Overview of MapReduce and Spark

Diego Rivera Correa

Spark Overview

- Spark builds on MapReduce's ideas but hides more details from the programmer.
  - Same concepts: data partitioning, combining, shuffling.
  - More operations than just map and reduce.

- Key difference: Spark lets programs keep data in memory across multiple rounds of computation and shuffling.
  - The data stays in the RAM of the executor processes scattered across the cluster. If they don't fit, they would have to be spilled to disk.

- Greater performance potential (especially for iterative algorithms), but also easier to write inefficient code.

Why Spark?

- In MapReduce, each job is independent. Between jobs, all data must go through the DFS:
  - Job finishes → replicate across network (HDFS) → next job reads from disk → start computing
  - A pipeline of 5 jobs = 5 round-trips through DFS.
- This is unavoidable in MapReduce since jobs don't share memory. The only way to pass data between them is through the distributed file system.

Why Spark?

- In Spark, a single application chains multiple stages. Data stays in executor memory between stages.

- For iterative algorithms (e.g., PageRank: 20 iterations over the same graph):
  - MapReduce: read the graph from DFS 20 times.
  - Spark: read the graph once, cache it in memory, reuse it 20 times.
    - Caching is not automatic — you must call .cache() or .persist() to tell Spark to keep data in memory.

Why Spark?

- **Lazy evaluation**: Spark can see the entire computation graph before executing anything, enabling automatic optimization (like a DBMS query optimizer).
  - We will see an example in a few slides...

- MapReduce optimizes one job at a time.

What is an RDD?

- RDD = Resilient Distributed Data Set. Spark's core data abstraction.
  - Immutable: one can read it, but not modify it.
  - Resilient: failures are transparent to the user, i.e., the user code does not need to handle those exceptions.
  - Distributed: data is split into partitions across executors. Each partition is processed by one task.

- Think of an RDD as a table of Objects, horizontally partitioned across executors.

What is an RDD?

- The partitions field of an RDD is an Array, whose size corresponds to the number of partitions.

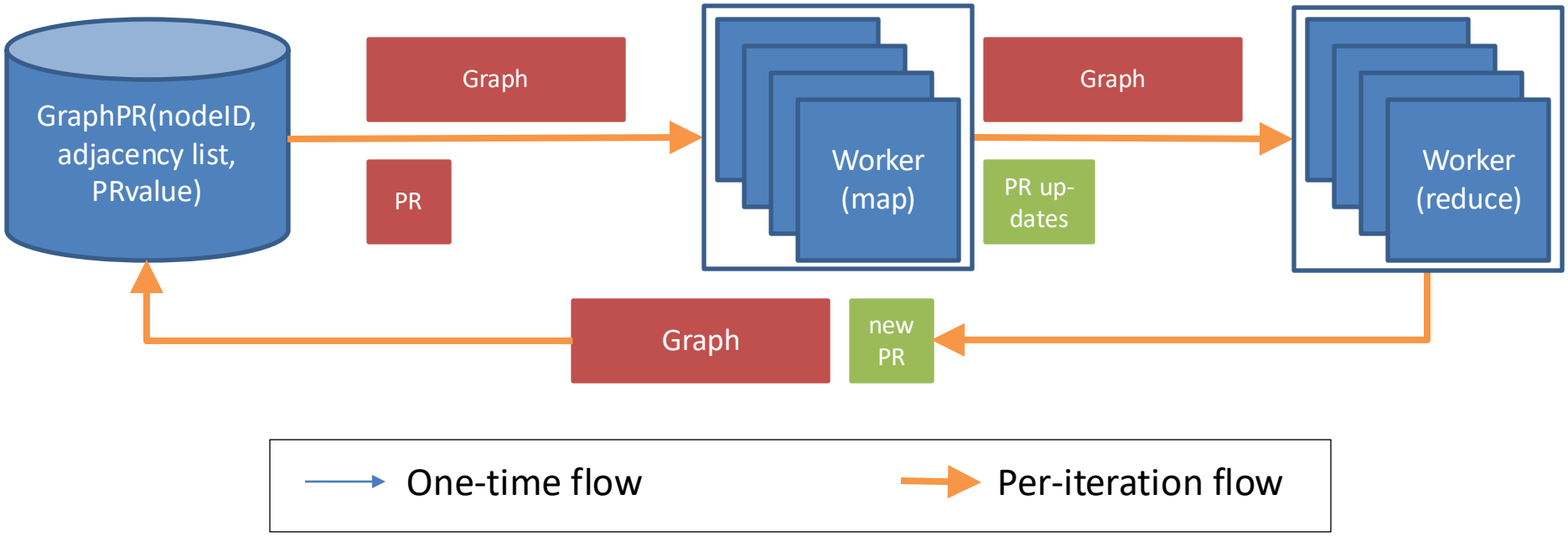
```
val rdd = sc.parallelize(List("apple", "banana", "cat", "dog", "egg", "fig"), 3)

 rdd.partitions.length // 3

// Partition 0: ["apple", "banana"]
// Partition 1: ["cat", "dog"]
// Partition 2: ["egg", "fig"]
```

  - More partitions = more parallelism (up to the number of available tasks an executor can hold).

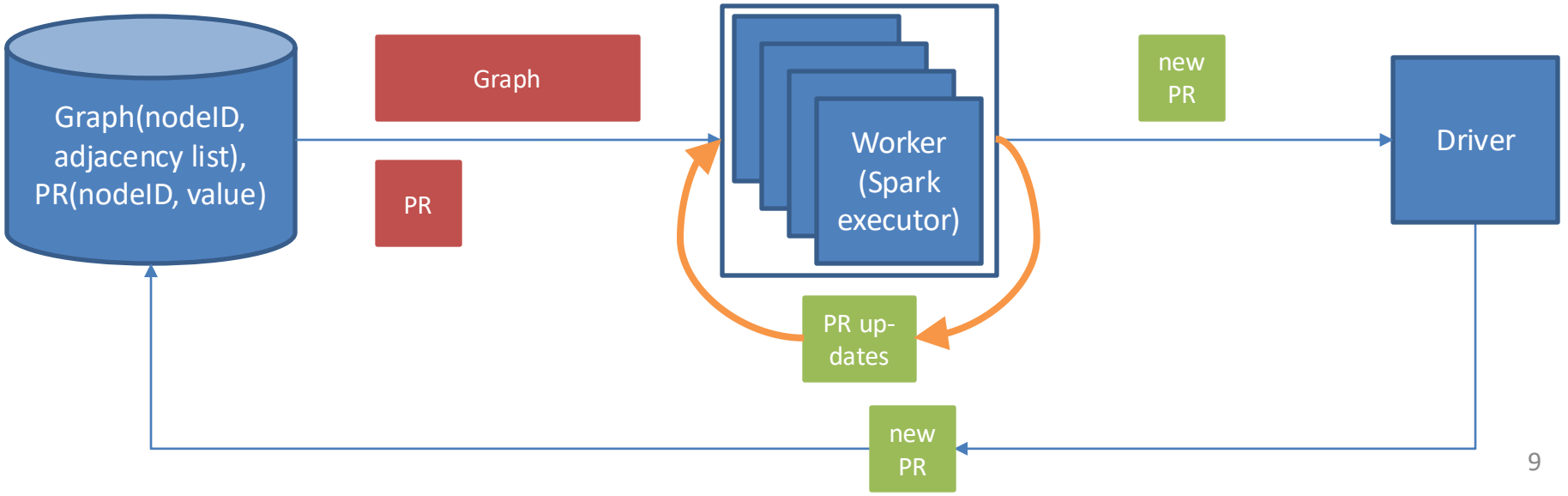
The following example illustrates the difference between MapReduce and Spark for an iterative computation.

Compare data movement in each iteration, focusing on accesses to the DFS on the left. We discuss details when we introduce PageRank in a future lecture.

Typical iterative job (PageRank) in Hadoop MapReduce:



Typical iterative job (PageRank) in Spark:



Spark Limitations

- No in-place updates. Like MapReduce, Spark transforms input into output — an "update" creates a new copy. all transactions on a large database).

- Optimized for batch processing. Spark excels at transforming big data in bulk, not at real-time stream processing or small random accesses via indexes.

Spark Terminology

- **SparkContext (sc)**: The object in your code that talks to Spark. Every time you write sc.textFile(...) or sc.parallelize(...), you're using it. One per application.

- **Driver**: The JVM process that runs your main() method. Contains the SparkContext, the Scheduler, and your application logic. It decides what work to do, breaks it into stages and tasks, tells executors to run them.

Spark Terminology

- **Executor**: A long-lived JVM inside a container in a worker machine. It receives tasks from the driver and runs them. Multiple tasks run simultaneously as threads.
  - like a Map or Reduce task in MapReduce, but runs as a thread, not a JVM

- **Task**: The smallest unit of work. Processes one partition of an RDD (similar to MR!).

- **Stage**: A group of tasks that can run without shuffling. A new stage begins wherever a shuffle is needed (a local phase!).

- **Job**: Triggered by one action (e.g., collect(),saveAsTextFile()). A job consists of one or more stages.

Spark Modes

- **Local mode**: driver and one executor in a single JVM on your machine. Use for development and debugging — like Hadoop's local mode.

- **YARN cluster**: YARN manages resources. Spark coexists with Hadoop and other applications. Most common in production. There are two options for where the driver runs:
  - **Cluster-deploy mode**: driver runs on the cluster (inside the application master container). Use for production — job survives if your laptop disconnects.
  - **Client-deploy mode**: driver runs on your local machine. Use for interactive development — you see output directly in your terminal.

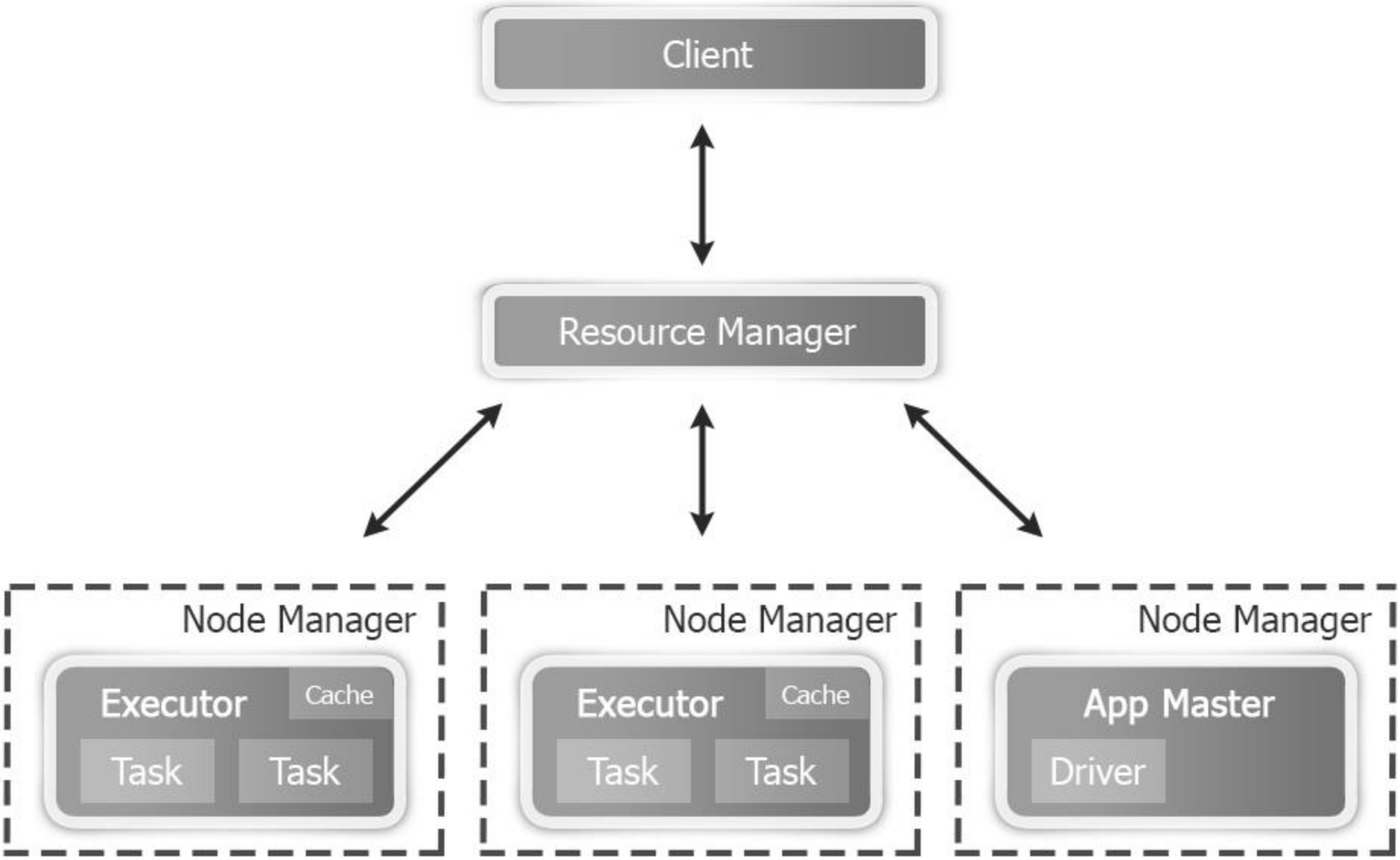
Cluster Terminology

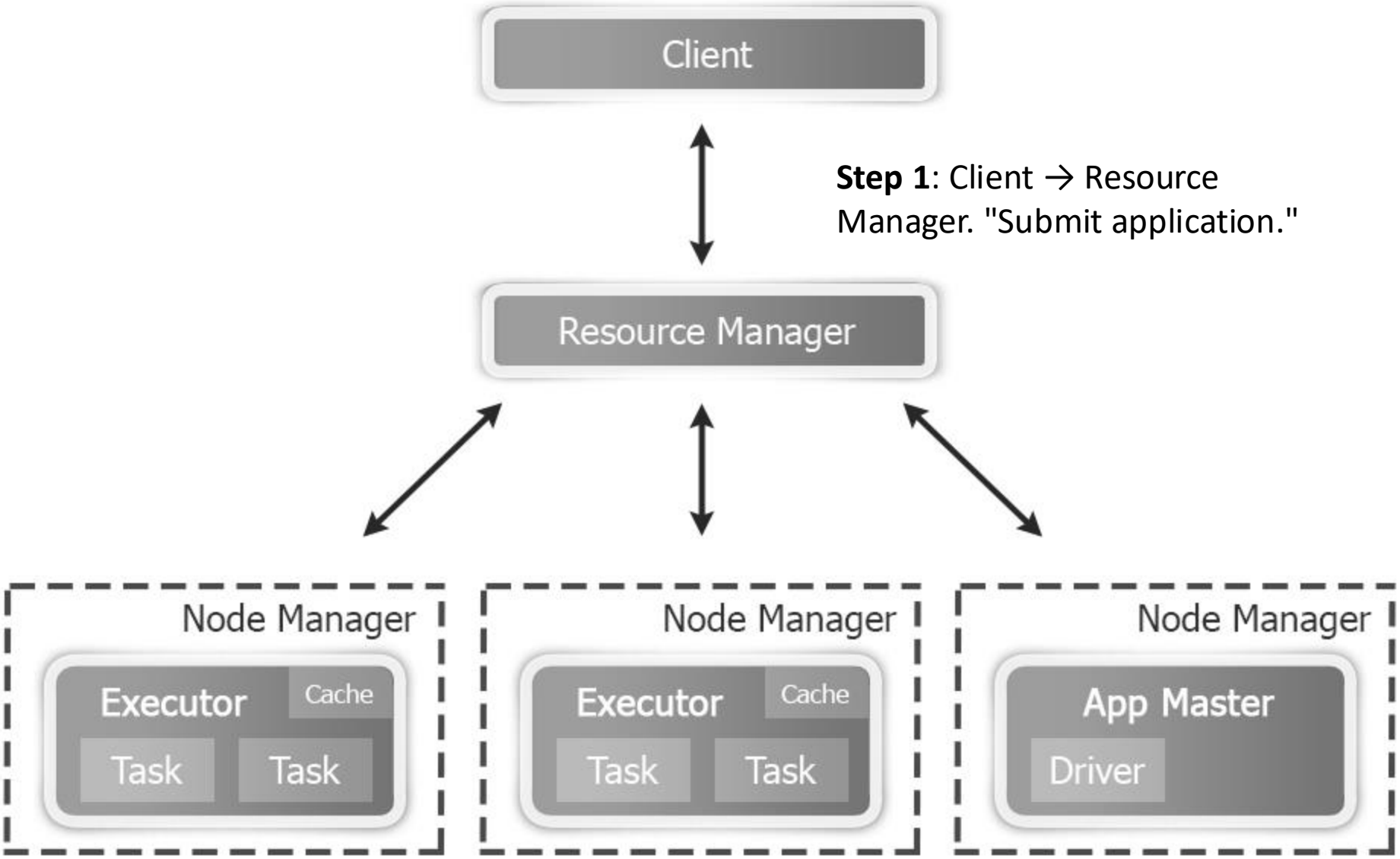
- **Container**: A YARN concept, not a Spark concept. A resource allocation (CPU + RAM) on a worker machine. YARN launches one container for the driver and one container per executor. Spark runs inside the containers but doesn't manage them.

- **Application Master**: Also a YARN concept. A container that YARN launches to manage your application's resources. In cluster-deploy mode, Spark's driver runs inside it. You never interact with it directly.
  - YARN talks to one Application Master per application — the Application Master then manages everything internally, so YARN never needs to understand Spark, MapReduce, or any other framework's internals.

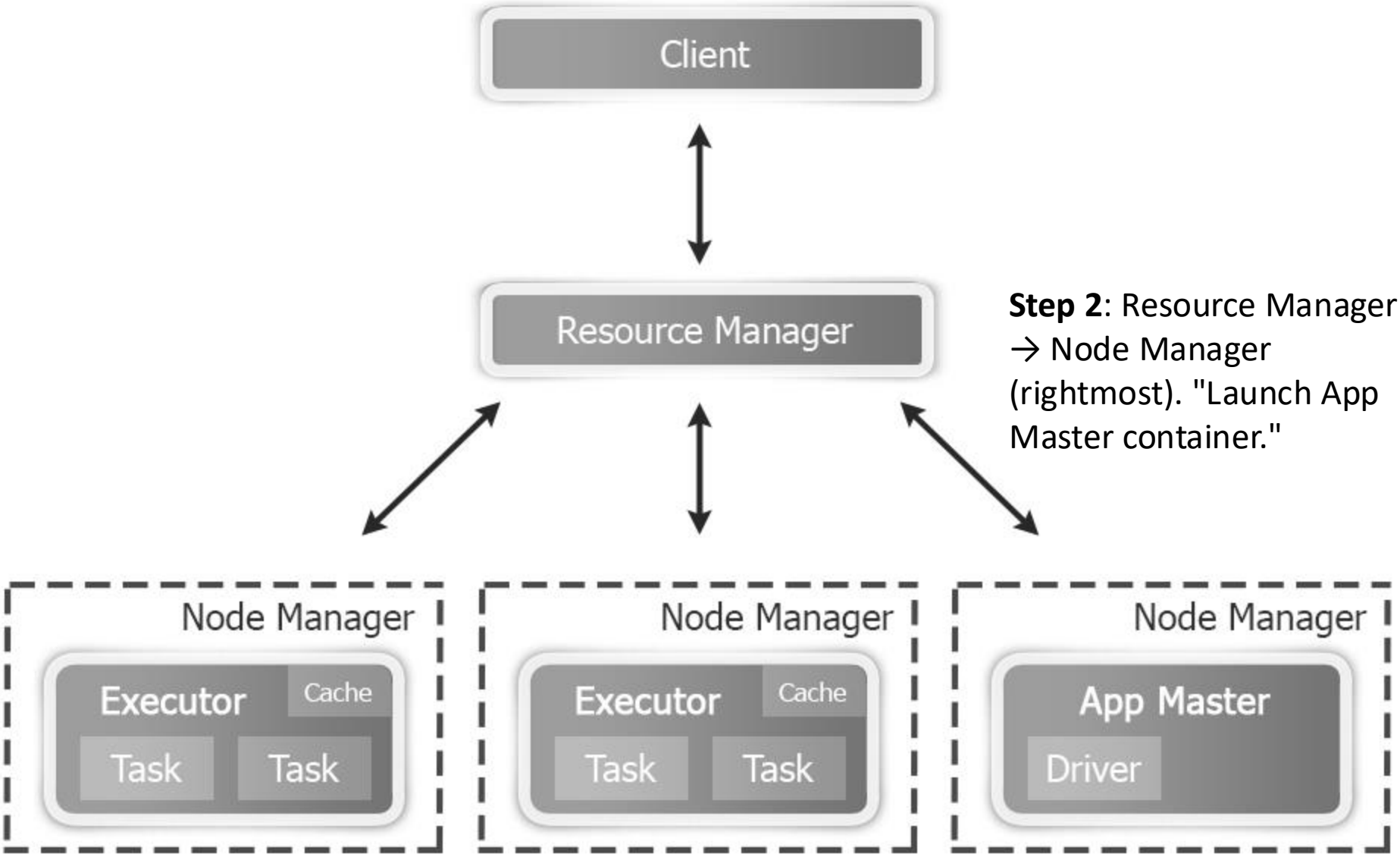
YARN Cluster

- **Resource Manager**: Runs on 1 node. Decides which nodes have spare resources for your application. It does NOT understand Spark (it just allocates containers).
- **Node Manager**: Runs on every worker node. Launches and monitors containers on that node.
- **Application Master**: Runs inside a container. For Spark, it contains the driver (in cluster-deploy mode). It requests executor containers from the Resource Manager and coordinates the application.
- **Executor**: Runs inside a container on a worker node. Receives tasks from the driver and executes them.

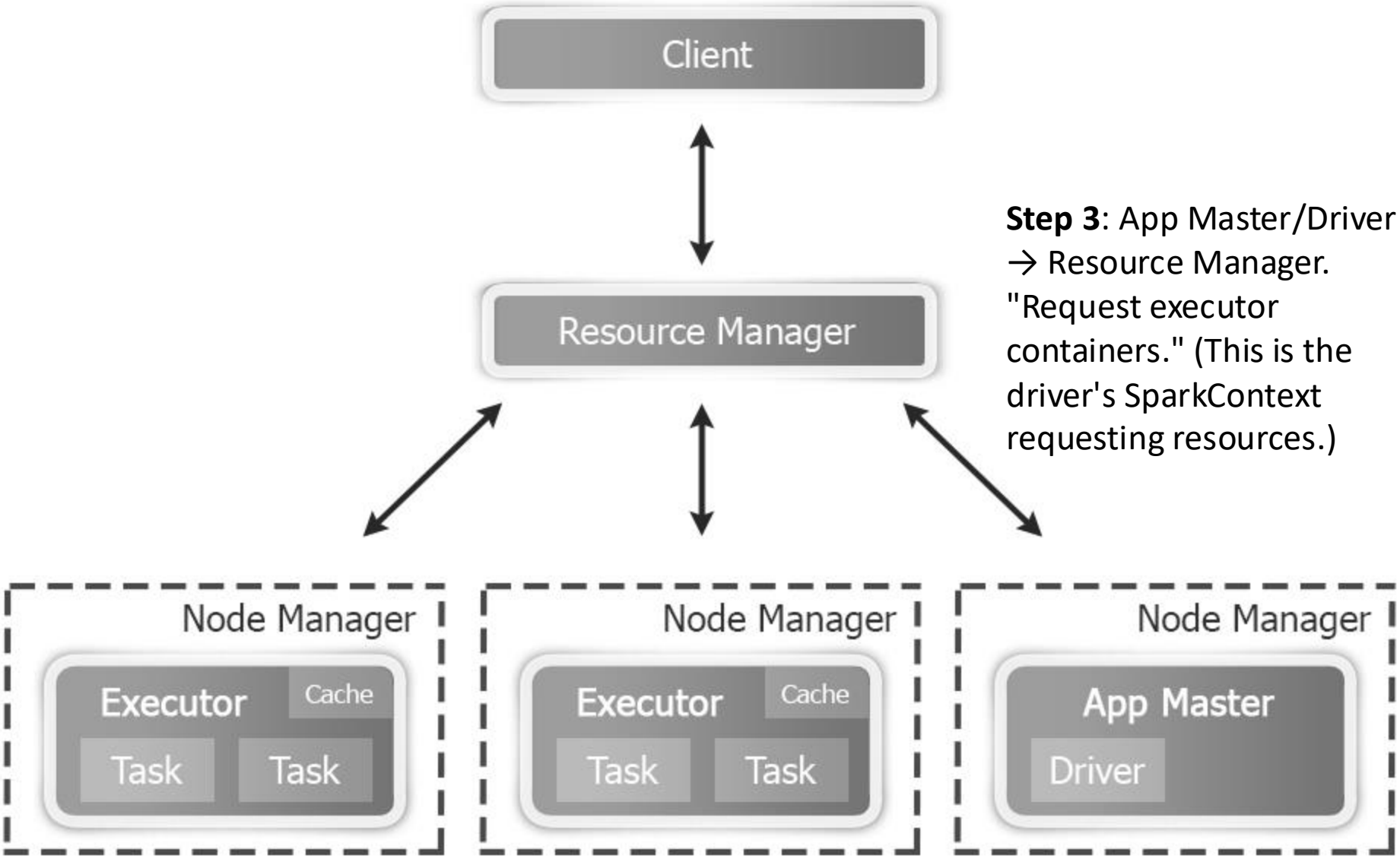

Client
Resource Manager
Node Manager
Executor
Cache
Task
Task
Node Manager
Executor
Cache
Task
Task
Node Manager
App Master
Driver



Step 1: Client → Resource Manager. "Submit application."

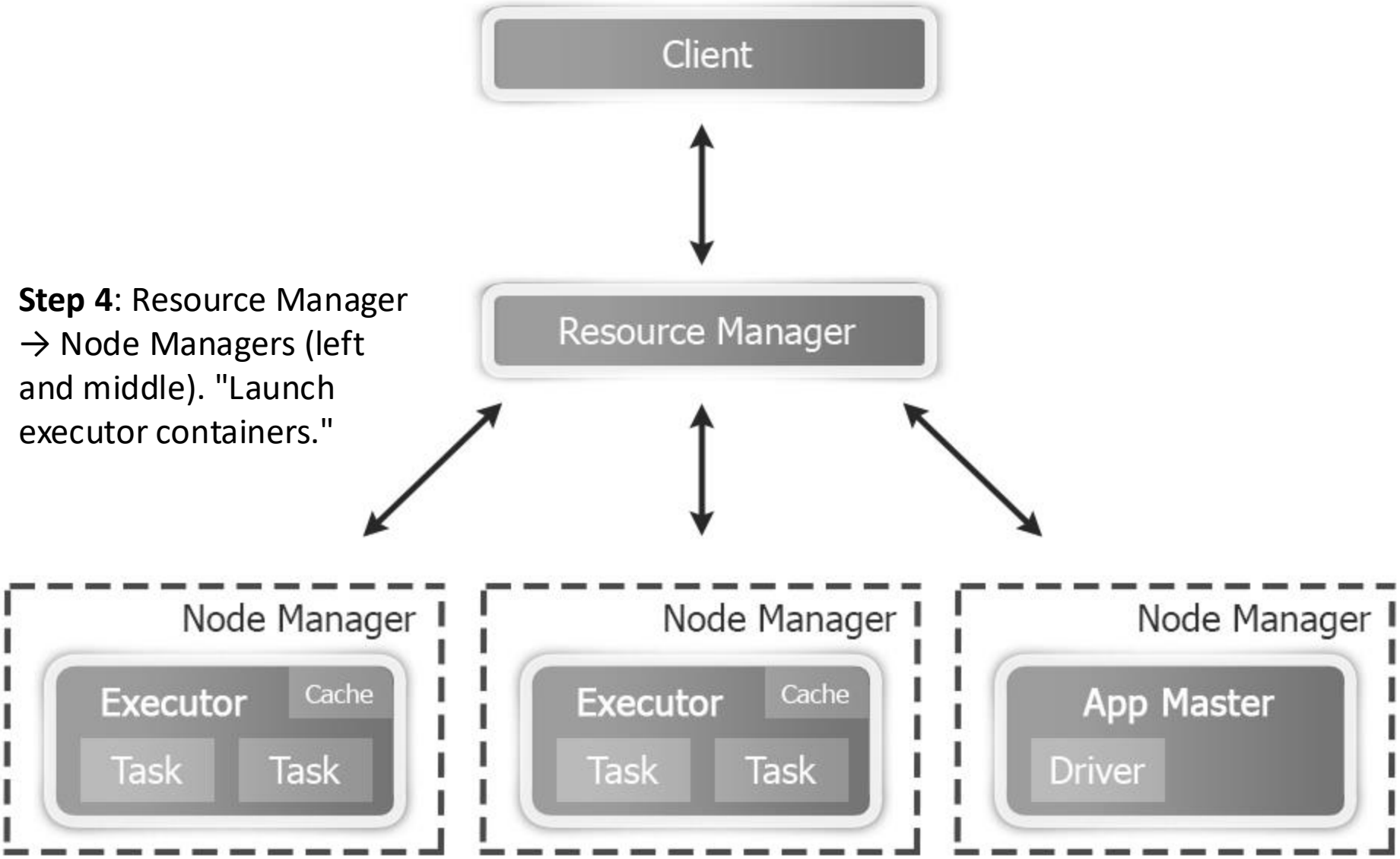


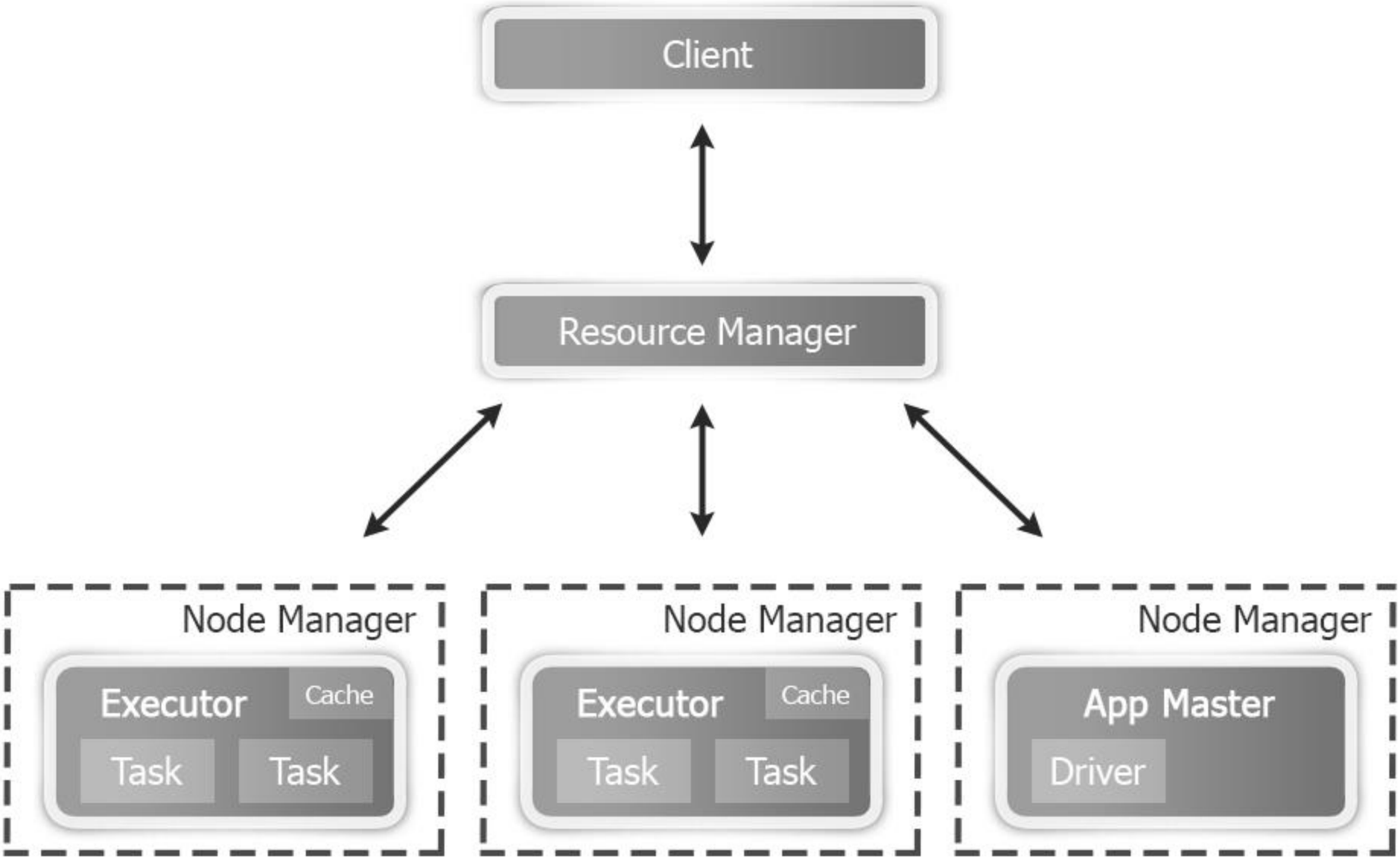
Step 2: Resource Manager → Node Manager (rightmost). "Launch App Master container."



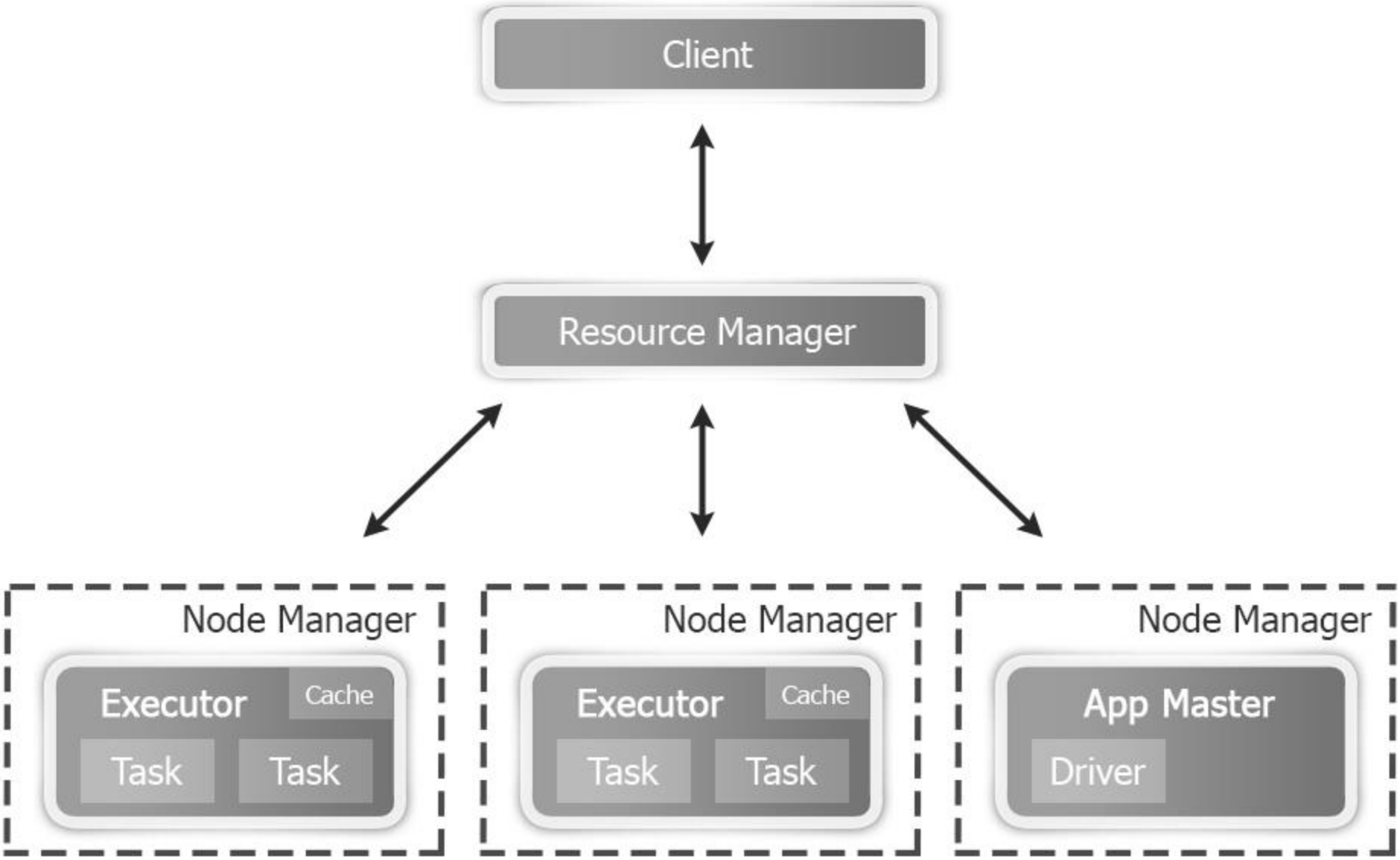
Step 3: App Master/Driver → Resource Manager. "Request executor containers." (This is the driver's SparkContext requesting resources.)

Step 4: Resource Manager → Node Managers (left and middle). "Launch executor containers."





Step 5: Driver ↔ Executors (direct arrows, bypassing Resource Manager).
"Driver assigns tasks to executors. YARN is no longer involved."



Step 6: Executors run tasks on their partitions. Results flow back to the driver.

Client-Deploy Mode

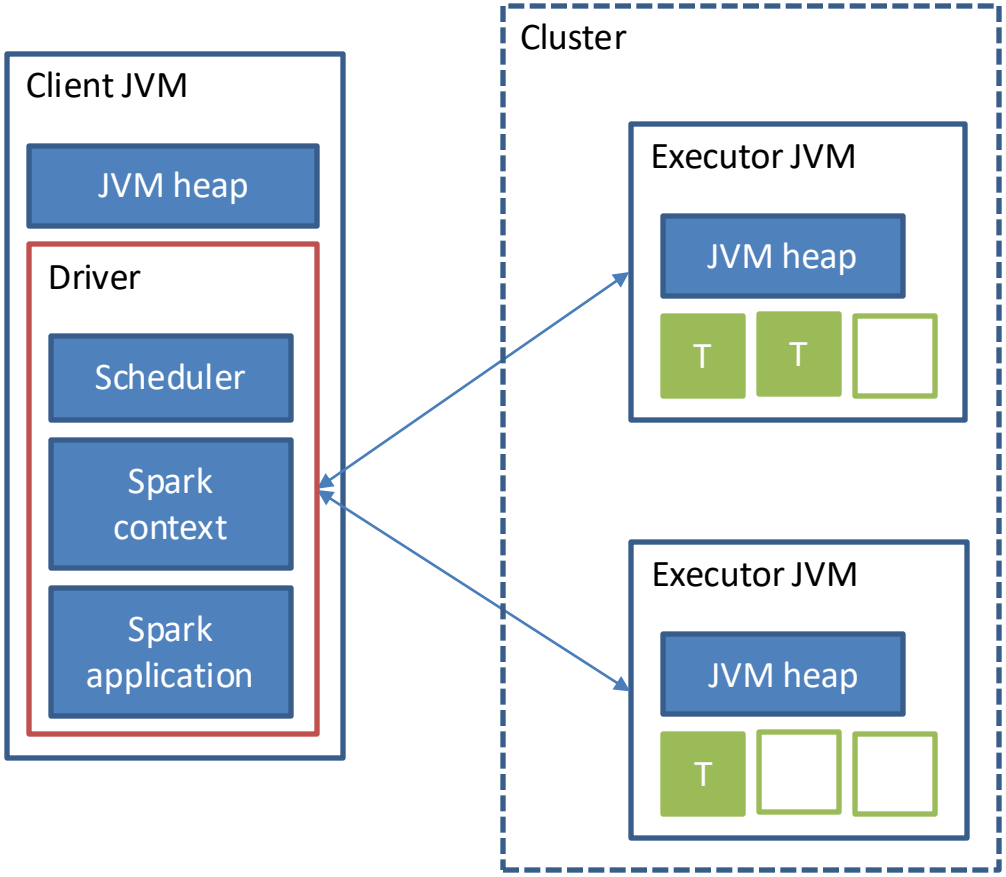


Illustration based on Zecevic/Bonaci book

Client vs. Cluster Deploy: When to Use Which

	Client Deploy	Cluster Deploy
Driver runs on	Your laptop	The cluster (inside App Master Container)
See output directly	Yes (check the print statements in your terminal)	No (check logs after finishing)
Laptop disconnects	Job dies	Job keeps running
Driver and executor communication	Across the network to your laptop	Fast (same cluster network)

- **Use client-deploy for**: interactive development, debugging, Spark shell, exploring data
- **Use cluster-deploy for**: production jobs, long-running jobs, anything you want to "submit and walk away"

Addressing Memory Problems

- Since Spark is optimized for in-memory execution, memory management is crucial for performance.

- The spark.executor.memory parameter determines the amount of memory allocated for an executor. This is the amount the cluster manager allocates. The Spark tasks running in the executor must work with this amount.

Avoiding Data Transfer

- Moving data across the network is expensive. When input data is already distributed across machines, the scheduler should assign tasks near their data.

- Like MapReduce, Spark's scheduler is location-aware. It prefers, in order: an executor that already has the partition cached in memory → a node that has the partition on local disk → a node on the same rack → anywhere.

Spark Programming Overview

- Programming in Spark is like writing a "local" program in Scala for execution on a single machine.
  - Parallelization, like in MapReduce, happens automatically under the hood.
- Pro: It is easy to write distributed programs.
  - Compare Word Count in Spark with Hadoop.
- Con: It is also easy to write inefficient distributed programs.

Word Count in Spark

- The Scala program below implements the same idea as the Hadoop program:
  - The flatMap call splits a line of text into words.
  - The map call converts word x to (x, 1). Like MapReduce, the first component of the pair is the key.
  - ReduceByKey automatically groups the (word, count) pairs by key and sums up the counts.
- The program is very concise, but it does not reveal when data is shuffled.

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Scala Function Literals

- Functions play a crucial role in Scala. To understand what a program does, we must understand the functions it uses.

- Let's start with one of the simplest programs: return all lines in a text file that contain string "Spark."

Scala Function Literals

- The program first loads the input data:
  val myFile = sc.textFile(“/somePath/Name”)

- sc is the Spark Context and textFile loads the file into an RDD called myFile where each element is a line of text. Keyword val, in contrast to var, indicates that myFile is immutable.

Scala Function Literals

- The next line does all the work:
  val sparkLines = myFile.filter(line =>
  line.contains("Spark"))
  - Filter returns all elements (i.e., lines of text) in myFile that satisfy the filter condition, i.e., contain "Spark."
  - The filter condition is also a function. It takes an element of myFile, named *line* here, and returns `true` if that element contains "Spark" and `false` otherwise.
    - Function (x => y) converts x to y. In the example, x is a line of text and y is the output of the contains() call on that line.

Alternative: Named Functions

Definition option 1:

```
def hasSpark(line: String) = { line.contains("Spark") }
```

Definition option 2:

```
val hasSpark = (line: String) => line.contains("Spark")
```

Usage:

```
val sparkLines = myFile.filter(hasSpark)
```

Spark Data Representation

- It helps to think of Spark's data abstractions as big tables that are horizontally partitioned across executors.
  - Each partition is processed by one task.
  - If the cluster has enough memory, all partitions fit in executor memory at the same time — the entire table is spread across the combined cluster memory.

- Spark's original data structure is the RDD. Intuitively, it is a table with a single column of type Object.

Spark Data Representation

- The pair RDD is a table with two columns: key and value. It supports special _ByKey operations.

- DataFrame (DataSet[Row]): A table with named columns and types, like a database table.
  - Spark sees the schema and can optimize automatically (e.g., drop unused columns before a shuffle). Columns accessed by name as strings — errors caught at runtime.

- DataSet[T]: Same as DataFrame but rows are typed Scala objects (e.g., DataSet[Student]).
  - Fields accessed as object properties — errors caught at compile time.

Spark Data Representation Example

RDD

Object of type String
(1, Amit, CS, 3.8)
(2, Bettina, CS, 4.0)
(3, Cong, ECE, 3.9)

Pair RDD

Key of type String	Value of type String
CS	(1, Amit, 3.8)
ECE	(2, Bettina, 4.0)
CS	(3, Cong, 3.9)

DataSet, DataFrame

ID: Int	Name: String	Department: String	GPA: Float
1	Amit	CS	3.8
2	Bettina	ECE	4.0
3	Cong	CS	3.9

Discussion of the Example

- Task: compute average GPA by department.
- With an RDD:

```
// Must parse every field manually, must manually drop unused fields
val avgByDeptRDD = studentsRDD
    .map(line => {
        val fields = line.split(",")
        (fields(2), fields(3).toDouble) // manually extract (dept, GPA), drop ID and Name
    })
    .groupByKey()                          // shuffle — but at least we dropped ID/Name first
    .mapValues(gpas => gpas.sum / gpas.size)
```

RDD

Object of type String
(1, Amit, CS, 3.8)
(2, Bettina, CS, 4.0)
(3, Cong, ECE, 3.9)

Discussion of the Example

- Task: compute average GPA by department.
- With a Pair RDD:

```
val avgByDeptPair = studentsPairRDD
    .mapValues(_._3)                    // manually drop ID and Name, keep GPA
    .aggregateByKey((0.0, 0))(
        ((sum, count), gpa) => (sum + gpa, count + 1), // within partition
    )
    .mapValues { (sum, count) => sum / count }
```

Pair RDD

Key of type String	Value of type String
CS	(1, Amit, 3.8)
ECE	(2, Bettina, 4.0)
CS	(3, Cong, 3.9)

Discussion of the Example

- Task: compute average GPA by department.
- With a Pair RDD:

```
val avgByDeptPair = studentsPairRDD
    .mapValues(_._3)                    // manually drop ID and Name, keep GPA
    .aggregateByKey((0.0, 0))(
        { case ((sum, count), gpa) => (sum + gpa, count + 1) }, // within partition
        { case ((s1, c1), (s2, c2)) => (s1 + s2, c1 + c2) }     // across partitions
    )
    .mapValues { case (sum, count) => sum / count }
```

Pair RDD

Key of type String	Value of type String
CS	(1, Amit, 3.8)
ECE	(2, Bettina, 4.0)
CS	(3, Cong, 3.9)

Discussion of the Example

- Task: compute average GPA by department.
- With a DataFrame:

```
val avgByDeptDF = studentsDF.groupBy("Department").avg("GPA")
```

DataSet, DataFrame

ID: Int	Name: String	Department: String	GPA: Float
1	Amit	CS	3.8
2	Bettina	ECE	4.0
3	Cong	CS	3.9

Discussion of the Example

- Task: compute average GPA by department.
- With a DataSet:

```
studentsDS
    .groupByKey(s => s.department) // type-safe: s.departmnet would fail at compile
    .mapGroups((dept, students) => {
        val gpas = students.map(_.gpa).toList
        (dept, gpas.sum / gpas.size)
    })
```

DataSet, DataFrame

ID: Int	Name: String	Department: String	GPA: Float
1	Amit	CS	3.8
2	Bettina	ECE	4.0
3	Cong	CS	3.9

Discussion of the Example

- **RDD**: Each row is an opaque object. Easy to load data, but the program must parse fields manually. Spark can't optimize what it can't see.

- **Pair RDD**: adds key-value structure, enabling built-in _byKey operations (groupByKey, reduceByKey). But you must choose the right key upfront (analyzing by a different column means reformatting).

- **DataFrame / DataSet**: full table schema with named, typed columns. Spark's optimizer can inspect the query and apply automatic optimizations (e.g., dropping unused columns before shuffling).

Discussion of the Example

- All four can solve the same problem. The difference is how much work you do vs. how much Spark does.

- Rule of thumb: use DataFrames and Datasets unless you have a specific reason notto.

Working with RDDs

- Transformation = an operation that converts an RDD to another RDD. It does not trigger an actual execution (lazy evaluation).

- Action = an operation that triggers execution. It may return a result to the client program.

- The intuition is simple: transformations describe *what to do*, actions say *do it now and give me something back*.
  - From the name of the function, it is often not obvious which is which. Typically, actions return small (final) results.

Lazy Evaluation

- Transformations define a graph of operations—think "workflow"—the lineage of the RDD.
- Given the lineage and initial input, any partition of an RDD can be re-created.
  - This is useful for fault tolerance or when RDD partitions are evicted from memory.
- An action triggers execution of *all* those transformations needed to produce the result of the action.

Lazy Evaluation

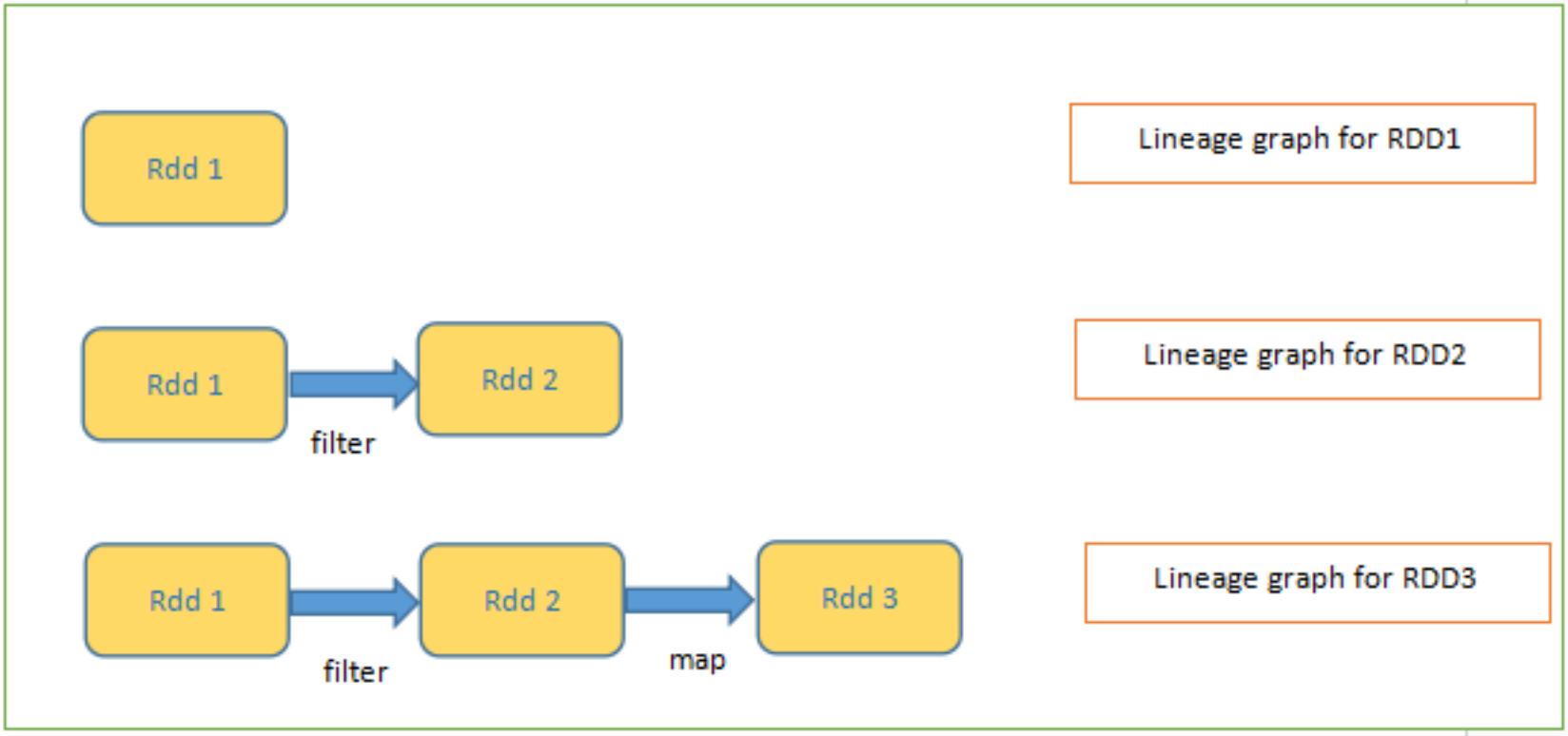
- Using Lazy Evaluation:

```
val rdd = sc.textFile("huge_file.txt") // 100 GB
val filtered = rdd.filter(_.contains("error")) // maybe 1 MB of results
val upper = filtered.map(_.toUpperCase)
upper.collect() // action triggers execution
```

- Spark doesn't read 100 GB, store the filtered result, then uppercase it in a second pass.

- It could elect to fuse the filter and map together into a single pass over the data. Read each line, check if it contains "error," if yes uppercase it, done. One pass, not two.

Lazy Evaluation



Program Execution

- A Spark job is divided into stages, defined by points where shuffles occur.
- For each stage, the driver creates tasks and sends them to the executors.
- The executors write their intermediate results to local disk. Shuffling is performed by special *shuffle-map* tasks.
- The last stage returns its results to the driver, using special *results* tasks.

RDD Dependencies

- RDD lineage forms a DAG: RDDs are vertices, dependencies are edges. Each transformation appends a new vertex.

- **Narrow dependency**: one-to-one relationship between input and output partitions. No shuffle.
  - Examples: map, filter, flatMap, mapValues.

- **Wide dependency**: output partitions depend on multiple input partitions. Requires a shuffle.
  - Examples: reduceByKey, groupByKey, join.

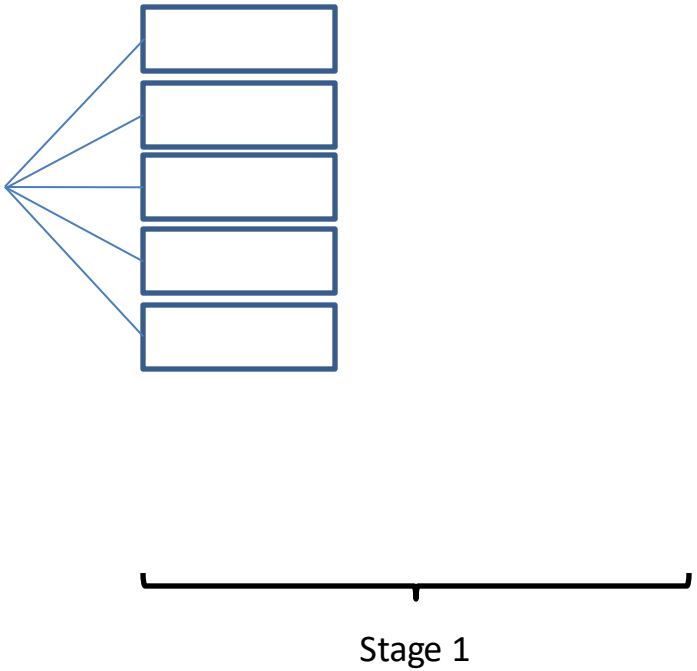
- Wide dependencies define **stage boundaries** — Spark executes all narrow transformations within a stage as a single pipeline, then shuffles before the next stage.

- Debugging tip: use println(myRDD.toDebugString) to see the DAG. "ShuffleRDD" indicates a wide dependency.

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
```

RDD lineage:

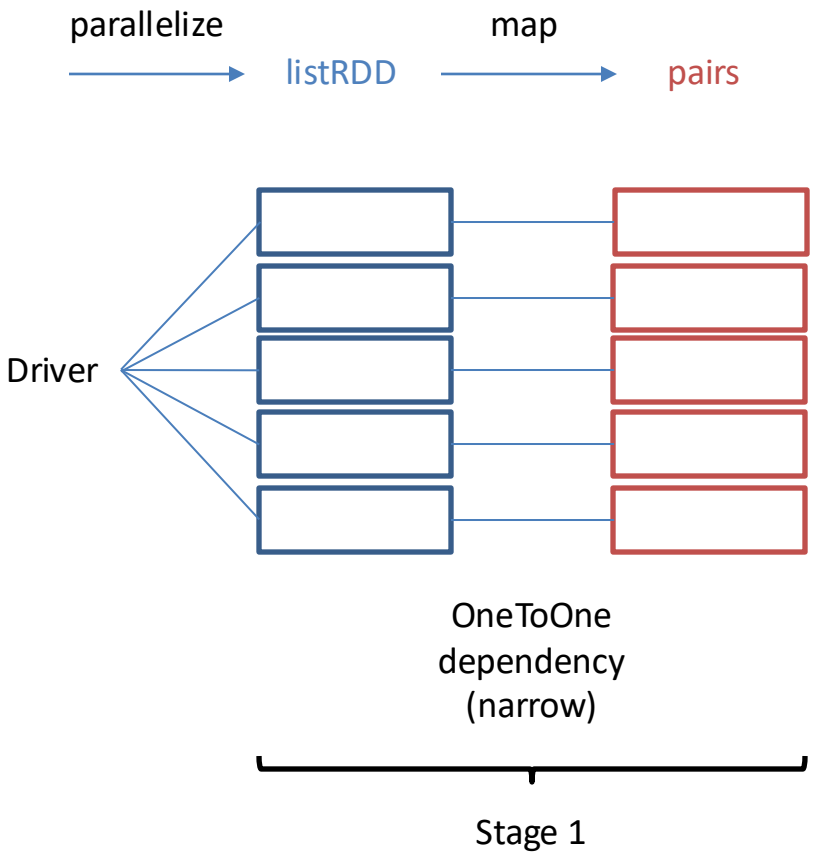
parallelize → listRDD



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
```

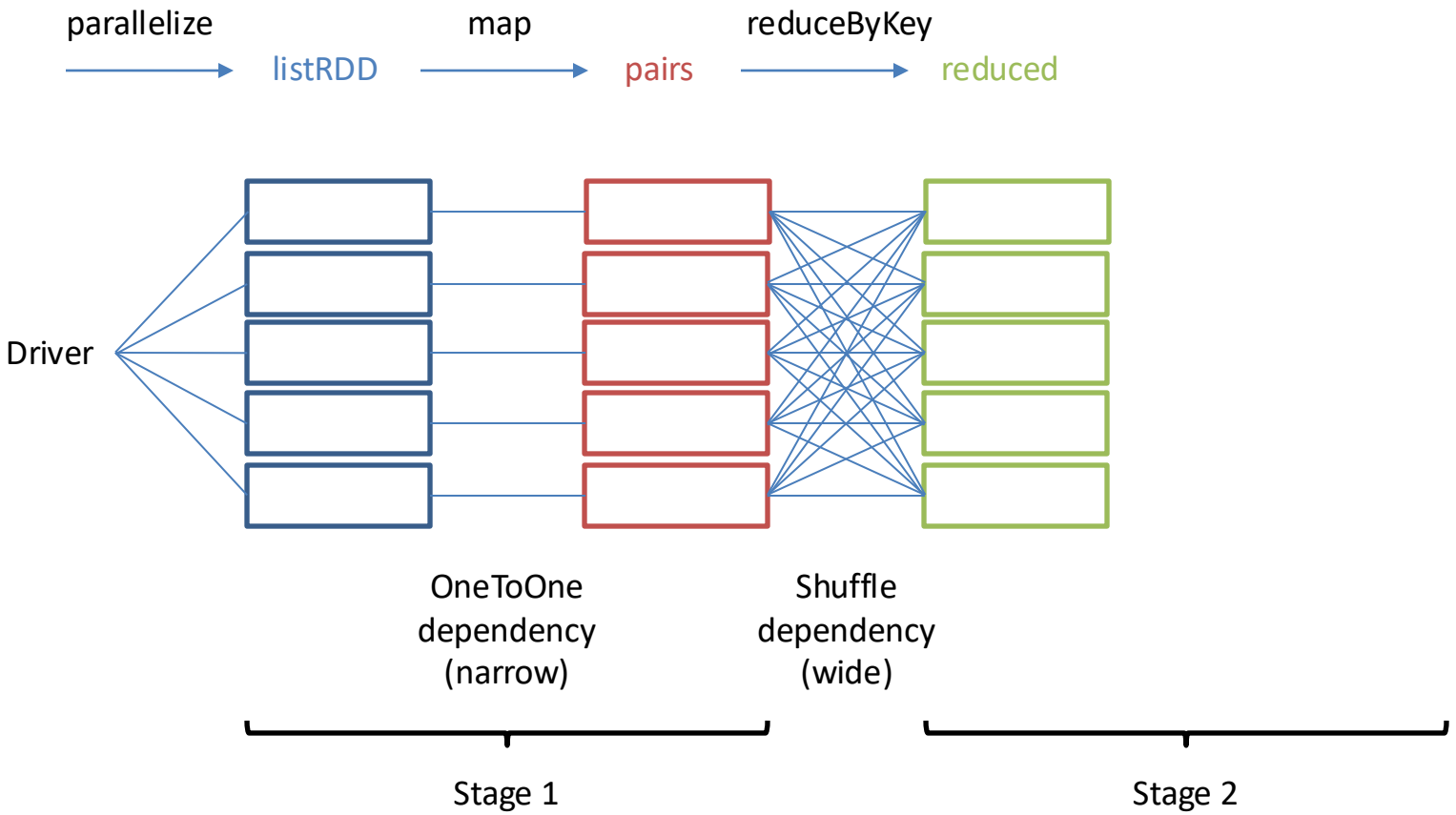
RDD lineage:



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
val reduced = pairs.reduceByKey( (v1, v2) => (v1+v2) )
```

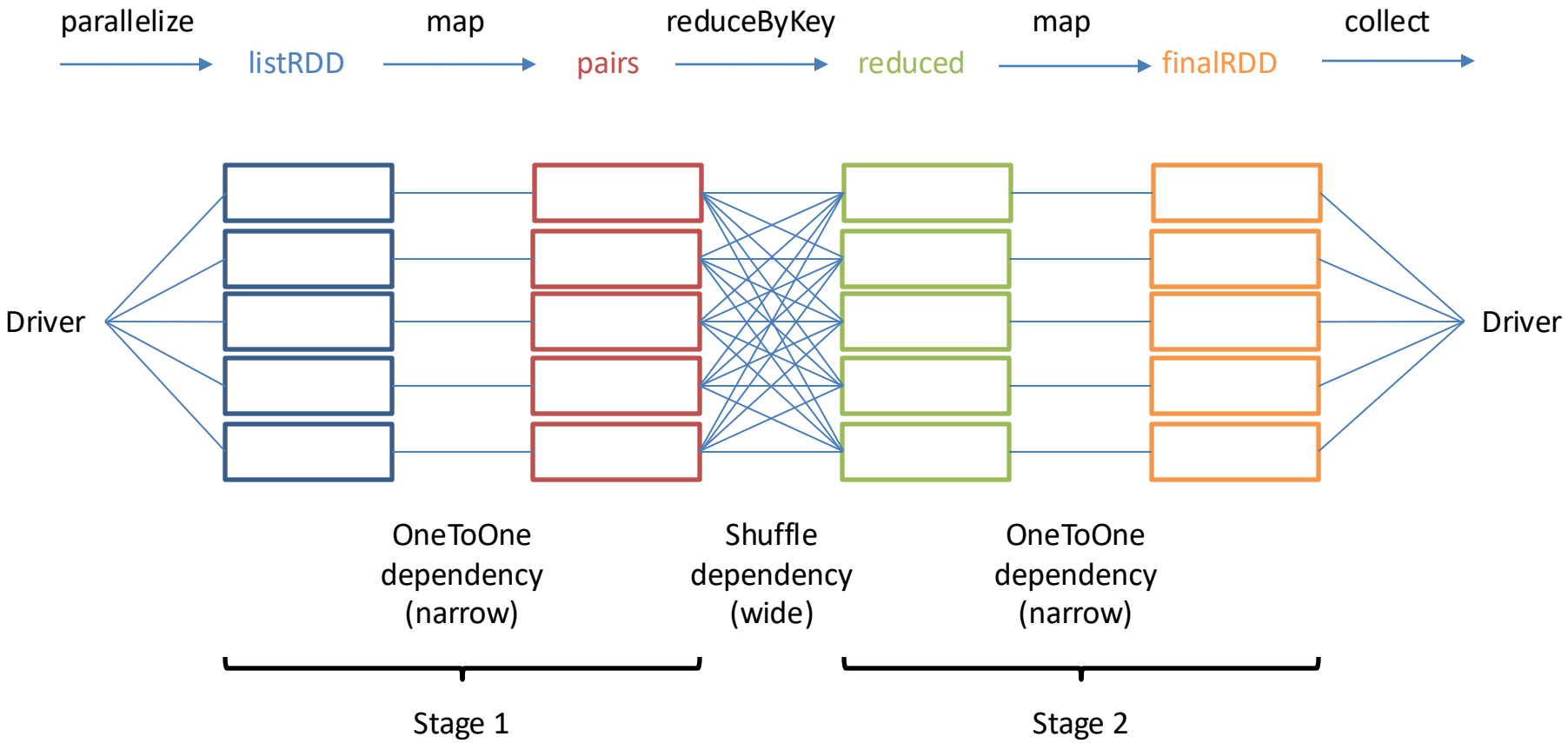
RDD lineage:



Example from on Zecevic/Bonaci book

```
val myList = List.fill(500) (scala.util.Random.nextInt(10) )
val listRDD = sc.parallelize(myList, 5)     // creates 5 partitions of the list
val pairs = listRDD.map(x => (x, x*x) )
val reduced = pairs.reduceByKey( (v1, v2) => (v1+v2) )
val finalRDD = reduced.map({ case (k, v) => "K=" + k + ", V=" + v }).collect()
```

RDD lineage:



Example from on Zecevic/Bonaci book

Pair RDD

- A pair RDD is an RDD where each element is a key-value pair (the same data model as MapReduce!).
  - Created by transformations that emit pairs, e.g., map(word => (word, 1)).

- Pair RDDs unlock _byKey operations that plain RDDs don't have: reduceByKey, groupByKey, aggregateByKey, sortByKey, join, and others.

- The key determines how data is partitioned and shuffled — just like the intermediate key in MapReduce determines which Reducer receives which data.

More Pair RDD Functions

- mapValues(oldVal => newVal) transformation replaces for each key the old value with a new value.
  - Here newVal stands not just for a constant but could be any kind of function applied to oldVal.

- flatMapValues(valueTransformationFunction) transformation associates each of the keys with zero or more new values.

More Pair RDD Functions

```
// Starting data
val scores = sc.parallelize(List(("alice", 80), ("bob", 90), ("alice", 70)))

// mapValues: apply a function to each value, keys unchanged
scores.mapValues(v => v + 10)
// → [("alice", 90), ("bob", 100), ("alice", 80)]
// No shuffle — keys didn't change, partitioning preserved.

// flatMapValues: each value produces zero or more new values
val tags = sc.parallelize(List(("alice", "CS,Math"), ("bob", "ECE")))
tags.flatMapValues(v => v.split(","))
// → [("alice", "CS"), ("alice", "Math"), ("bob", "ECE")]
// One input pair → multiple output pairs. No shuffle.
```

More Pair RDD Functions

- reduceByKey(mergeFunction) transformation merges all values associated with the same key into a single value, using a merge function with signature (U, U) => U.
  - Input and output values of the merge function have the same type. It merges two input values at a time, hence must be an associative function.

More Pair RDD Functions

```
// Starting data
val scores = sc.parallelize(List(("alice", 80), ("bob", 90), ("alice", 70), ("bob", 60)))

// reduceByKey: merge all values for the same key
// Function must be associative: (U, U) => U
scores.reduceByKey((a, b) => a + b)
// → [("alice", 150), ("bob", 150)]
// Triggers shuffle. Combines locally first (like a MapReduce Combiner).

// Compare with groupByKey:
scores.groupByKey()
// → [("alice", [80, 70]), ("bob", [90, 60])]
// Also triggers shuffle, but does NOT combine locally.
// All values shipped across the network, then grouped.
// Usually less efficient than reduceByKey.
```

More Pair RDD Functions

- foldByKey(zeroVal)(mergeFunction) transformation is the same as reduceByKey, but the additional parameter zeroVal allows more flexibility in defining the initial value.
  - The zeroVal is 0 for addition and counting, 1 for multiplication, and Nil for lists.

foldByKey Example

```
myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()
```

RDD myNumbers
partition 0: [(k1,2),(k5,4),(k1,6)]

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [**(k1,2)**,(k5,4),(k1,6)]

for key k1:
(0, 2) => 2

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [(k1,2),**(k5,4)**,(k1,6)]

for key k1:
(0, 2) => 2

for key k5:
(0, 4) => 4

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [(k1,2),(k5,4),**(k1,6)**]

for key k1:
(0, 2) => 2
(2, 6) => 8

for key k5:
(0, 4) => 4

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [(k1,2),(k5,4),(k1,6)]

for key k1:
(0, 2) => 2
(2, 6) => 8

for key k5:
(0, 4) => 4

RDD myNumbers
partition 1: [(k5,1),(k5,3),(k1,5)]

for key k5:
(0, 1) => 1
(1, 3) => 4

for key k1:
(0, 5) => 5

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [(k1,2),(k5,4),(k1,6)]

for key k1:
(0, 2) => 2
(2, 6) => 8

for key k5:
(0, 4) => 4

RDD myNumbers
partition 1: [(k5,1),(k5,3),(k1,5)]

for key k5:
(0, 1) => 1
(1, 3) => 4

for key k1:
(0, 5) => 5

Final aggregation:
for key k1:
(0, 8) => 8
(8, 5) => 13

for key k5:
(0, 4) => 4
(4, 4) => 8

foldByKey Example

myNumbers.foldByKey(0)((n1, n2) => n1+n2).collect()

RDD myNumbers
partition 0: [(k1,2),(k5,4),(k1,6)]

for key k1:
(0, 2) => 2
(2, 6) => 8

for key k5:
(0, 4) => 4

RDD myNumbers
partition 1: [(k5,1),(k5,3),(k1,5)]

for key k5:
(0, 1) => 1
(1, 3) => 4

for key k1:
(0, 5) => 5

Final aggregation:
for key k1:
(0, 8) => 8
(8, 5) => 13

for key k5:
(0, 4) => 4
(4, 4) => 8

What happens is we set zeroVal to 10, instead of 0?

aggregateByKey

- reduceByKey: combines values directly. Works when the partial result is the same type as the values.
  - Sum: 80 + 70 = 150. Partial result is a number, values are numbers. Same shape.
- aggregateByKey: accumulates values into a different shape. Works when the partial result needs more information than a single value.
  - Average: you need (sum, count) = Different shape.
- Two functions:
  - **"Put a value into the bag"**: (bag, value) → updated bag (runs within each partition)
  - **"Merge two bags"**: (bag, bag) → combined bag (runs across partitions)

What does this compute?

```
myNumbers.aggregateByKey((0, 0))
                                    (((s, c), v) => (s+v, c+1))
                                    (((s1,c1), (s2,c2)) => ((s1+s2), (c1+c2))).collect()
```

DataFrame and DataSet

- Introduced in Spark 1.3, DataFrame is an RDD with a schema. Like for a database table, the schema defines the name and type of each column.
  - The schema enables Spark to perform better optimization of a computation.
- Introduced in Spark 1.6, DataSet generalizes DataFrame.
  - They offer additional functionality compared to RDDs, similar to the key-based operations for pair RDDs.
- Internally, they are managed as RDDs. Hence all RDD functionality applies.

DataSet vs. Pair RDD

- DataSet overall offers a better abstraction than the limited pair RDD.

- Consider a table with columns (year, month, day, sale). To compute annual sales, a pair RDD needs to have year as the key. For an analysis of seasonal sales patterns, we need month as the key, or maybe a composite key (year, month).

DataSet vs. Pair RDD

- With pair RDDs, this requires clumsy data "re-formatting," just to make the right column(s) become the key. With DataSets, all these different groupings can be performed on the same DataSet instance.

- Unfortunately, the semantics of DataSet functions can be a little more complex. This will be discussed in future modules.

Checkpointing

- Problem: for long computations (e.g., 100 iterations), the lineage grows very long. Recomputing a lost partition means replaying the entire chain from the beginning.

- Solution: rdd.checkpoint() saves the RDD to the DFS and cuts the lineage at that point.
  - Before checkpoint: partition lost → replay 100 steps to rebuild it
  - After checkpoint: partition lost → read it from DFS (one disk read)

- Tradeoff: checkpointing is expensive (you're writing to DFS, just like MapReduce). But for very long lineages, paying that cost once is cheaper than risking a full replay.

Global State in Spark

- Global state requires communication across the cluster, which is expensive. Spark intentionally limits global state to two mechanisms:

- **Accumulators**: tasks write, driver reads. Use for counters and simple aggregates (e.g., counting parse errors across partitions).

- **Broadcast variables**: driver writes, tasks read. Use for sharing large read-only data with all tasks (e.g., a lookup table).

Accumulators

- Like global counters in MapReduce, accumulators implement global sums and counts, allowing only adding of values.
  - Define custom accumulators by extending AccumulatorV2[IN, OUT], implementing functions add, merge, copy, reset, isZero, and value.
  - It might be fun to subvert the accumulator by encoding complex objects in it. This may result in poor performance and does not substitute for good distributed-algorithm design!

Accumulators

- How they should be used:
  - Create accumulator with Spark context in the driver; set initial value in the driver.
  - Update it in the executors.
  - Read its value in the driver. The value cannot be read by the executors, i.e., in a task.

```
// abstract class AccumulatorV2[IN, OUT] extends Serializable
// class LongAccumulator extends AccumulatorV2[Long, Long]
// spark refers to a SparkSession object.
val lineCount = spark.sparkContext.longAccumulator

myRDD.foreach(line => lineCount.add(1))    // This operation is applied to an RDD partition, i.e., in a task
println(lineCount.value)    // For println, the argument is sent to the driver. Hence this operation happens in the driver.

// This throws an exception
myRDD.foreach(line => lineCount.value)    // This operation is applied to an RDD partition, i.e., in a task
```

Broadcast Variables

- Broadcast variables are read-only data shared with all executors. Created in the driver, sent once per machine (not once per task).
- Use for large read-only data that many tasks need — e.g., a lookup table, a set of stop words.
- Without broadcast: if your code references a variable from the driver, Spark ships a copy with every task.
  - 100 tasks on the same executor = 100 copies of the same data.
- With broadcast: one copy per executor, shared across all its tasks.

```
val stopWords = spark.sparkContext.broadcast(Set("the", "a", "is", "of"))
myRDD.filter(word => !stopWords.value.contains(word))
```

Controlling RDD Partitions

- Partitioning answers the same question as in MapReduce: "which task is responsible for which data?"

- **HashPartitioner** (default): hash(key) % numPartitions. Fast and simple. Keys are scattered randomly across partitions — good load balance, no ordering.
  - If partitions are heavily skewed in size, you can rebalance with repartition(n) (triggers a shuffle) or coalesce(n) (avoids shuffle when reducing partitions).

- **RangePartitioner**: Keys are sorted and split into ranges. Partition 0 gets A–F, partition 1 gets G–M, etc. Useful when you need sorted output.

Data Shuffling

- The rule: if an operation needs data from other partitions, it shuffles. If it can work partition-by-partition, it doesn't.

- **No shuffle** (each task works on its own partition independently):
  - map, flatMap, filter, mapValues, flatMapValues, union

- **Shuffle** (data must move between partitions):
  - reduceByKey, groupByKey, aggregateByKey, foldByKey
  - join, leftOuterJoin, rightOuterJoin, fullOuterJoin

Data Shuffling

- Subtle point: map and flatMap remove the Partitioner (because they might change keys) but don't shuffle.

- A subsequent reduceByKey will then trigger a shuffle. mapValues preserves the Partitioner because it can't touch keys — so a following reduceByKey may skip the shuffle entirely.

References

- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
  - https://scholar.google.com/scholar?cluster=10940266603640308767&hl=en&as_sdt=0,22
- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proc. of the 2nd USENIX conference on Hot topics in cloud computing* (HotCloud'10)
  - https://scholar.google.com/scholar?cluster=14934743972440878947&hl=en&as_sdt=0,22
- Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. Abstract Cost Models for Distributed Data-Intensive Computations. In *Distributed and Parallel Databases* 37(3): 411-439 (2019), Springer
  - http://www.ccs.neu.edu/home/mirek/papers/2018-DAPD-AbstractCostModels-preprint.pdf

References

- Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* 1, 1 (January 1997), 29-53
  - https://scholar.google.com/scholar?cluster=10836794633940449959&hl=en&as_sdt=0,22