An Introduction to Apache Spark

Anastasios **S**karlatidis
@anskarl

Software Engineer/Researcher
IIT, NCSR "Demokritos"

Outline

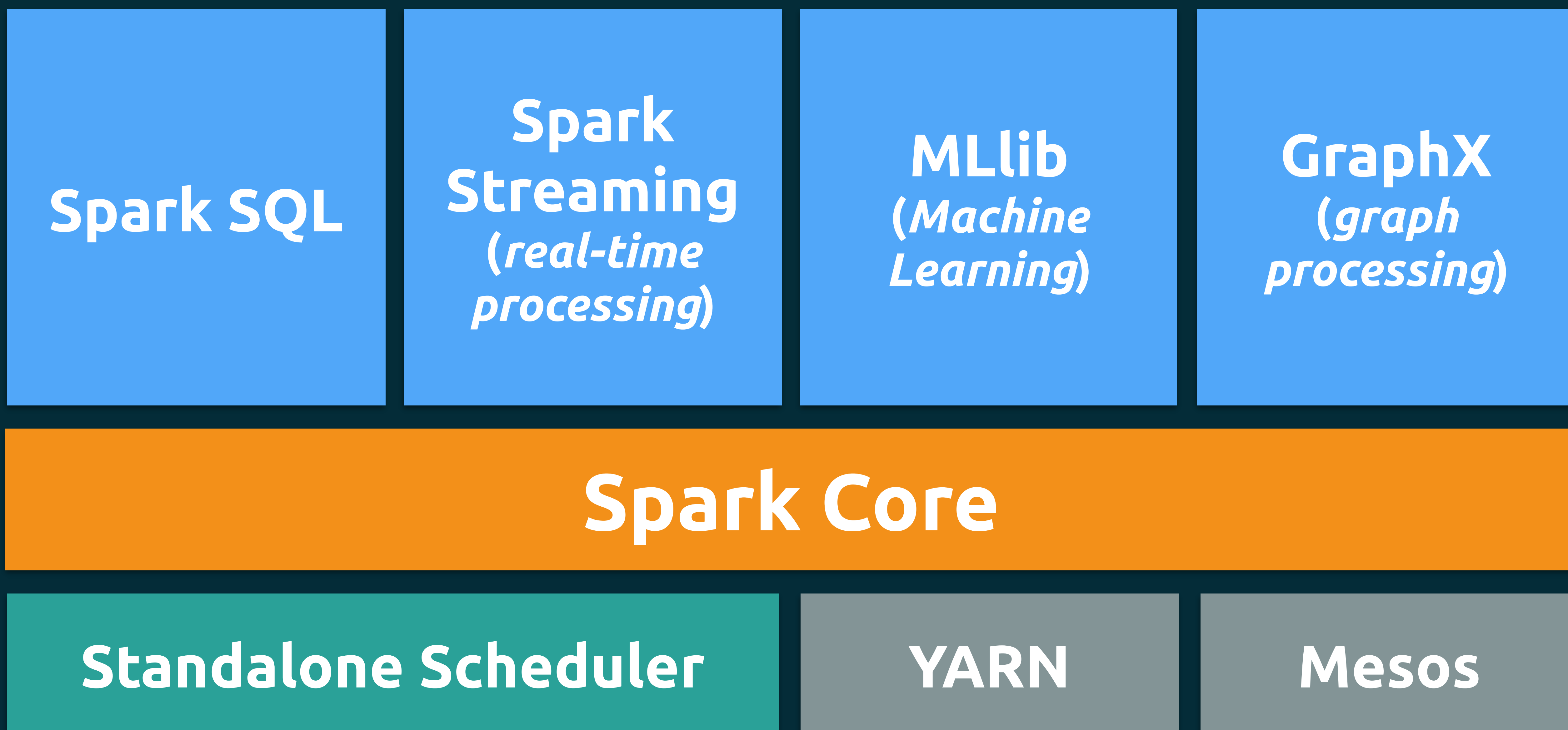
- Part I: Getting to know Spark
- Part II: Basic programming
- Part III: Spark under the hood
- Part IV: Advanced features

Part I:
Getting to know Spark

Spark in a Nutshell

- **General cluster computing platform:**
  - **Distributed in-memory** computational framework.
  - SQL, Machine Learning, Stream Processing, etc.
- **Easy to use, powerful, high-level API:**
  - Scala, Java, Python and R.

Unified Stack



High Performance

- **In-memory** cluster computing.
- Ideal for **iterative algorithms**.
- Faster than Hadoop:
  - **10x on disk.**
  - **100x in memory.**

Brief History

- Originally developed in 2009, **UC Berkeley AMP Lab**.
- **Open-sourced** in 2010.
- As of 2014, Spark is a **top-level Apache project**.
- Fastest open-source engine for **sorting 100 TB**:
  - Won the 2014 Daytona GraySort contest.
  - Throughput: **4.27 TB/min**

Who uses Spark, and for what?

A. Data Scientists:

- Analyze and model data.
- Data transformations and prototyping.
- Statistics and Machine Learning.

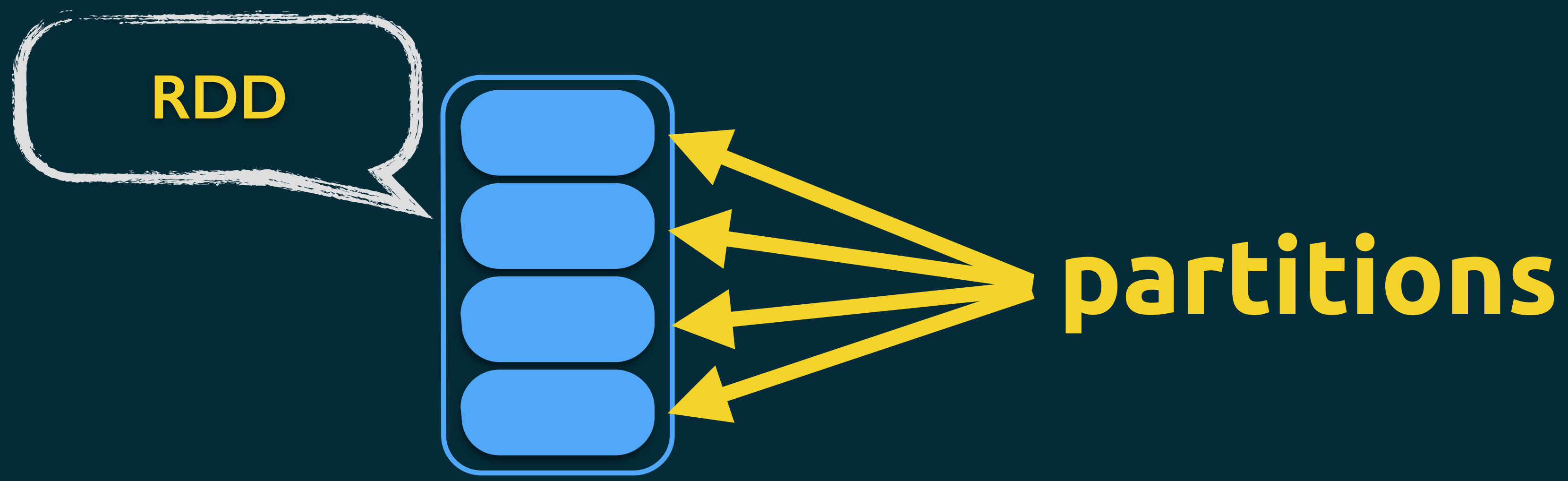
B. Software Engineers:

- Implement production data processing systems.
- Require a reasonable API for distributed processing.
- Reliable, high performance, easy to monitor platform.

Resilient Distributed Dataset

RDD is an **immutable** and **partitioned** collection:

- **R**esilient: it can be **recreated**, when data in memory is lost.
- **D**istributed: stored **in memory** across the **cluster**.
- **D**ataset: data that comes from file or created programmatically.



Resilient Distributed Datasets

- Feels like coding using typical Scala collections.
- RDD can be build:
  1. **Directly** from a datasource (e.g., text file, HDFS, etc.),
  2. or by applying a ***transformation*** to another RDD(s).
- **Main features:**
  - RDDs are **computed *lazily***.
  - Automatically ***rebuild on failure***.
  - ***Persistence*** for reuse (RAM and/or disk).

Part II:
Basic programming

Spark Shell

```
$ cd spark
$ ./bin/spark-shell

Spark assembly has been built with Hive, including Datanucleus jars on classpath
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.

scala>
```

Standalone Applications

Sbt:

```
"org.apache.spark" %% "spark-core" % "1.2.1"
```

Maven:

```
groupId: org.apache.spark
artifactId: spark-core_2.10
version: 1.2.1
```

Initiate Spark Context

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp extends App {

  val conf = new SparkConf().setAppName("Hello Spark")

  val sc = new SparkContext(conf)

}
```

Rich, High-level API

`map` `reduce`

Rich, High-level API

map	**reduce**	sample
filter	count	take
sort	fold	first
groupBy	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
…	zip	save
	…	…

Loading and Saving

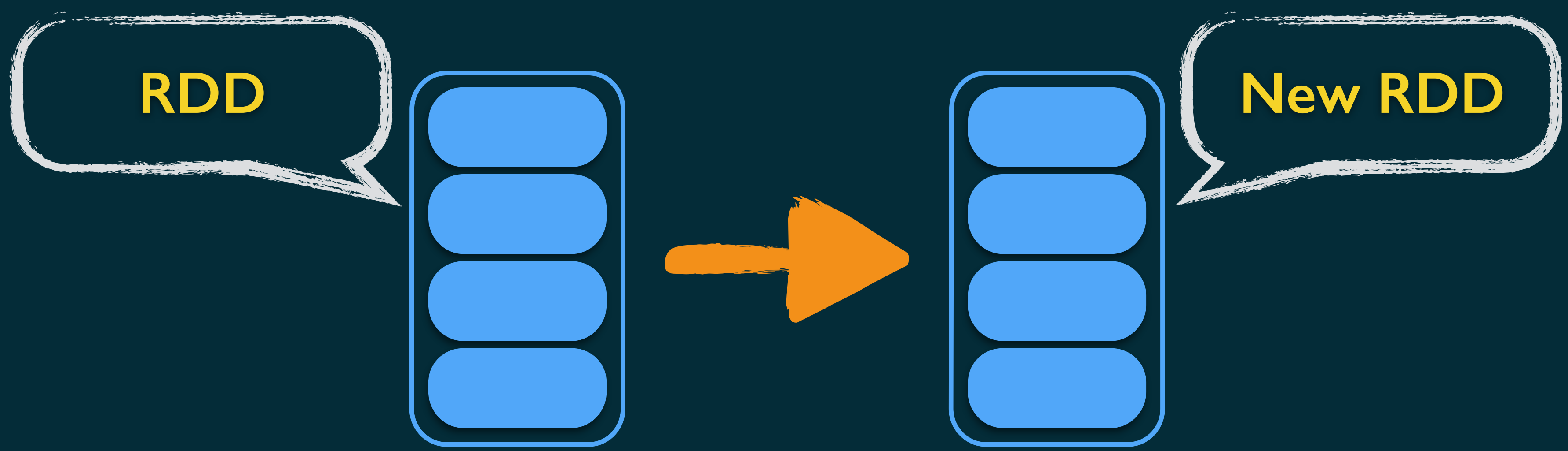
- **File Systems:** Local FS, Amazon S3 and HDFS.
- **Supported formats:** Text files, JSON, Hadoop sequence files, parquet files, protocol buffers and object files.
- **Structured data with Spark SQL:** Hive, JSON, JDBC, Cassandra, HBase and ElasticSearch.

Create RDDs

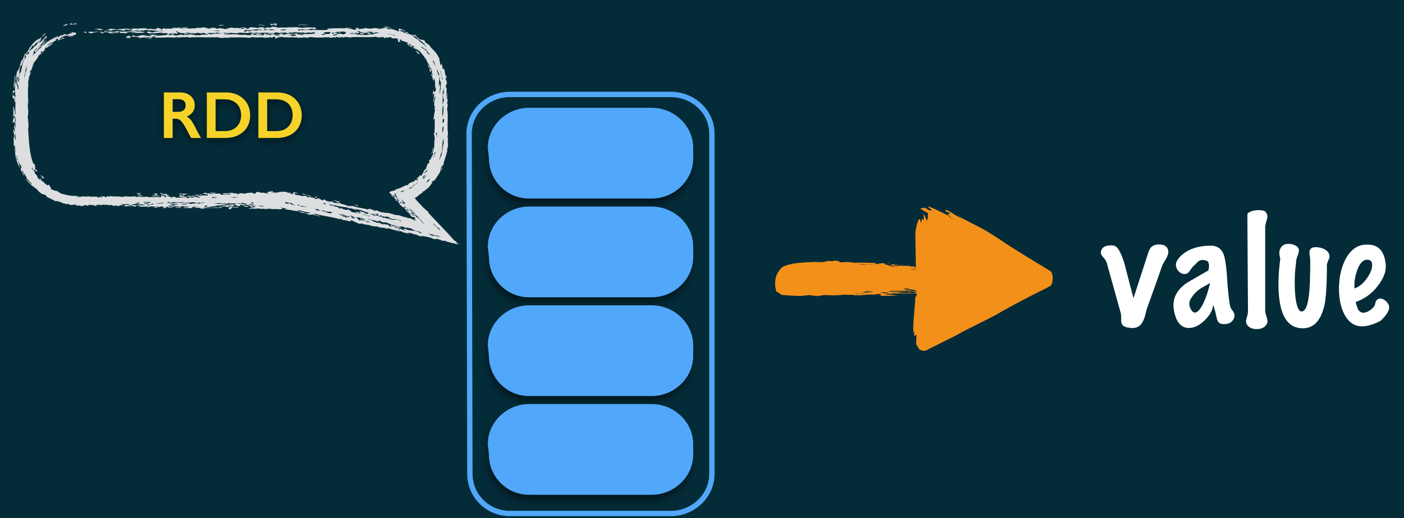
```scala
// sc: SparkContext instance

// Scala List to RDD
val rdd0 = sc.parallelize(List(1, 2, 3, 4))

// Load lines of a text file
val rdd1 = sc.textFile("path/to/filename.txt")

// Load a file from HDFS
val rdd2 = sc.hadoopFile("hdfs://master:port/path")

// Load lines of a compressed text file
val rdd3 = sc.textFile("file:///path/to/compressedText.gz")

// Load lines of multiple files
val rdd4 = sc.textFile("s3n://log-files/2014/*.log")
```

RDD Operations

1. **Transformations**: define new RDDs based on current one, e.g., filter, map, reduce, groupBy, etc.



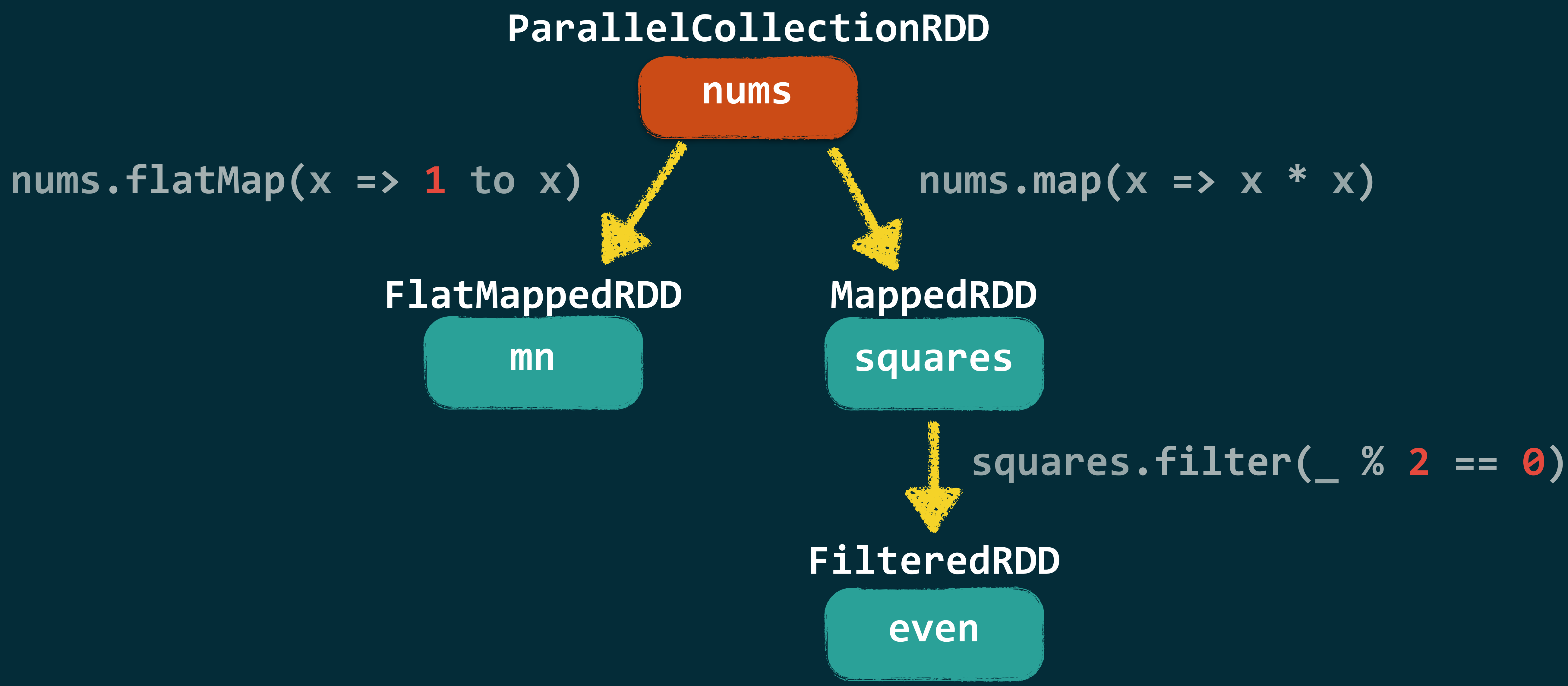
2. **Actions**: return values, e.g., count, sum, collect, etc.



Transformations (I): basics

```scala
val nums = sc.parallelize(List(1, 2, 3))

// Pass each element through a function
val squares = nums.map(x => x * x) //{1, 4, 9}

// Keep elements passing a predicate
val even = squares.filter(_ % 2 == 0) //{4}

// Map each element to zero or more others
val mn = nums.flatMap(x => 1 to x) //{1, 1, 2, 1, 2, 3}
```

Transformations (I): illustrated



Transformations (II): key - value

```scala
val pets = sc.parallelize(List(("cat", 1), ("dog", 1),
("cat", 2)))
```

Key Value

```scala
pets.filter{case (k, v) => k == "cat"}
// {(cat,1), (cat,2)}

pets.map{case (k, v) => (k, v + 1)}
// {(cat,2), (dog,2), (cat,3)}

pets.mapValues(v => v + 1)
// {(cat,2), (dog,2), (cat,3)}
```

Transformations (II): key - value

```
val pets = sc.parallelize(List(("cat", 1), ("dog", 1),
("cat", 2)))
```

Key

Value

```
// Aggregation
pets.reduceByKey((l, r) => l + r) //{(cat,3), (dog,1)}

// Grouping
pets.groupByKey() //{(cat, Seq(1, 2)), (dog, Seq(1)}

// Sorting
pets.sortByKey()  //{(cat, 1), (cat, 2), (dog, 1)}
```

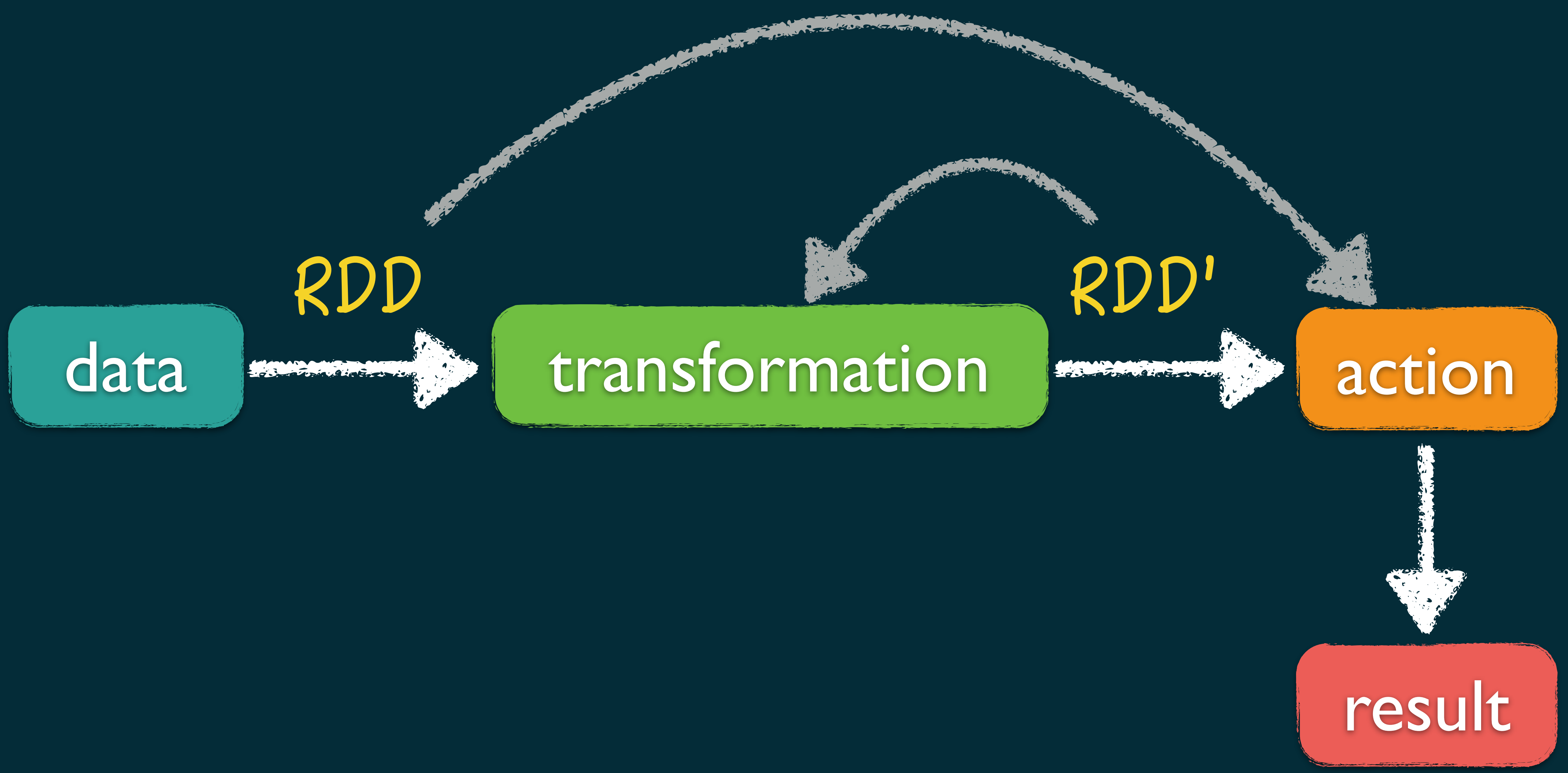
Transformations (III): key - value

```
//RDD[(URL, page_name)] tuples
val names = sc.textFile("names.txt").map(…)…

//RDD[(URL, visit_counts)] tuples
val visits = sc.textFile("counts.txt").map(…)…

//RDD[(URL, (visit counts, page name))]
val joined = visits.join(names)
```

Basics: Actions

```
val nums = sc.parallelize(List(1, 2, 3))

// Count number of elements
nums.count()   // = 3

// Merge with an associative function
nums.reduce((l, r) => l + r)  // = 6

// Write elements to a text file
nums.saveAsTextFile("path/to/filename.txt")
```

Workflow

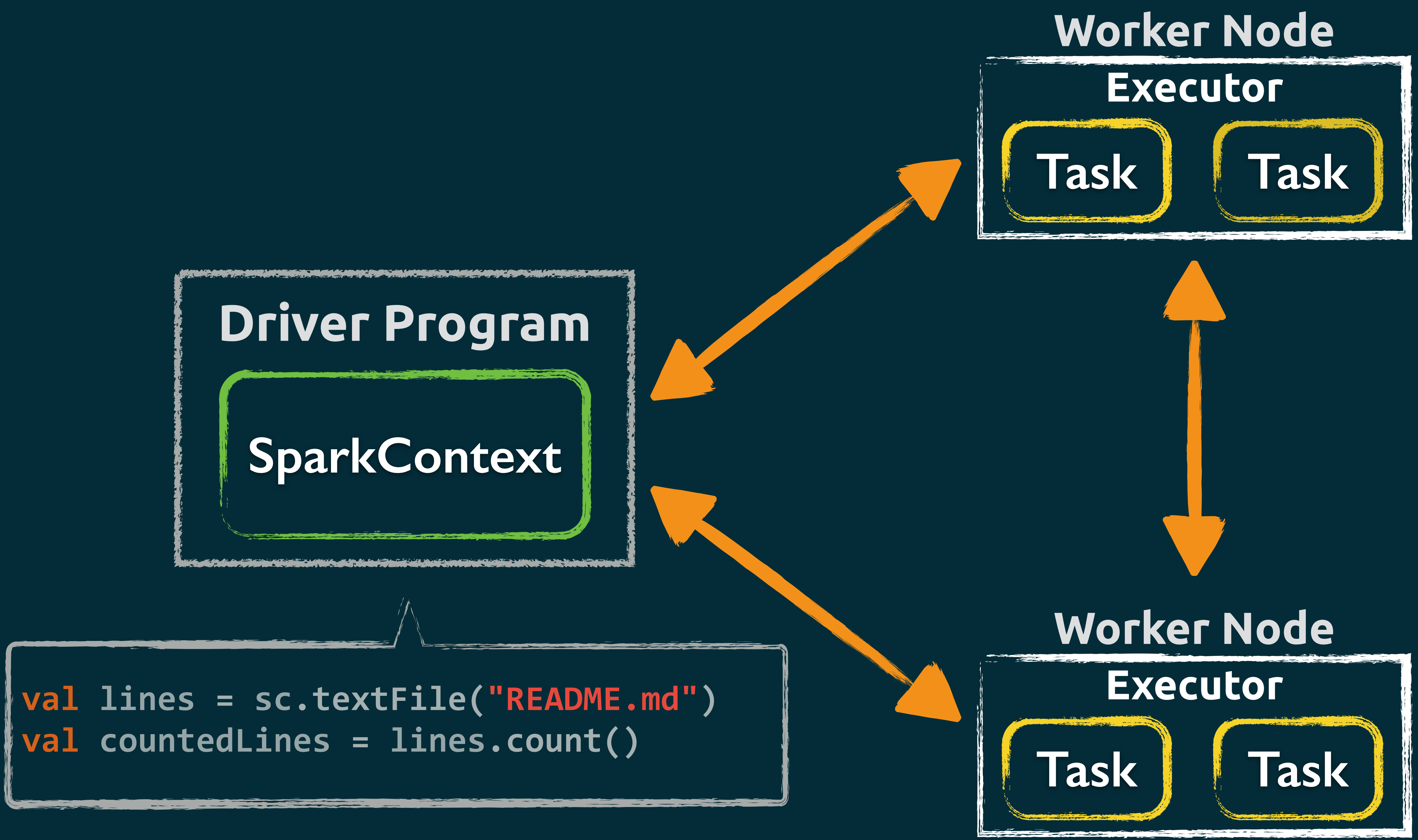


Part III:
Spark under the hood

Units of Execution Model

1. **Job**: work required to compute an RDD.
2. Each job is divided to **stages**.
3. **Task**:
   - Unit of work within a stage
   - Corresponds to **one RDD partition**.

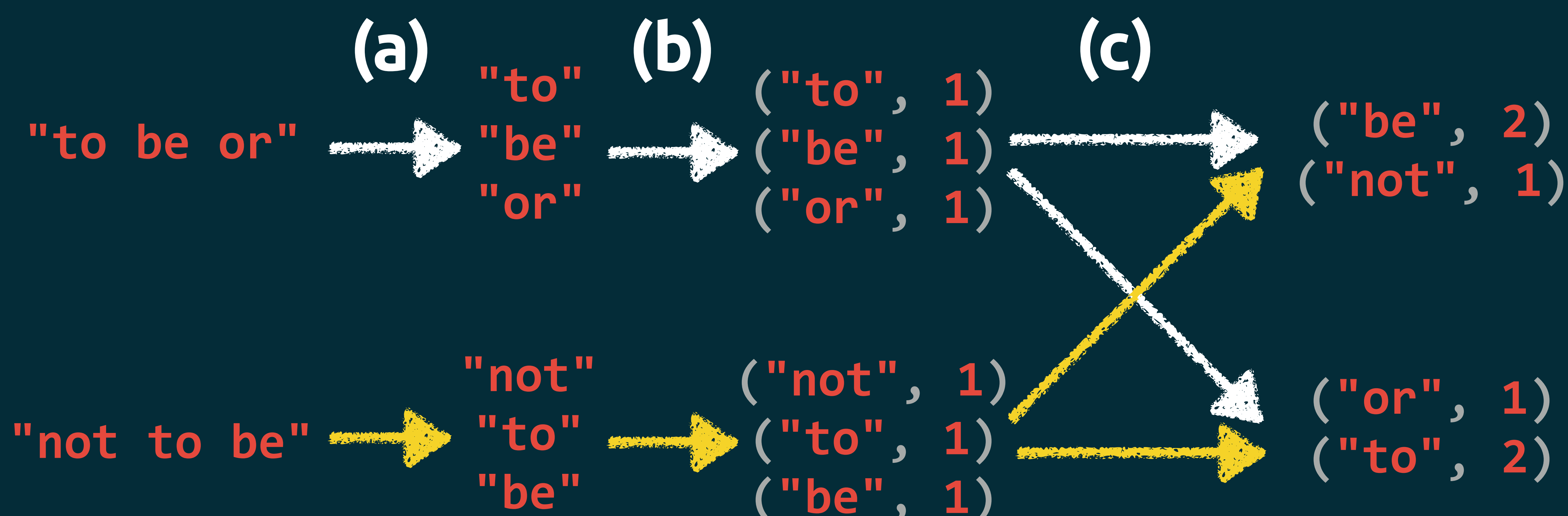
Execution Model



Example: word count

```
val lines = sc.textFile("hamlet.txt")

val counts = lines.flatMap(_.split(" "))   // (a)
                  .map(word => (word, 1))  // (b)
                  .reduceByKey(_ + _)      // (c)
```

to be or
not to be



Visualize an RDD

```
12: val lines = sc.textFile("hamlet.txt") // HadoopRDD[0], MappedRDD[1]
13:
14: val counts = lines.flatMap(_.split(" ")) // FlatMappedRDD[2]
15:                   .map(word => (word, 1))    // MappedRDD[3]
16:                   .reduceByKey(_ + _)        // ShuffledRDD[4]
17:
18: counts.toDebugString
```

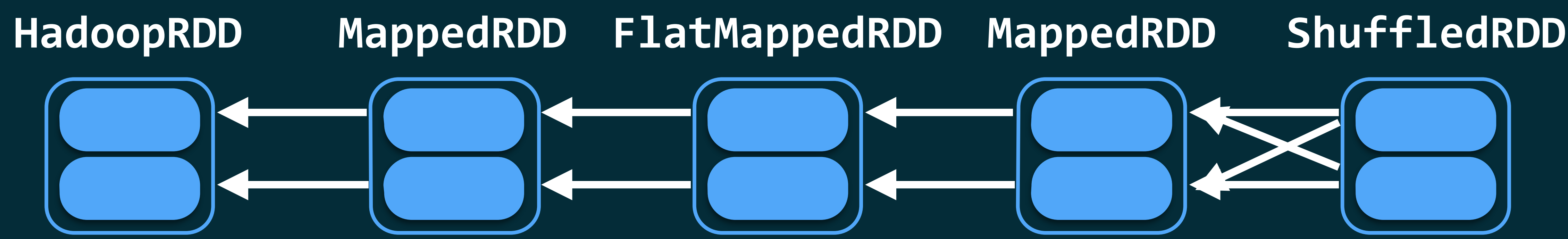
```
res0: String =
(2) ShuffledRDD[4] at reduceByKey at <console>:16 []
 +-(2) MappedRDD[3] at map at <console>:15 []
    |  FlatMappedRDD[2] at flatMap at <console>:14 []
    |  hamlet.txt MappedRDD[1] at textFile at <console>:12 []
    |  hamlet.txt HadoopRDD[0] at textFile at <console>:12 []
```

Lineage Graph

```
val lines = sc.textFile("hamlet.txt") // MappedRDD[1], HadoopRDD[0]

val counts = lines.flatMap(_.split(" "))  // FlatMappedRDD[2]
                  .map(word => (word, 1)) // MappedRDD[3]
                  .reduceByKey(_ + _)     // ShuffledRDD[4]
```

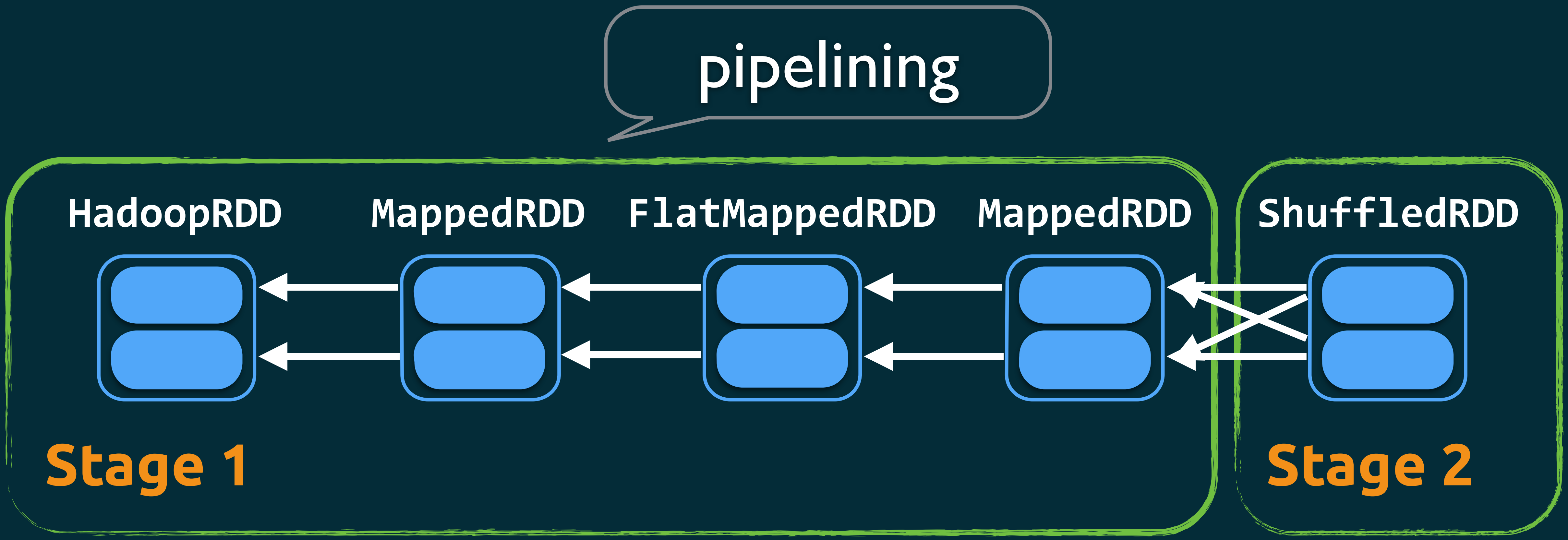
Lineage Graph

```
val lines = sc.textFile("hamlet.txt") // MappedRDD[1], HadoopRDD[0]

val counts = lines.flatMap(_.split(" "))  // FlatMappedRDD[2]
                  .map(word => (word, 1)) // MappedRDD[3]
                  .reduceByKey(_ + _)     // ShuffledRDD[4]
```



Execution Plan

```
val lines = sc.textFile("hamlet.txt") // MappedRDD[1], HadoopRDD[0]

val counts = lines.flatMap(_.split(" "))  // FlatMappedRDD[2]
                  .map(word => (word, 1)) // MappedRDD[3]
                  .reduceByKey(_ + _)     // ShuffledRDD[4]
```



Part IV:
Advanced Features

Persistence

- When we use the same RDD multiple times:
  - Spark will recompute the RDD.
  - Expensive to iterative algorithms.
- Spark can ***persist*** RDDs, avoiding recomputations.

Levels of persistence

```
val result = input.map(expensiveComputation)
result.persist(LEVEL)
```

LEVEL	Space Consumption	CPU time	In memory	On disk
MEMORY_ONLY (default)	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

Persistence Behaviour

- Each node will **store** its computed **partition**.
- In case of a **failure**, Spark **recomputes** the missing partitions.
- **Least Recently Used** cache policy:
  - Memory-only: recompute partitions.
  - Memory-and-disk: recompute and write to disk.
- Manually remove from cache: `unpersist()`

Shared Variables

1. **Accumulators:** **aggregate** values from worker nodes back to the driver program.
2. **Broadcast variables:** **distribute** values to all worker nodes.

Accumulator Example

```
val input = sc.textFile("input.txt")

val sum = sc.accumulator(0)
val count = sc.accumulator(0)

input
  .filter(line => line.size > 0)
  .flatMap(line => line.split(" "))
  .map(word => word.size)
  .foreach{
      size =>
        sum += size // increment accumulator
        count += 1  // increment accumulator
   }

val average = sum.value.toDouble / count.value
```

initialize the accumulators

driver only

Accumulators and Fault Tolerance

- **Safe**: Updates inside **actions** will only applied once.
- **Unsafe**: Updates inside **transformation** may applied more than once!!!

Broadcast Variables

- Closures and the variables they use are send **separately** to each task.
- We may want to **share** some variable (e.g., a Map) across tasks/operations.
- This can **efficiently** done with broadcast variables.

Example without broadcast variables

```scala
// RDD[(String, String)]
val names = … //load (URL, page name) tuples

// RDD[(String, Int)]
val visits = … //load (URL, visit counts) tuples

// Map[String, String]
val pageMap = names.collect.toMap

val joined = visits.map{
  case (url, counts) =>
    (url, (pageMap(url), counts))
}
```

CAUTION pageMap is sent along with every task

Example with broadcast variables

```
// RDD[(String, String)]
val names = … //load (URL, page name) tuples

// RDD[(String, Int)]
val visits = … //load (URL, visit counts) tuples

// Map[String, String]
val pageMap = names.collect.toMap

val bcMap = sc.broadcast(pageMap)

val joined = visits.map{
  case (url, counts) =>
    (url, (bcMap.value(url), counts))
}
```

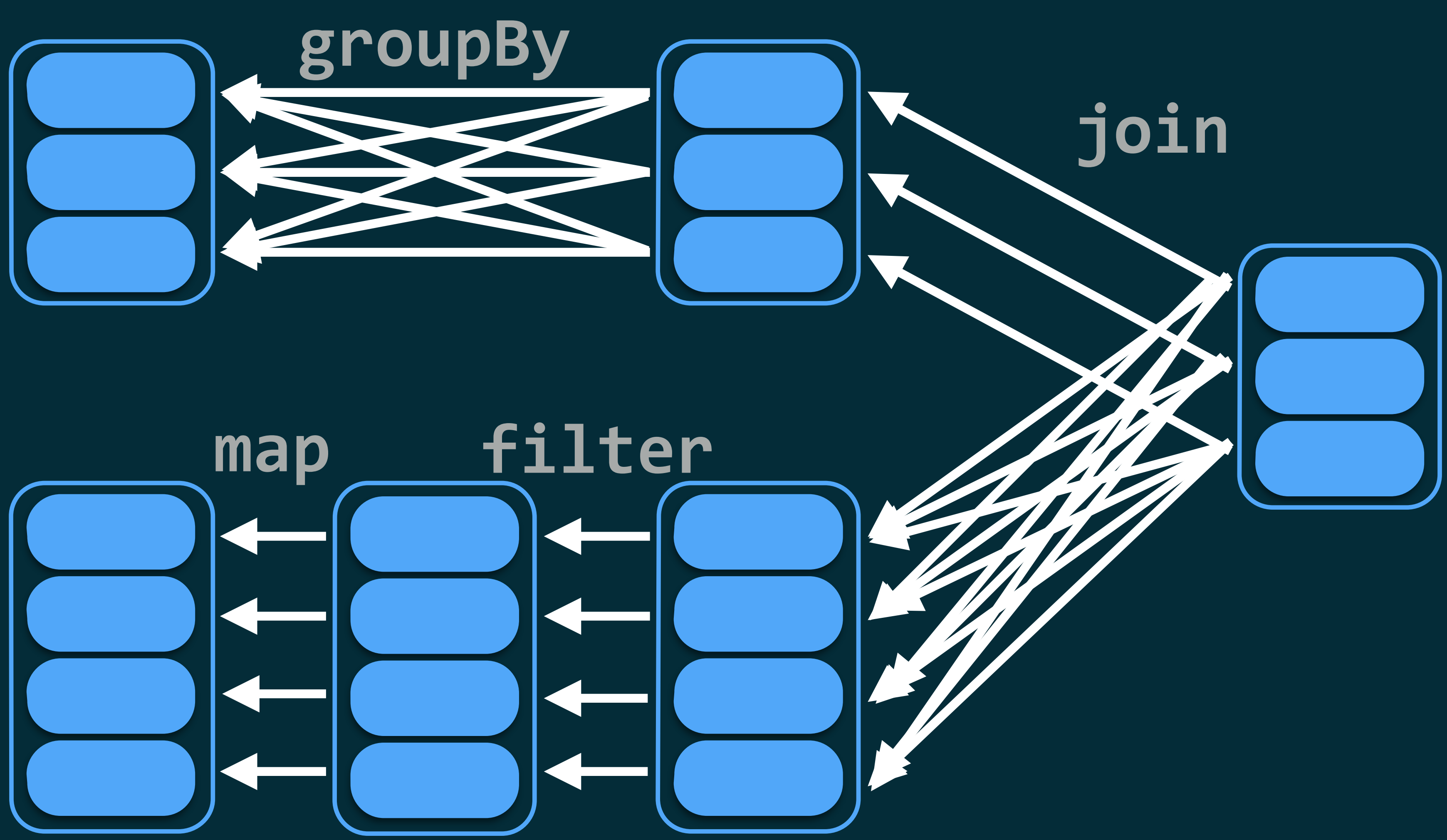
Broadcast variable

pageMap is sent only to each node once

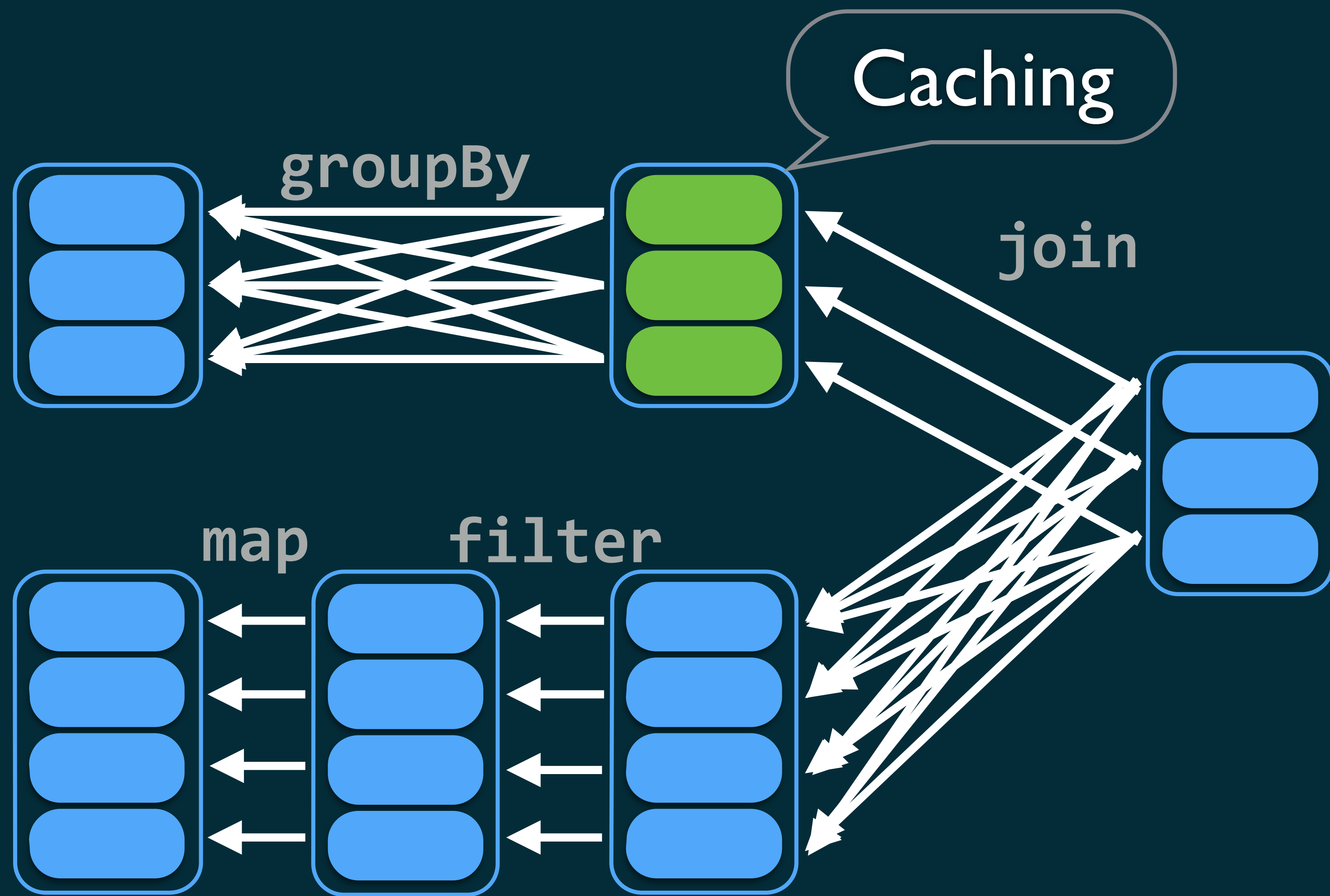

ANY
QUESTIONS
?

Appendix

Staging



Staging



Staging