Big Data Systems

Djellel Difallah

Spring 2023

Lecture 10 – Apache Spark

Outline

- Introduction
- Apach Spark Motivation
- RDD: Resilient Distributed Datasets
- Spark Internals

MapReduce Problems

- Many problems aren’t easily mapped into a MapReduce job
- Persistence to disk is typically slower than in-memory processing
  - Shuffle phase is disk intensive
- Jobs reload data from disk storage on each new execution
  - No-reutilization

Motivation

- Towards “distributed data programming”
  - Iterative data processing
    - Multiple runs of a Map/Reduce program
  - Interactive data processing with intermediary data reuse
    - Arbitrary code + parallel data processing
- Many specialized frameworks on top of M/R have been created
- Increasingly better hardware in Hadoop deployments:
  - High-speed networks
  - Larger memory capacity

APACHE SPARK

Design Ideas

- Retain the attractive properties of MapReduce
  - Scalability
  - Data locality
  - Fault tolerance
- Support more operation
- Lesson learned from other systems: Execution are DAGs (Directed A-cyclic Graphs) of tasks.
  - Unification has benefits for user (learning curve) and the system (code base, complexity etc.)

Keep intermediary/computed data in-memory

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

- Install dependencies:
  - Java and Scala
- On Mac:
  - brew install apache-spark
- On Windows
  - [Tutorial]

Interact with Spark using Scala or PySpark shell

dd115 — java ◂ spark-shell — 146×28

```
dd115@ADUAEI12744LPMX ~ % spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/04/11 10:04:25 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context Web UI available at http://10.225.94.244:4040
Spark context available as 'sc' (master = local[*], app id = local-1681193065734).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.3.2
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 20)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 23/04/11 10:04:37 WARN GarbageCollectionMetrics: To enable non-built-in garbage collector(s) List(G1 Concurrent GC), users should configure
 it(them) to spark.eventLog.gcMetrics.youngGenerationGarbageCollectors or spark.eventLog.gcMetrics.oldGenerationGarbageCollectors


scala>

scala>

scala>
```

dd115 — java ◂ Python — 146×28

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/04/11 10:30:06 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.3.2
      /_/

Using Python version 3.11.3 (main, Apr  7 2023 19:29:16)
Spark context Web UI available at http://10.225.94.244:4040
Spark context available as 'sc' (master = local[*], app id = local-1681194607212).
SparkSession available as 'spark'.
>>>
```

Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)

RESILIENT DISTRIBUTED DATASET

Apache Spark RDD

- 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)

- In other words: A Distributed Data Collection
  - Is this sharding? 🤔
  - mylist = [1, 2, 3]
  - mylistRDD = [1, 2, 3]

RDD Characteristics

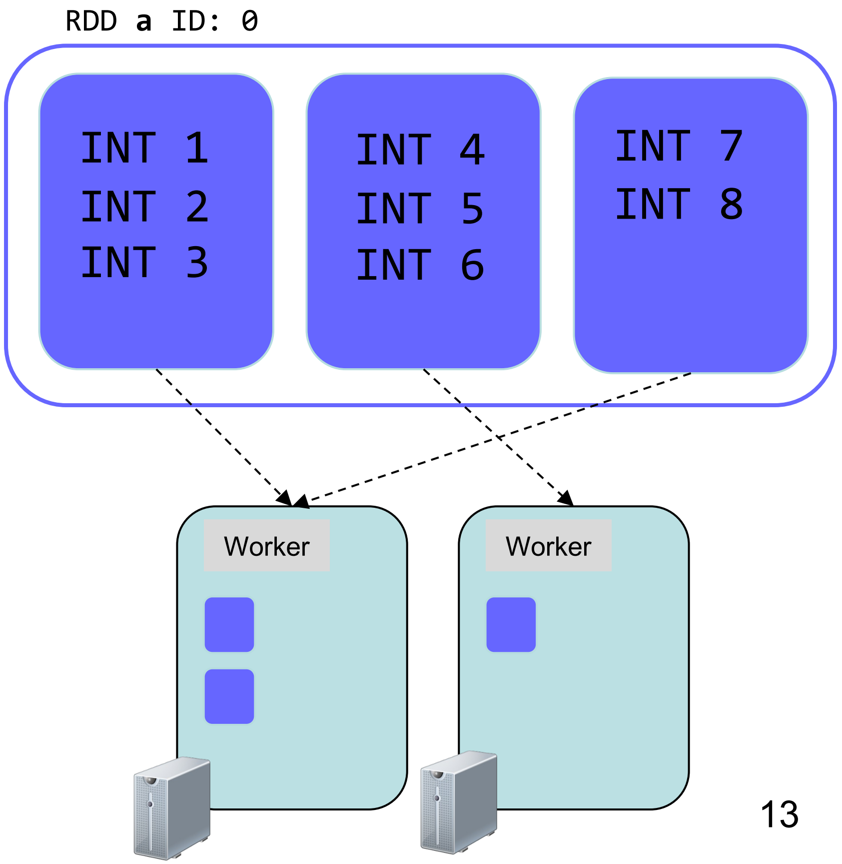
- **In-Memory first**
- **Immutable** or **Read-Only**
- **Lazy evaluated**
- **Cacheable**
- **Parallel**
- **Typed**
- **Partitioned**

RDD Creation and Partition

An RDD can be created in 2 ways:
- Parallelize a collection (list, set, dictionary, etc.)
- Read data from an external source (HDFS, S3, etc.)

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

```
firstRDD = sc.parallelize(range(1, 9))
firstRDD.cache()
firstRDD.count()
```

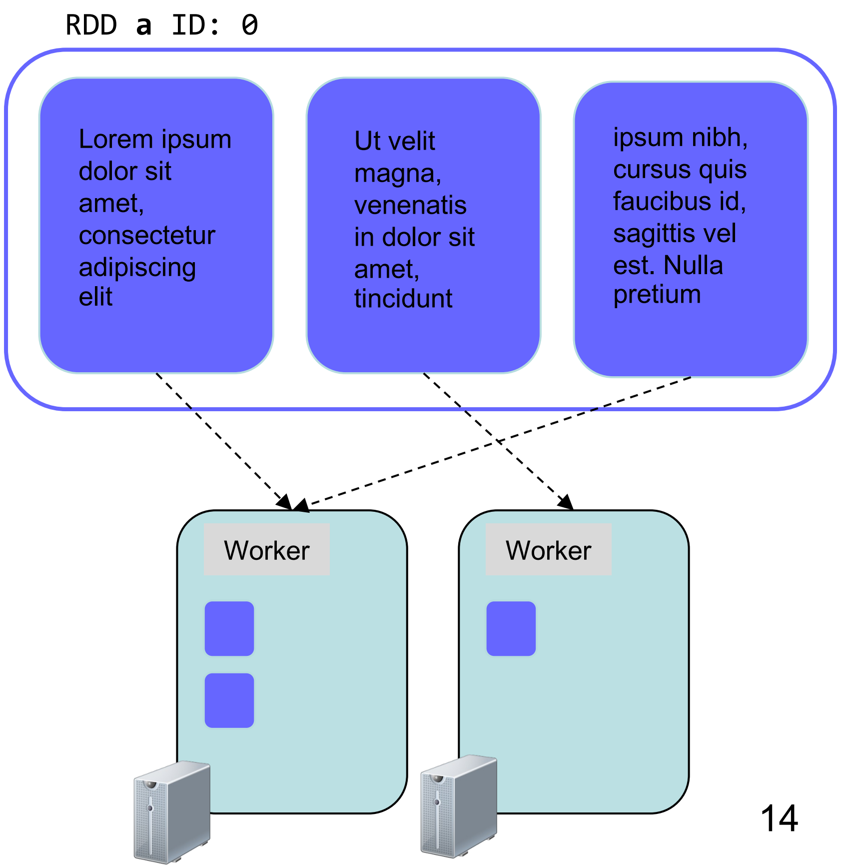


RDD Creation and Partition

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

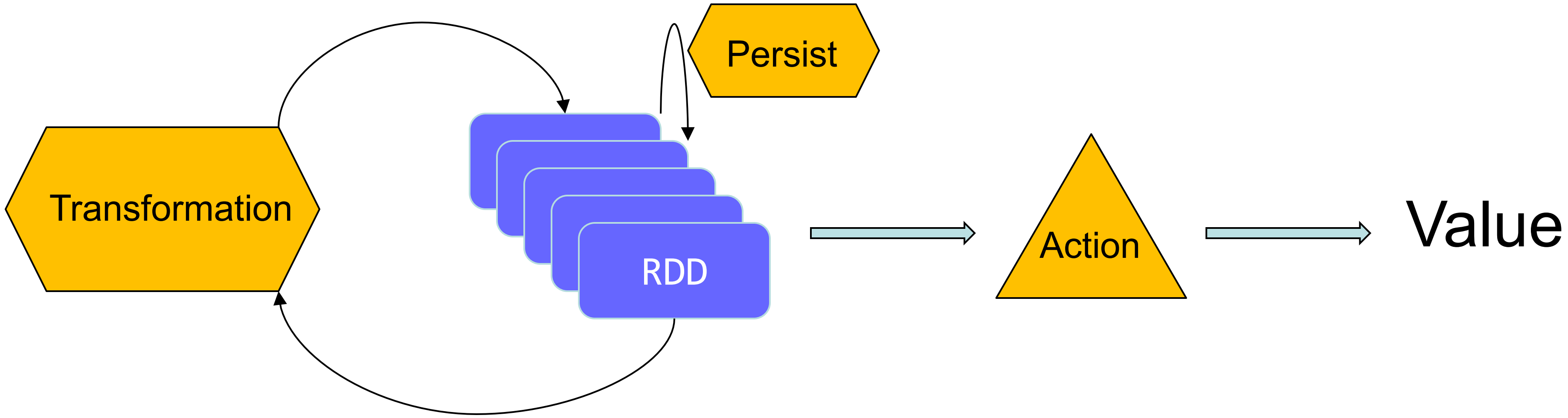
```
var secondRDD = sc.textFile(“input”)
secondRDD.cache()
```

```
secondRDD = sc.textFile(“input”)
secondRDD.cache()
```



RDD Operation and Lifecycle

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



Example in PySpark

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

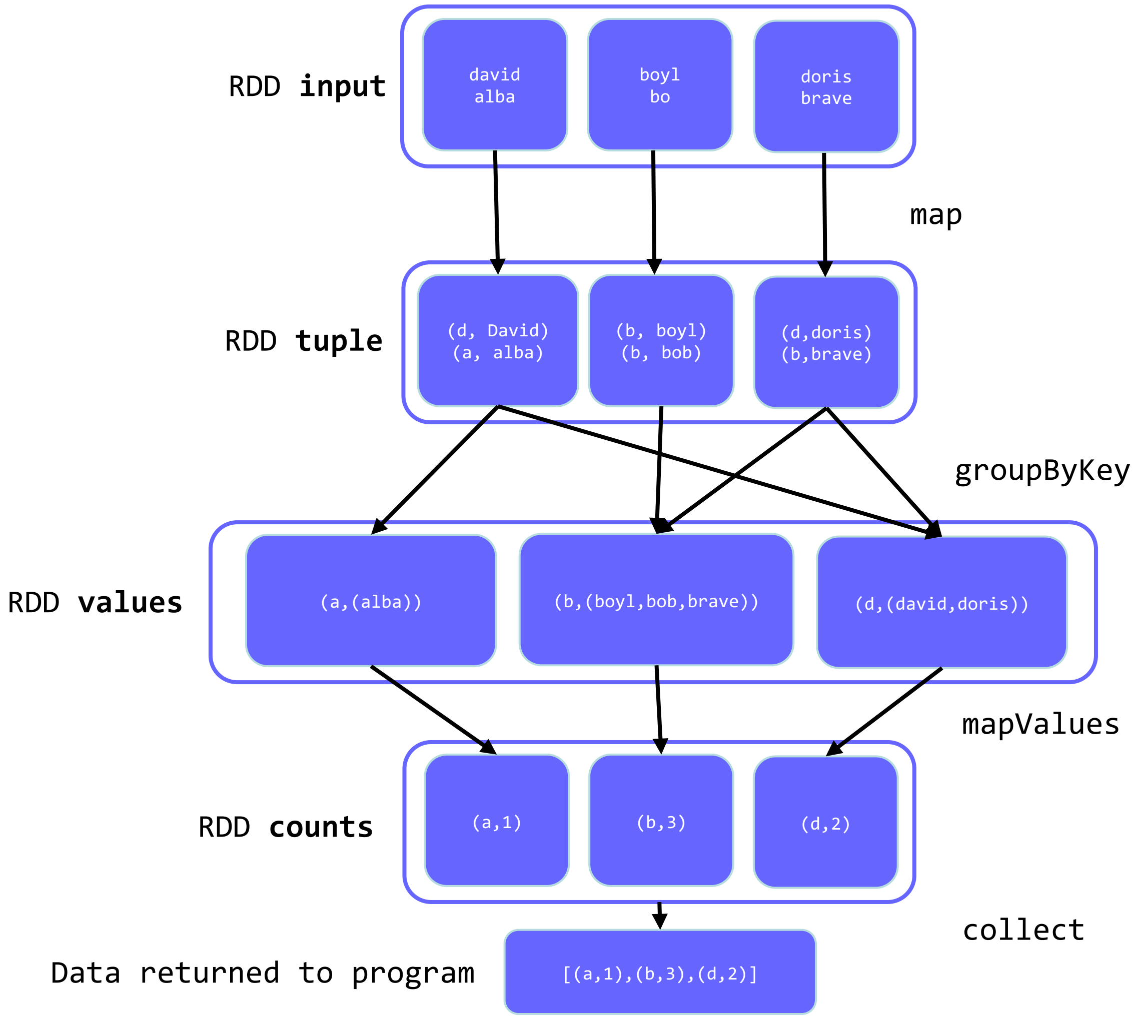
```
# Example
# input : alba, david, boyl, doris, bob, brave
# output: (d,2), (b,3), (a,1)

input = sc.textFile("hdfs://names")
tuple = input.map(lambda name: (name[0], name))
values = tuple.groupByKey()
counts = values.mapValues(lambda name: len(set(name)))
counts.collect() # Action!
```

Example in Scala

Find the number of distinct *names* by “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 bo
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)]

Misc. Examples

```
###
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)]

###
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])]

###
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])]

###
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()
sortByKey()
partitionBy()
sample()
join()
union()
...
```

`persist()`

`cache()`

Special transformations: Mark the RDD for persistance

Actions

(return results to program)

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

RDD Types

`HadoopRDD`

`JdbcRDD`

`JsonRDD`

`SchemaRDD`

`ShuffledRDD`

`UnionRDD`

`CassandraRDD`

…

Specialized RDDs; Check out the code repository:
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)

Base RDD code:
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

RDD Manipulation

Users can control two aspects of RDDs:

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

RDD Lineage

- Lineage: the sequence of RDDs (Resilient Distributed Datasets) that form the dependencies between the RDDs in a Spark application
- Fault Tolerance: 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()
```

HadoopRDD ← FilteredRDD ← MappedRDD ← CachedRDD

RDD vs. Distributed Shared Memory

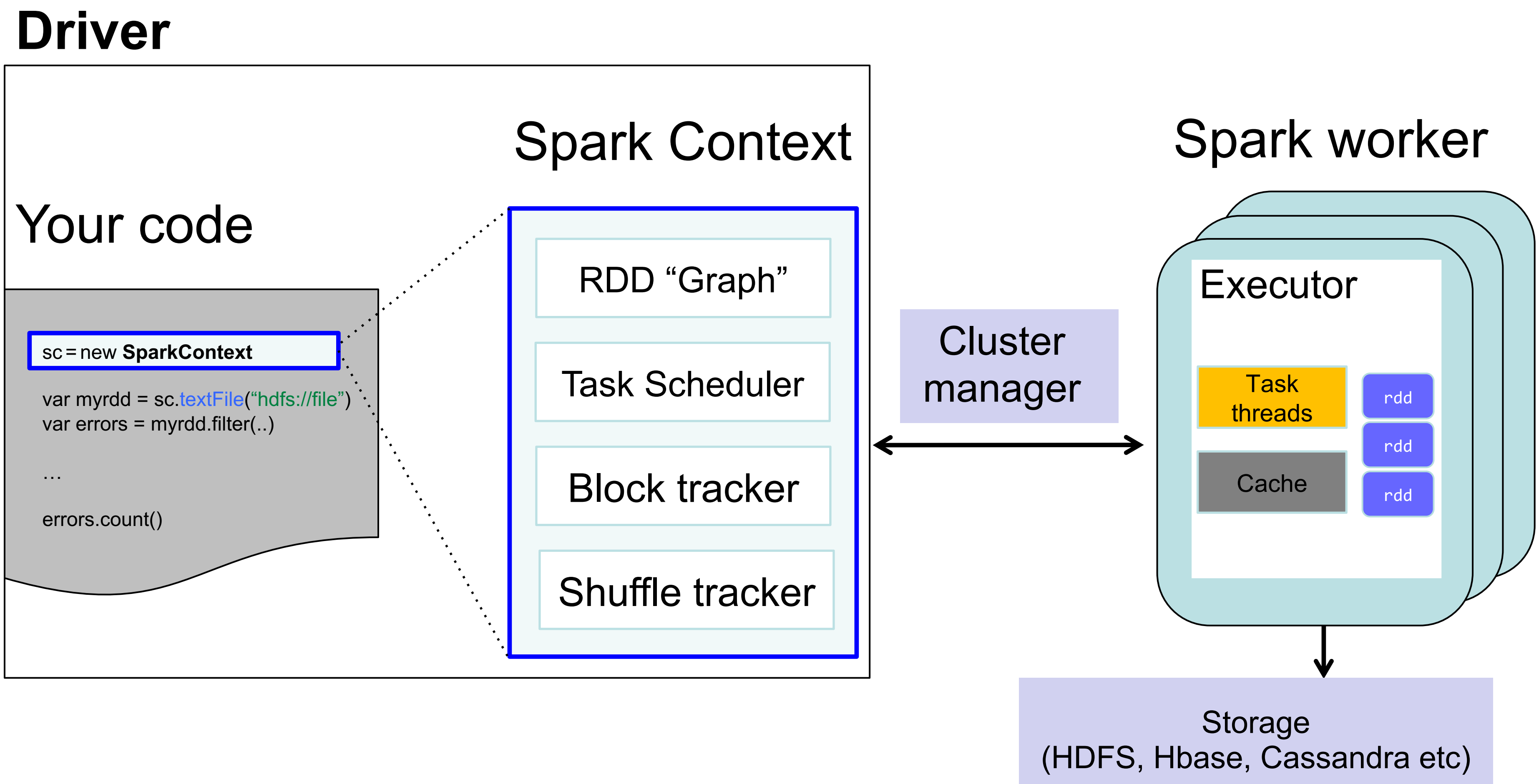
Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)

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

APACHE SPARK INTERNALS

Spark Components



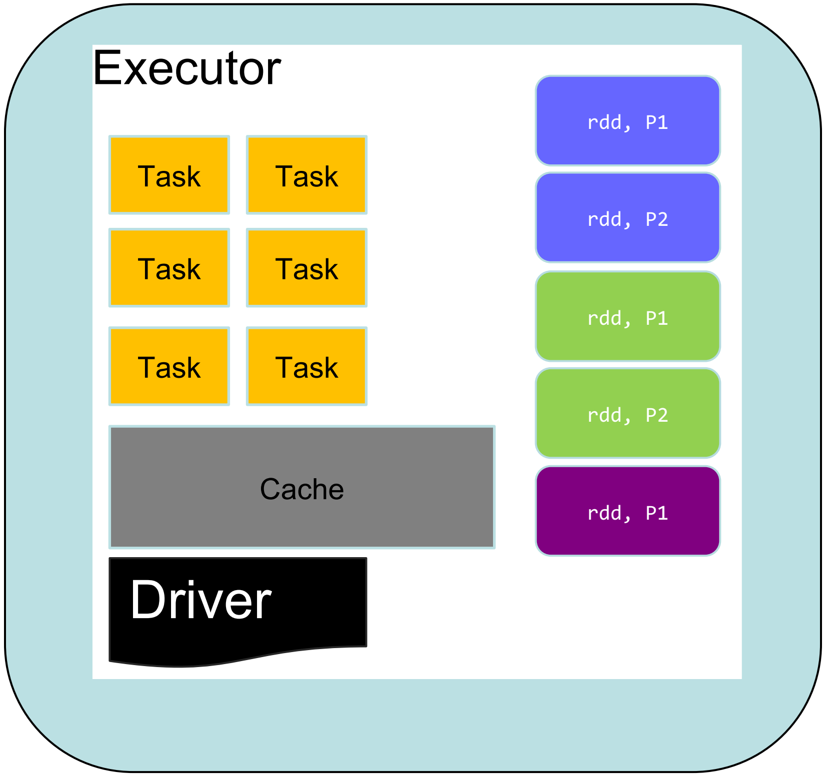
Spark Execution Modes

- Local (Local machine)
- Standalone “Cluster) (manually configured cluster)
- YARN (Hadoop cluster)

Using Container Orchestration Engines:

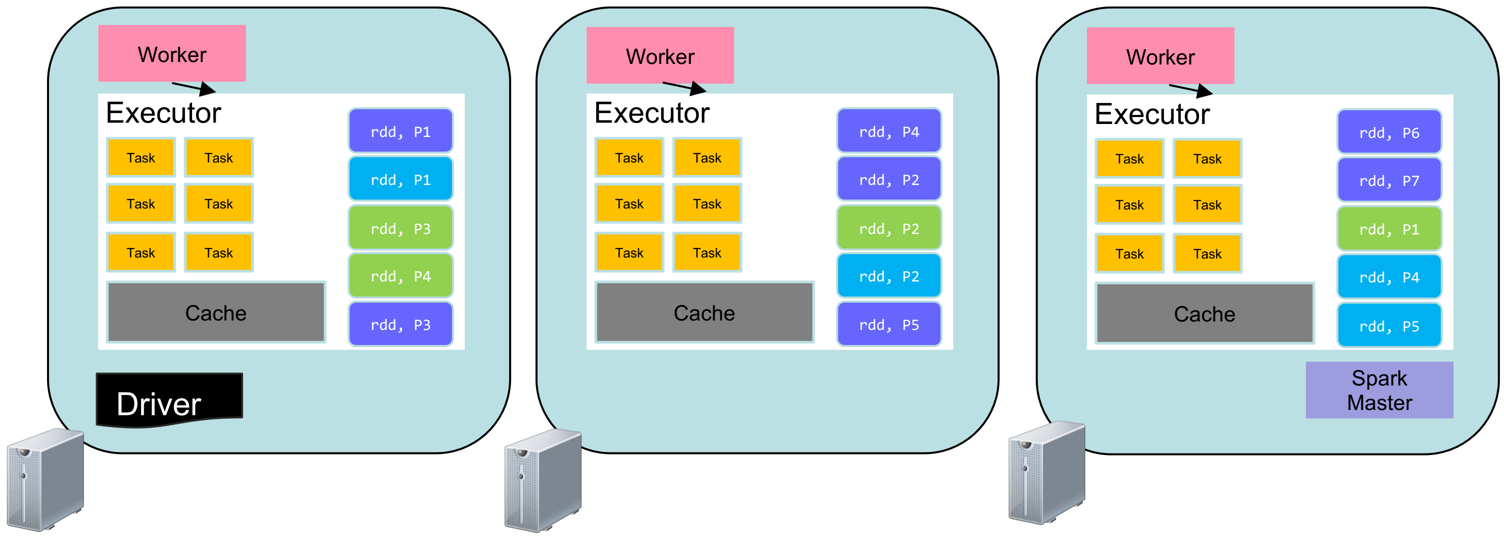
- Mesos
- Kubernetes

Local Mode



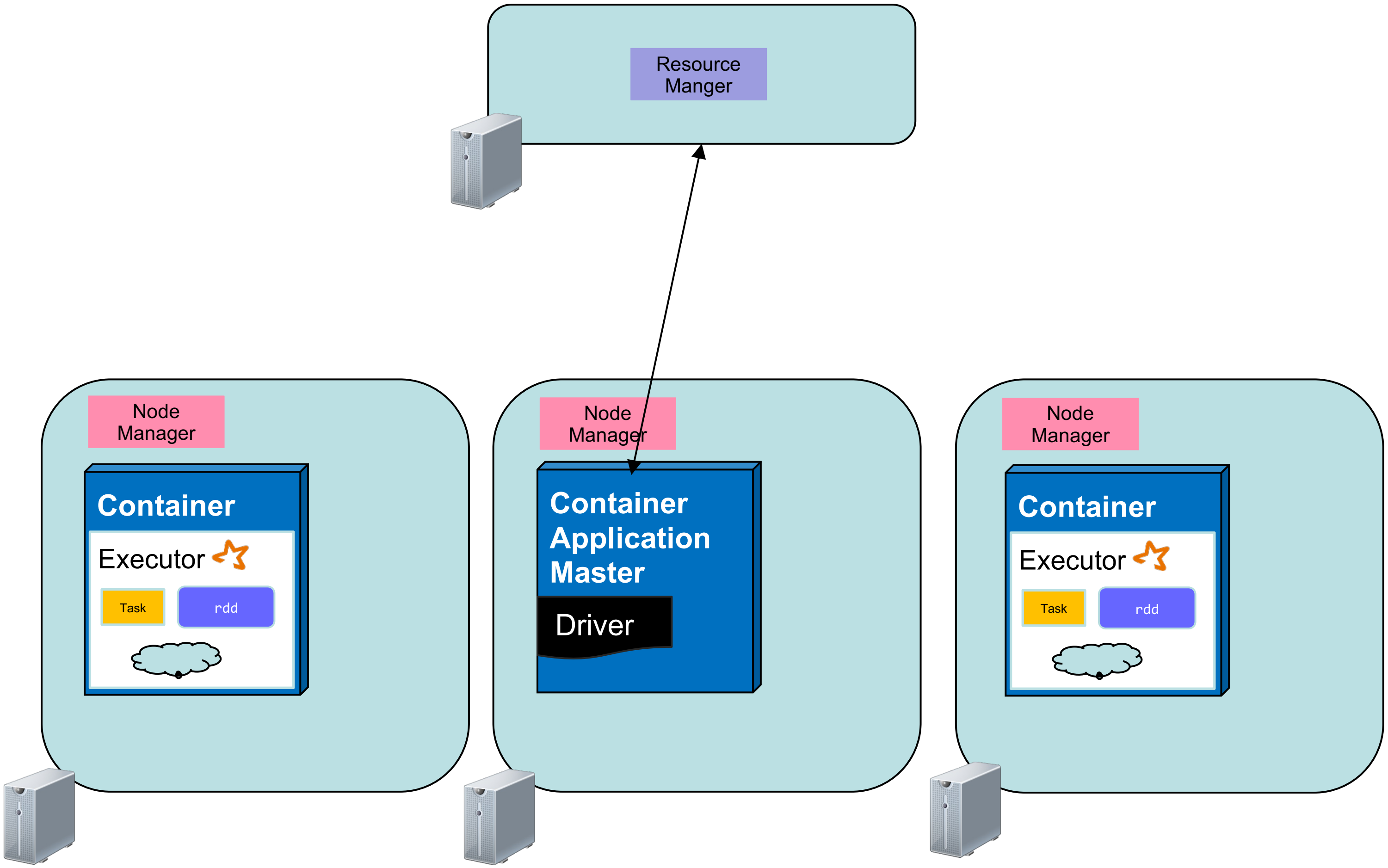
./bin/spark-shell --master local**[6]**

Standalone "Cluster" Mode



./bin/spark-submit --name "SecondApp" --master **spark://host:port** myApp.jar

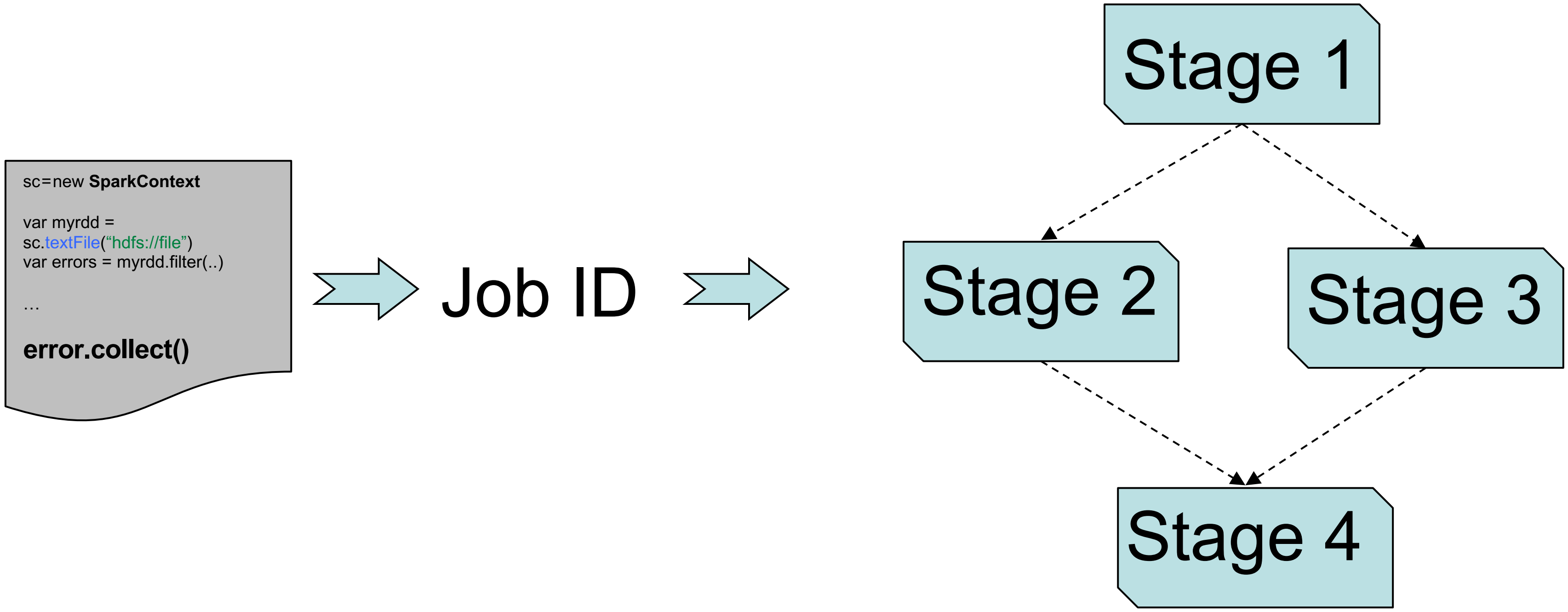
YARN Mode



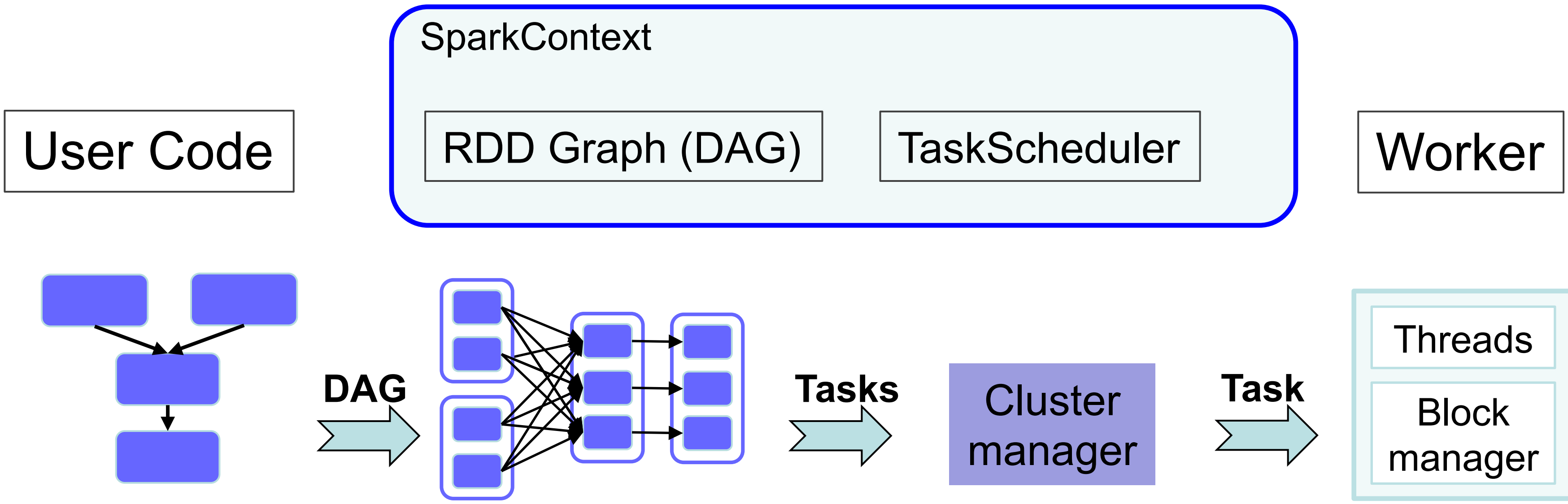
Resource
Manger
Node
Manager
Container
Executor
Task
rdd
Node
Manager
Container
Application
Master
Driver
Node
Manager
Container
Executor
Task
rdd

Staged Execution

Given a job, Spark generates the stages of execution



Scheduling Process



rdd1.join(rdd2)
.groupBy(…)

1. split graph into *stages* of tasks

2. launch tasks via cluster manager

3. 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.
  - How to represent dependencies between RDDs?

- In practice, 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.

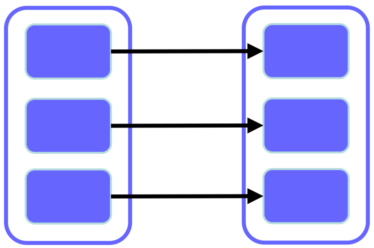
(Zaharia 2012)

DAG Scheduling

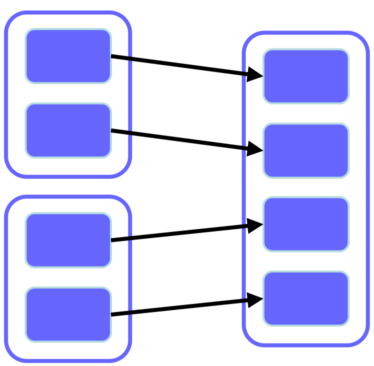
1. Spark creates an operator graph on the user code (RDD lineage)
2. When an Action, the operator graph is submitted to the DAG Scheduler.
3. The DAG Scheduler breaks the lineage into stages based on the presence of wide dependencies.
   - Each stage consists of a set of tasks that can be executed together on the same set of input data. 💡 Spark optimizes the execution plan to minimize the number of shuffle operations required.
4. The stages are then passed on to the Task Scheduler, which launches tasks through the cluster manager.
5. The workers execute the tasks on the worker node. Spark coordinates the execution of tasks across the executors to ensure fault-tolerance and efficient resource utilization

Dependency Types

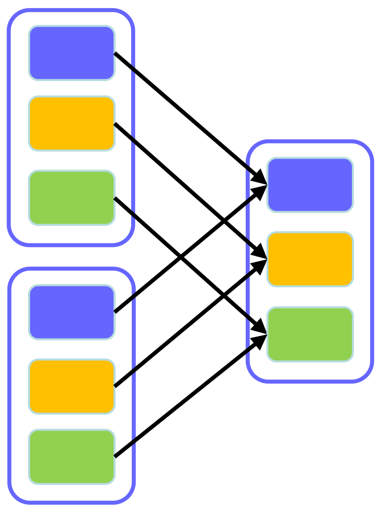
"Narrow" dependencies:



map, filter

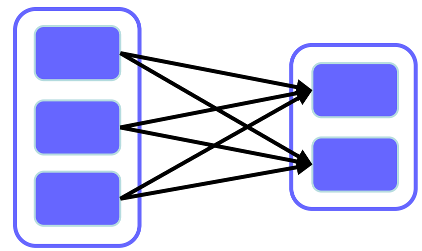


union

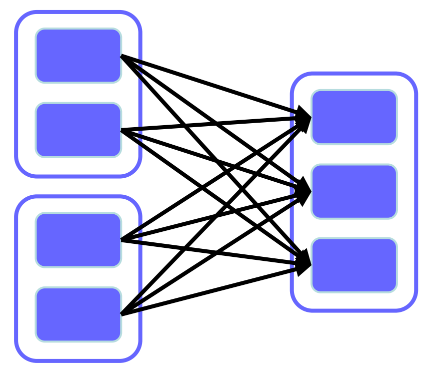


join with inputs co-partitioned

"Wide" (shuffle) dependencies :



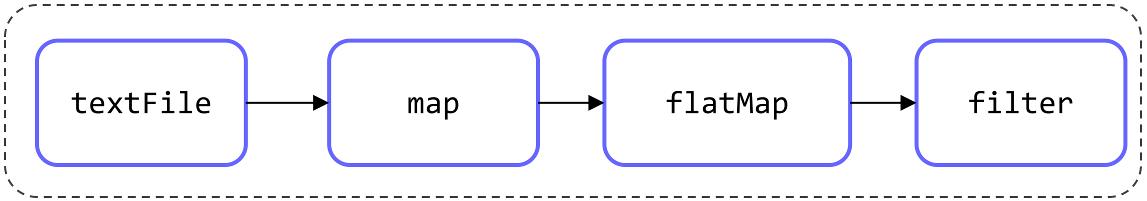
groupByKey



join with inputs not co-partitioned

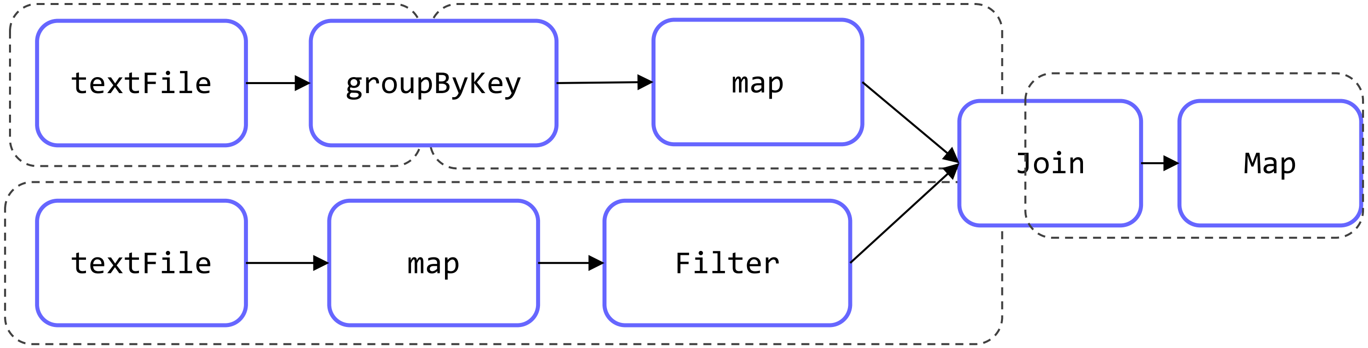
How Many Stages?

```
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 is used for efficient distributed data processing:
  - In-memory
  - Lazy execution
  - MapReduce principles
  - Graph of executions
- RDD
  - A distribtributed data structure in Spark.
  - It defines how a collection (e.g., a list) is distributed on a cluster of machines.
- Getting Started with RDDs:
  - Programming Guide