Big Data Infrastructures

Rana Hussein

Fall 2018

Lecture 9 – Apache Spark

MapReduce Problems

- Many problems aren’t easily described as map/reduce
- Persistence to disk typically slower than in-memory work
- Jobs reload data from disk storage on each new execution

Motivation

- Iterative data processing
  - multiple runs of a Map/Reduce program
- Interactive data processing with intermediary data reuse
  - Arbitrary code + parallel data processing
- Specialized frameworks on top of M/R have been created
- Increasingly better hardware in hadoop deployments (more RAM for example)

Design Ideas

- Observation: DAGs (Directed A-cyclic Graphs) of tasks and a shared distributed data model are enough to represent these models in the same engine
  - Unification has benefits for user (learning curve) and the system (code base, complexity etc.)
- Retain the attractive properties of MapReduce
  - Fault tolerance, data locality, scalability
- Keep intermediary/computed data in memory for efficient reuse

Apache Spark

A general purpose data processing engine

- Defines a large set of operations (as opposed to simple “map” and “reduce”)
- Operations can be arbitrarily combined in any order
- Programming at a higher level of abstraction; work with distributed dataset as if it was local
- Combines multiple data processing types (SQL, ML, Graph)

Getting Started with Spark

- http://spark.apache.org/downloads.html
  - Need Java JDK
- ./bin/spark-shell

spark — ded@diufpc04: ~ — java ◂ bash bin/spark-shell — 150×34

```
scala> xis-MacBook-Pro:spark xi$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/11/30 12:05:54 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
16/11/30 12:05:55 WARN SparkContext: Use an existing SparkContext, some configuration may not take effect.
Spark context Web UI available at http://134.21.148.22:4040
Spark context available as 'sc' (master = local[*], app id = local-1480503954837).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_20)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 
```

Apache Spark

- Key construct: Resilient Distributed Dataset (RDD)

  "Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner." (Zaharia 2012)

- A distributed data collection

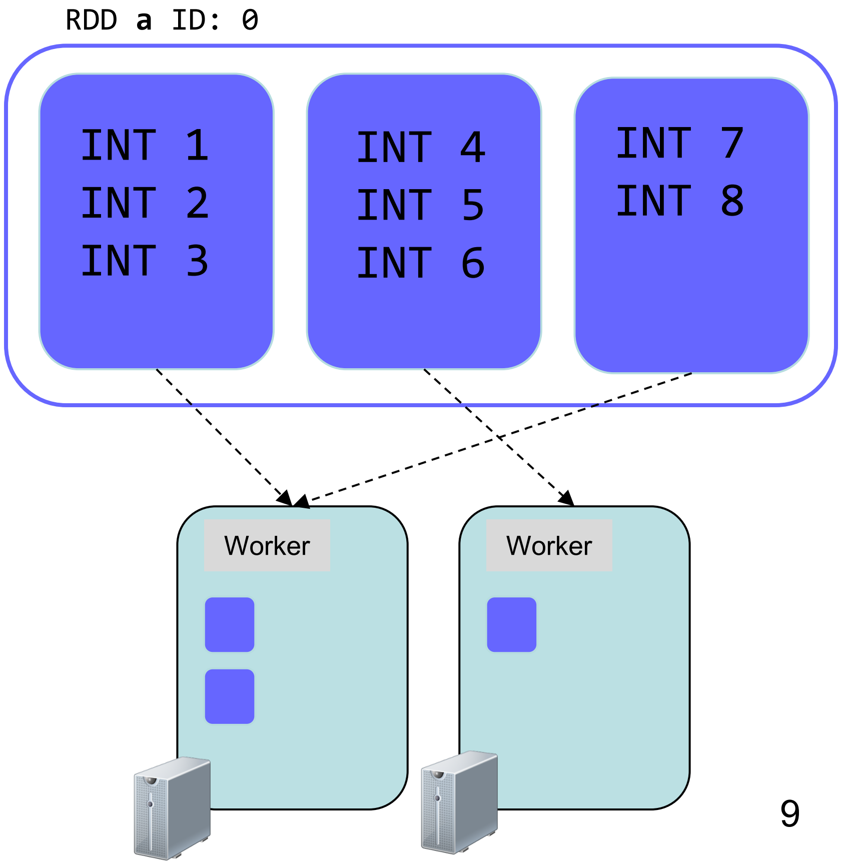
RDD Characteristics

- **In-Memory first**
- **Immutable** or **Read-Only**
- **Lazy evaluated**
- **Cacheable**
- **Parallel**
- **Typed**
- **Partitioned**

RDD Creation and Partition

An RDD can be created 2 ways:
- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

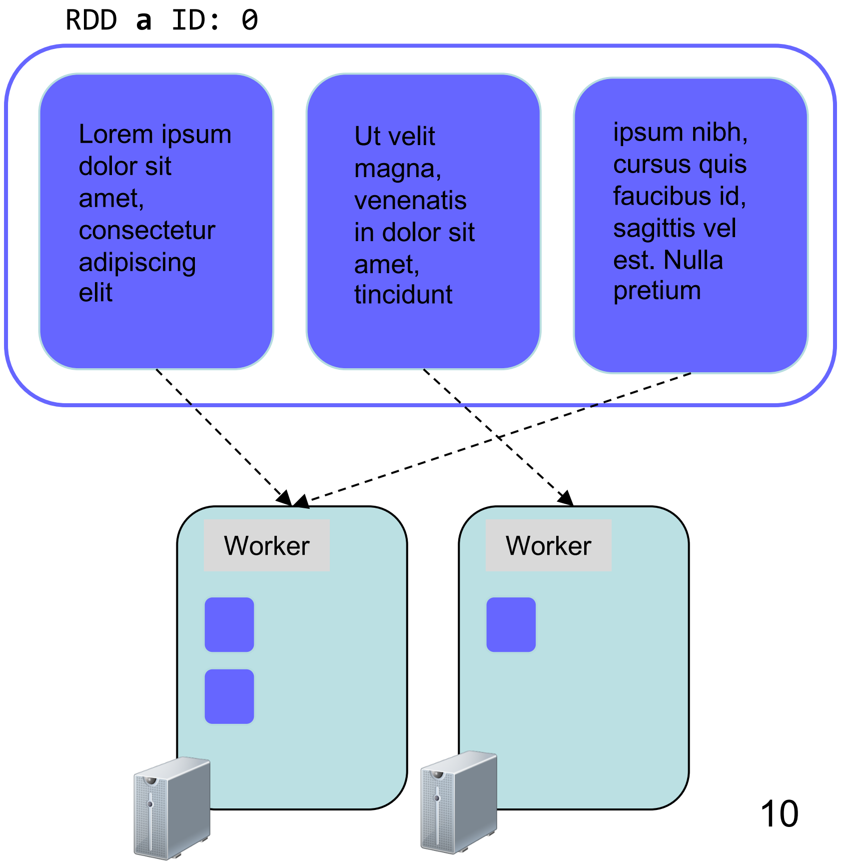
```
var firstRDD = sc.parallelize(1 to 8)
firstRDD.count()
```



RDD Creation and Partition

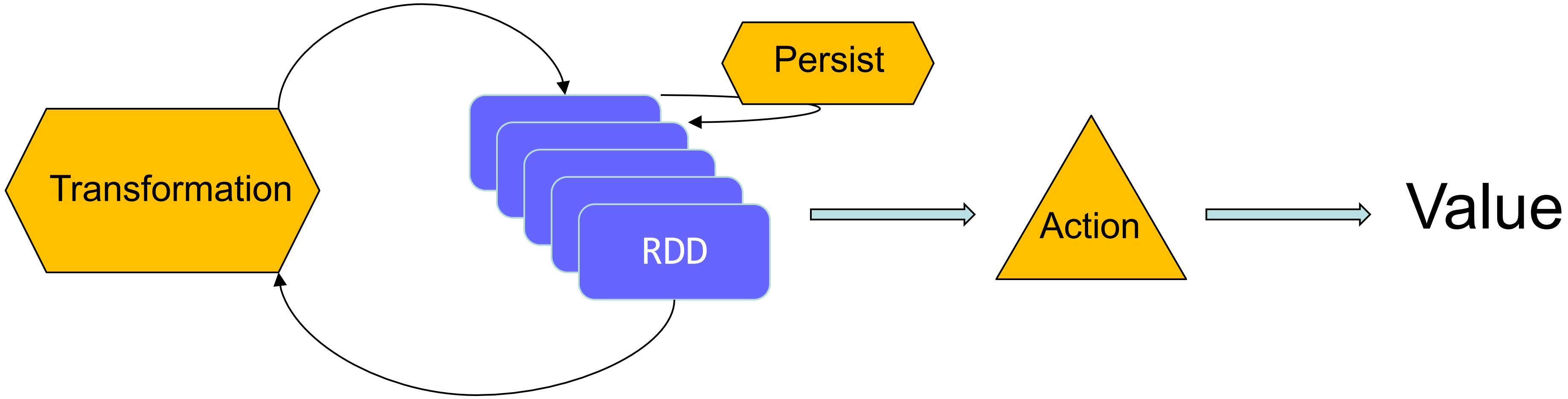
An RDD can be created 2 ways:
- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

```
var secondRDD = sc.textFile("input")
secondRDD.count()
```



RDD Operation and Life

- Transformations
  - Lazy operations that return another RDD
- Actions
  - Operations that trigger computation and return values

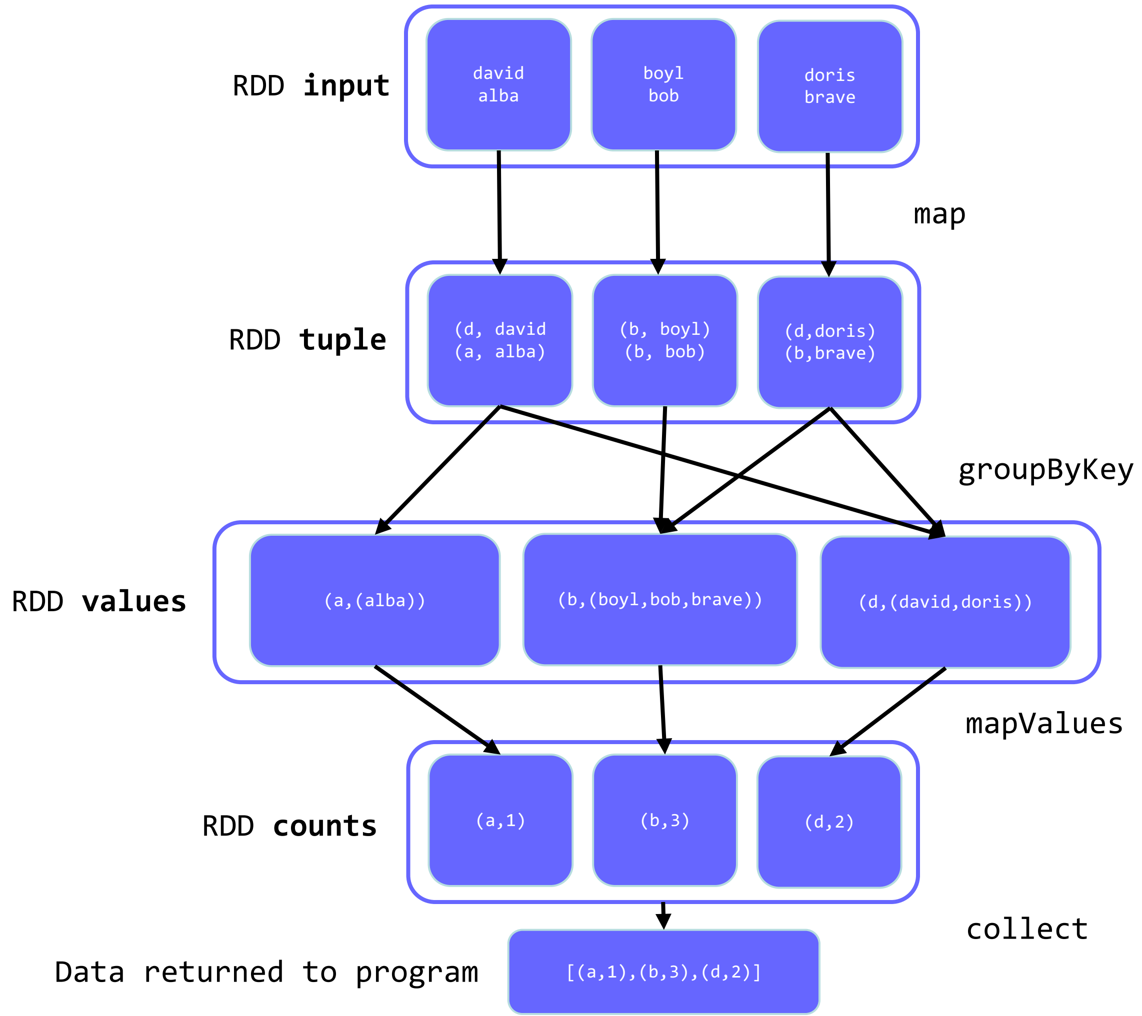


Example in Scala

Find the number of distinct *names* per “first letter”

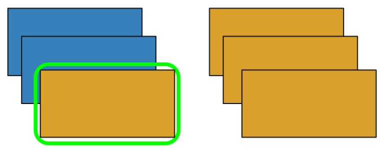
```
// Example
// input : alba, david, boyl, doris, bob, brave
// output: (d,2), (b,3), (a,1)
var input = sc.textFile(“hdfs://names”)
var tuple = input.map(name => (name.charAt(0), name))
var values = tuple.groupByKey()
var counts = values.mapValues(name => name.toSet.size)
counts.collect() // Action !
```

Example RDD


RDD input
david
alba
boyl
bob
doris
brave
map
RDD tuple
(d, david
(a, alba)
(b, boyl)
(b, bob)
(d,doris)
(b,brave)
groupByKey
RDD values
(a,(alba))
(b,(boyl,bob,brave))
(d,(david,doris))
mapValues
RDD counts
(a,1)
(b,3)
(d,2)
collect
Data returned to program
[(a,1),(b,3),(d,2)]

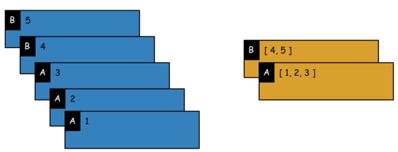
Examples

map



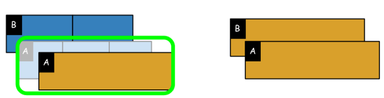
```
x = sc.parallelize([1,2,3])
y = x.map(lambda x: (x,x**2))
x: [1, 2, 3]
y: [(1, 1), (2, 4), (3, 9)]
```

groupByKey



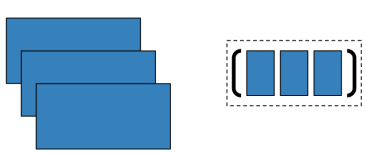
```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
x: [('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
y: [('A', [3, 2, 1]), ('B', [5, 4])]
```

mapValues



```
x = sc.parallelize([('A',(1,2,3)),('B',(4,5))])
y = x.mapValues(lambda x: [i**2 for i in x])
x: [('A', (1, 2, 3)), ('B', (4, 5))]
y: [('A', [1, 4, 9]), ('B', [16, 25])]
```

collect



```
x = sc.parallelize([1,2,3])
y = x.collect()
y: [1, 2, 3]
```

RDD Operations

Transformations

(define a new RDD)

```
map()
flatMap()
distinct()
filter()
groupByKey()
reduceByKey()
coalesce()
repartition()
sortByKey()
partitionBy()
sample()
join()
union()
cogroup()
...
```

Actions

(return results to program)

```
reduce()
collect()
saveAsTextFile()
count()
first()
take()
countByKey()
takeSample()
foreach()
saveToCassandra()
...
```

Visual Documentation

http://nbviewer.jupyter.org/github/jkthompson/pyspark-pictures/blob/master/pyspark-pictures.ipynb

RDD Types

`HadoopRDD`	`DoubleRDD`
`FilteredRDD`	`JdbcRDD`
`MappedRDD`	`JsonRDD`
`PairRDD`	`SchemaRDD`
`ShuffledRDD`	`VertexRDD`
`UnionRDD`	`EdgeRDD`
`PythonRDD`	`CassandraRDD`
…	…

For more in-depth info => read the code !
https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/rdd

RDD Interface

- Set of partitions (“splits”)
- List of dependencies on parent RDDs
- Function to compute a partition given parents
- Optional preferred locations
- Optional partitioning information for Key/Value RDDs (Partitioner)

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala

Example: HadoopRDD

- partitions = one per HDFS block
- dependencies = none
- compute*(partition)* = read corresponding block

- preferredLocations*(part)* = HDFS block location
- partitioner = none

Example: FilteredRDD

- partitions = same as parent RDD
- dependencies = "one-to-one" on parent
- compute*(partition)* = compute parent and filter it

- preferredLocations*(part)* = none (ask parent)
- partitioner = none

Example: JoinedRDD

- partitions = one per reduce task
- dependencies = “shuffle” on each parent
- compute*(partition)* = read and join shuffled data

- preferredLocations*(part)* = none
- partitioner = HashPartitioner(numTasks)

RDD

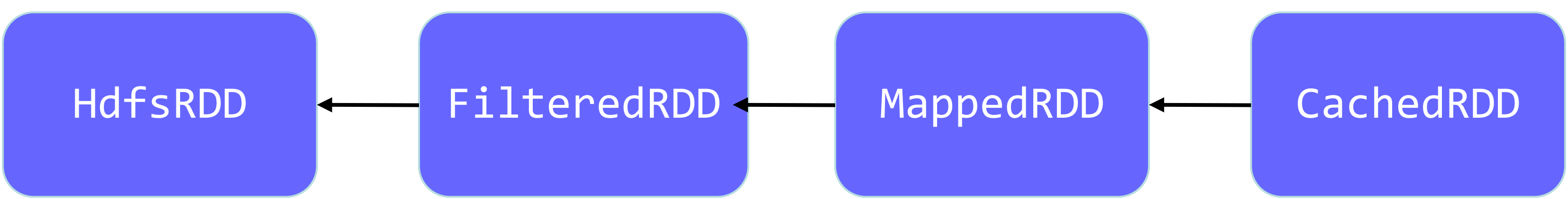
Users can control two aspects of RDDs:

- *Persistence* (in RAM, reuse)
- *Partitioning (hash, range, [<k, v>])*

RDD Fault Tolerance

- Replication ?
- Data lineage
  - Upon node failure RDDs recompute lost data by reapplying the transformations used to build them

```
var errors = sc.textFile(“hdfs://logs”)
.filter(_.contains(“error”))
.map(_.split(‘\t’)(2))
.cache()
```

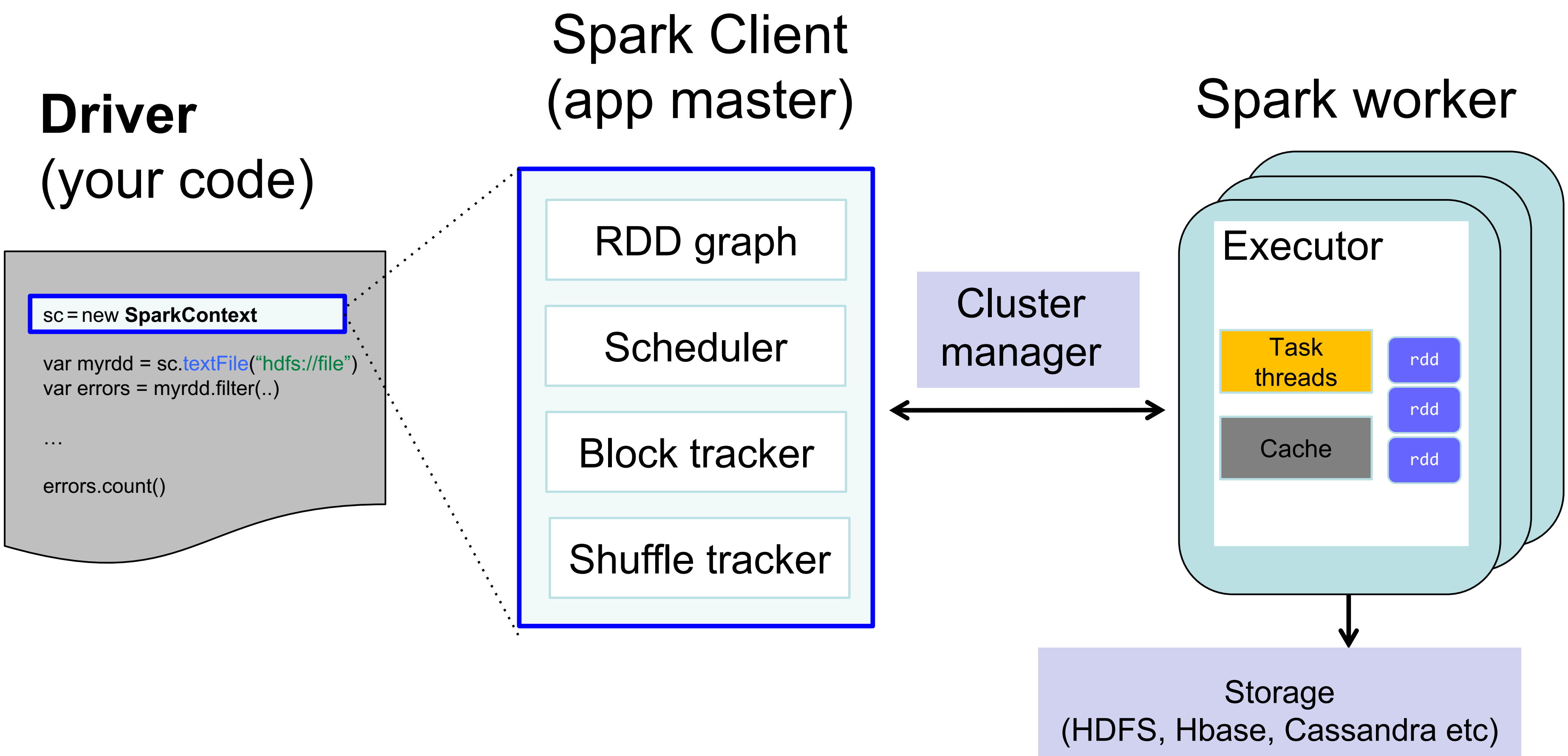


Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

SPARK INTERNALS

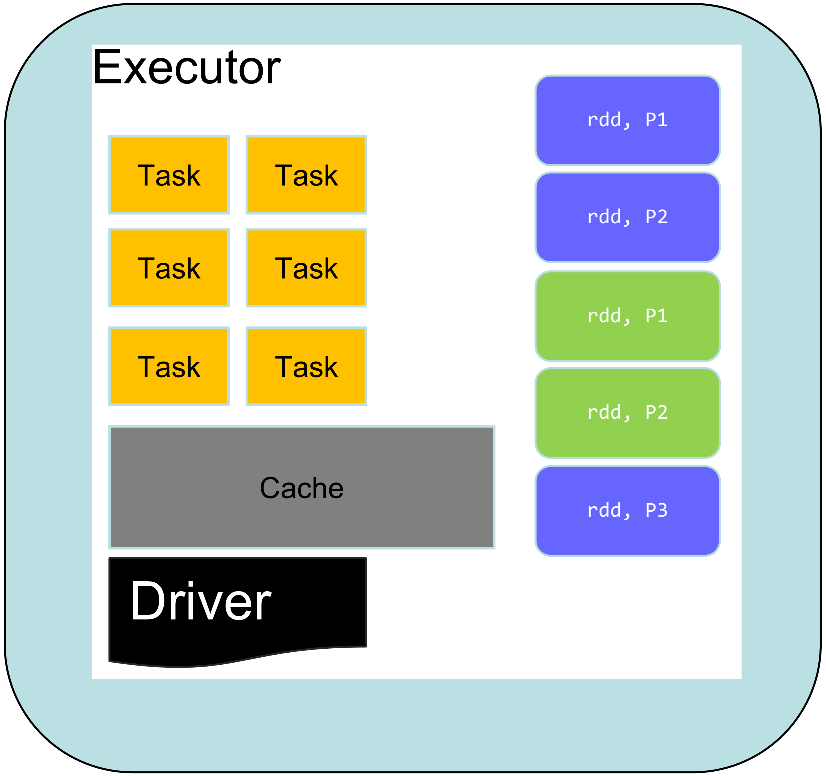
Spark Components



Spark Execution Modes

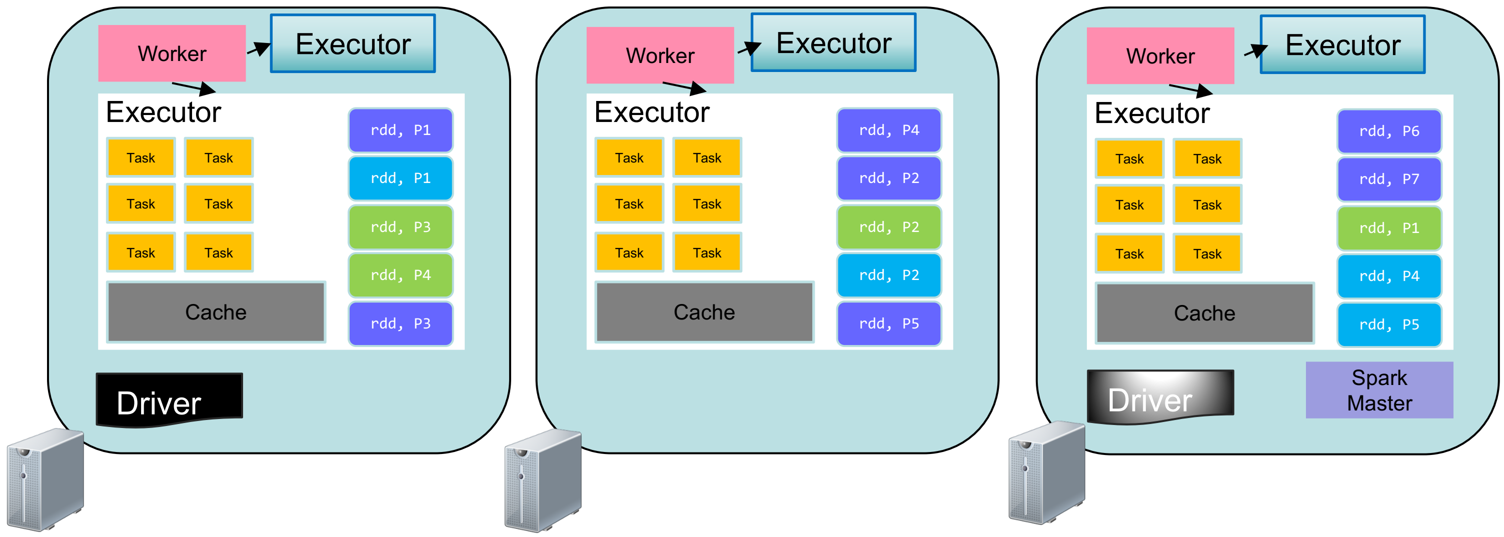
- Local
- Standalone Scheduler
- YARN

Local Mode



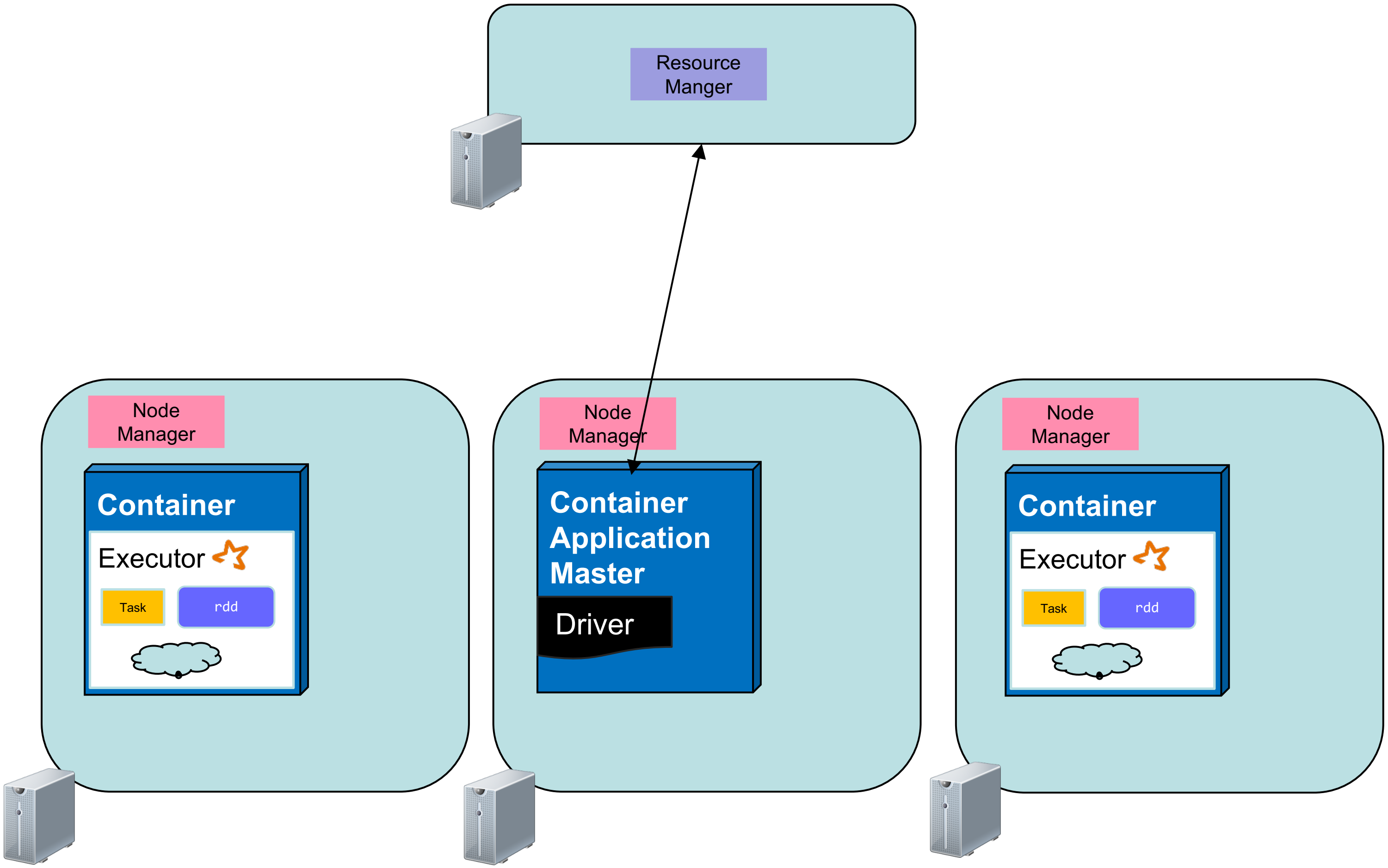
./bin/spark-shell --master local[6]

Standalone Mode



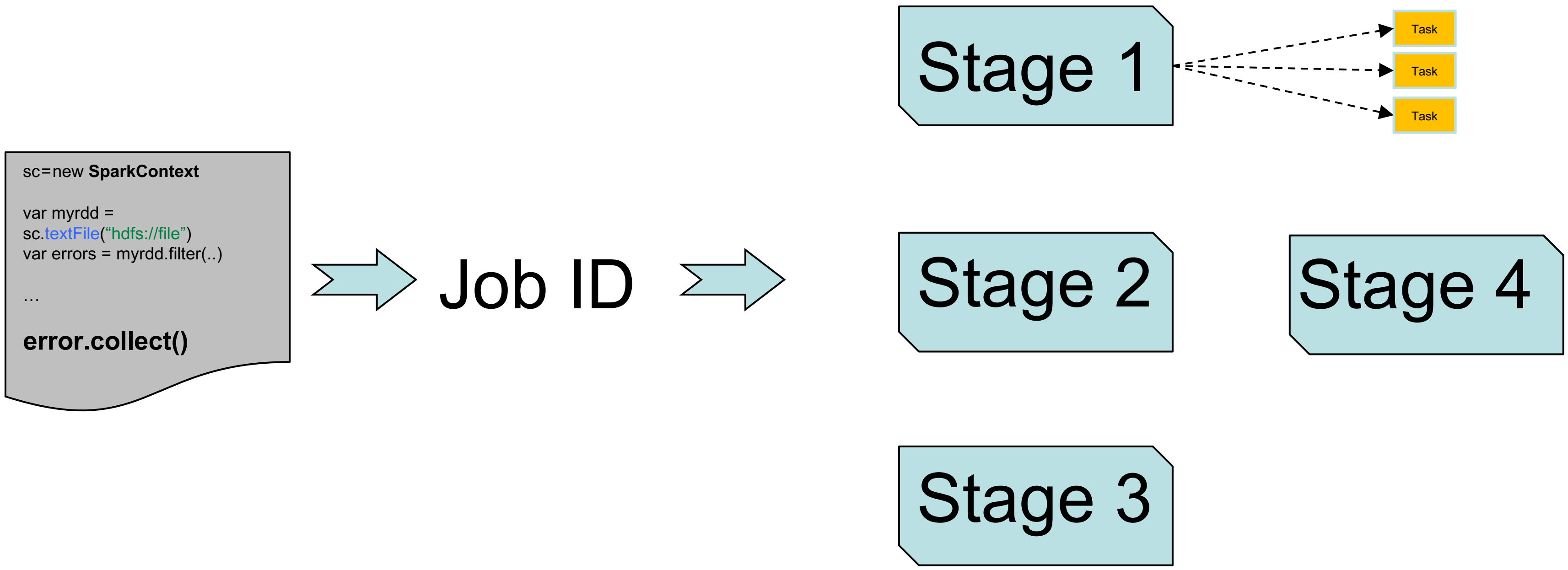
Place a compiled version of Spark on each node on the cluster

YARN Mode

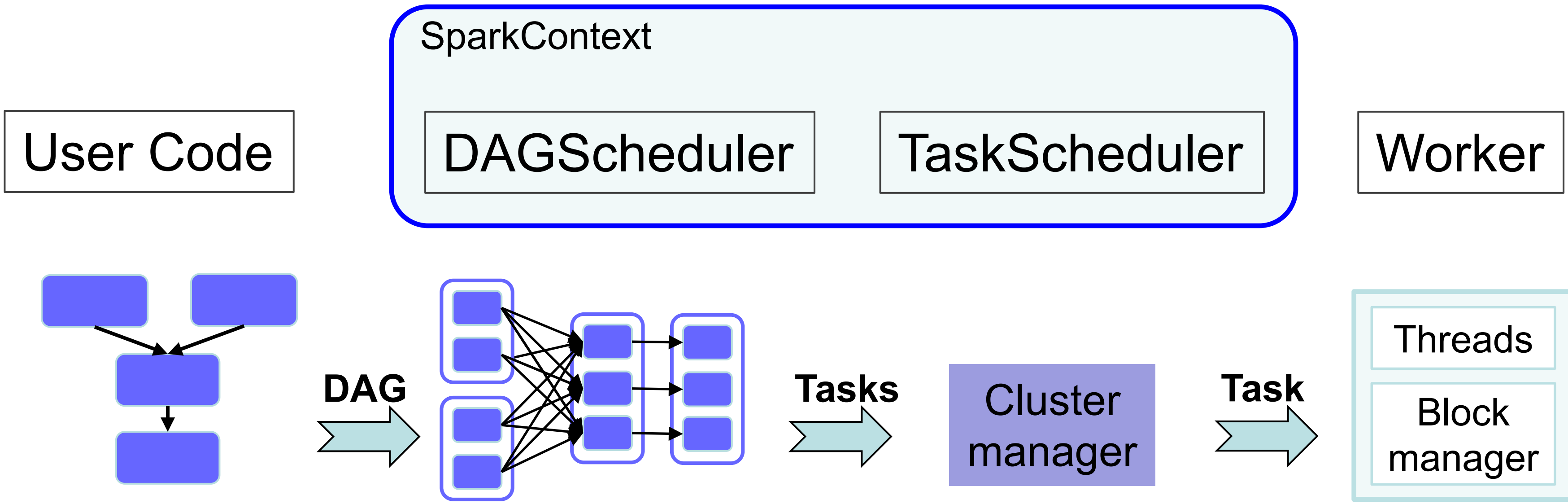


Resource Manger
Node Manager
Container
Executor
Task
rdd
Node Manager
Container
Application
Master
Driver
Node Manager
Container
Executor
Task
rdd

Staged Execution



Scheduling Process




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

split graph into *stages* of tasks

launch tasks via cluster manager

execute tasks

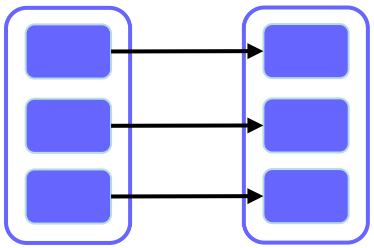
Lineage

- “One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations.”
- “The most interesting question in designing this interface is how to represent dependencies between RDDs.”
- “We found it both sufficient and useful to classify dependencies into two types:
  - **narrow dependencies**, where each partition of the parent RDD is used by at most one partition of the child RDD
  - **wide dependencies**, where multiple child partitions may depend on it.”

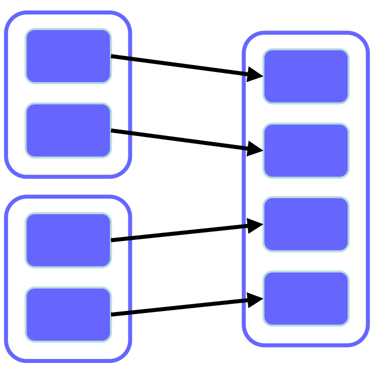
(Zaharia 2012)

Dependency Types

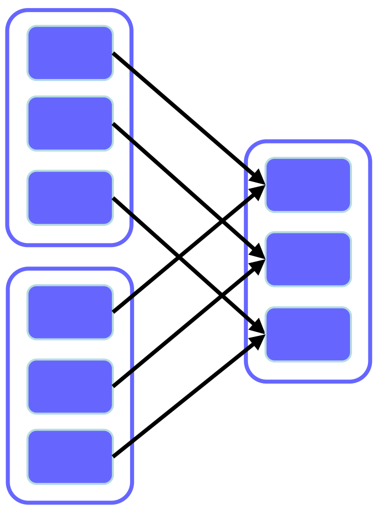
"Narrow" dependencies:



map, filter

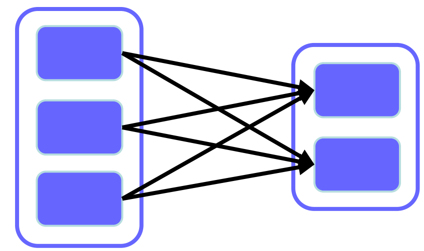


union

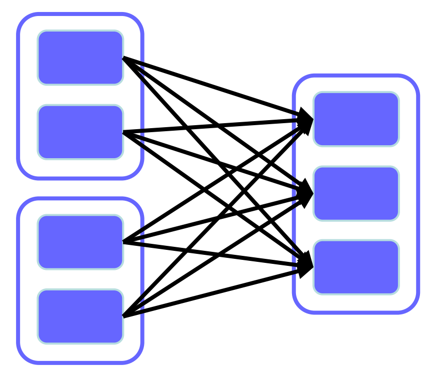


join with inputs co-partitioned

"Wide" (shuffle) dependencies :



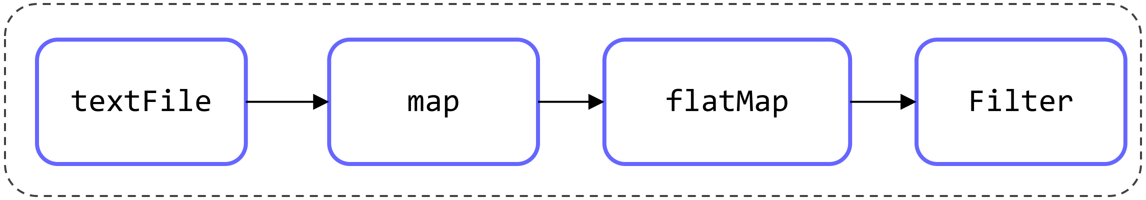
groupByKey



join with inputs not co-partitioned

How Many Stages?

```
var a = sc.textFile("someFile.txt”)
.map(mapFunc)
.flatMap(flatMapFunc)
.filter(filterFunc)
.count()
```

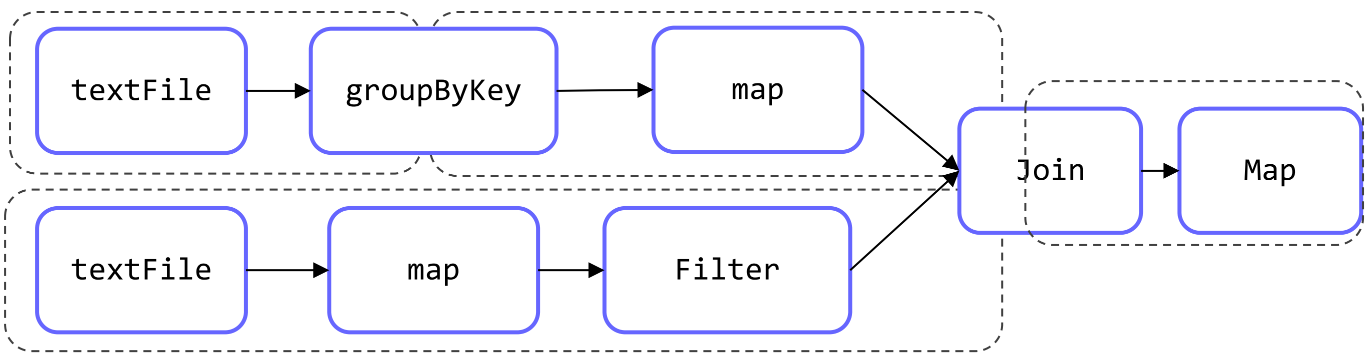


How Many Stages?

```
var s = sc.textFile("sales")

var l = sc.textFile("locations”)
.groupByKey()
.map()

s.map()
.filter()
.join(l)
.map()
.collect()
```



Summary

- Spark can overcome some of the problems of MapReduce
- Spark provides RDDs (Distributed – fault tolerant - Immutable)
- Spark can run on different modes

Thanks to Djellel Eddine Difallah for providing the original slides.