▷ Yogesh Simmhan
▷ simmhan@iisc.ac.in

▷ Department of Computational and Data Sciences
▷ Indian Institute of Science, Bangalore

DS256 (3:1)

Scalable Systems for Data Science

Module 2

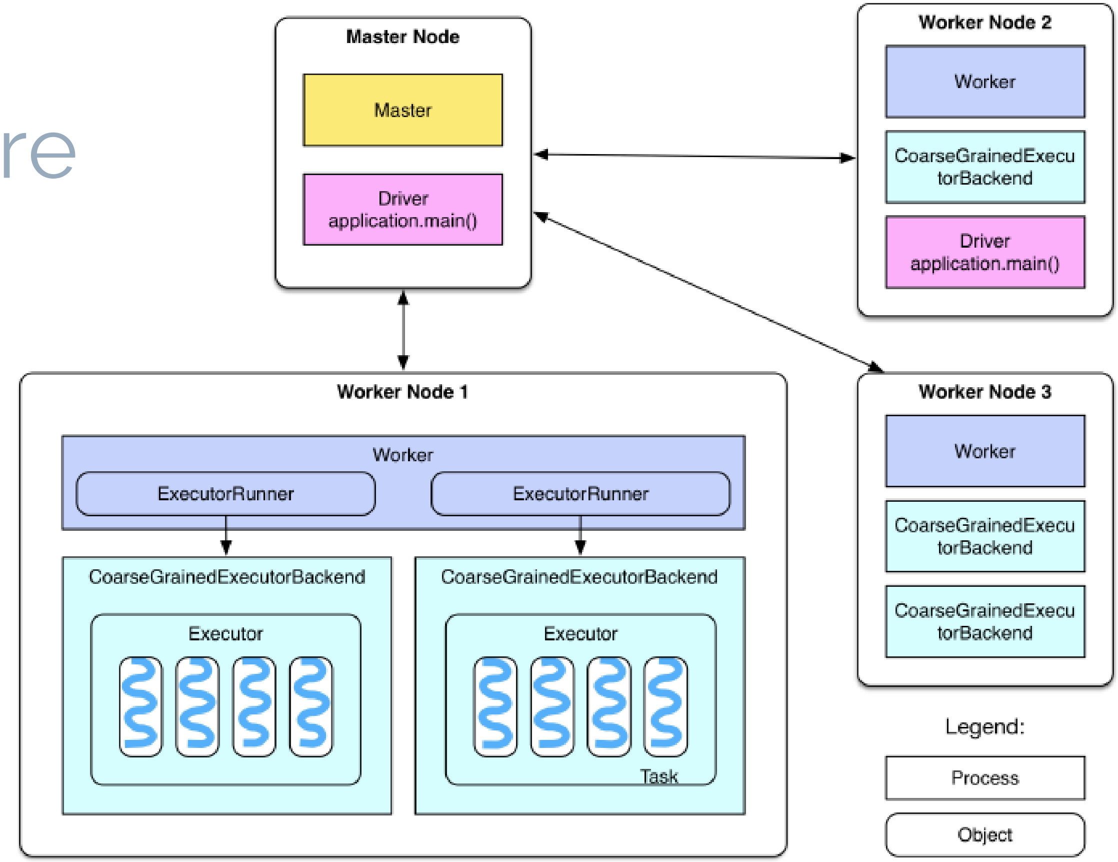
Processing Large Volumes of Big Data

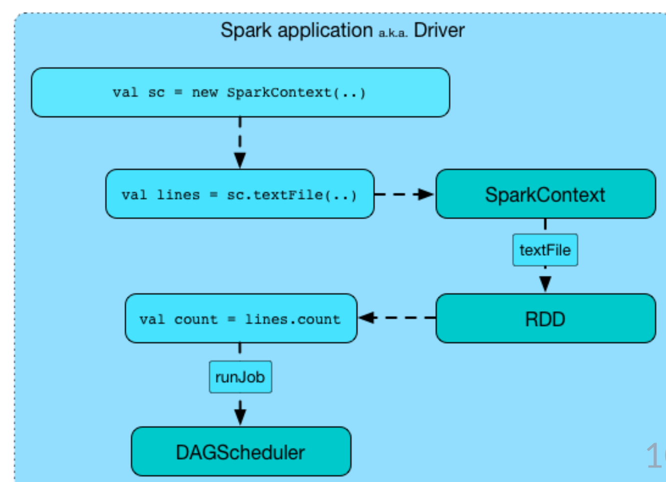
Spark Internals

Select topics from external sources

Spark Architecture

- Spark runs on a cluster of machines
- Driver interfaces with SparkContext
- Logical Plan
  - Converts application to a dataflow of dependencies
- Physical Plan
  - Converts dataflow into specific tasks for execution
  - Tasks executed within Workers/Executors

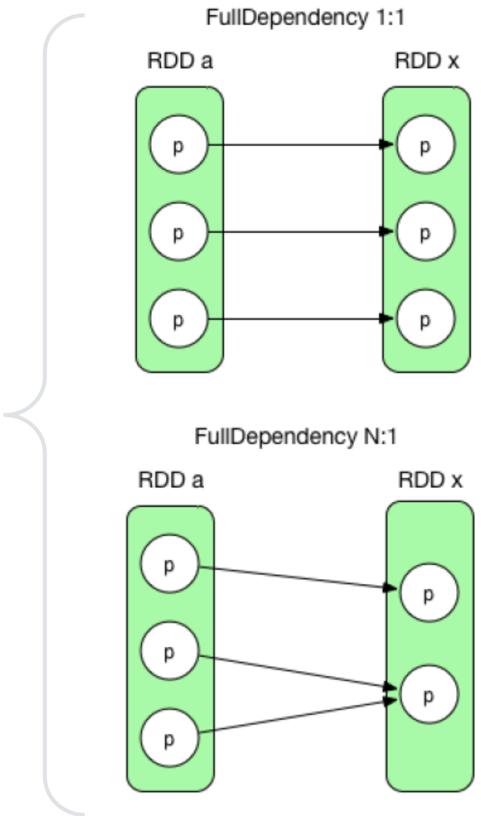


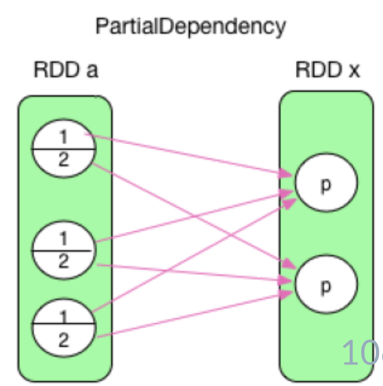


Logical Plan

Narrow and Wide Dependencies

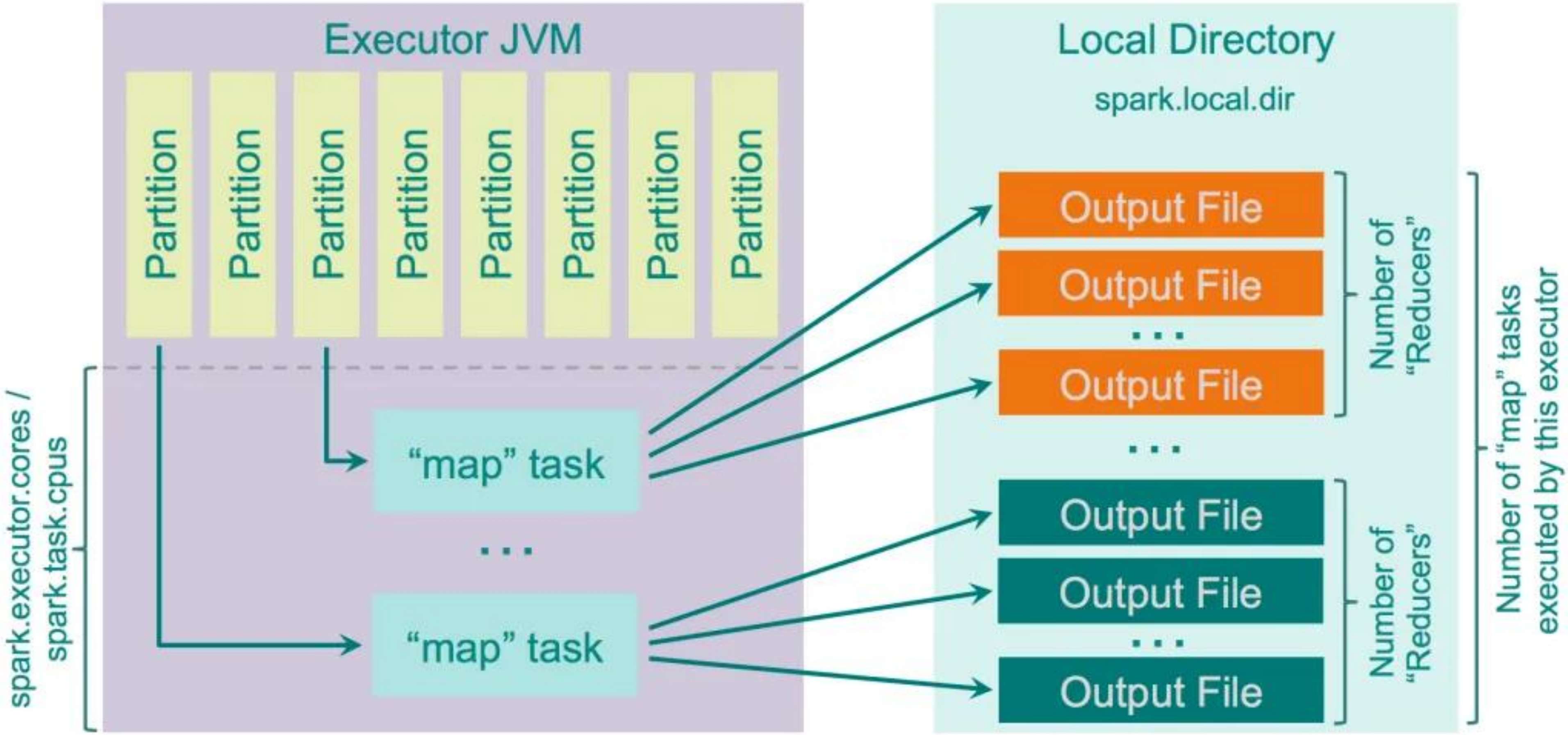
- ▷ Narrow Dependency
  - Each output partition depends on exactly one or a few input partitions
  - Each input partition is used by exactly one output partition
  - So output and input partitions can be on same Worker
  - Also called full dependency
  - E.g., map, cogroup
- ▷ Wide Dependency
  - Each output partition depends on parts of one or more input partition
  - Each input partition can be used by one or more output partitions
  - So forces a shuffle across Workers
  - Also called Shuffle or partial dependency
  - E.g., join, groupByKey





Hash Shuffle in Spark

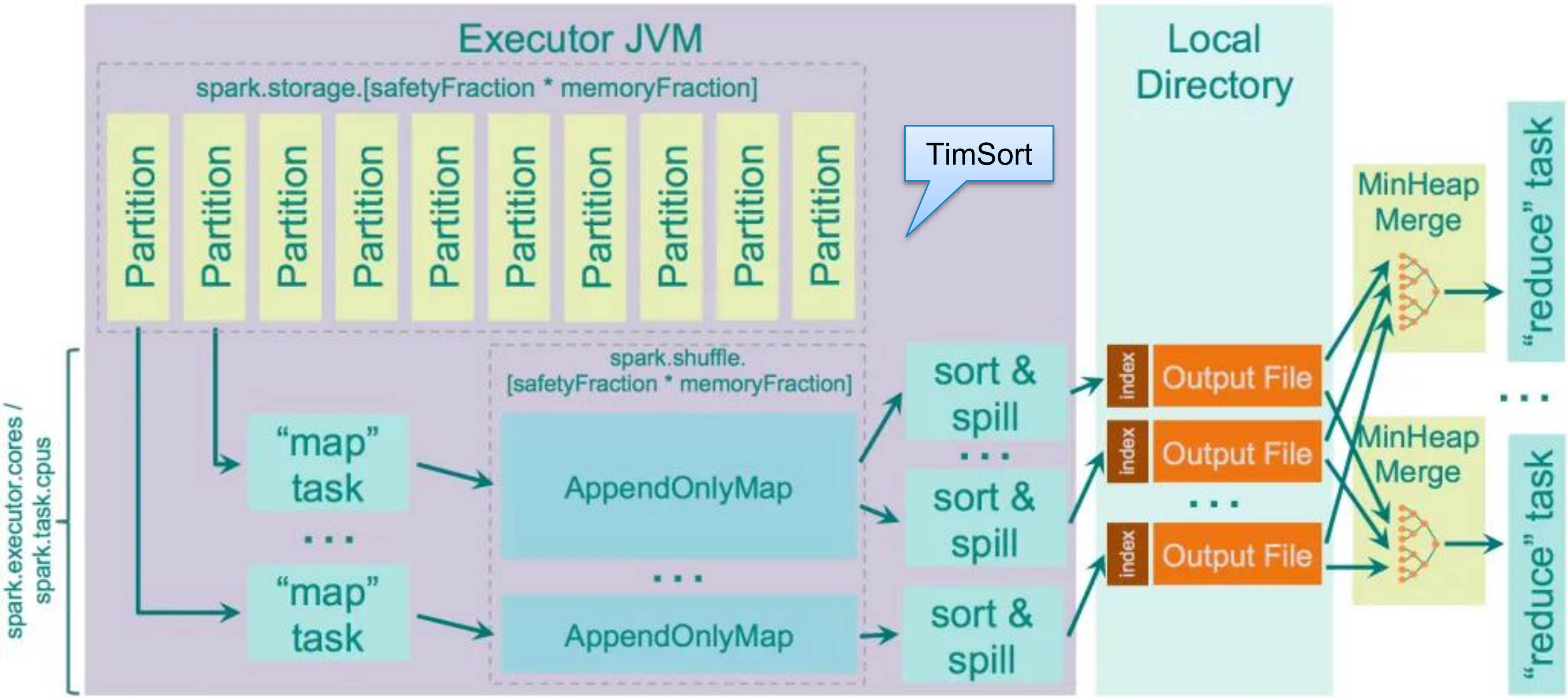
- ▷ Write to one file per reducer per map task



https://0x0fff.com/spark-architecture-shuffle/

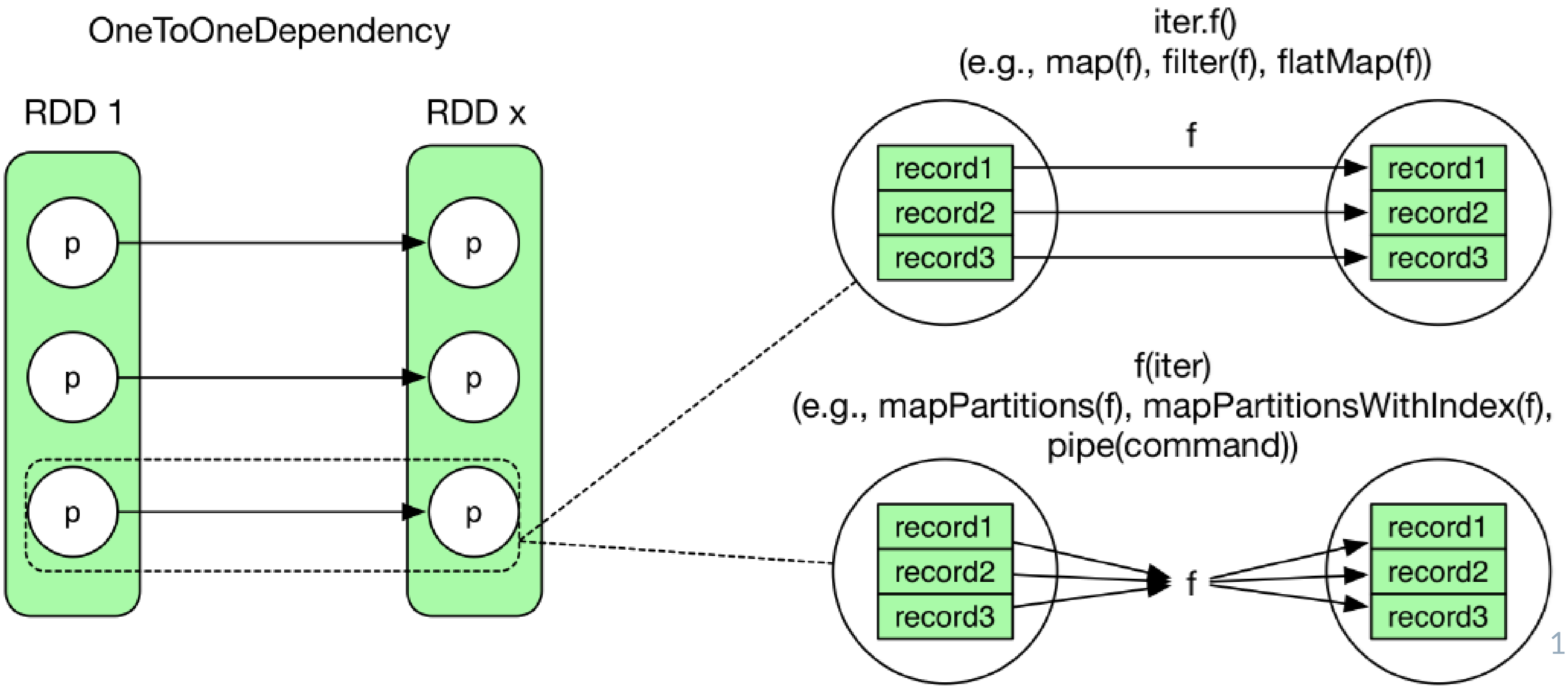
Sort Shuffle in Spark >= v1.2.0 (~MapReduce)

- Write to a single (spill) file per Map task, maintain offset for each reducer



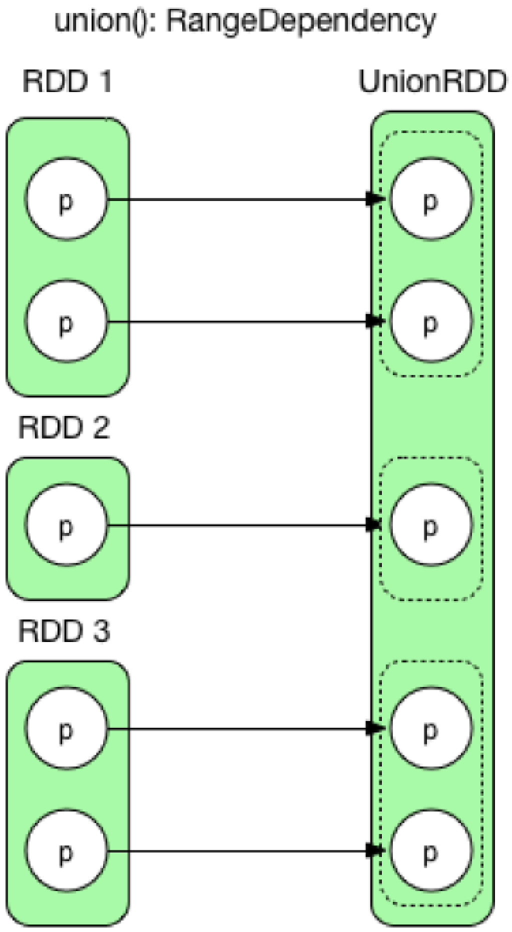
Narrow Dependency (One to One)

- 1:1 mapping between output and input records, e.g., map, filter
- 1:1 mapping between output and input partitions, e.g., mapPartitions



Narrow Dependency (Range)

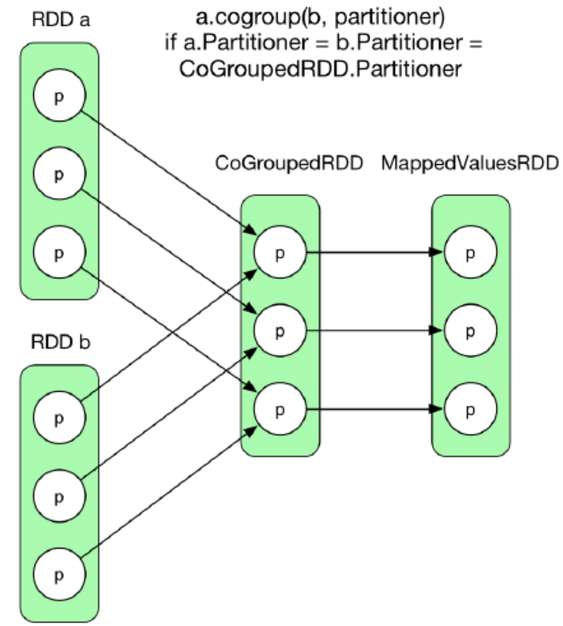
- Range dependency
  - Ranges are retained between input and output
  - E.g., union

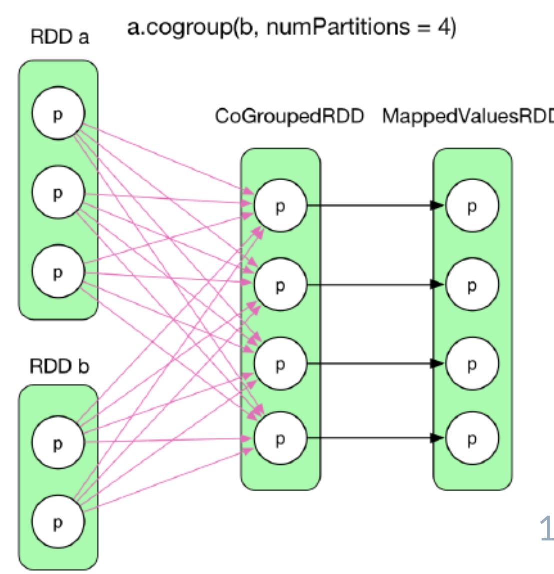


Narrow Dependency (N:1)

- ▷ Many to One Dependency
  - ○ Multiple input partitions generate a single output partition
  - ○ Operates on multiple RDDs
  - ○ Requires both RDDS to
    - ▪ Have same number of partitions
    - ▪ Use same partitioner
  - ○ E.g., cogroup, colease

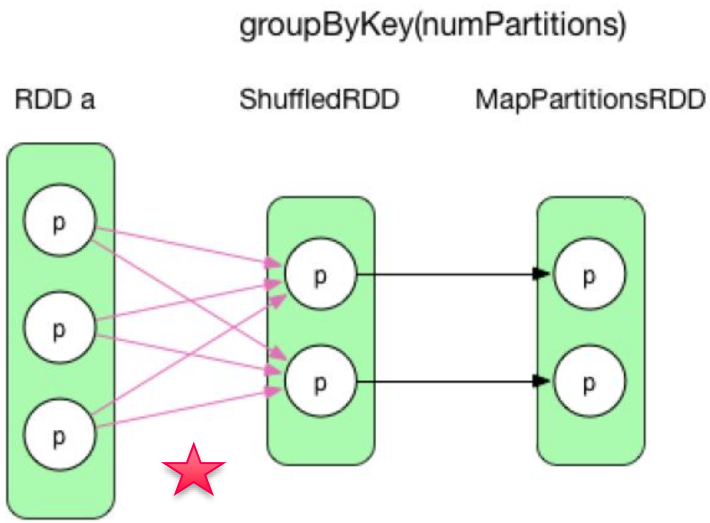
- ▷ If # parts or partitioner not same, this can become wide dependency

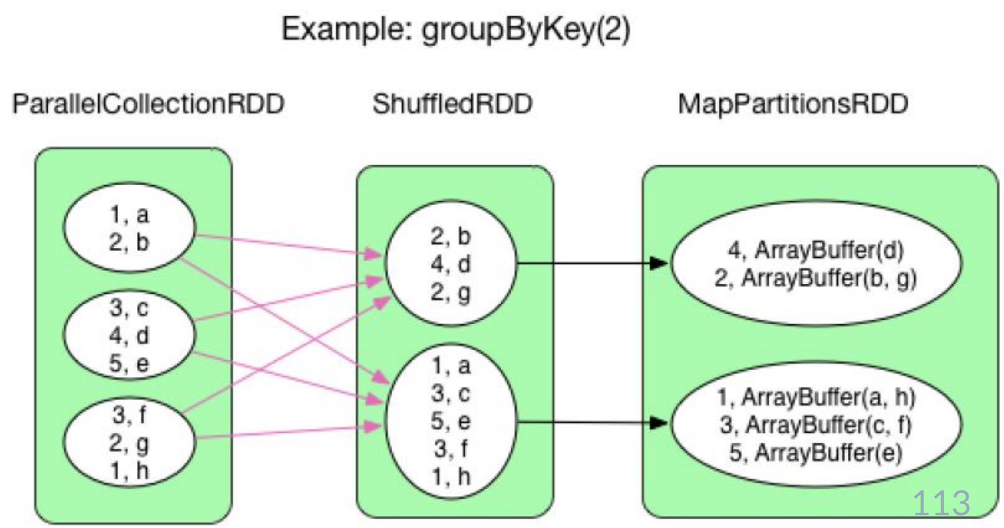




Wide Dependency (Single RDD)

- Each input partition can contribute to multiple output partitions
- Since data moves from an input to multiple output workers, this causes a shuffle
- Shuffles are costly!

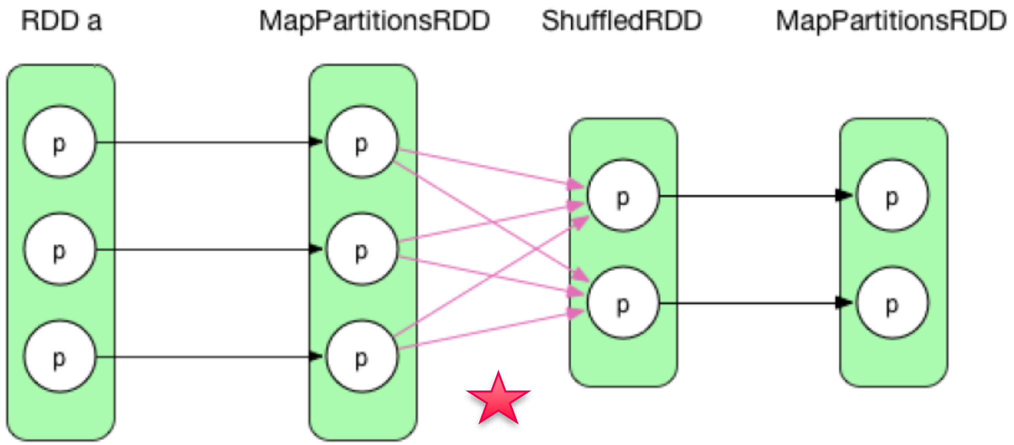




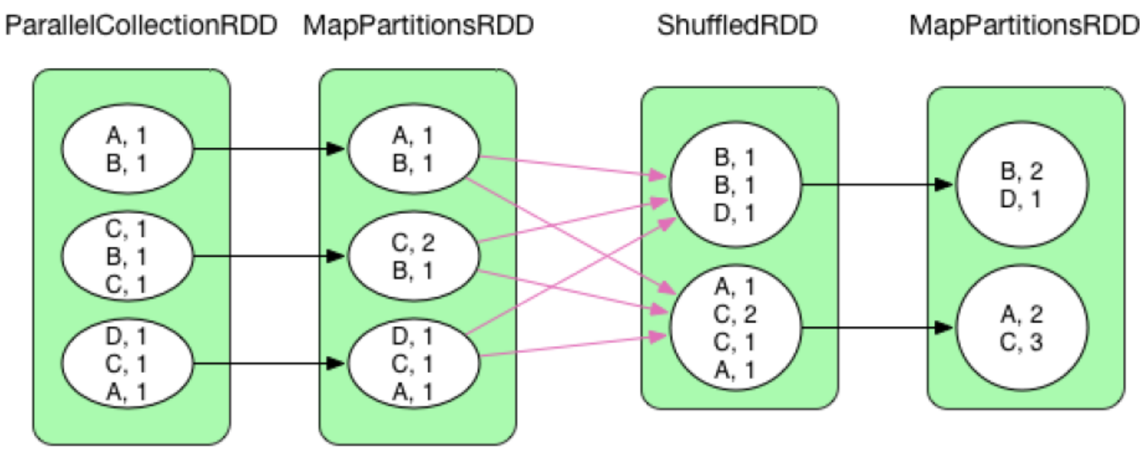
- Shuffle can be across partitions for one RDD
  - E.g., groupyKey, reduceByKey, sort, distinct

Wide Dependency (Single RDD)

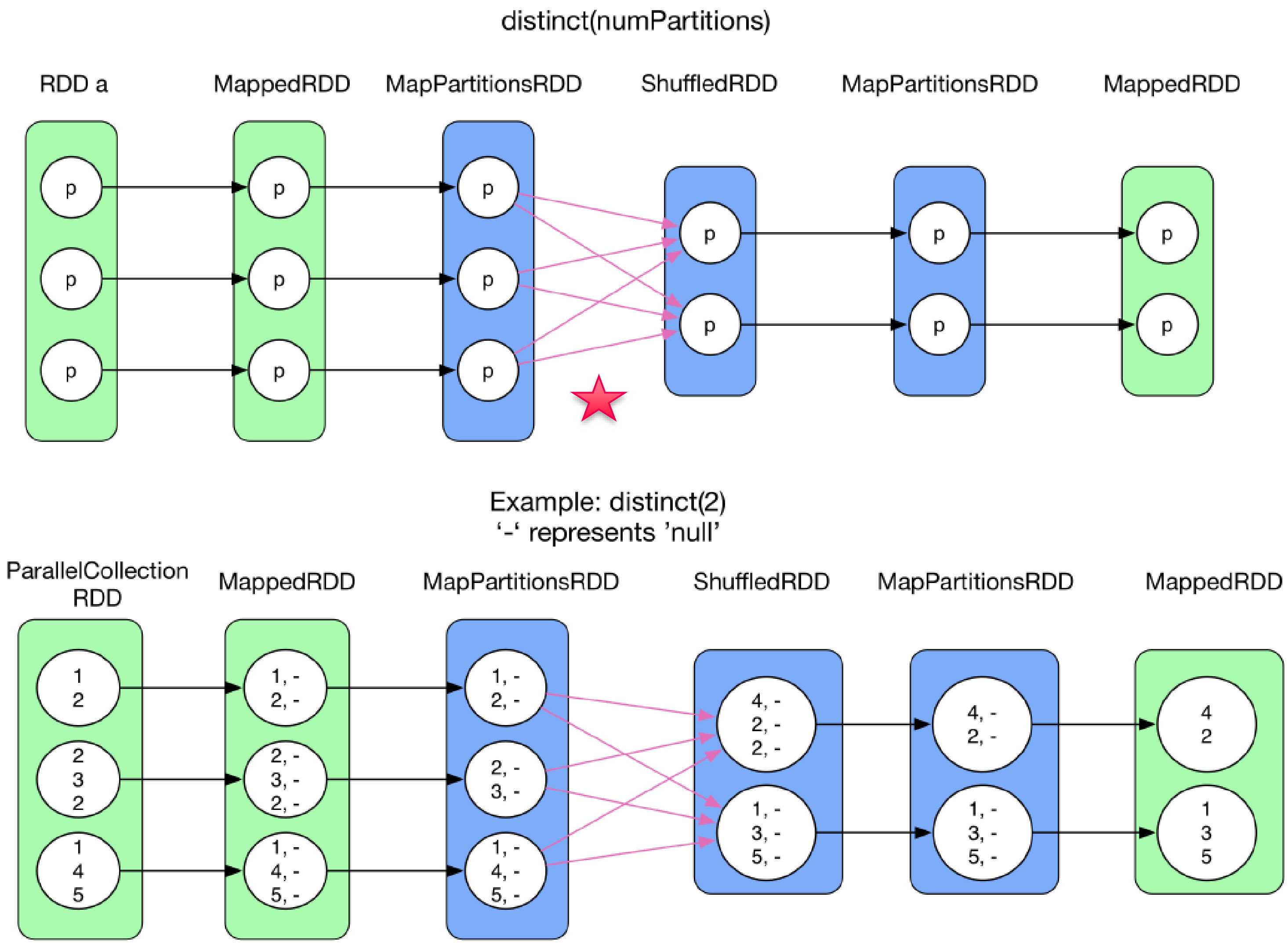
reduceByKey(f, numPartitions)



Example (WordCount): reduceByKey(_ + _, 2)

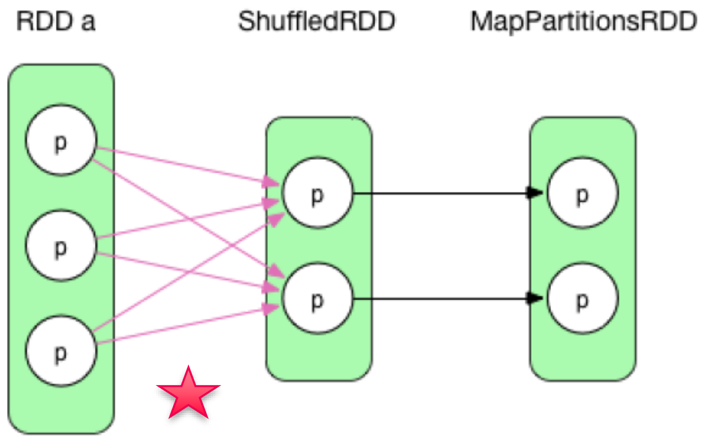


Wide Dependency (Single RDD)

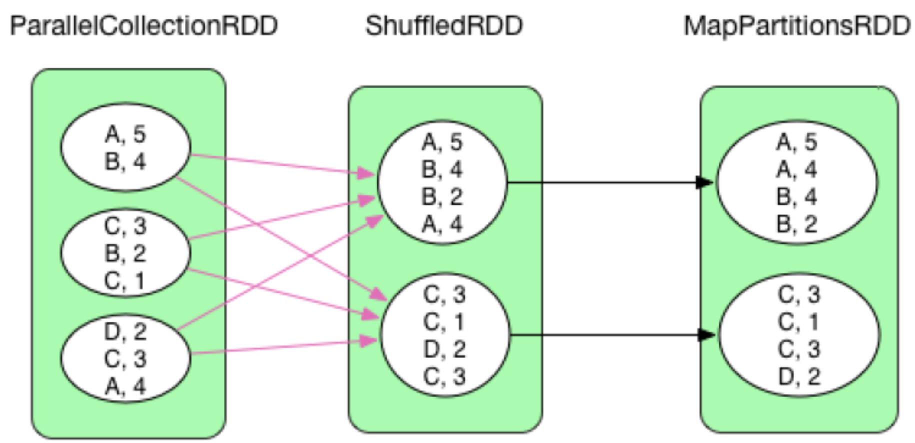


Wide Dependency (Single RDD)

sortByKey(ascending, numPartitions)

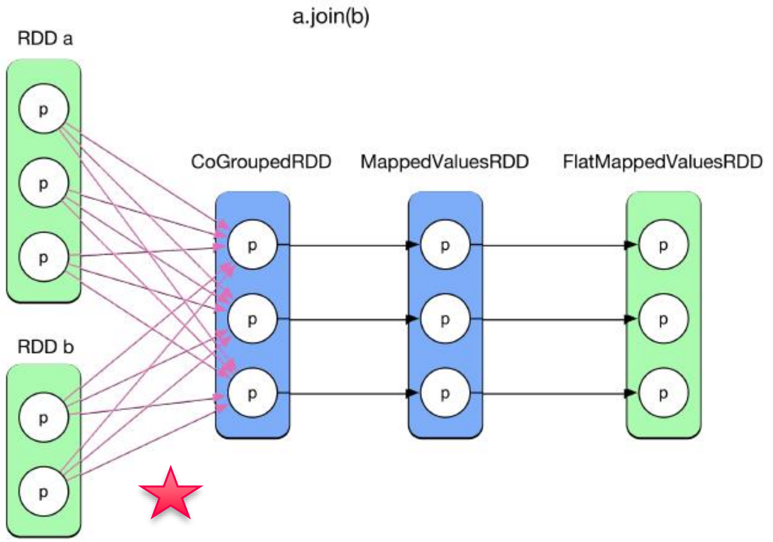


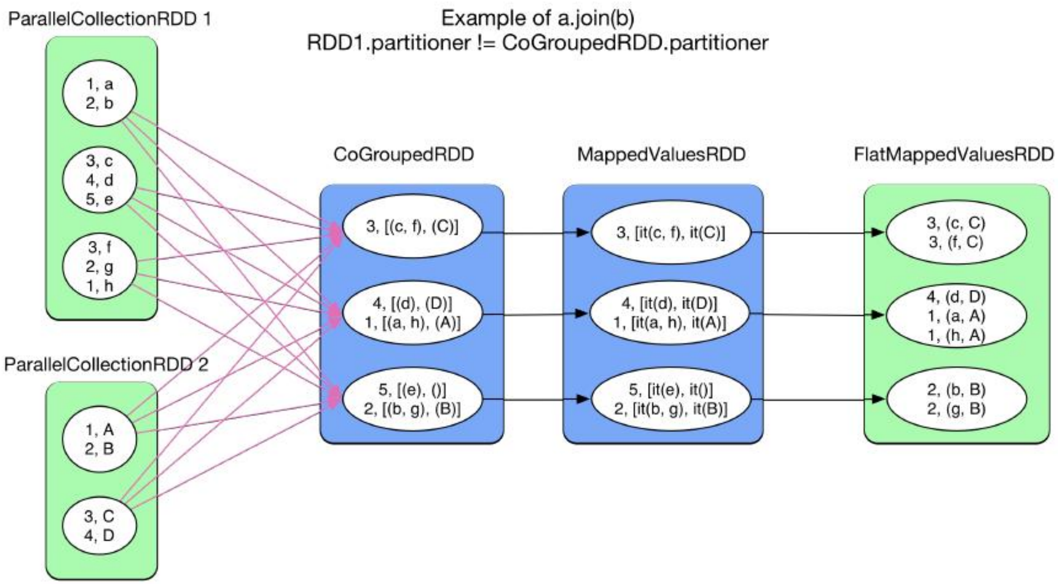
Example of sortByKey(true, 2)



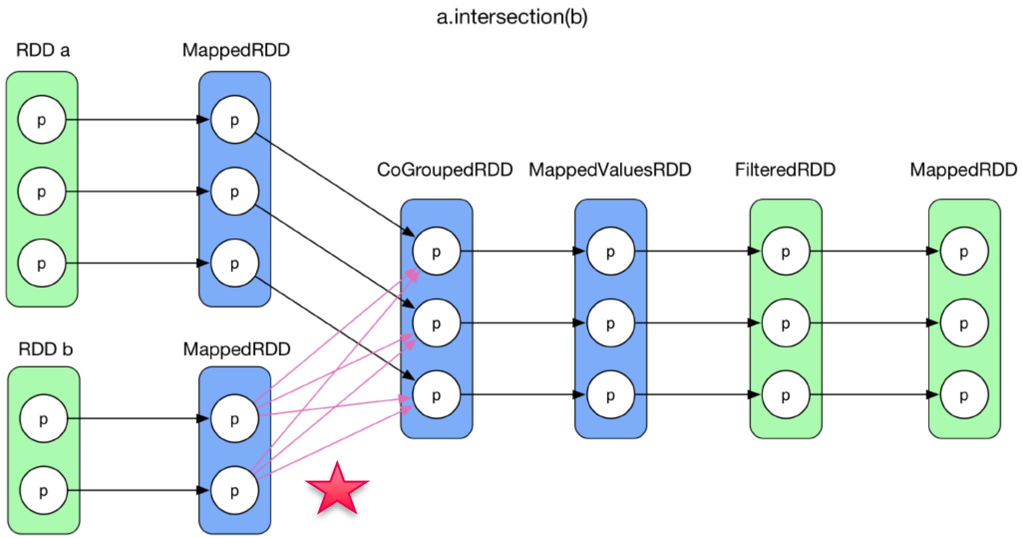
Wide Dependency (Multiple RDDs)

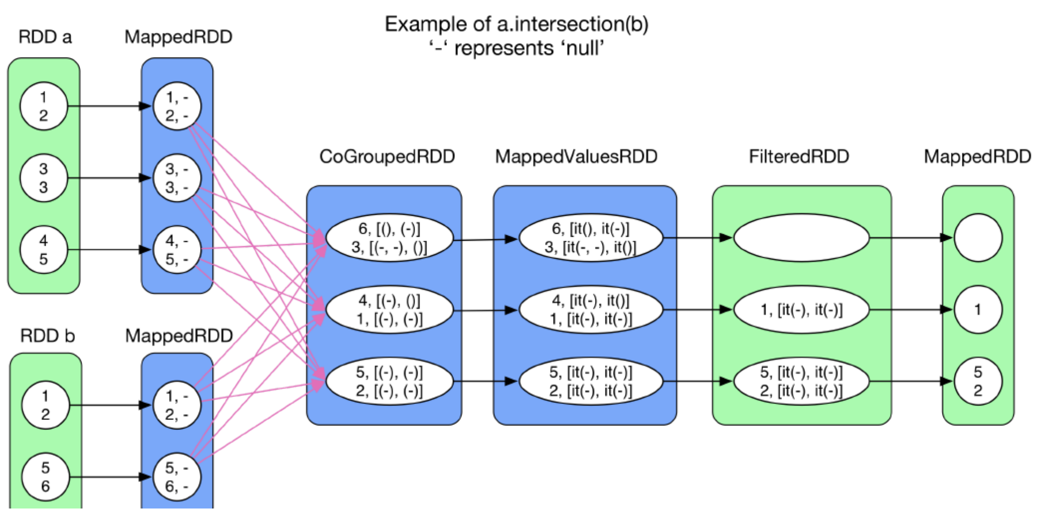
- Shuffle can also combine partitions from multiple RDDs
  - E.g., join, intersect



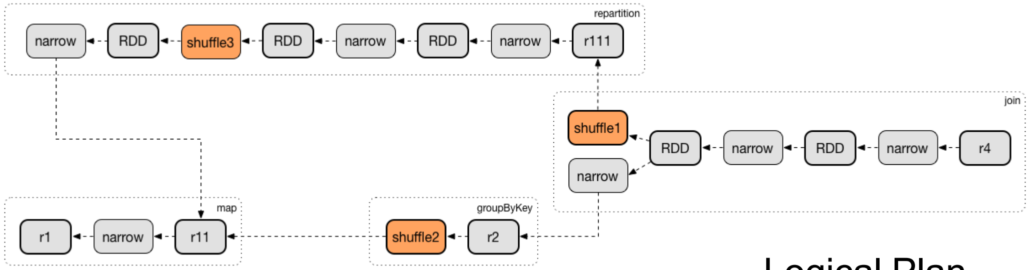


Wide Dependency (Multiple RDDs)





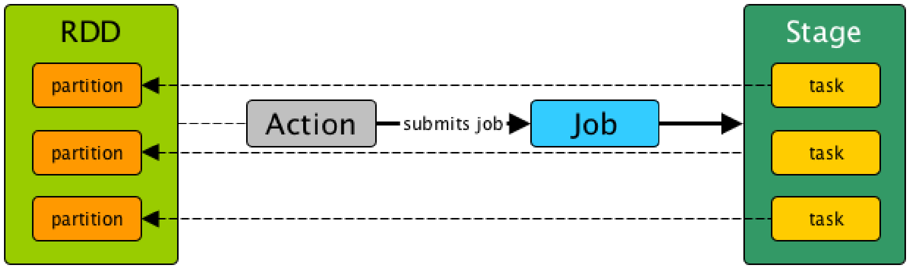
Physical Plan and Scheduling



Logical Plan

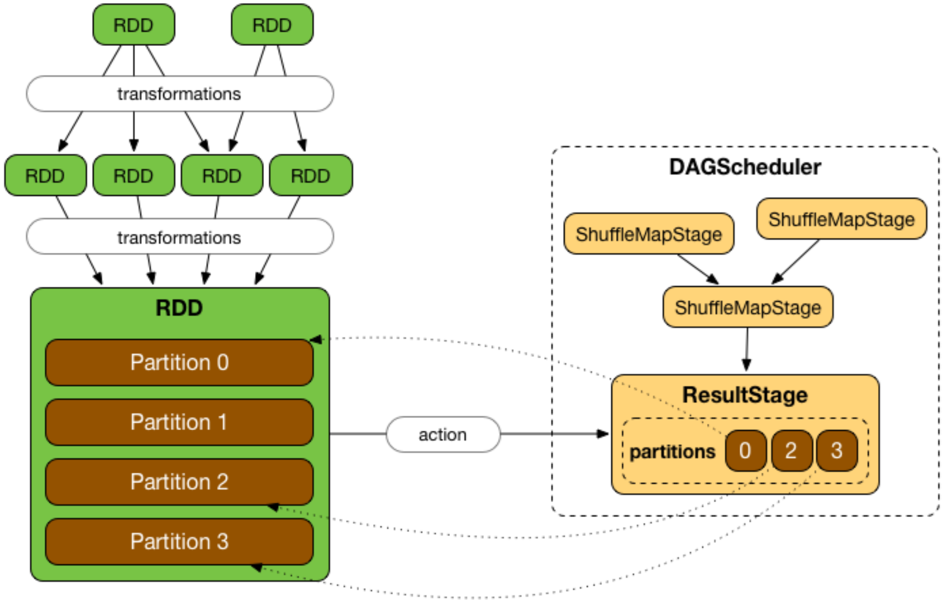
- Convert logical plan to execution on cluster
- Create a Job for each Action
- Use DAG scheduler to create stages, Tasks for Job
- Schedule and coordinates tasks on cluster

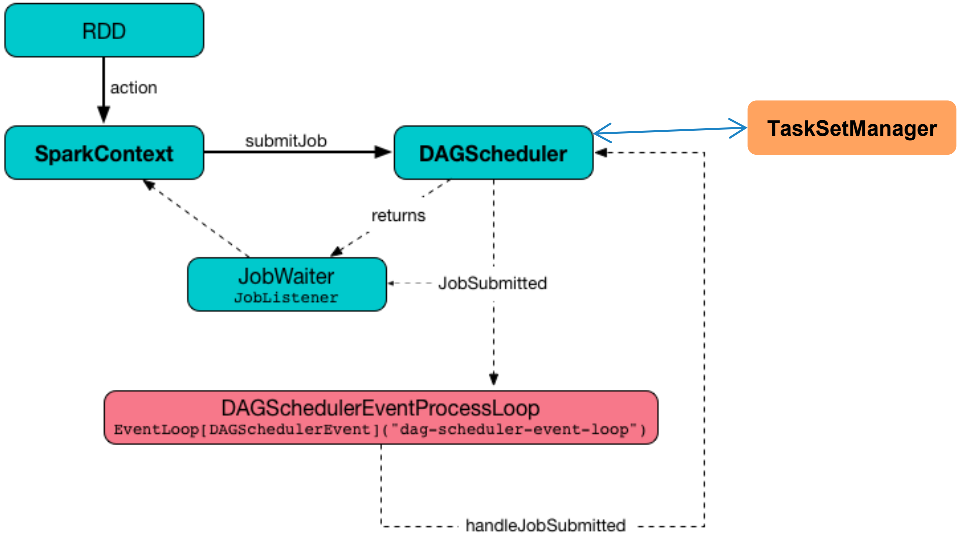
Jobs: Stages and Tasks

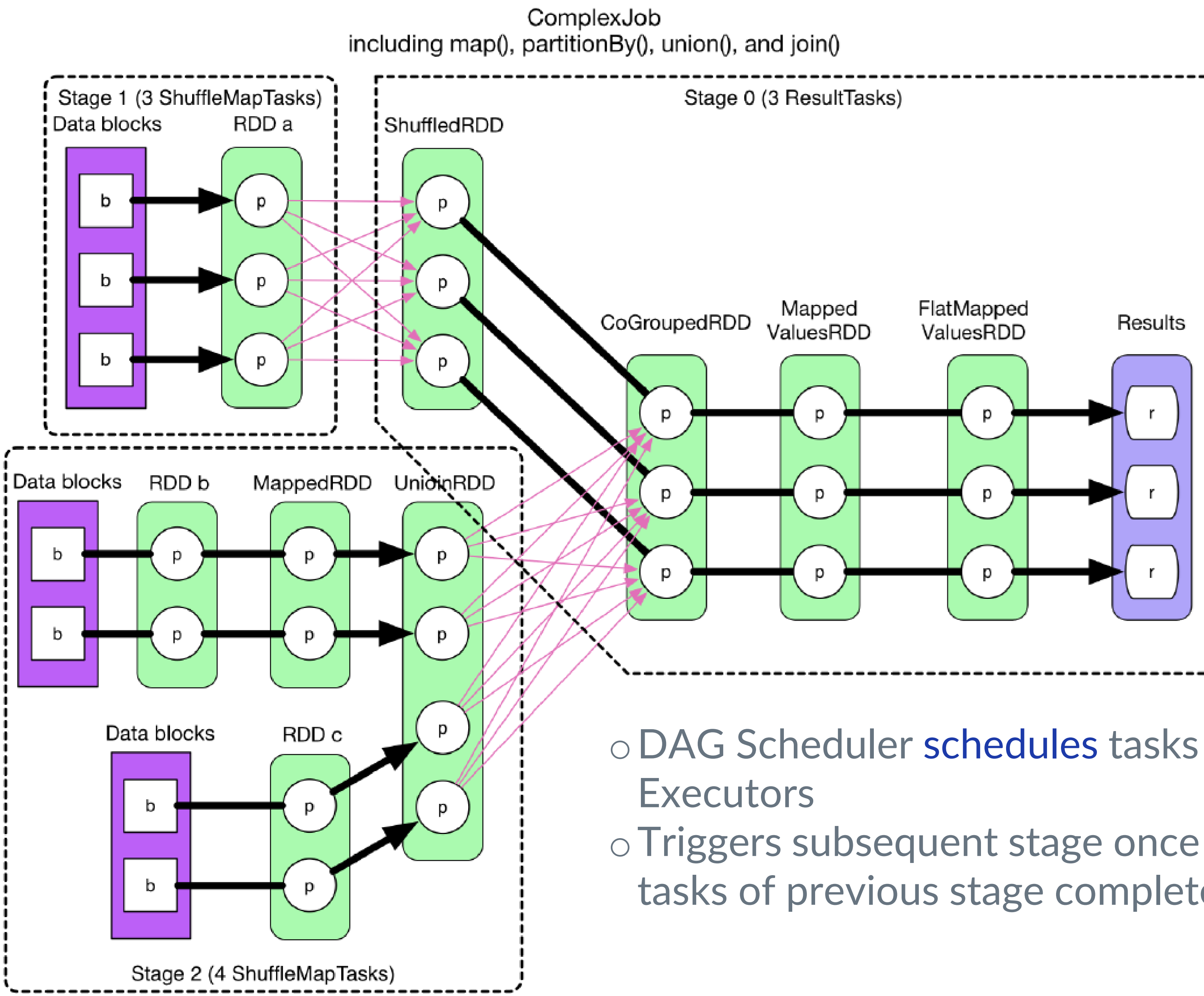


DAG Scheduler

- ▷ **Stages** are separated by a wide dependency
- ▷ Creates tasks for different stages
  - ShuffleStage
  - ResultStage
- ▷ Each **Task** is responsible for one output partition
- ▷ **Task set** is a collection of tasks to generate an output RDD

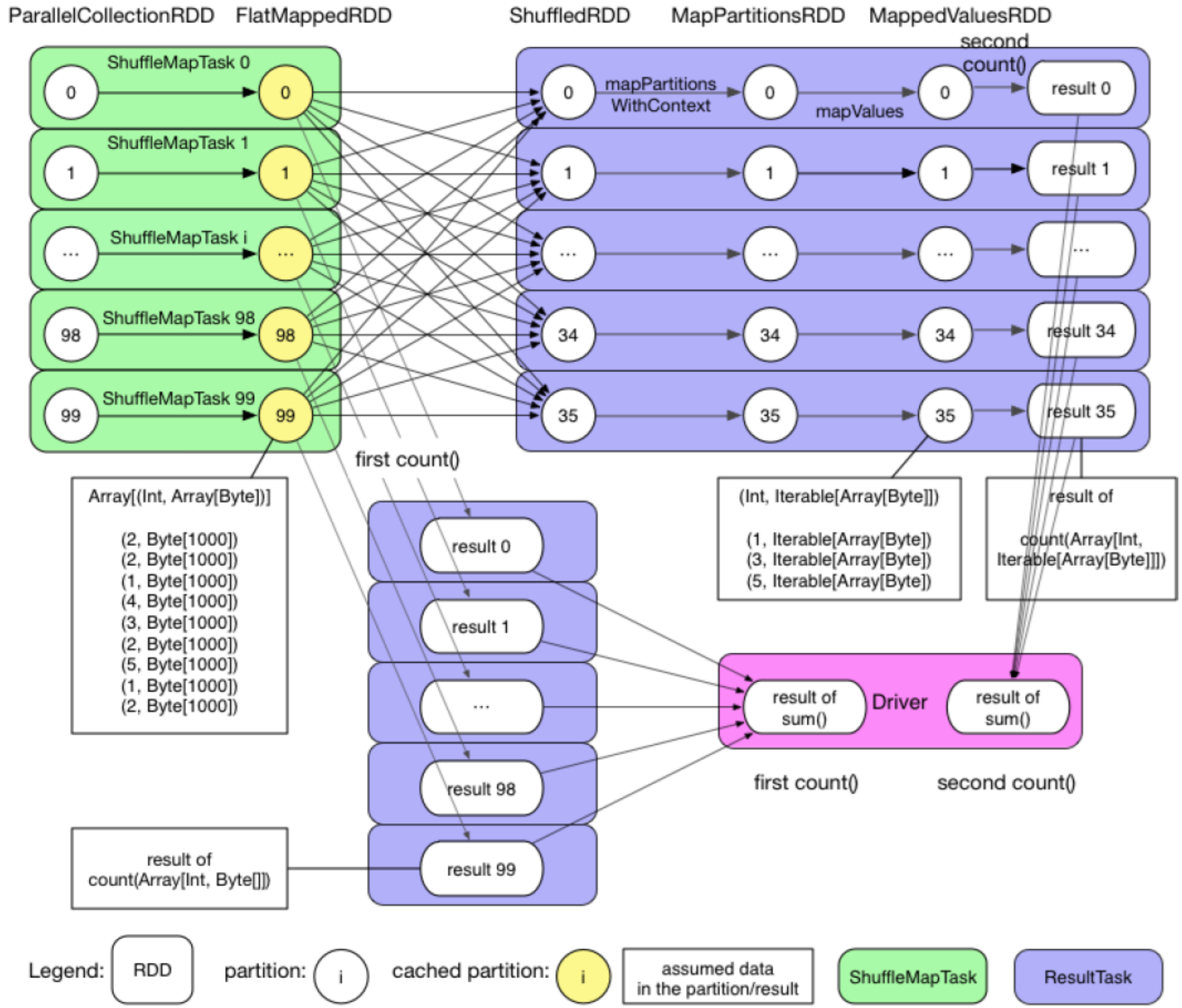






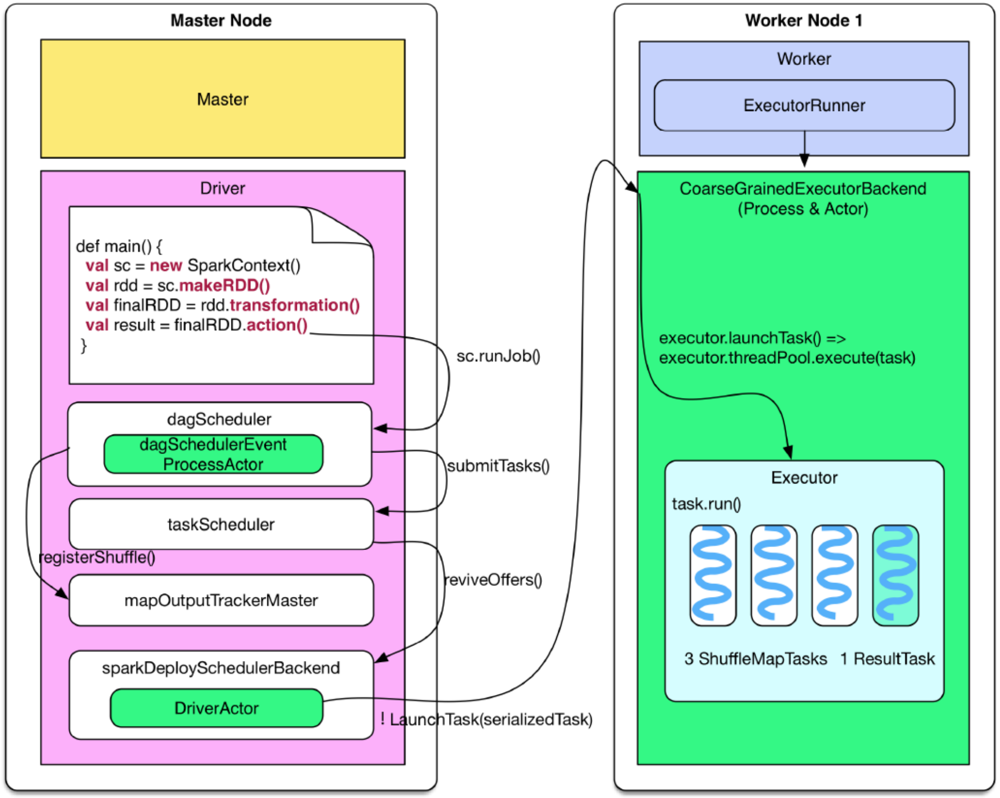
- DAG Scheduler **schedules** tasks onto Executors
- Triggers subsequent stage once tasks of previous stage complete

- Narrow dependencies are pipelined within a single Task
  - Reduces scheduling overheads, data copies, barrier sync delays



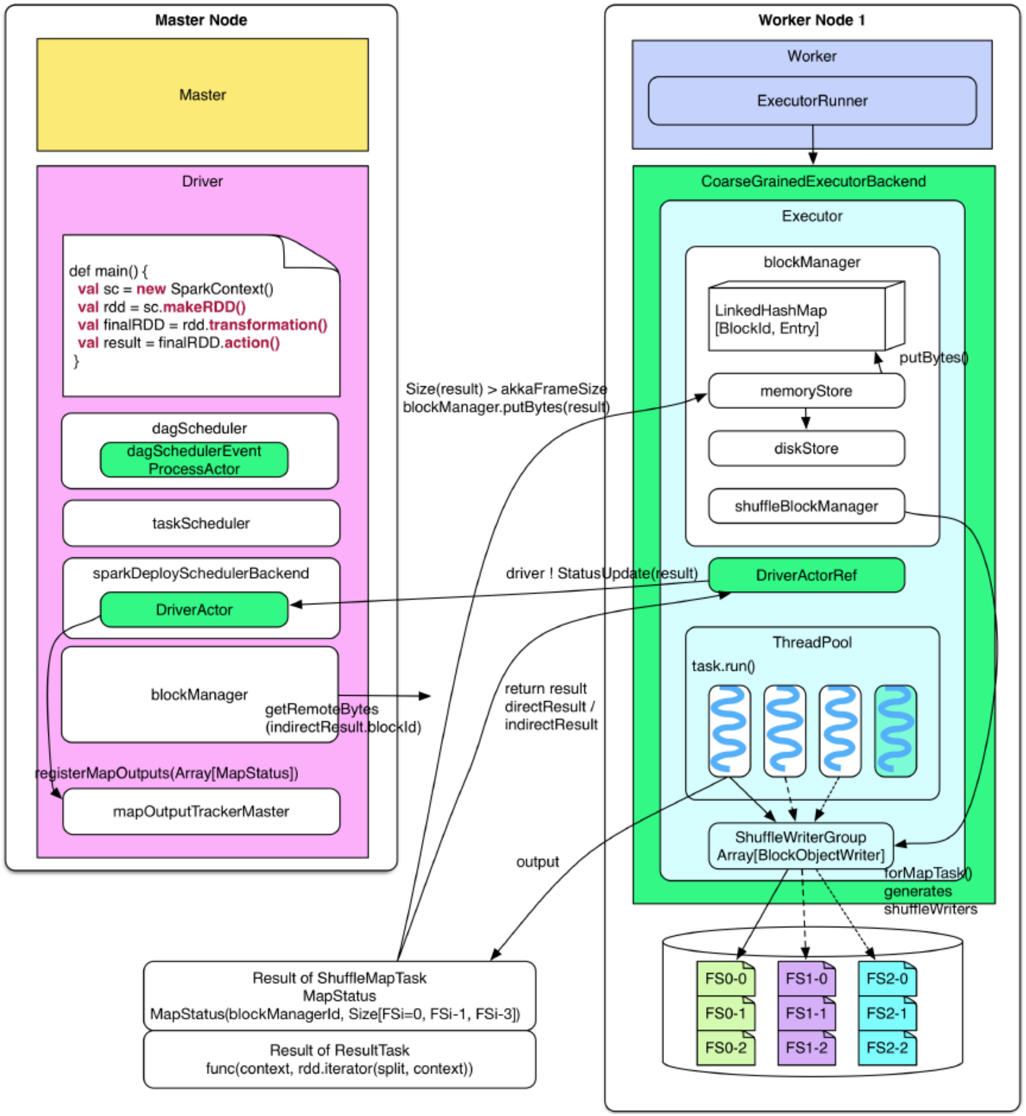
Worker and Executor Model

- **Workers** host one or more **Executors**
  - Worker is one machine
  - Executor is one Process
- Executors execute Tasks
  - Each task is responsible for one output partition
- Tasks run on separate **Threads**
  - Thread pool for concurrency
  - Low thread overhead for executing a task

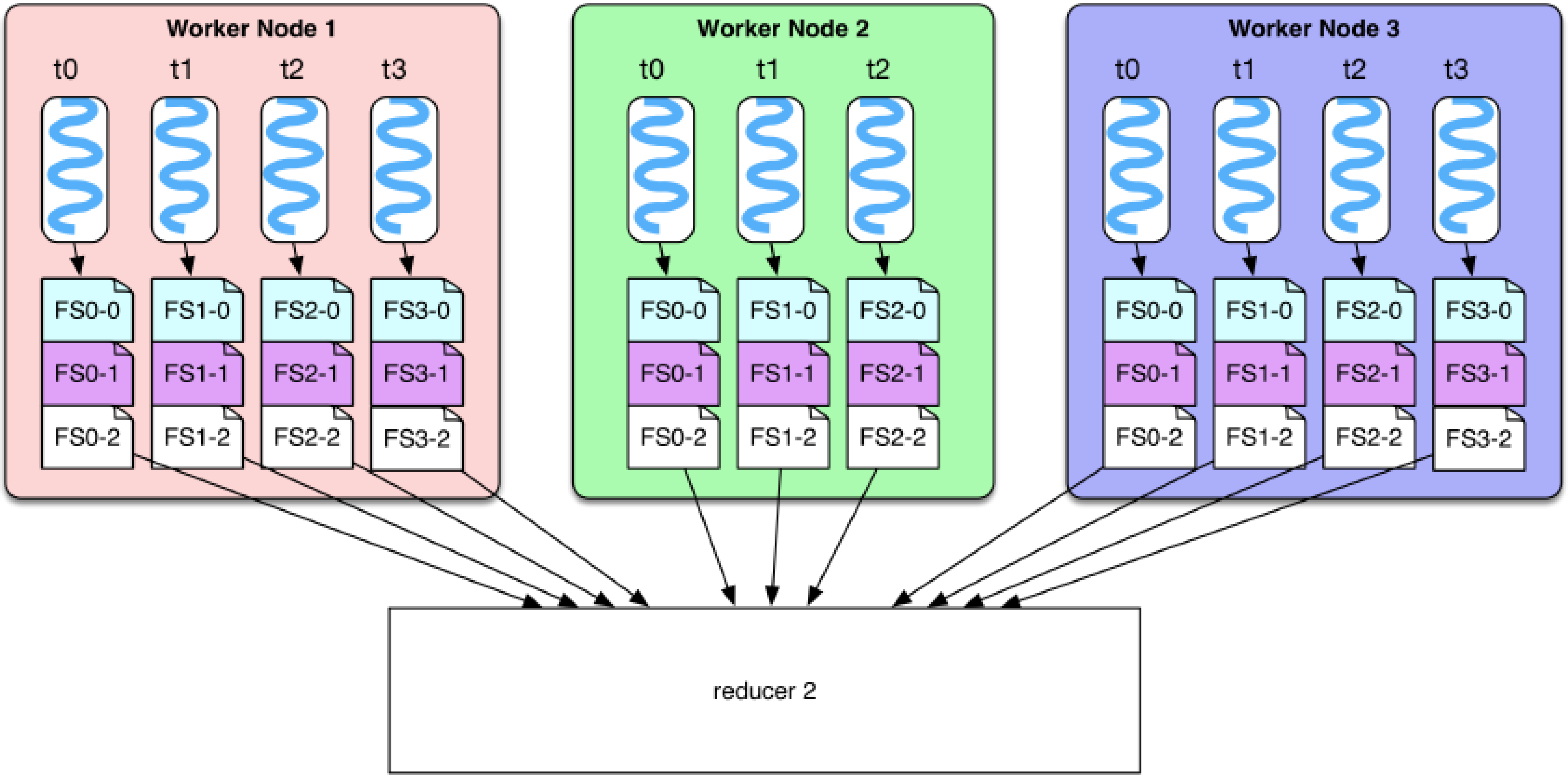


Execution Model

- Executors manage **blocks**, with DAG Scheduler
- Executors manage block **persistence**
- Executors coordinate shuffle

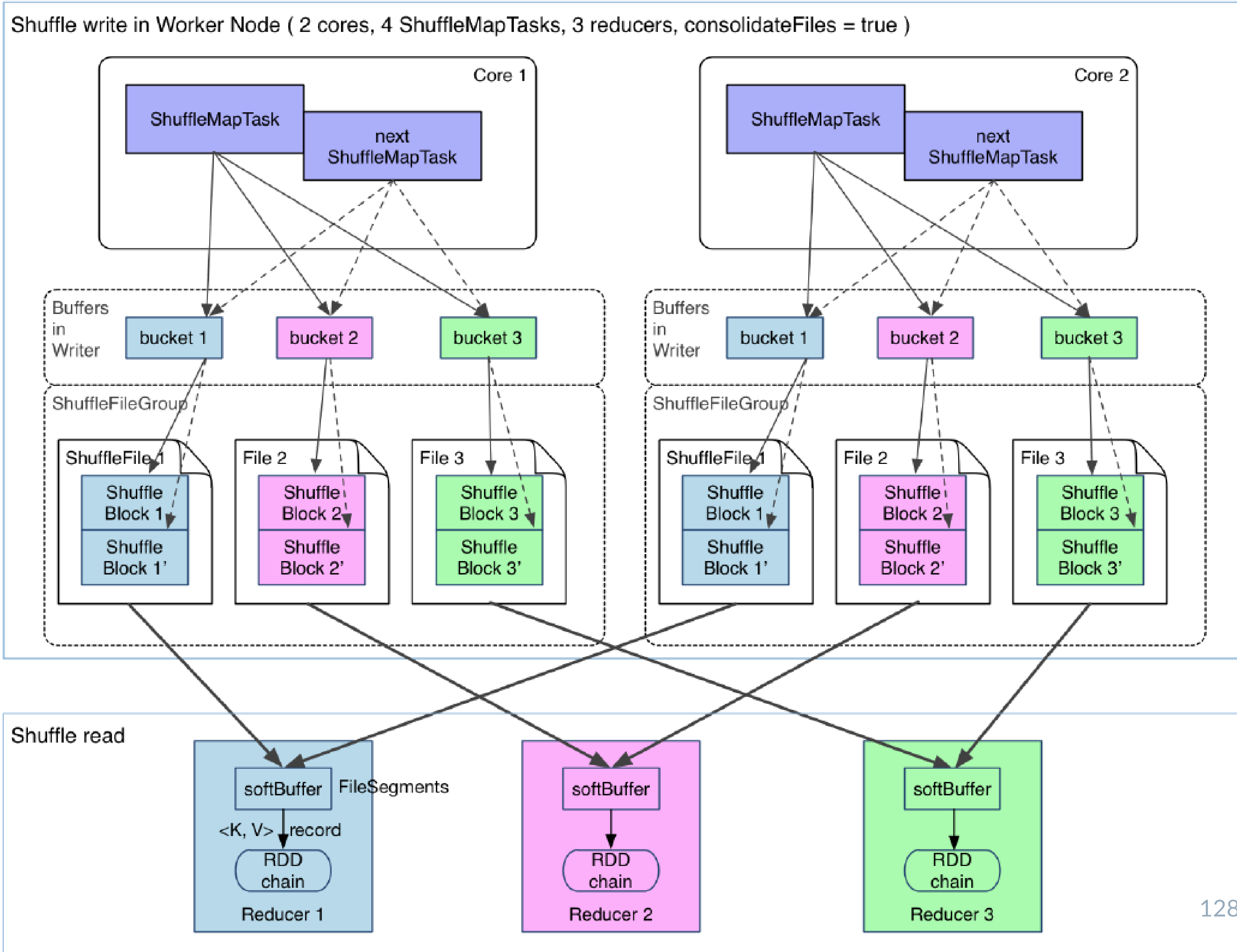


Executors Coordinate Shuffle



Shuffle

- Allows items from one partition to be used by multiple partitions
- Shuffle sits at a stage boundary
- Requires coordination between tasks sets at the boundary of two stages
- Shuffle is costly
  - Requires a lot of disk I/O
  - Requires a lot of Network communication
  - Requires a barrier synchronization



Shuffle write in Worker Node (2 cores, 4 ShuffleMapTasks, 3 reducers, consolidateFiles = true)
Core 1
ShuffleMapTask
next ShuffleMapTask
Core 2
ShuffleMapTask
next ShuffleMapTask
Buffers in Writer
bucket 1
bucket 2
bucket 3
Buffers in Writer
bucket 1
bucket 2
bucket 3
ShuffleFileGroup
ShuffleFile 1
Shuffle Block 1
Shuffle Block 1'
File 2
Shuffle Block 2
Shuffle Block 2'
File 3
Shuffle Block 3
Shuffle Block 3'
ShuffleFileGroup
ShuffleFile 1
Shuffle Block 1
Shuffle Block 1'
File 2
Shuffle Block 2
Shuffle Block 2'
File 3
Shuffle Block 3
Shuffle Block 3'
Shuffle read
softBuffer
FileSegments
<K, V> record
RDD chain
Reducer 1
softBuffer
RDD chain
Reducer 2
softBuffer
RDD chain
Reducer 3

Spark Tuning

Spark Tuning

- Improving Parallelism
  - Data parallel execution reduces overall time
- More partitions allows data parallel execution
  - Number of tasks map to number of partitions
  - Number of threads map to number of tasks
- Control the number of partitions,
  - numPart params for wide transforms

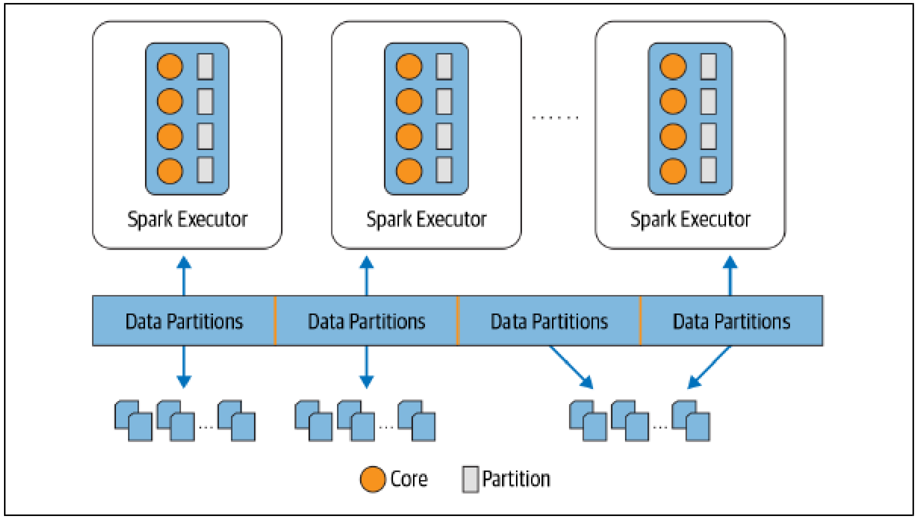


Figure 7-3. Relationship of Spark tasks, cores, partitions, and parallelism

Data Partitioning

- RDDs are spread on partitions across nodes of a cluster
- Controlling the number of partitions
  - Helps balance compute load
  - Helps reduce *shuffle* communication across nodes
- Partitioning function to map key to specific partitions
  - Set of “related” keys are placed in same partition
  - E.g., Hash partitioning, Range partitioning
- Can also be used for specific algorithms
  - Operate on partition at a time

Tracking Partitioners

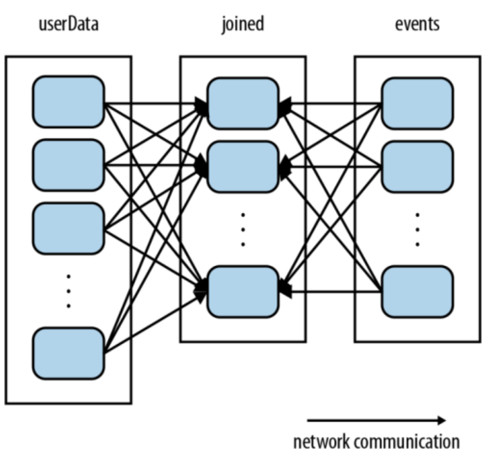
- Spark tracks partitioner used to generate an RDD
  - Uses this to optimize operations
- A partitioner option may or may not be present
  - *cogroup*, *joins*, *group/reduce/combineByKey*, *partitionBy*, *sort* will set a partitioner
  - Get the partitioner used to generate an RDD using **partitioner()** method
- Use partitioner to optimize future operations
  - E.g., Join output has been hash partitioned. So joined RDD tagged with that partitioner. Helps future **reduceByKey**.
- Many transformations can unset the partitioner
  - Map following a Join can unset Hash partitioner
  - Filter, mapValue & flatMapValues retain partitioner

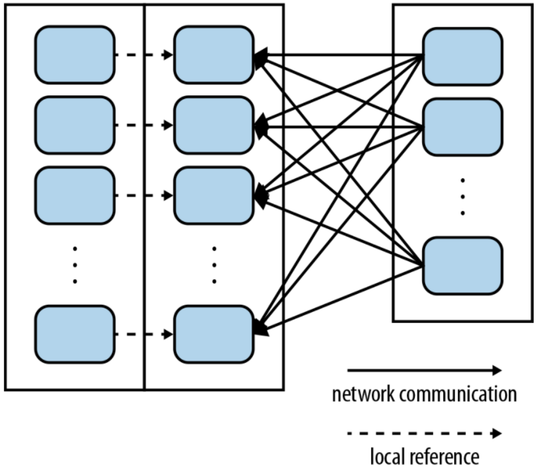
Impact of Partitioning: Limit Shuffle

- Join of big RDD with small RDD
  - **Default**: All to all shuffle without any partitioner
  - **HashPartition**: Selectively move small RDD to big RDD

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()
val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
val joined = userData.join(events)
```

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100))   // Create 100 partitions
                 .persist()
```





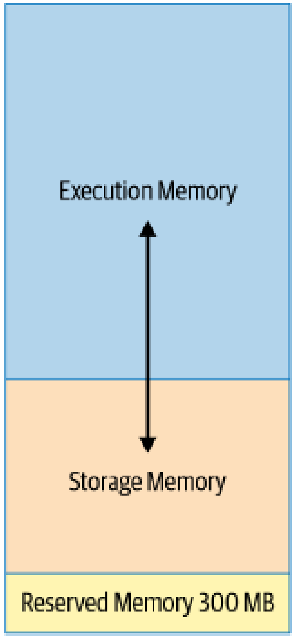
Passing Partitioners

- Can explicitly call **partitionBy** transformation
- Pass a number of partitions/partitioner as part of wide transformations
  - E.g., *join*, *groupByKey*, *sort*, etc.
- Default Partitioners
  - **HashPartitioner**: Uses key/value's hash to decide partition
  - **RangePartitioner**: Creates roughly equal ranges, determined by sampling the RDD contents
- Custom Partitioners
  - **numPartitions**: number of partitions to create
  - **getPartition**(key): Returns 0 to (numPartitions-1)

Tuning Spark

- Static vs. Dynamic resource allocation
  - Static fixes number of executors at submission time
    - Guarantees fixed resources before job starts
  - Dynamic allows resources to scale out and in by up to min/max executors
    - Based on task queue demand, idle time threshold

- Spark Memory Allocation
  - Execution memory used by shuffle, join
  - Storage memory used by user data, to cache blocks/partitions
  - Reserved memory is buffer to avoid out of memory exceptions



Spark Tuning

- Caching and Persistence
- Caching attempts to retain partitions in memory
  - Use when RDDs used across actions
  - Limited by available memory across executors
  - Recreated if OOM on executors
- Persist on disk allows moving to disk
  - “Unlimited” space, but slower than cache
- Cache/persist for iterative operations on RDDs, common RDDs used by several transformations
- Do no cache if too big for memory, or cheap to recompute

Additional Reading

- **Spark Internals**
  - Lijie Xu (Jerry Lead)
  - https://github.com/JerryLead/SparkInternals
- **The Internals of Apache Spark 3.1.1**
  - Jacek Laskowski
  - https://books.japila.pl/apache-spark-internals/overview/
- **Learning Spark**, Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia, O'Reilly
  - Chapter 7 (2nd Ed)