
# Apache Spark Core - RDD

# Anatomy of Spark Application

- Every Spark application starts at instantiating a *Spark Context*. Without a Spark context no computation can ever be started using Spark services.
- For it to work, you have to create a *Spark configuration using SparkConf* or use a *custom SparkContext constructor* (old versions).
- When a Spark application starts (using spark-submit script or as a standalone application), it connects to Spark master as described by master URL. It is part of Spark context's initialization.

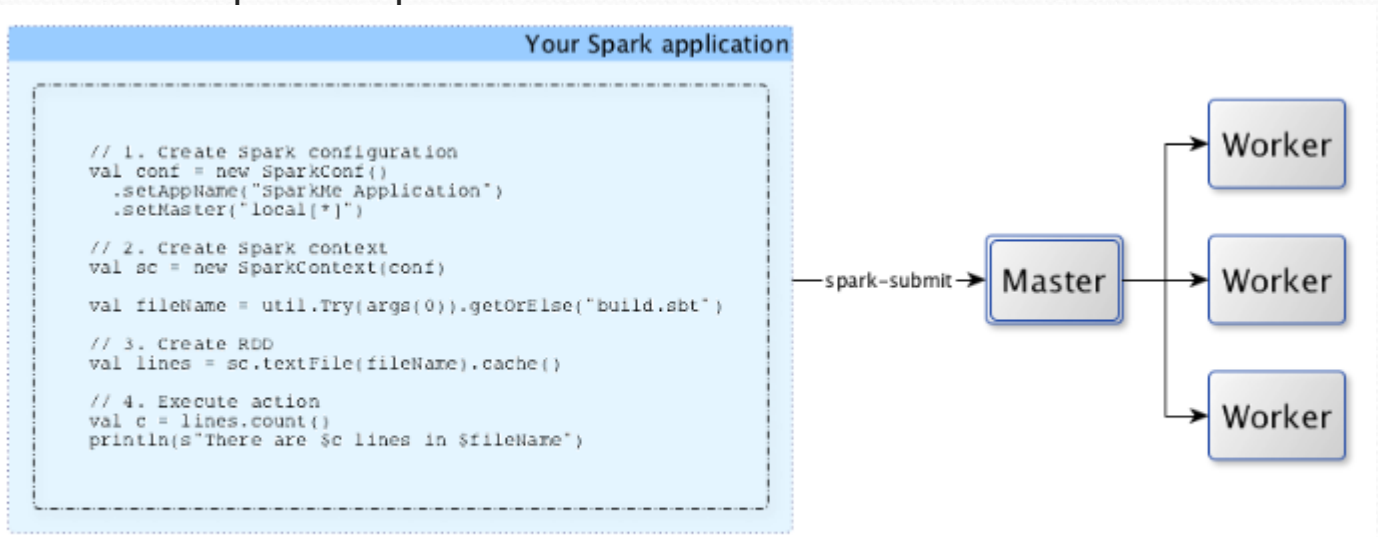



Figure 1: Submitting Spark application to master using master URL

# Spark Context – Entry point to Spark (Core)

- ***SparkContext*** is the entry point to Spark for a Spark application. (You could also assume that a SparkContext instance is a Spark application.
- It sets up internal services and establishes a connection to a Spark execution environment (deployment mode).
- Once a *SparkContext* instance is created you can use it to create RDDs, accumulators and broadcast variables, access Spark services and run jobs (until *SparkContext* is stopped)
- A *SparkContext* is essentially a client of Spark's execution environment and acts as the master of your Spark application.

# Spark Context – Entry point to Spark (Core)

- ***SparkContext*** offer the following functions:
  - Getting current configuration (SparkConf, appname, deploy mode etc)
  - Setting configuration (master URL, logging level etc)
  - Creating Distributed entities (RDDs, accumulators etc)
  - Accessing services e.g. Task Scheduler etc
  - Running jobs
  - Cancelling job etc..

```
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")
```

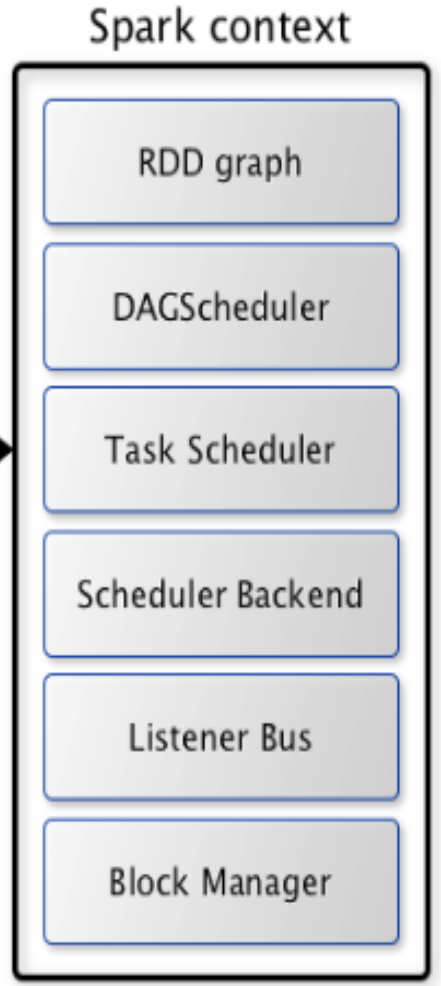



Figure 1. Spark context acts as the master of your Spark application

# RDD – Resilient Distributed Dataset

- A big collection of data having following properties:
  - Immutable
  - Distributed
  - Lazily evaluated
  - Cacheable
  - Type inferred

# Immutability

- Immutability means once created it never changes
- Big data by default immutable in nature
- Immutability helps to
  - Parallelize
  - Caching
- Immutability is about value

  not about reference

| Mutable | Immutable |
| --- | --- |
| var collection = [1,2,4,5]<br><br>for ( i = 0; i<collection.length; i++) {<br>collection[i]+=1;<br><br>}<br><br>for ( i = 0; i<collection.length; i++) {<br>collection[i]+=2;<br><br>} | val collection = [1,2,4,5]<br><br>val newCollection = collection.map(value => value +1)<br><br>val nextNewCollection = newCollection.map(value => value +2)<br><br>Creates 3 copies of same collection |

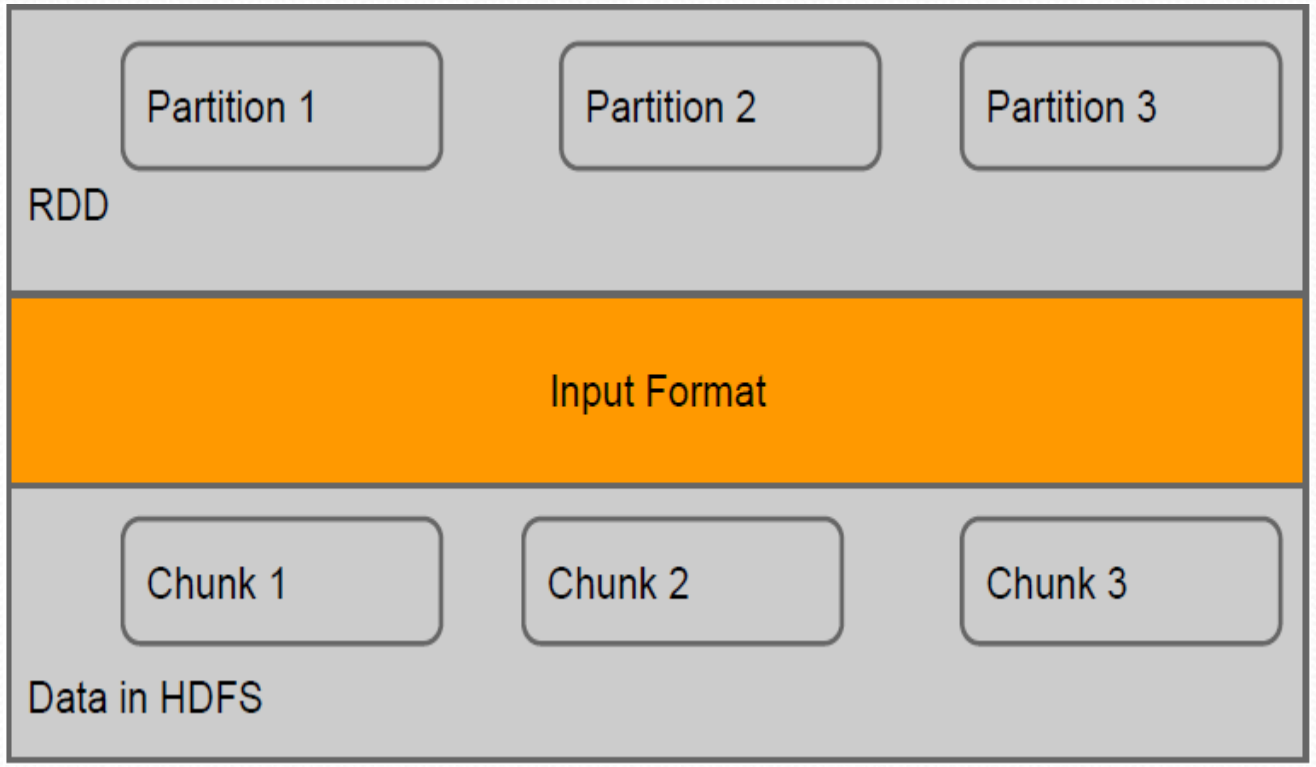
# Challenges of Immutability

- Immutability is great for parallelism but not good for space
- Doing multiple transformations result in
  - Multiple copies of data
  - Multiple passes over data
- In big data, multiple copies and multiple passes will have poor performance characteristics.

# Distributed via Partitions

Partitions

- Logical division of data
- Derived from Hadoop Map/Reduce
- All input, intermediate and output data will be represented as partitions
- Partitions are basic unit of parallelism
- RDD data is just collection of partitions



# Let's get lazy

- Laziness means not computing transformation till its need
- Laziness defers evaluation
- Laziness allows separating execution from evaluation

## Laziness in action:

```
val c1 = collection.map(value => value + 1)  //do not compute anything

val c2 = c1.map(value => value +2) // don't compute

print c2
```

Now transform into,

```
val c2 = collection.map ( value => { var result = value + 1;   result = result + 2 })
```

## Multiple transformations are combined to one.

# Laziness and Immutability

- You can be lazy only if the underneath data is immutable
- You cannot combine transformation if transformation has side effect
- So combining laziness and immutability gives better performance and distributed processing.

# Caching

- Immutable data allows you to cache data for long time
- Lazy transformation allows to recreate data on failure
- Transformation can be saved also
- Caching data improves execution engine performance

# Type Inference

- Type Inference is part of compiler to determining the type by value
- As all the transformation are side effect free, we can determine the type by operation
- Every transformation has specific return type
- Having type inference relieves you think about representation for many transforms.

```
val collection = [1,2,4,5] // explicit type Array

val c1 = collection.map( value => value + 1) // Array only

val c2 = c1.map( value => value +2) // Array only

val c3 = c2.count() // inferred as Int

val c4 = c3.map( value => value +3) // gets error. As you cannot map over an integers
```

# RDD Operations

RDDs support two kinds of operations

- Transformations – lazy operations that return another RDD
- Actions – operations that trigger computation and return values (to a Spark driver – the user program)

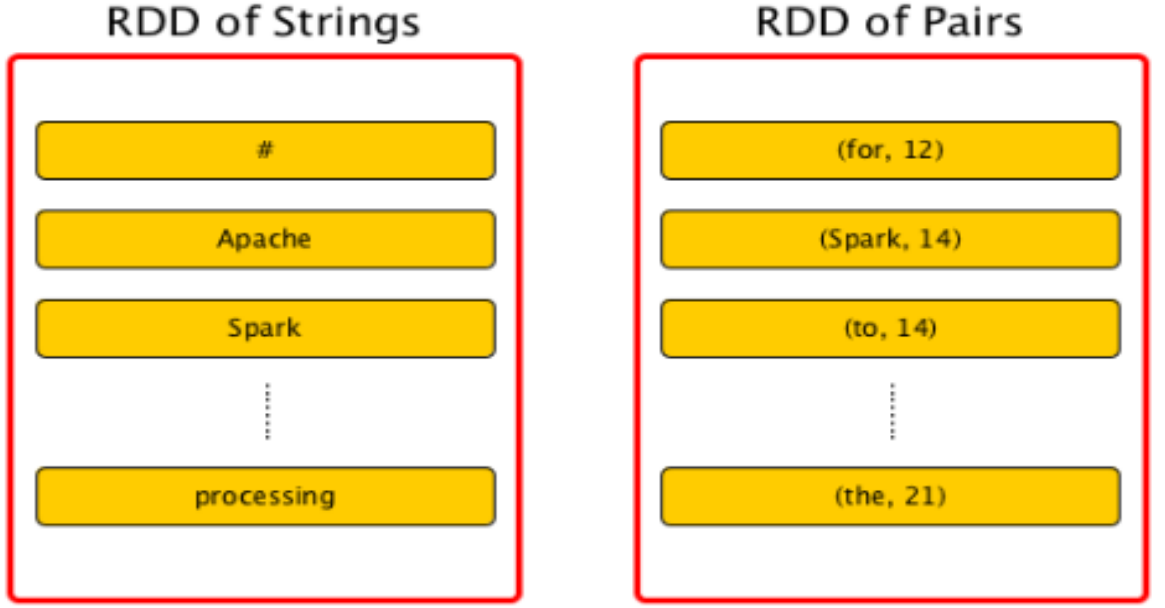



Figure 1. RDDs

# RDD Properties

## An RDD is defined by five main intrinsic properties:

- List of *parent RDDs* that is the list of the dependencies an RDD depends on for records
- An array of partitions that a dataset is divided to
- A compute function to do a computation on partitions
- An optional partitioner that defines how keys are hashed, and the pairs partitioned (for key-value RDDs)
- Optional preferred locations (aka locality info), i.e. hosts for a partition where the data will have been loaded

An RDD is a named (by name) and uniquely identified (by id) entity inside a SparkContext.

It lives in a SparkContext and as a SparkContext creates a logical boundary,

RDDs can't be shared between SparkContexts.

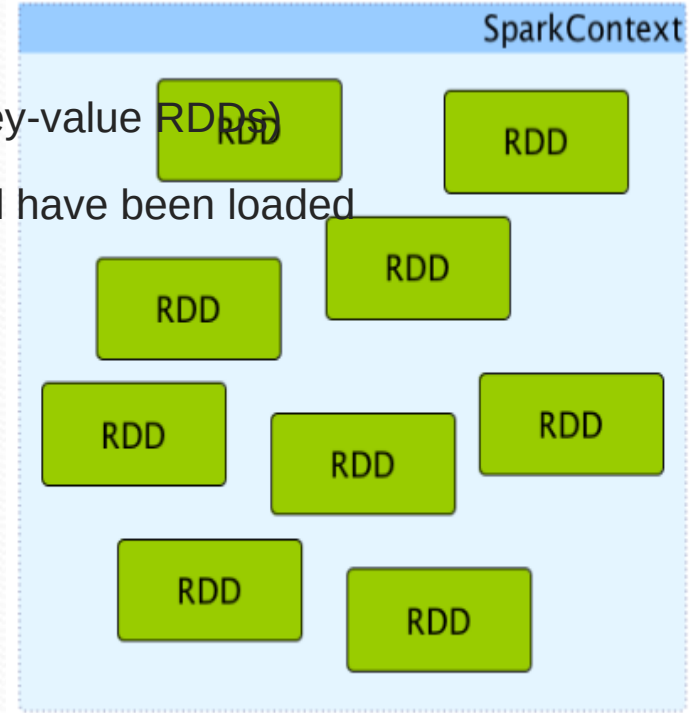



Figure 2. A Spark context creates a living space for RDDs.

# Types of RDDs

There are some of the most interesting types of RDDs:

- ParallelCollectionRDD
- HadoopRDD is an RDD that provides core functionality for reading data stored in HDFS. The most notable use case is the return RDD of SparkContext.textFile
- MapPartitionsRDD – a result of calling operations like map, flatMap, filter, mapPartitions, etc
- CoalescedRDD – a result of calling operations like repartition and coalesce
- ShuffledRDD – a result of shuffling
- PairRDD – is an RDD of key-value pairs that is a result of groupBy and join operations.
- DoubleRDD – is an RDD of double type
- SequenceFileRDD – is an RDD that can be saved as SequenceFile

# RDD Lineage – Logical Execution Plan

A RDD Lineage Graph (aka RDD operator graph) is a graph of all the parent RDDs of a RDD. It is built as a result of applying transformations to the RDD and creates a logical execution plan.

A RDD lineage graph is hence a graph of what transformations need to be executed after an action has been called.

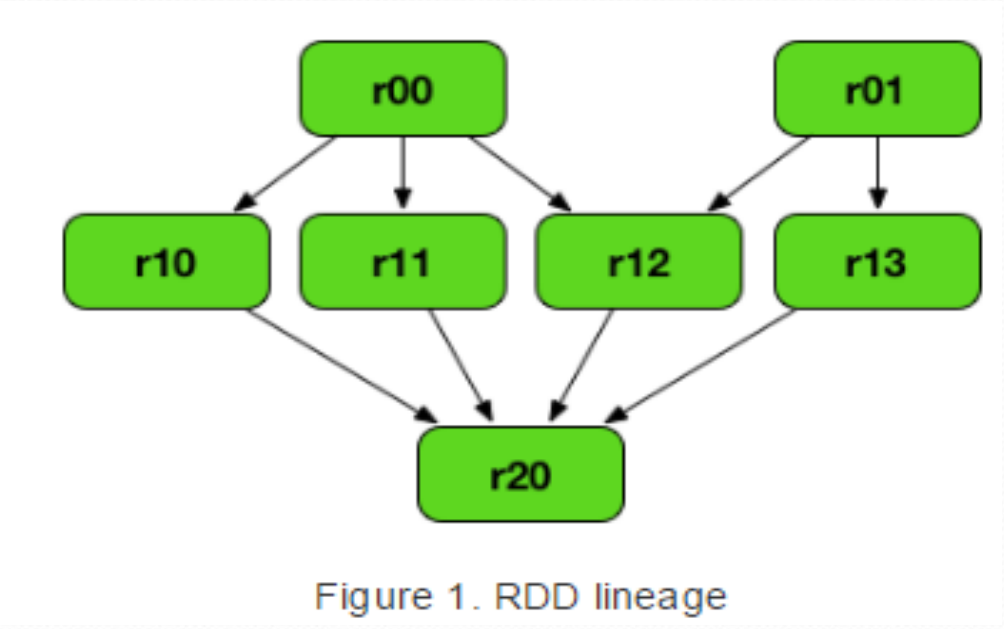



Figure 1. RDD lineage

# Kinds of Transformations

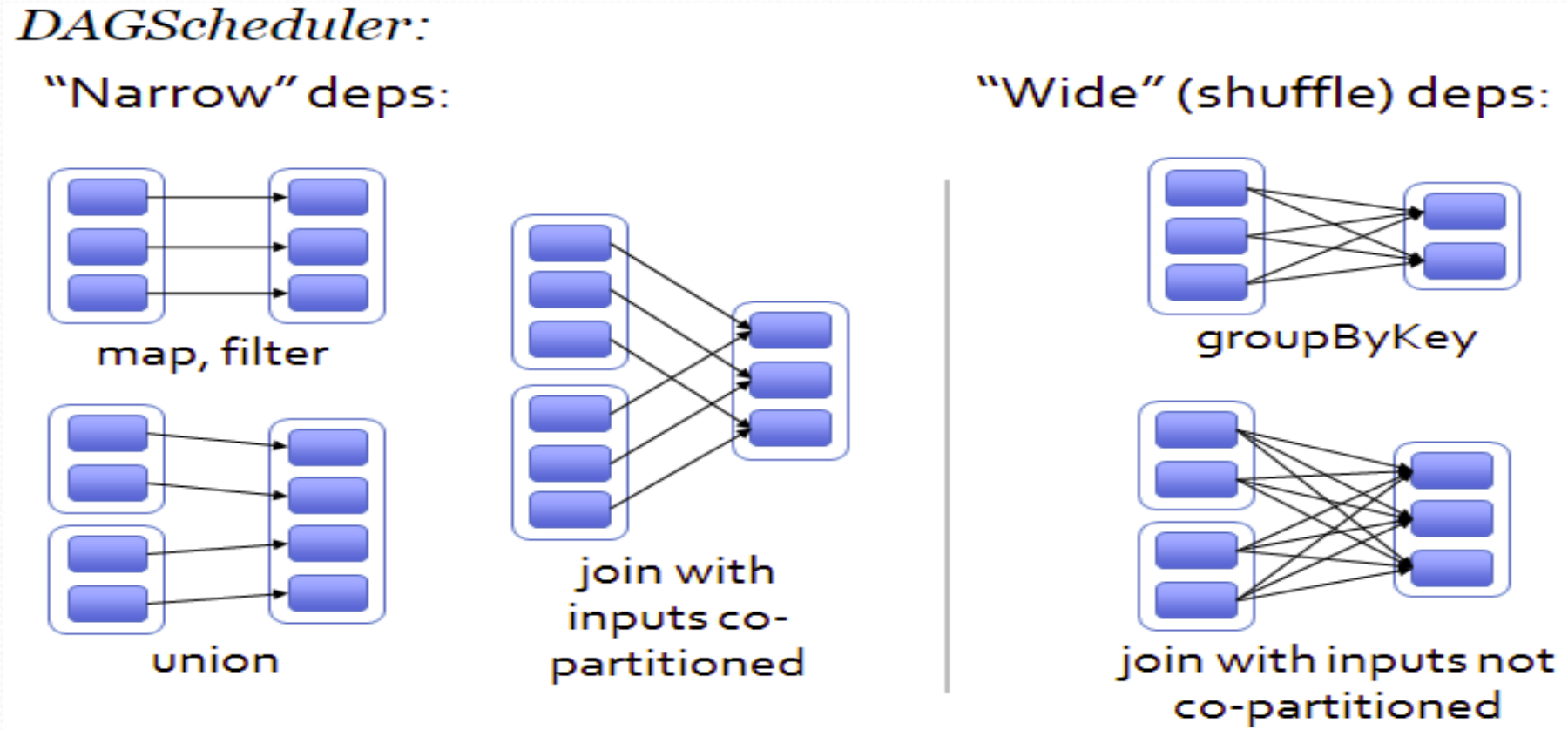
**Narrow Transformations**

- Narrow transformations are the result of map, filter and such that is from the data from a single partition only, i.e. it is self-sustained.
- An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.
- Spark groups narrow transformations as a stage which is called *pipelining*.

**Wide(Shuffle) Transformations**

- Wide transformations are the result of *groupByKey* and *reduceByKey*. The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
- All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, Spark must execute *RDD shuffle*, which transfers data across cluster and results in a new stage with a new stage with a new set of partitions.

# Kinds of Transformations



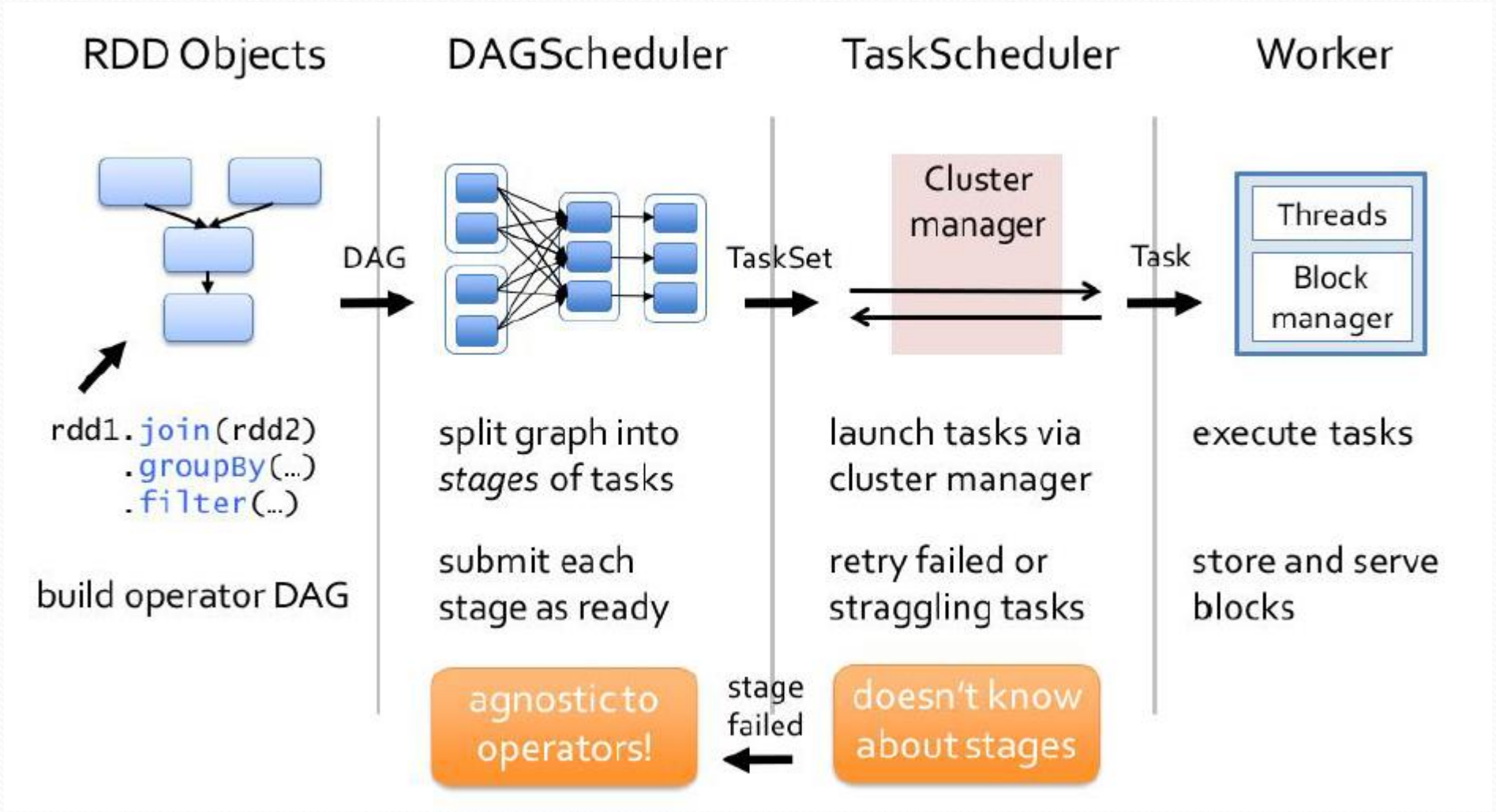

**Transformations**

map, flatMap, filter, mapPartitions, union, groupByKey, reduceByKey, distinct, join

**Actions**

reduce, collect, count, first, take(num), saveAsTextFile, foreach(f: T => Unit)

# Internals of Spark



# **Spark Deconstructed:** Log Mining Example

```
// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

# **Spark Deconstructed:** Log Mining Example

```
// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

**discussing the other part**

# **Spark Deconstructed:** Log Mining Example

At this point, take a look at the transformed RDD *operator graph*:

```
scala> messages.toDebugString
res5: String =
MappedRDD[4] at map at <console>:16 (3 partitions)
  MappedRDD[3] at map at <console>:16 (3 partitions)
    FilteredRDD[2] at filter at <console>:14 (3 partitions)
      MappedRDD[1] at textFile at <console>:12 (3 partitions)
        HadoopRDD[0] at textFile at <console>:12 (3 partitions)
```

# **Spark Deconstructed:** Log Mining Example

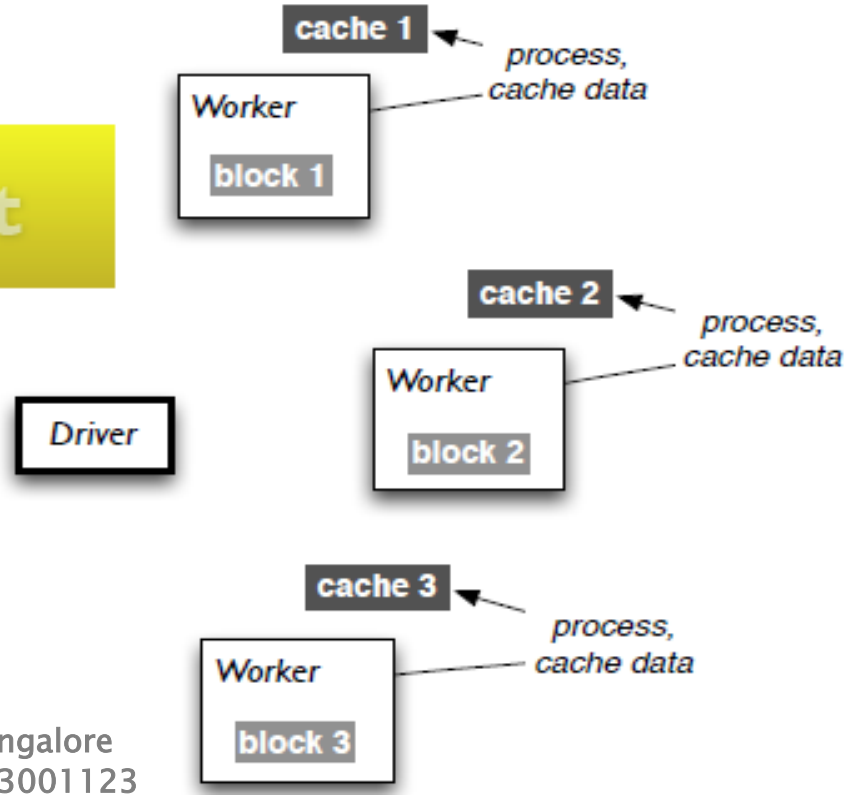
```
// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

discussing the other part



# **Spark Deconstructed:** Log Mining Example

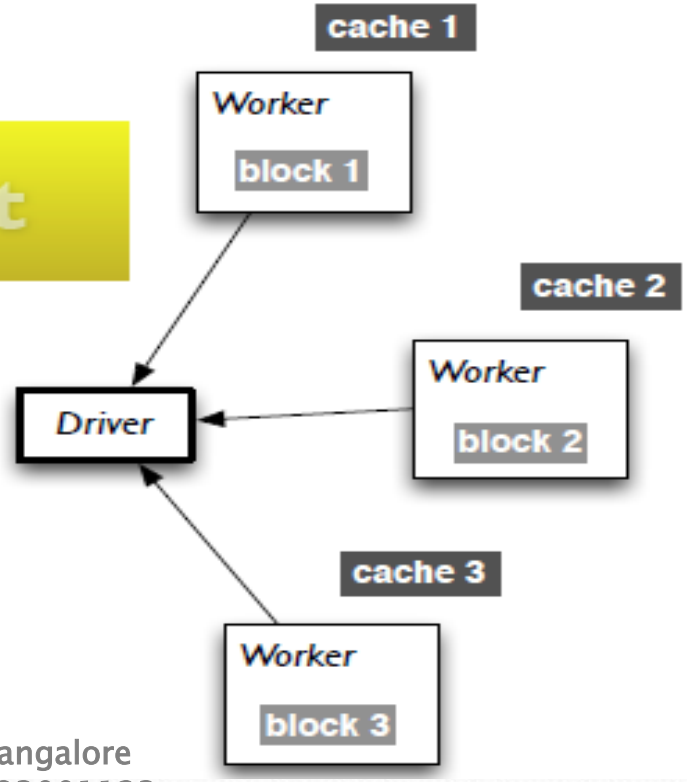
```
// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```

discussing the other part



# **Spark Deconstructed:** Log Mining Example

## Looking at the RDD transformations and actions from another perspective…

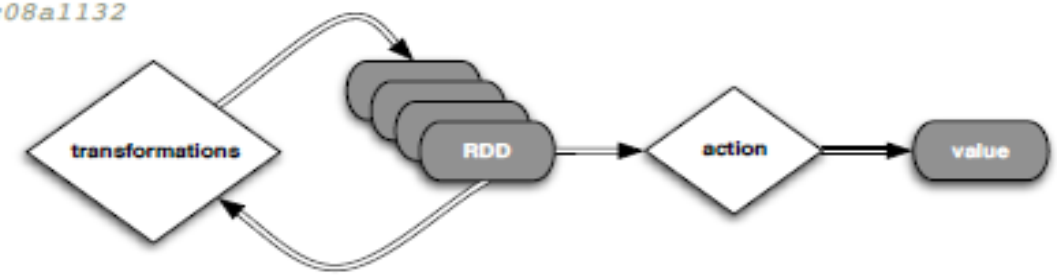
```
// load error messages from a log into memory
// then interactively search for various patterns
// https://gist.github.com/ceteri/8ae5b9509a08c08a1132

// base RDD
val lines = sc.textFile("hdfs://...")

// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()

// action 2
messages.filter(_.contains("php")).count()
```



# RDD transformations

```
val dataRDD =
sc.textFile(args
(1))

val splitRDD =
dataRDD.
flatMap(value =>
value.split(" ")
```