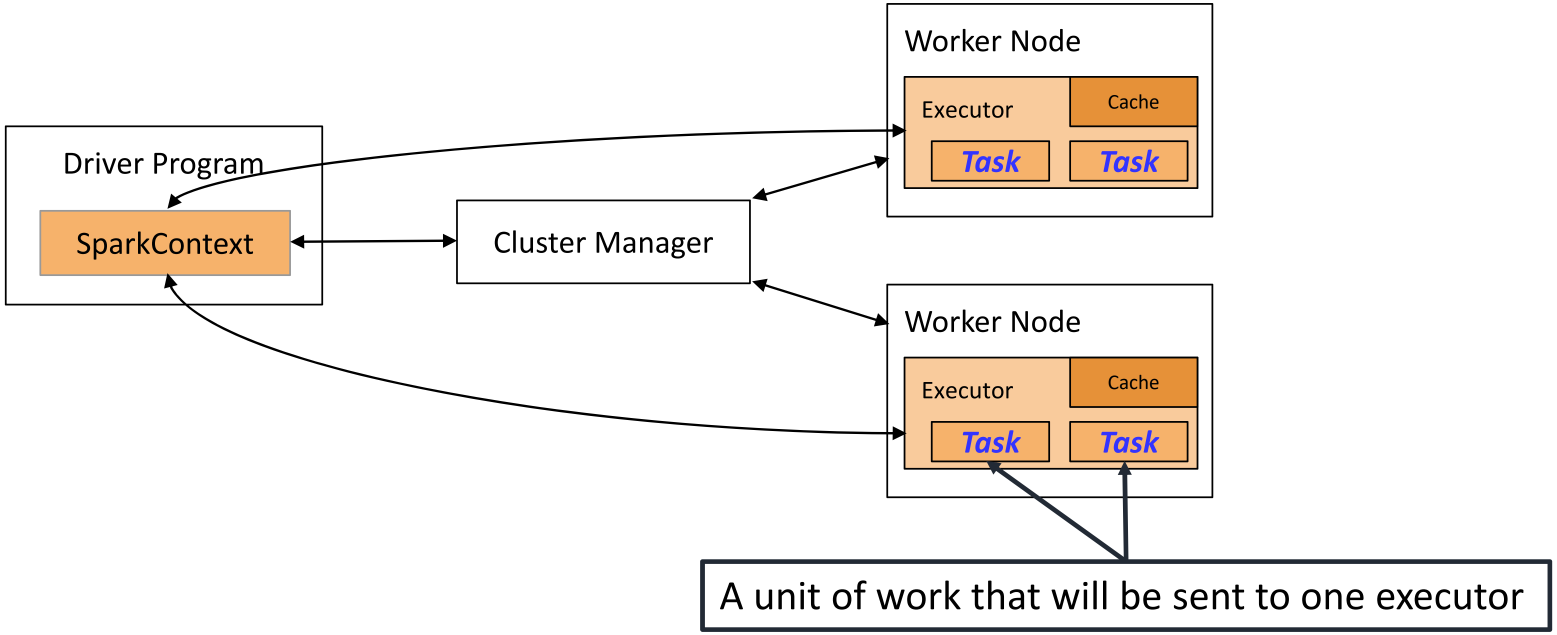
- Once connected, Spark acquires ***executors*** on nodes in the cluster, which are processes that run computations and store data for your application.



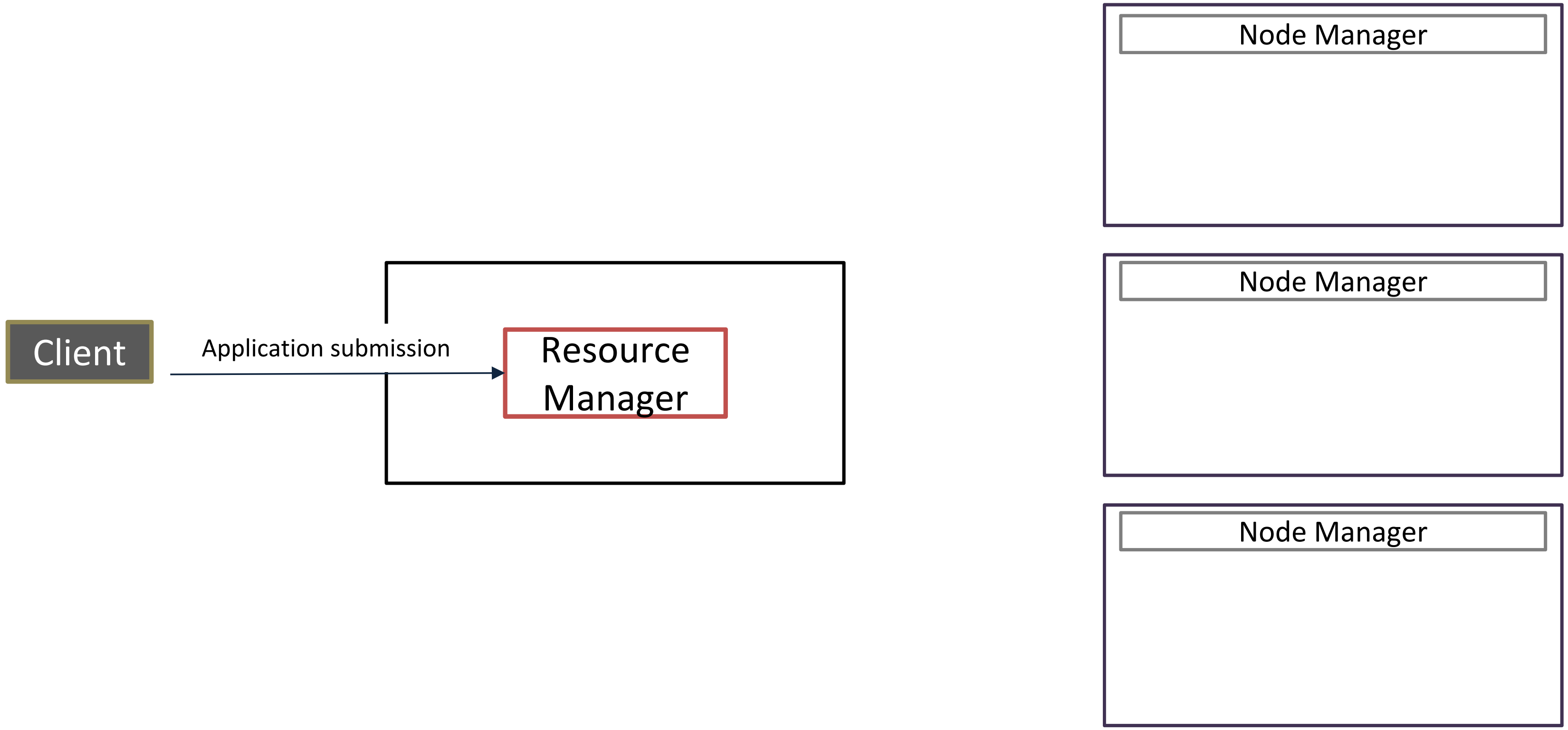
- A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them.
- Each application has its own executors

Spark Cluster Architecture

- Next, it sends your application code (defined by JAR) to the executors.
- Finally SparkContext sends ***tasks*** to the executors to run

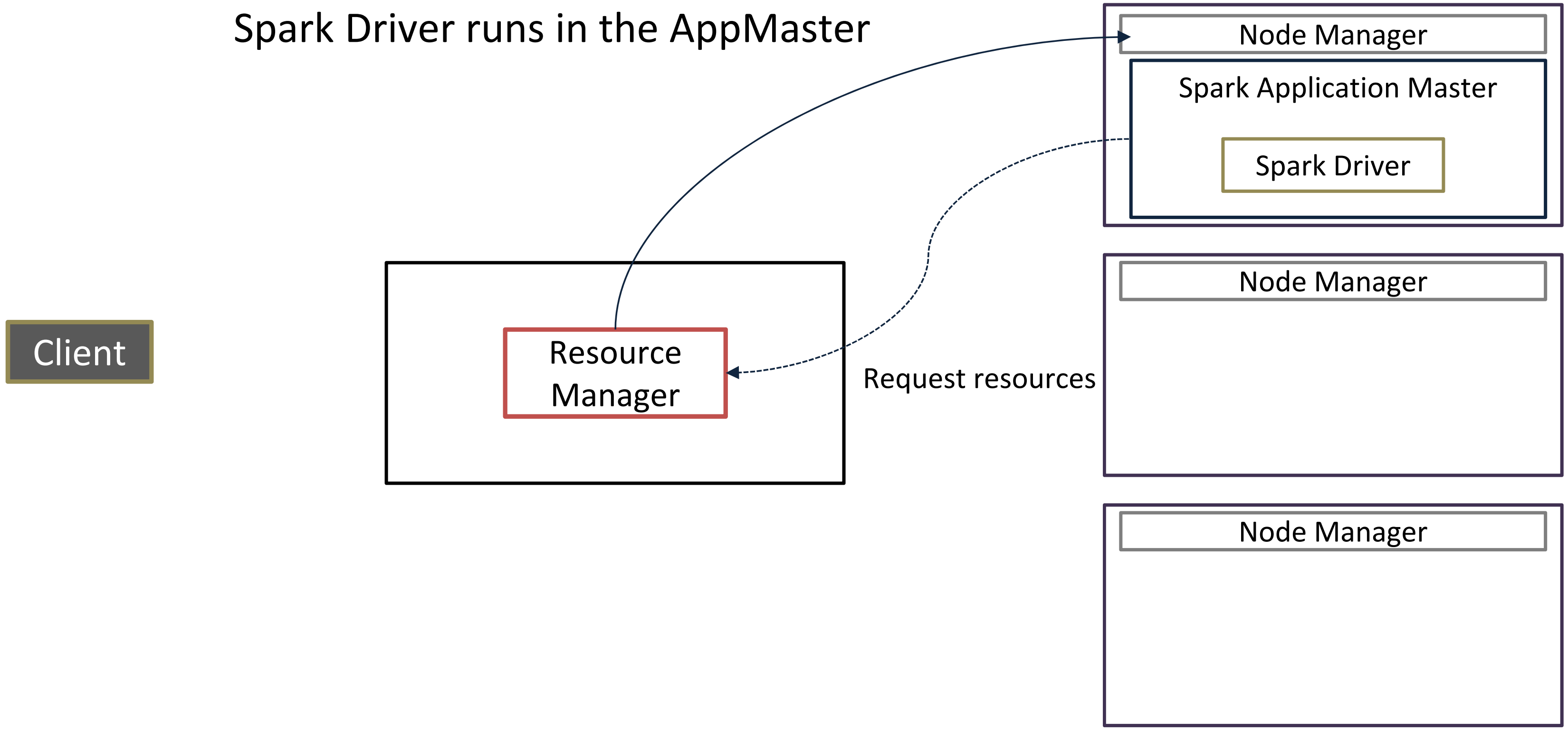


Different view of Spark Architecture

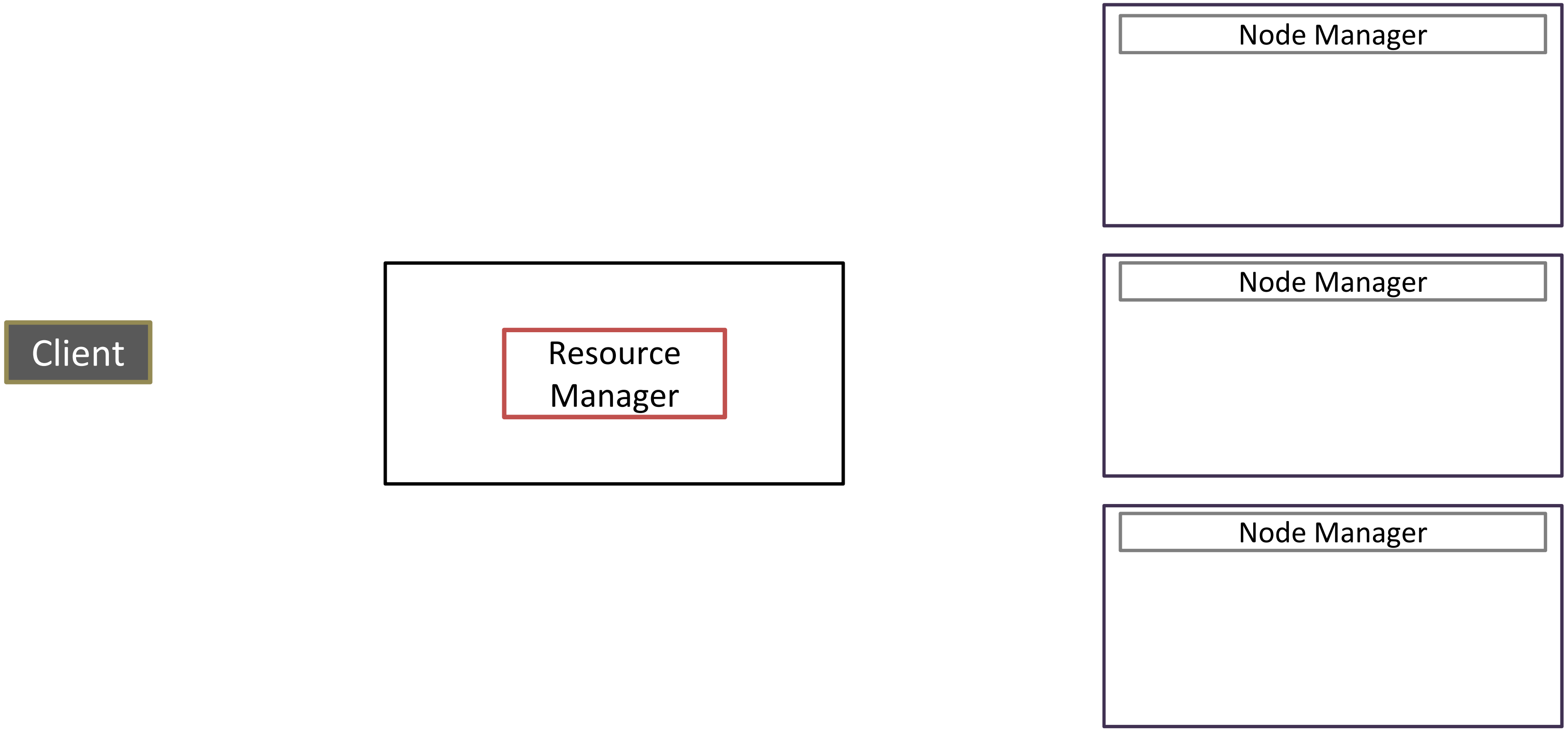


Different view of Spark Architecture

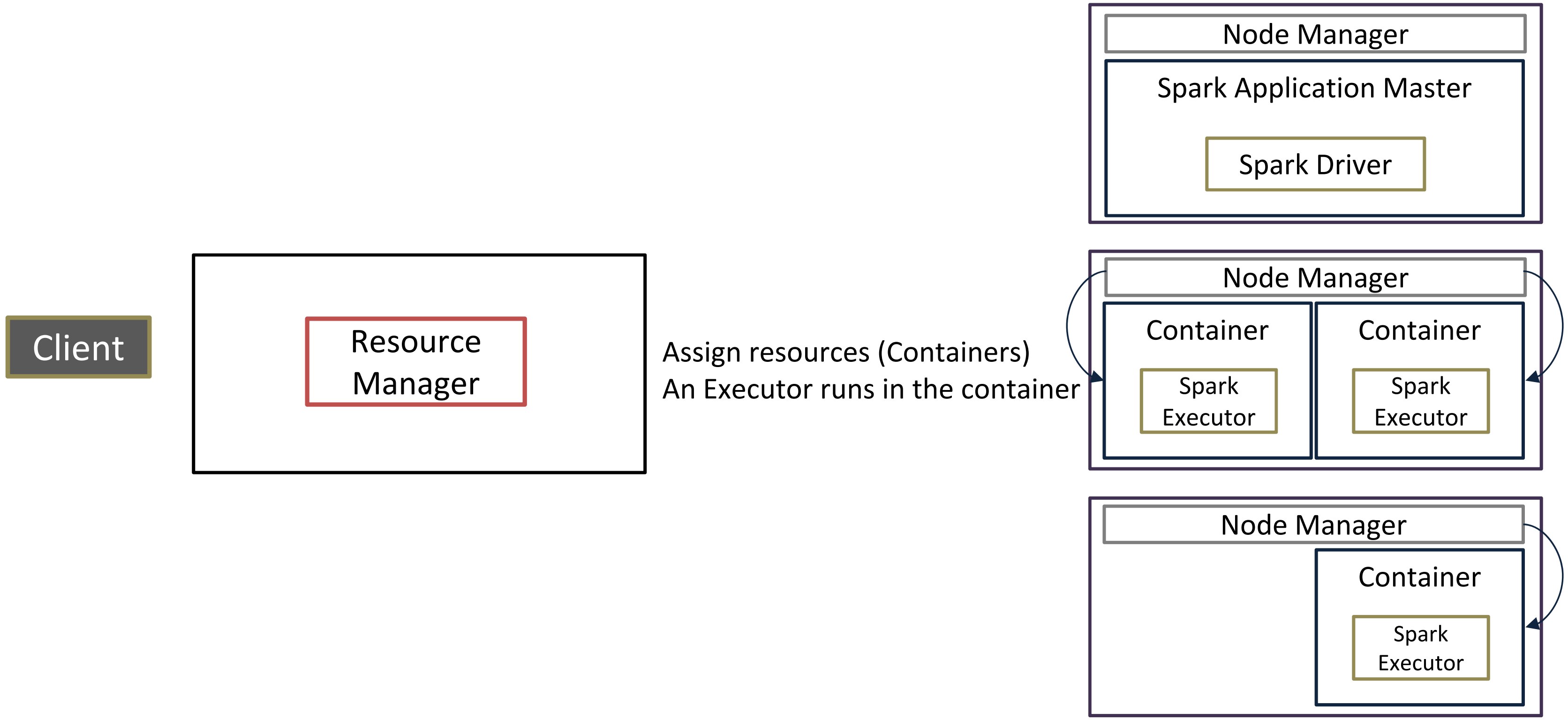
Assign Application Master
Spark Driver runs in the AppMaster



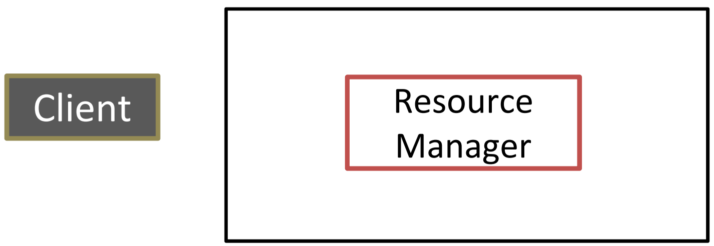
Different view of Spark Architecture



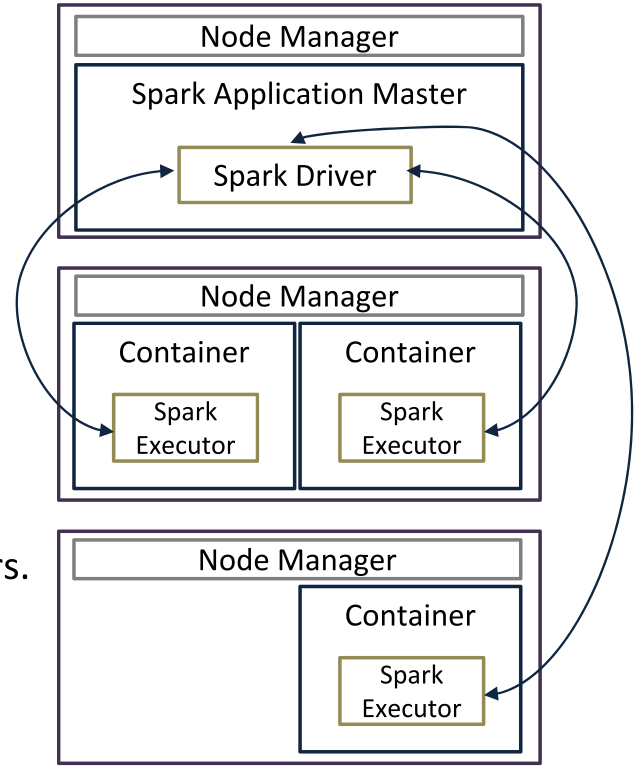
Different view of Spark Architecture



Different view of Spark Architecture

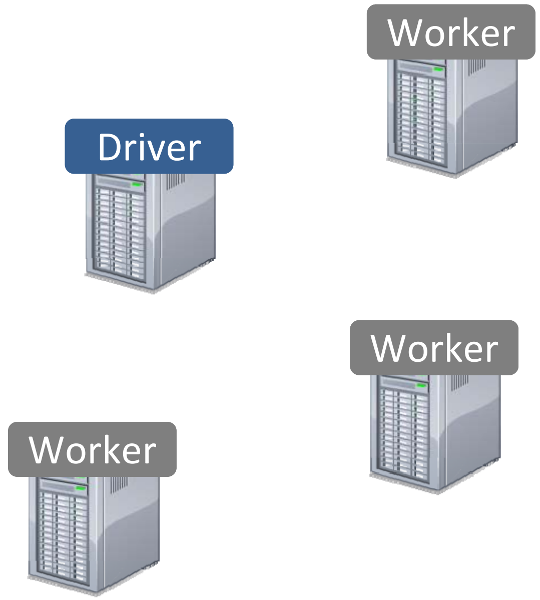


Driver directly controls the Executors.



Log Mining Example

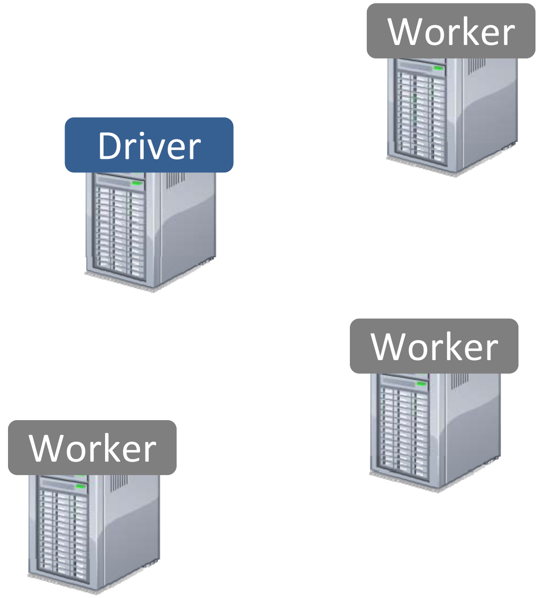
Load error messages from a log into memory, then interactively search for various patterns



Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
```

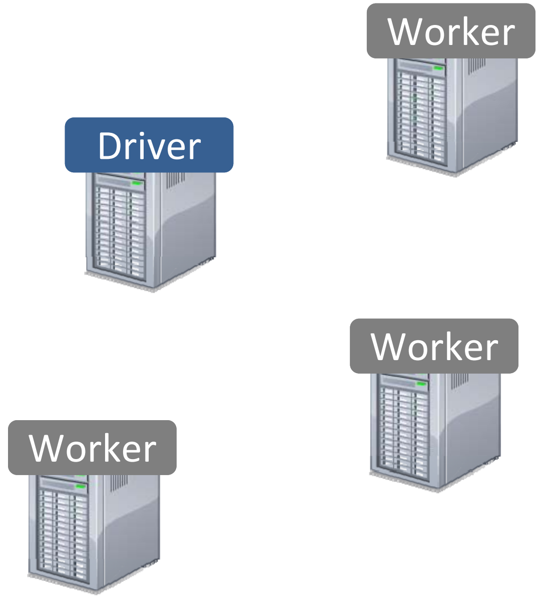


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

Base RDD

```
val lines = spark.textFile("hdfs://...")
```

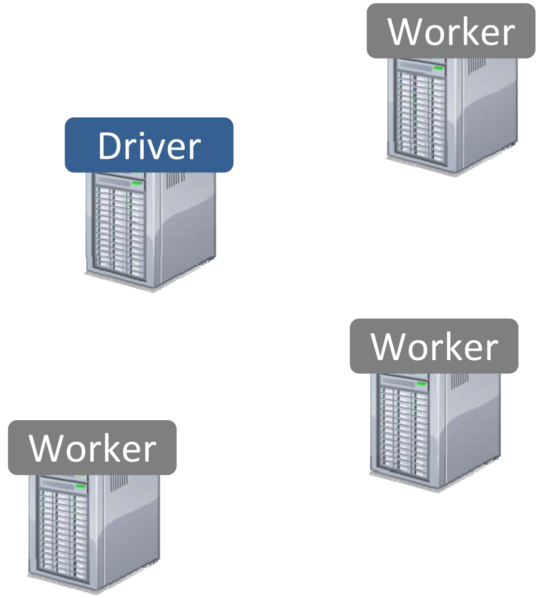


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

```
val lines = spark.textFile(“hdfs://...”)
val errors = lines.filter(_.startswith(“ERROR”))
```



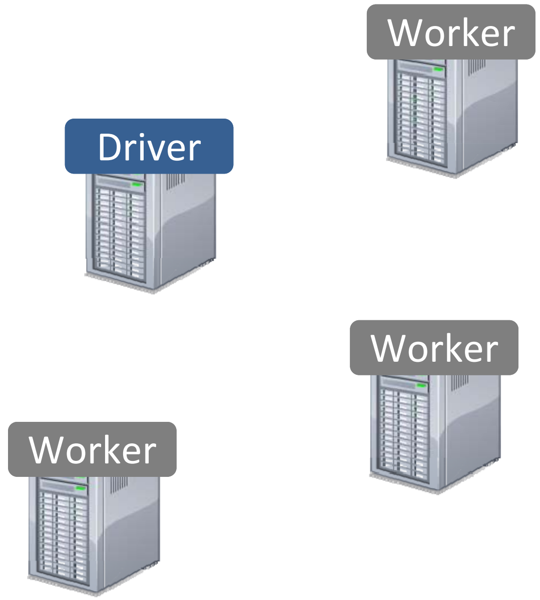
Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile(“hdfs://...”)
val errors = lines.f                ROR”))
val messages = e    rs.map(_.split(“\t”)(2))
messages.cache()

messages.filter(_.contains(“mysql”)).count()
```

Cache the RDD



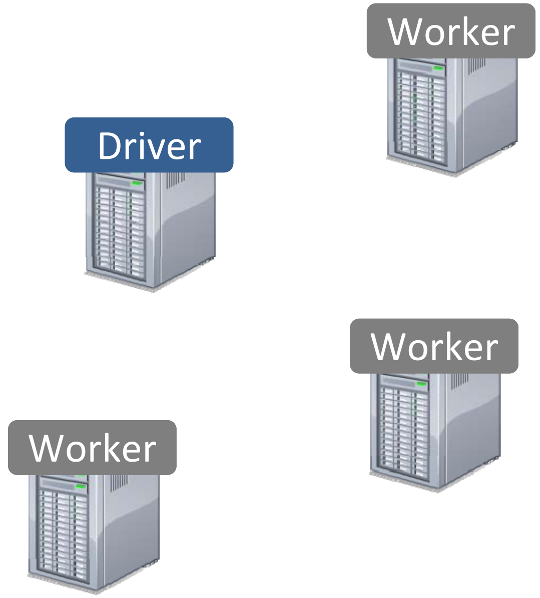
Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile(“hdfs://...”)
val errors = lines.filter(_.startswith(“ERROR”))
val messages = errors.map(_.split(“\t”)(2))
messages.cache()

messages.filter(_.contains(“mysql”)).count()
```

Action

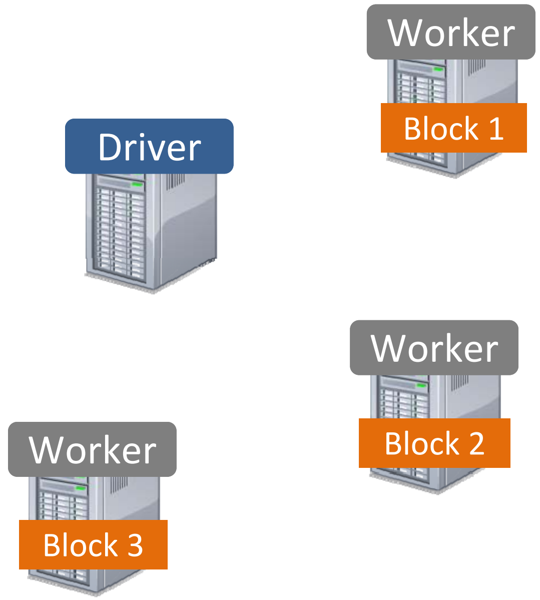


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

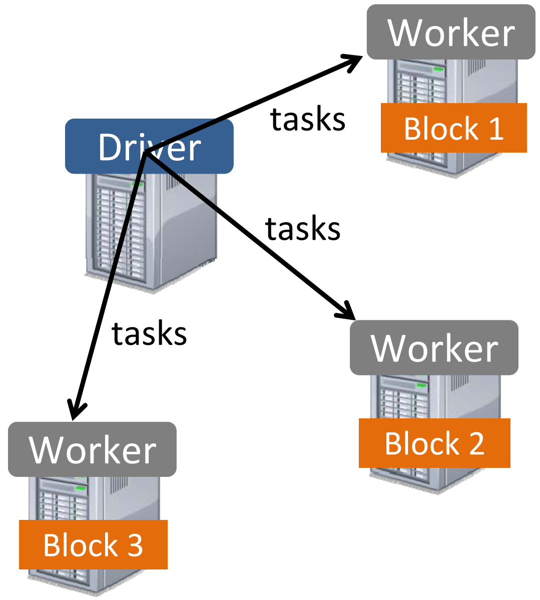


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

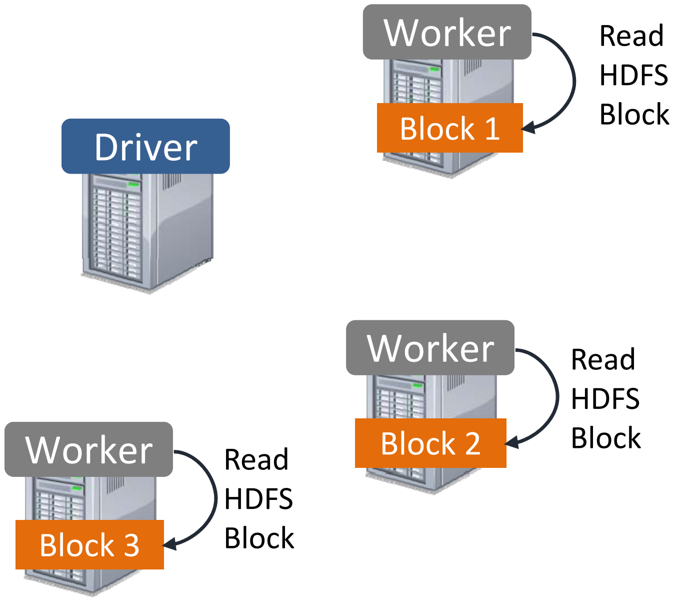


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

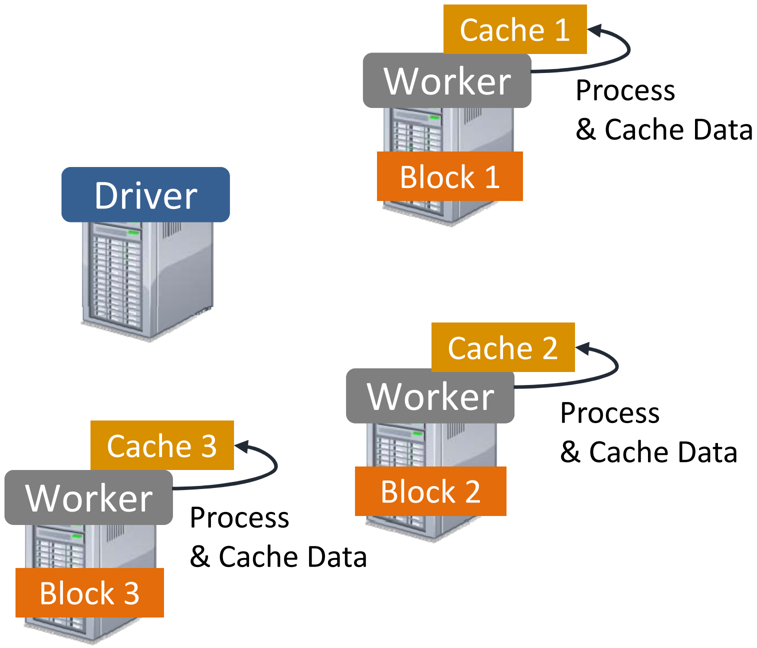


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

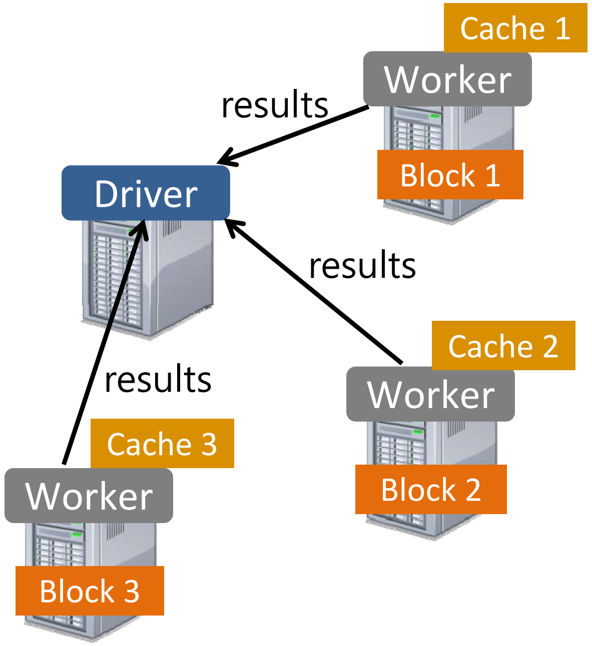


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
```

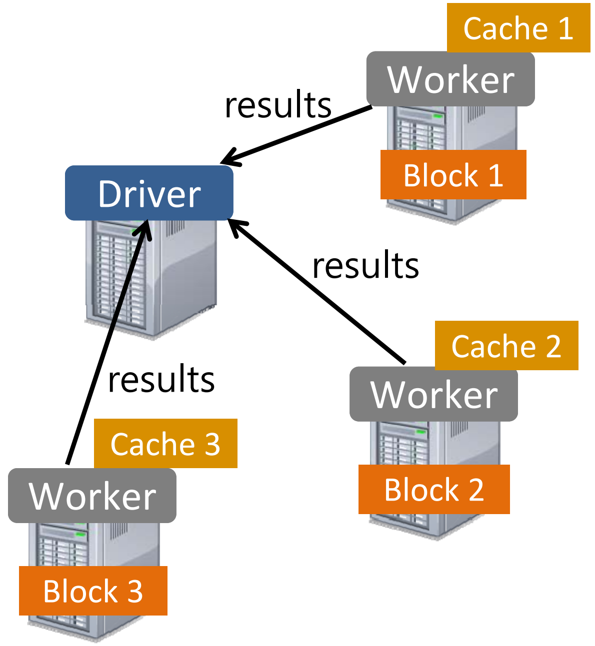


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

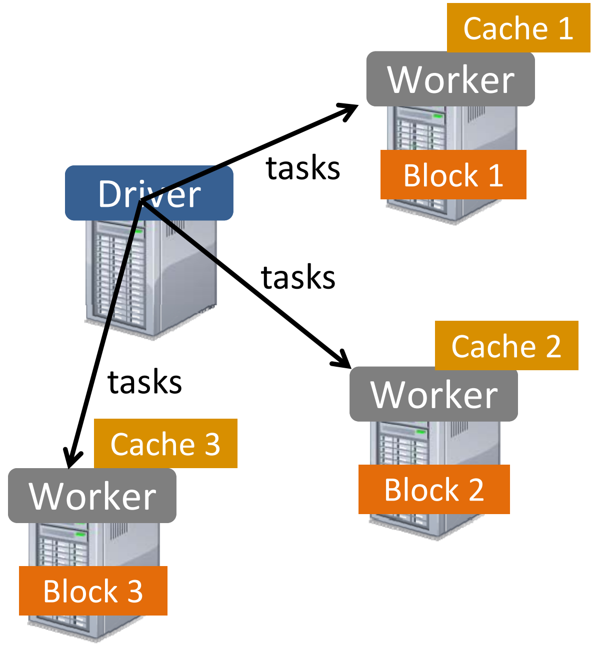


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

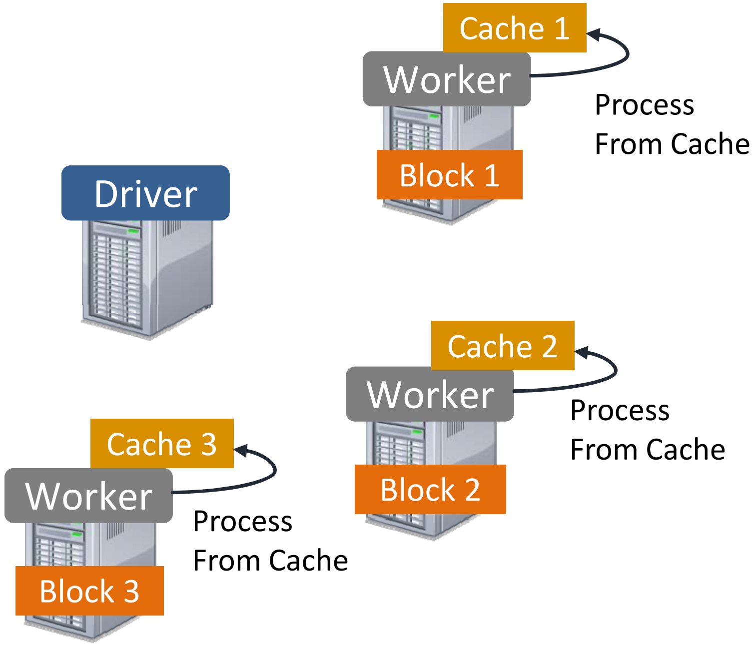


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

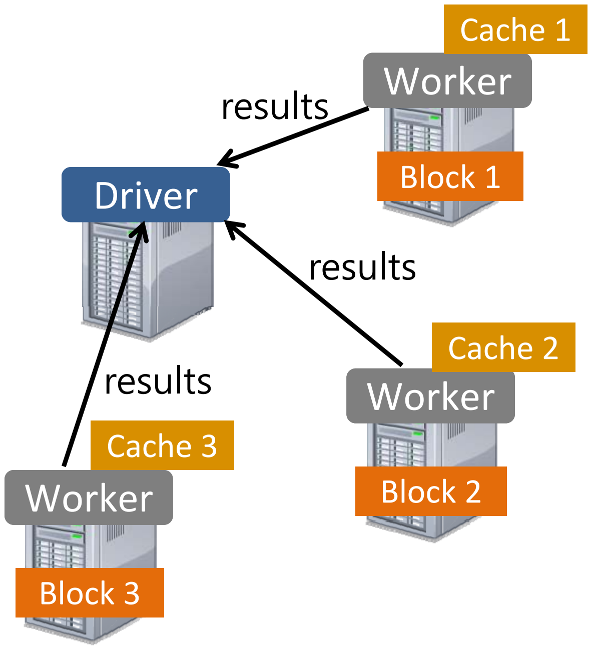


Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```



Log Mining Example

Load error messages from a log into memory, then interactively search for various patterns

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startswith("ERROR"))
val messages = errors.map(_.split("\t")(2))
messages.cache()

messages.filter(_.contains("mysql")).count()
messages.filter(_.contains("php")).count()
```

Cache data → Faster Results
1 TB of log data
- *5-7 sec from cache vs. 170s for on-disk*