Apache Spark Internals

Pietro Michiardi

Eurecom

Acknowledgments & Sources

- **Sources**
  - Research papers:
    - ★ `https://spark.apache.org/research.html`
  - Presentations:
    - ★ M. Zaharia, “Introduction to Spark Internals”, `https://www.youtube.com/watch?v=49Hr5xZyTEA`
    - ★ A. Davidson, “A Deeper Understanding of Spark Internals”, `https://www.youtube.com/watch?v=dmL0N3qfSc8`
  - Blogs:
    - ★ Quang-Nhat Hoang-Xuan, Eurecom, `http://hxquangnhat.com/`
    - ★ Khoa Nguyen Trong, Eurecom, `https://trongkhoanguyenblog.wordpress.com/`

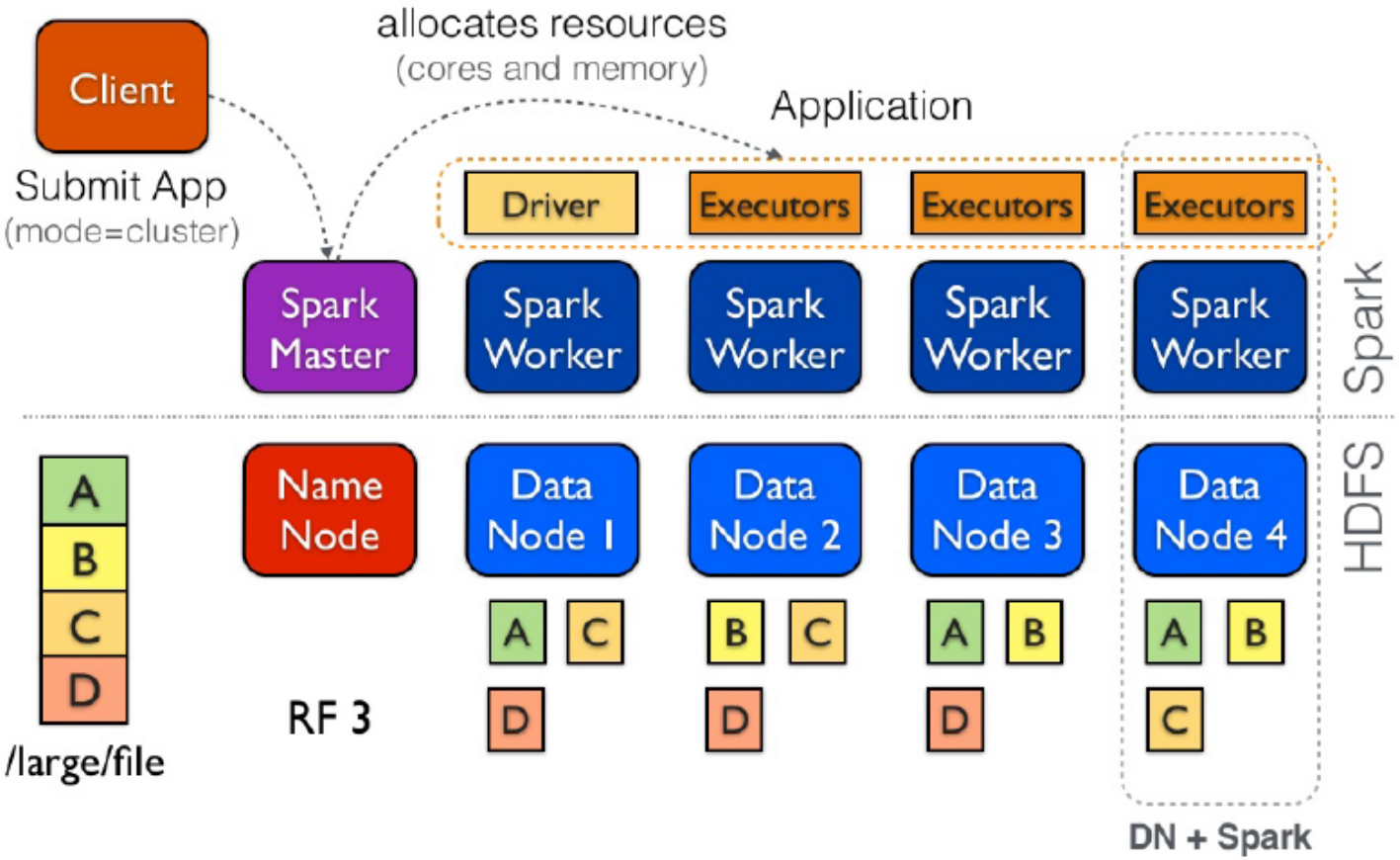
Anatomy of a Spark Application

A Very Simple Application Example

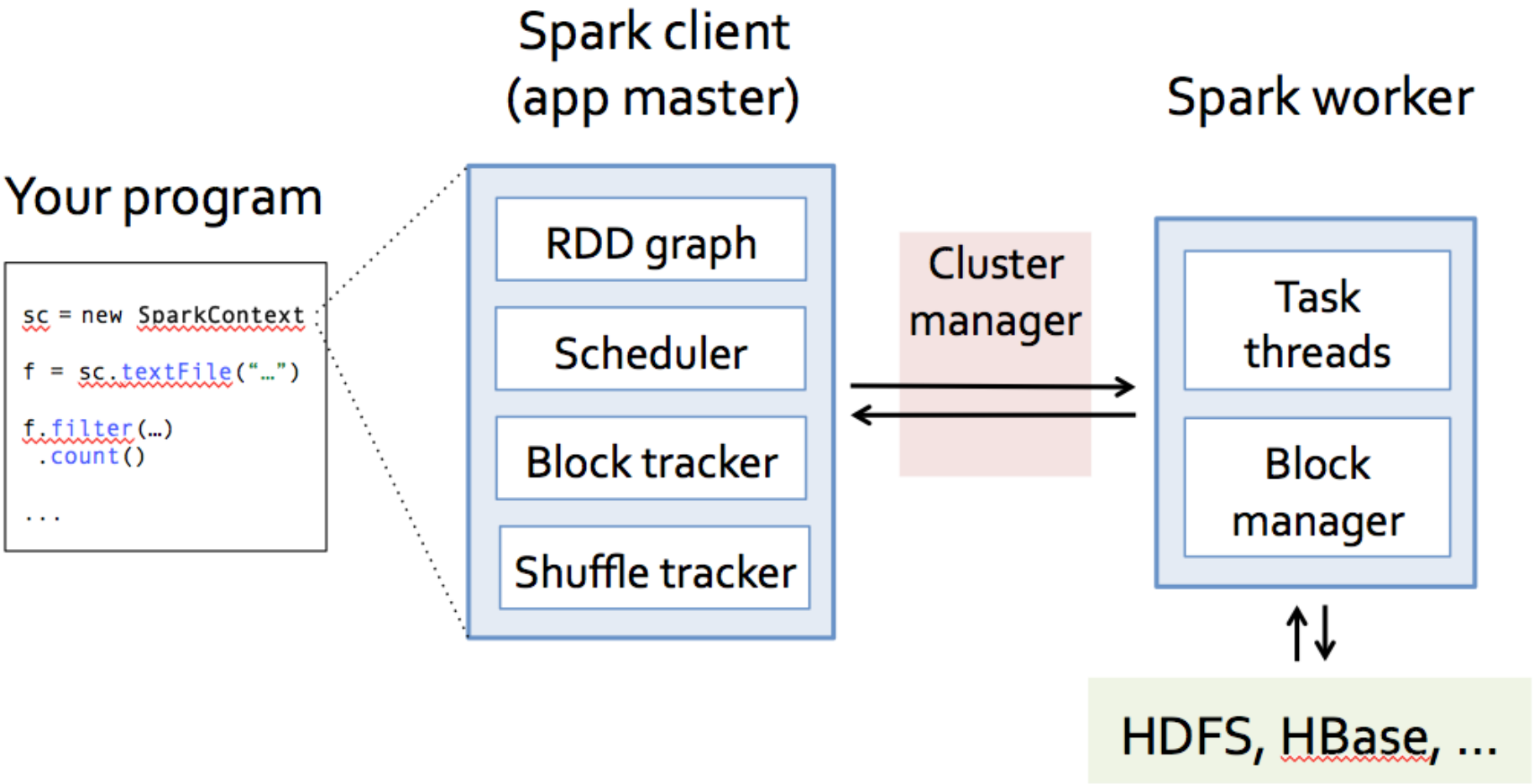
```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

Spark Applications: The Big Picture

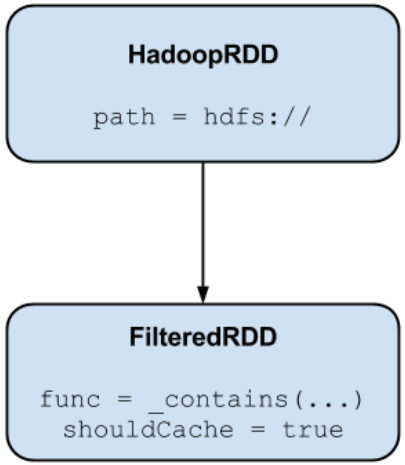
- There are two ways to manipulate data in Spark
  - Use the interactive shell, *i.e.,* the REPL
  - Write standalone applications, *i.e.,* driver programs

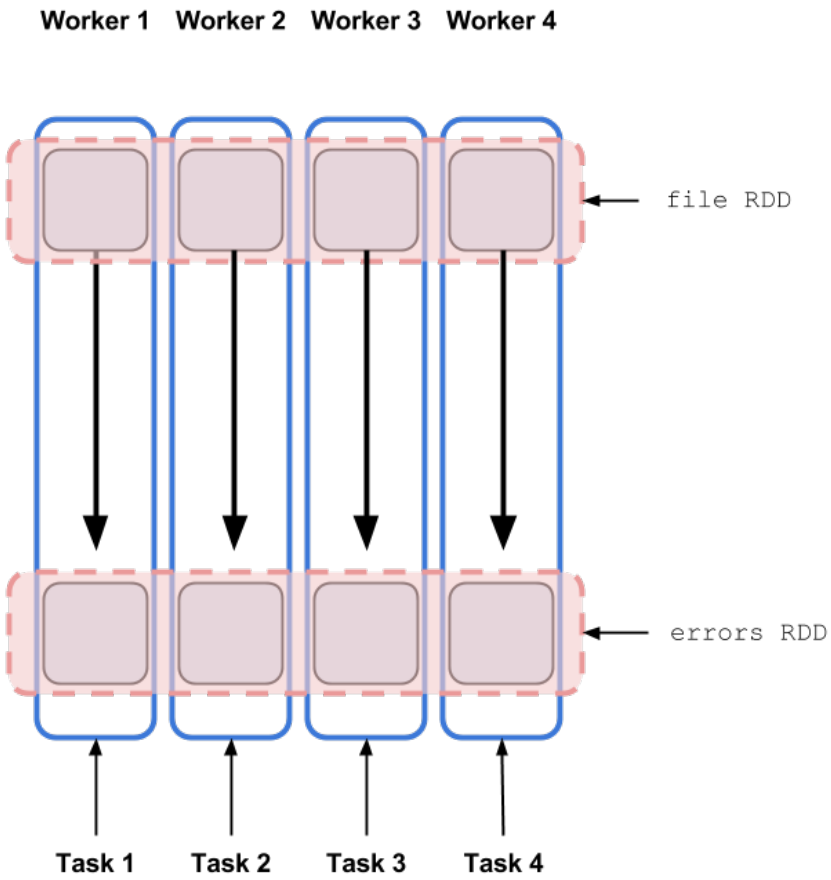


Spark Components: details



The RDD graph: dataset vs. partition views





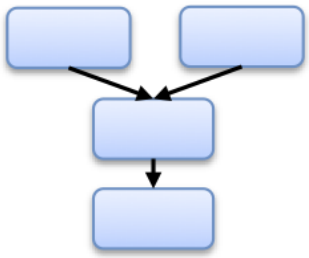
Data Locality

- **Data locality principle**
  - Same as for Hadoop MapReduce
  - Avoid network I/O, workers should manage local data

- **Data locality and caching**
  - First run: data not in cache, so use HadoopRDD's locality prefs (from HDFS)
  - Second run: FilteredRDD is in cache, so use its locations
  - If something falls out of cache, go back to HDFS

Lifetime of a Job in Spark

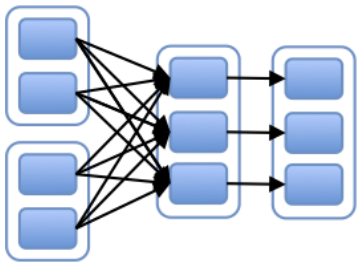
RDD Objects



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

Build the operator DAG

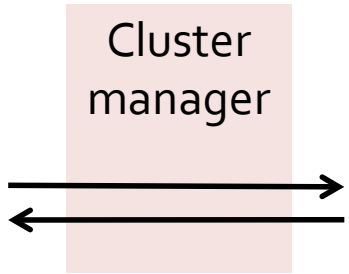
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

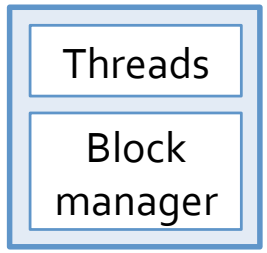
Task Scheduler



Launch tasks via Master

Retry failed and straggler tasks

Worker



Execute tasks

Store and serve blocks

Application model for scheduling

Application: Driver code that represents the DAG

Job: Subset of application triggered for execution by an “action” in the DAG

Stage: Job sub-divided into stages that have dependencies with each other

Task: Unit of work in a stage that is scheduled on a worker

http://spark.apache.org/docs/latest/cluster-overview.html

In Summary

- **Our example Application**: a **jar** file
  - Creates a `SparkContext`, which is the core component of the driver
  - Creates an input RDD, from a file in HDFS
  - Manipulates the input RDD by applying a `filter(f: T => Boolean)` transformation
  - Invokes the action `count()` on the transformed RDD
- **The DAG Scheduler**
  - Gets: RDDs, functions to run on each partition and a listener for results
  - Builds *Stages* of *Tasks* objects (code + preferred location)
  - Submits Tasks to the **Task Scheduler** as ready
  - Resubmits failed *Stages*
- **The Task Scheduler**
  - Launches *Tasks* on executors
  - Relaunches failed *Tasks*
  - Reports to the DAG Scheduler

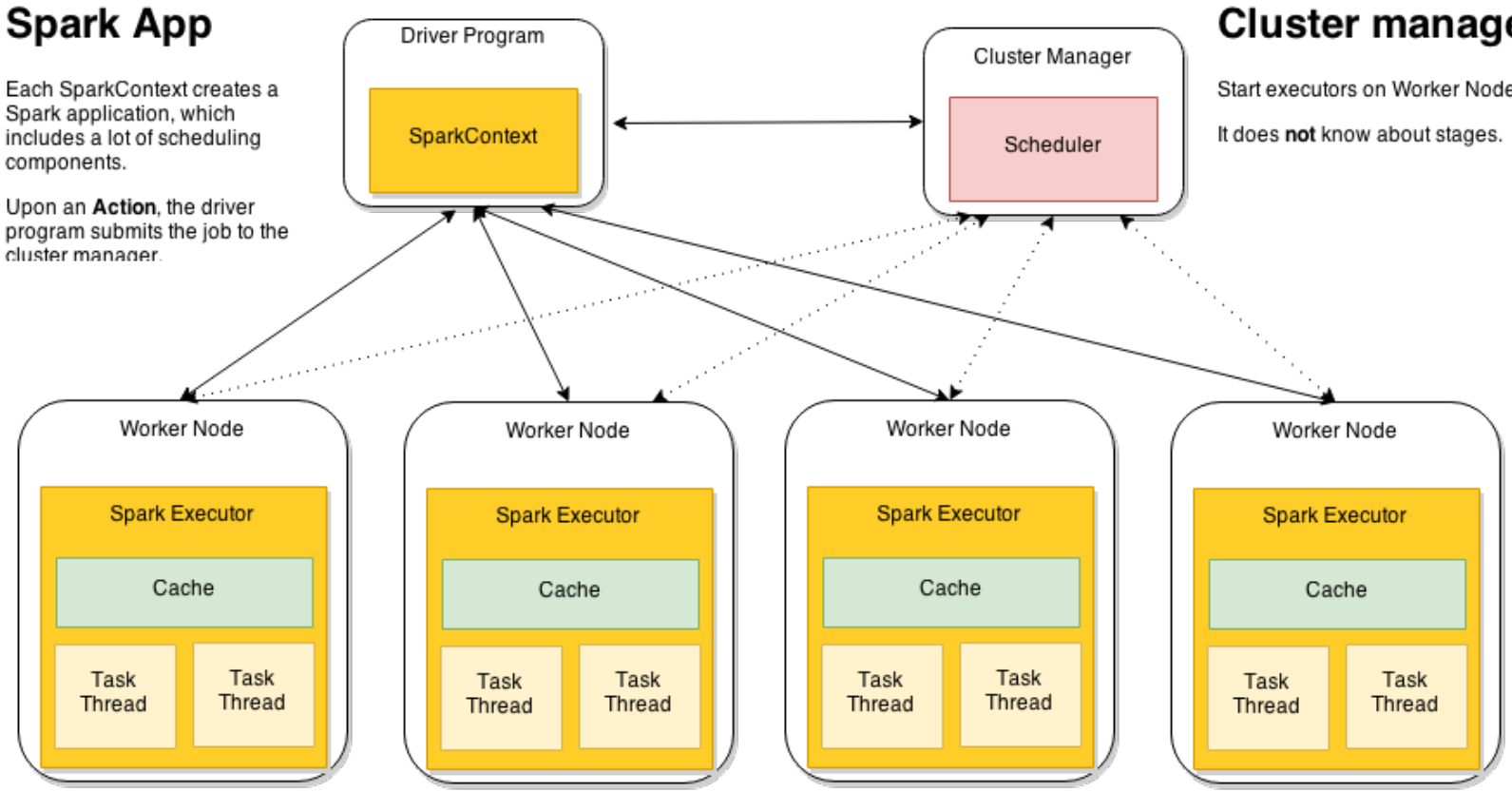
Spark Deployments

Spark Components: System-level View

Spark App

Each SparkContext creates a Spark application, which includes a lot of scheduling components.

Upon an **Action**, the driver program submits the job to the cluster manager.



Cluster manager

Start executors on Worker Nodes.

It does **not** know about stages.

Worker

Launch Spark Executor in a process.

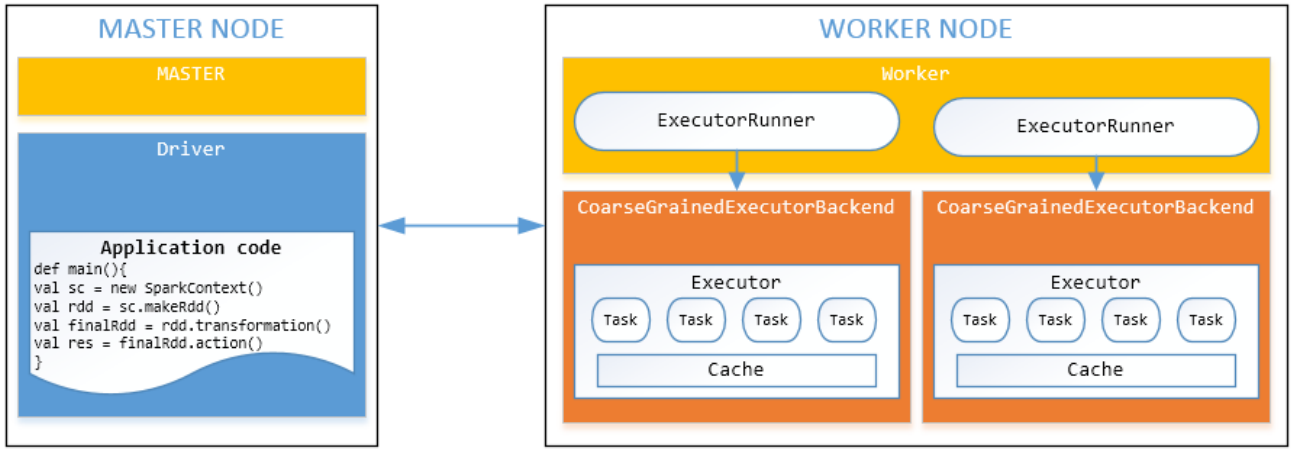
Tasks are launched in separate threads, one per each core on the worker node (can be configured)

Spark Deployment Modes

- **The Spark Framework can adopt several cluster managers**
  - *Local Mode*
  - *Standalone mode*
  - *Apache Mesos*
  - *Hadoop YARN*

- **General "workflow"**
  - Spark application creates `SparkContext`, which initializes the `DriverProgram`
  - Registers to the `ClusterManager`
  - Ask resources to allocate Executors
  - Schedule Task execution

Worker Nodes and Executors



- **Worker nodes are machines that run executors**
  - Host one or multiple `Workers`
  - One JVM (= 1 UNIX process) per `Worker`
  - Each `Worker` can spawn one or more `Executors`
- **Executors run tasks, used by 1 application, for whole lifetime**
  - Run in child JVM (= 1 UNIX process)
  - Execute one or more task using threads in a `ThreadPool`

Comparison to Hadoop MapReduce

Hadoop MapReduce

- One Task per UNIX process (JVM), more if JVM reuse
- `MultiThreadedMapper`, advanced feature to have threads in Map Tasks

→ **Short-lived** Executor, with one **large Task**

Spark

- Tasks run in one or more Threads, within a single UNIX process (JVM)
- Executor process statically allocated to worker, even with no threads

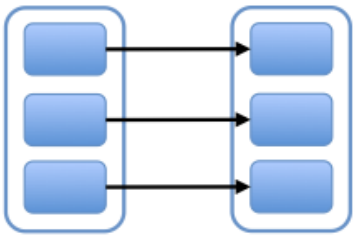
→ **Long-lived** Executor, with many **small Tasks**

Benefits of the Spark Architecture

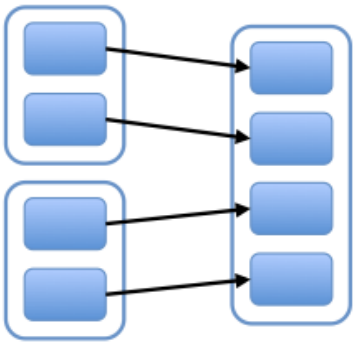
- **Isolation**
  - Applications are completely isolated
  - Task scheduling *per application*
- **Low-overhead**
  - Task setup cost is that of spawning a thread, not a process
  - 10-100 times faster
  - Small tasks $\rightarrow$ mitigate effects of data skew
- **Sharing data**
  - Applications cannot share data in memory natively
  - Use an external storage service like Tachyon
- **Resource allocation**
  - Static process provisioning for executors, even without active tasks
  - Dynamic provisioning under development

RDD Partition Dependency Types

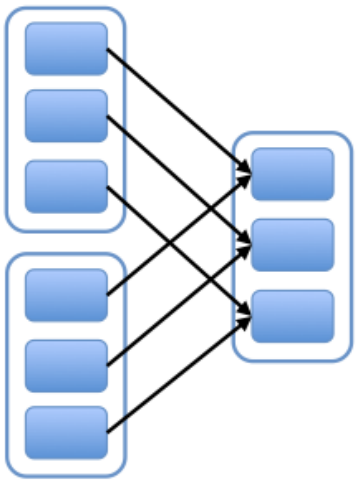
Narrow dependencies



map, filter

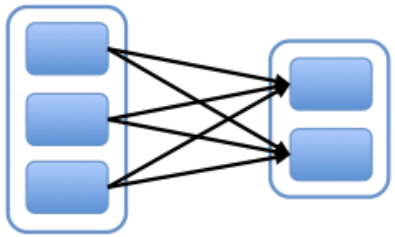


union

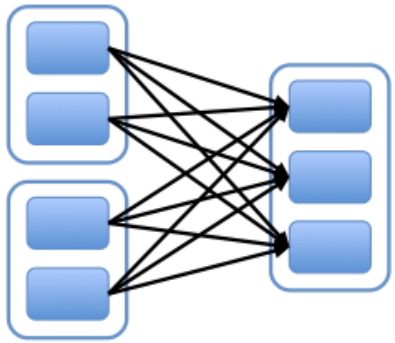


join with co-partitioned inputs

Wide dependencies



groupByKey



join with inputs not co-partitioned

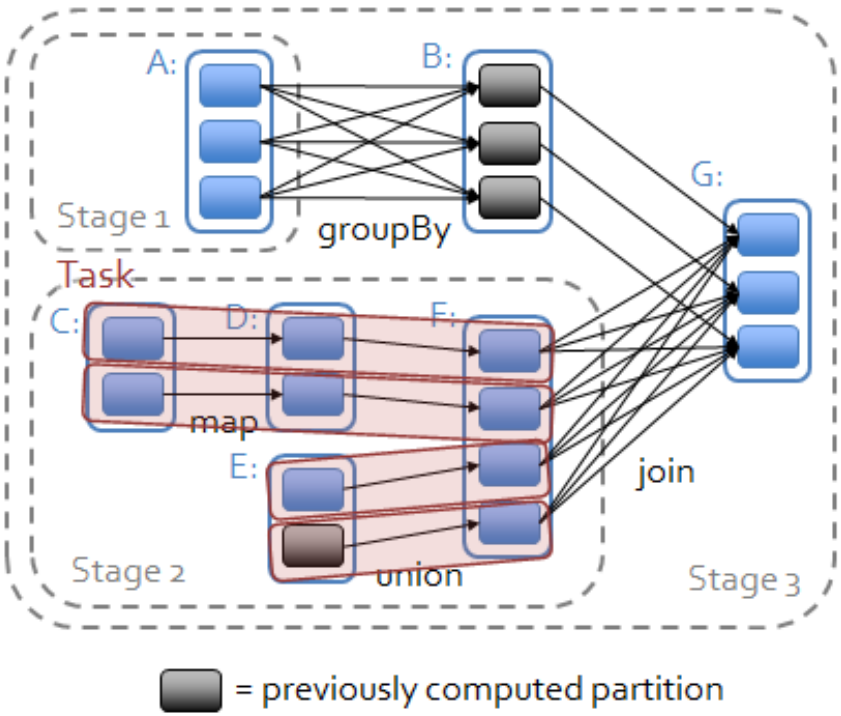
Dependency Types (2)

- **Narrow dependencies**
  - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Task can be executed locally and we don't have to shuffle. (Eg: `map`, `flatMap`, `filter`, `sample`)

- **Wide Dependencies**
  - Multiple child partitions may depend on one partition of the parent RDD
  - This means we have to shuffle data **unless the parents are hash-partitioned** (Eg: `sortByKey`, `reduceByKey`, `groupByKey`, `cogroupByKey`, `join`, `cartesian`)

Dependency Types: Optimizations

- **Benefits of Lazy evaluation**
  - The DAG Scheduler optimizes *Stages* and *Tasks* before submitting them to the Task Scheduler
  - **Piplining** narrow dependencies within a Stage
  - **Join plan selection** based on partitioning
  - **Cache reuse**



Detailed Example: Word Count

Spark Word Count: the driver

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

val sc = new SparkContext("spark://...", "MyJob", "spark
    home", "additional jars")
```

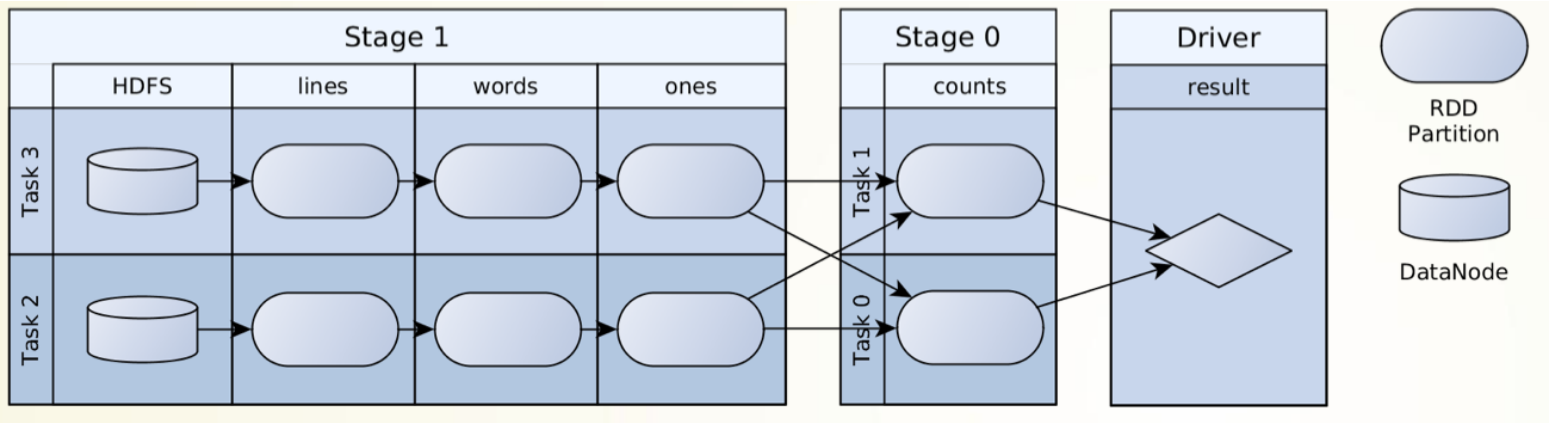
- **Driver and `SparkContext`**
  - A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
  - Then, the driver takes full control of the Spark job

Spark Word Count: the code

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

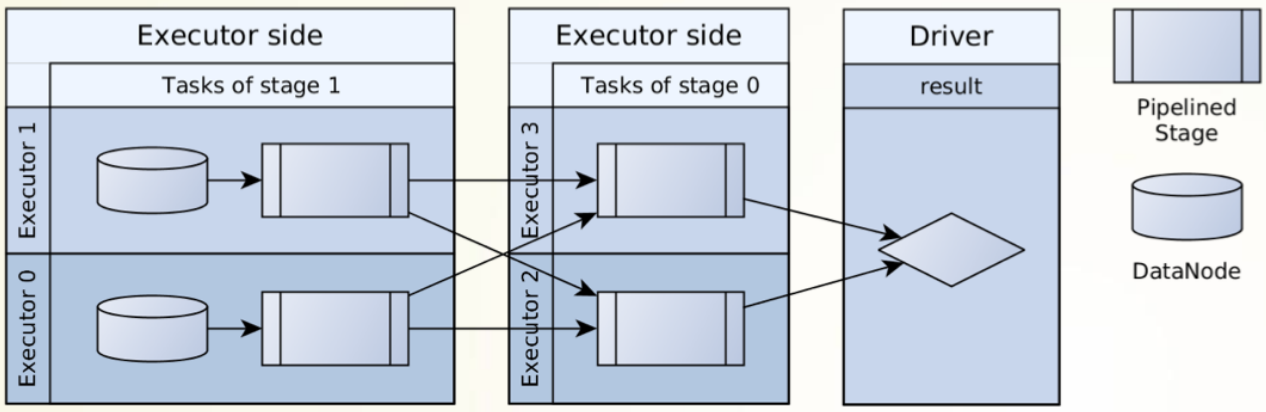
- **RDD lineage DAG is built on driver side with**
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations
- **Job submission**
  - An *action* triggers the DAG scheduler to submit a job

Spark Word Count: the DAG



- **Directed Acyclic Graph**
  - Built from the RDD lineage
- **DAG scheduler**
  - Transforms the DAG into stages and turns each partition of a stage into a single task
  - Decides what to run

Spark Word Count: the execution plan



- **Spark Tasks**
  - Serialized RDD lineage DAG + closures of transformations
  - Run by Spark executors

- **Task scheduling**
  - The driver side task scheduler launches tasks on executors according to resource and locality constraints
  - The task scheduler decides where to run tasks

Spark Word Count: the Shuffle phase

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- **reduceByKey transformation**
  - Induces the shuffle phase
  - In particular, we have a *wide dependency*
  - Like in Hadoop MapReduce, intermediate <key,value> pairs are stored on the local file system

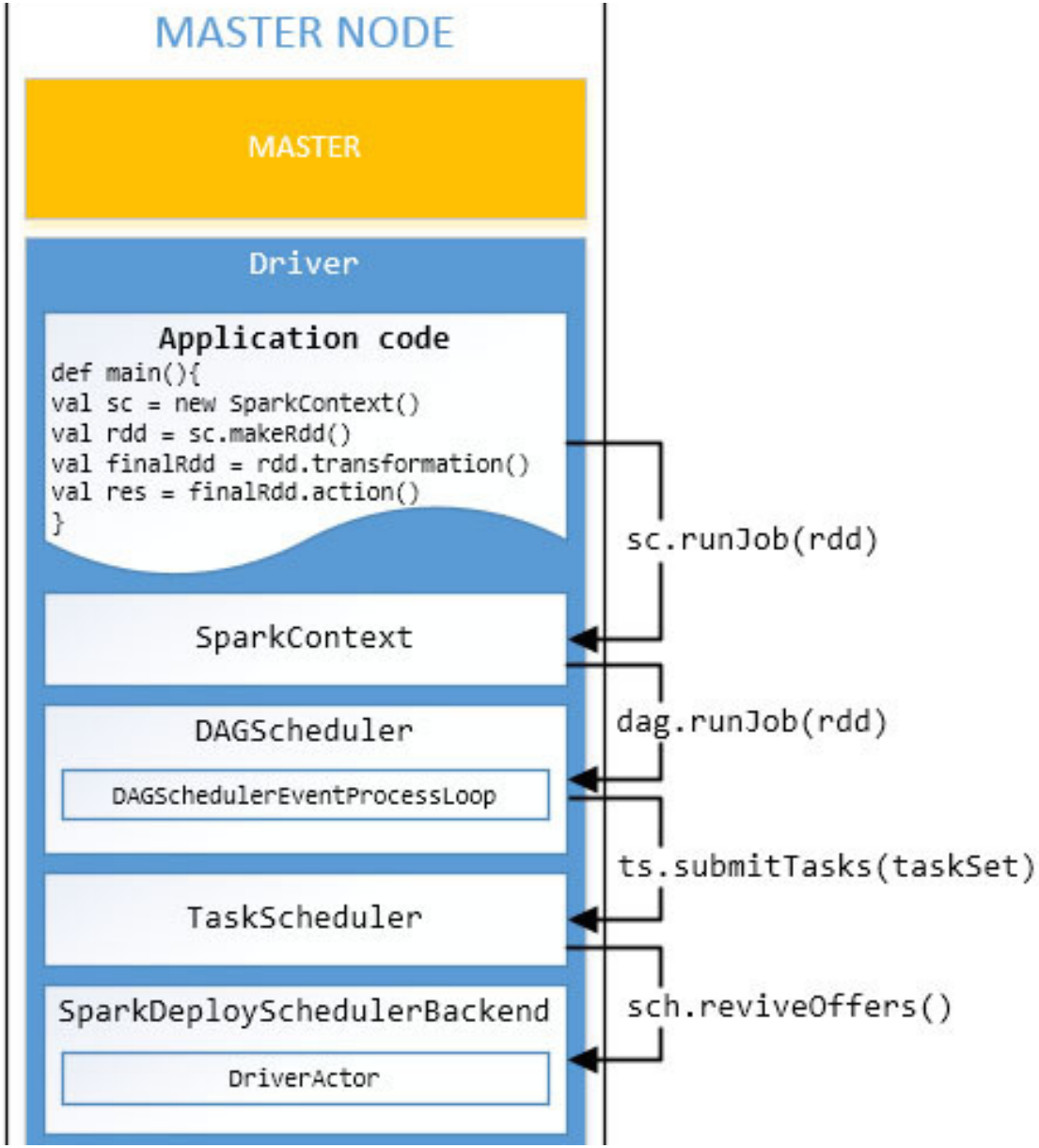
- **Automatic combiners!**
  - The `reduceByKey` transformation implements map-side combiners to pre-aggregate data

Spark Schedulers

- **Two main scheduler components, executed by the driver**
  - The DAG scheduler
  - The Task scheduler

- **Objectives**
  - Gain a broad understanding of how Spark submits Applications
  - Understand how *Stages* and *Tasks* are built, and their optimization
  - Understand interaction among various other Spark components

Submitting a Spark Application: A Walk Through



The DAG Scheduler

- **Stage-oriented scheduling**
  - Computes a DAG of stages for each job in the application
  - Keeps track of which RDD and stage output are materialized
  - Determines an optimal schedule, minimizing stages
  - Submit stages as sets of Tasks (`TaskSets`) to the Task scheduler
- **Data locality principle**
  - Uses “preferred location” information (optionally) attached to each RDD
  - Package this information into Tasks and send it to the Task scheduler
- **Manages Stage failures**
  - Failure type: (intermediate) data loss of shuffle output files
  - Failed stages will be resubmitted
  - NOTE: Task failures are handled by the Task scheduler, which simply resubmit them if they can be computed with no dependency on previous output

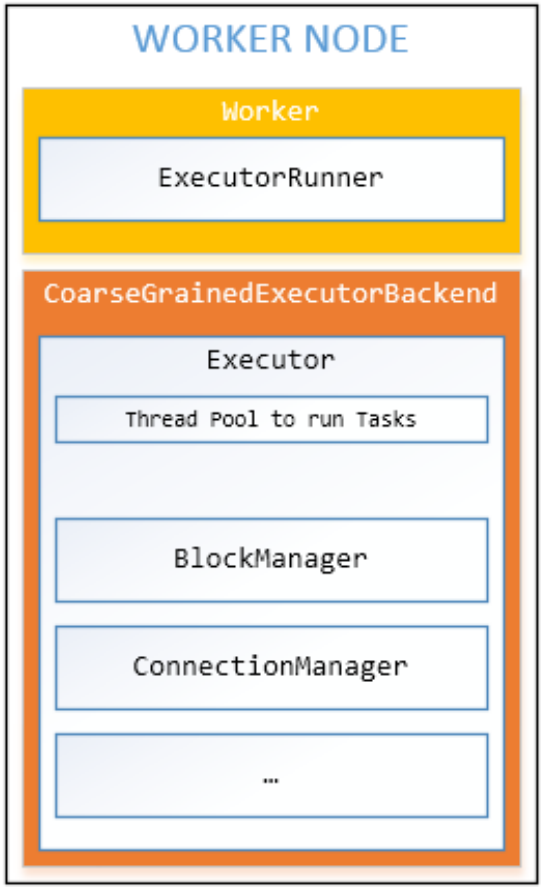
More About Stages

- **What is a DAG**
  - Directed acyclic graph of stages
  - Stage boundaries determined by the shuffle phase
  - Stages are run in *topological order*
- **Definition of a Stage**
  - Set of *independent* tasks
  - All tasks of a stage apply the same function
  - All tasks of a stage have the same dependency type
  - All tasks in a stage belong to a `TaskSet`
- **Stage types**
  - Shuffle Map Stage: stage tasks results are inputs for another stage
  - Result Stage: tasks compute the final action that initiated a job (e.g., `count()`, `save()`, etc.)

The Task Scheduler

- **Task oriented scheduling**
  - Schedules tasks for a *single* `SparkContext`
  - Submits tasks sets produced by the DAG Scheduler
  - Retries failed tasks
  - Takes care of *stragglers* with speculative execution
  - Produces events for the DAG Scheduler
- **Implementation details**
  - The Task scheduler creates a `TaskSetManager` to wrap the `TaskSet` from the DAG scheduler
  - The `TaskSetManager` class operates as follows:
    - ★ Keeps track of each task status
    - ★ Retries failed tasks
    - ★ Imposes data locality using *delayed scheduling*
  - Message passing implemented using *Actors*, and precisely using the *Akka framework*

Running Tasks on Executors



Executor.scala

```
1 def launchTask(serializedTask){
2       val tr = new TaskRunner(serializedTask)
3       threadPool.execute(tr)
4 }
```

TaskRunner.scala

```
5 def run(){
6       executorBackend.statusUpdate(RUNNING)
7       val task = serializer.deserialize(serializedTask)
8       val value = task.run()
9       val res = new DirectTaskResult(serializer.serialize(value))
10      executorBackend.statusUpdate(FINISHED)
11      if res.size > akkaFrameSize
12              blockManager.putBytes(taskId, res)
13      else
14              return res
15 }
```

```
16 task.run()
17 if task is ResultTask
18      val (rdd, func) = ser.deserialize(RDD, taskContext)
19      return result = func(rdd.iterator(partition))
20
21 if task is ShuffleMapTask
22      val (rdd, dep) = ser.deserialize(RDD, ShuffleDependency, taskContext)
23      shuffleWriter = shuffleManager.getWriter(dep.shuffleHandler, partitionId)
24      shuffleWriter.write(rdd.iterator(partition))
25      return shuffleWriter.stop().get()
```

ShuffleRDD.scala / CoGroupedRDD.scala

```
26 def compute(split, taskContext){
27      shuffleManager.getReader(dep.shuffleHandler, split.index, taskContext).read()
28 }
```

Running Tasks on Executors

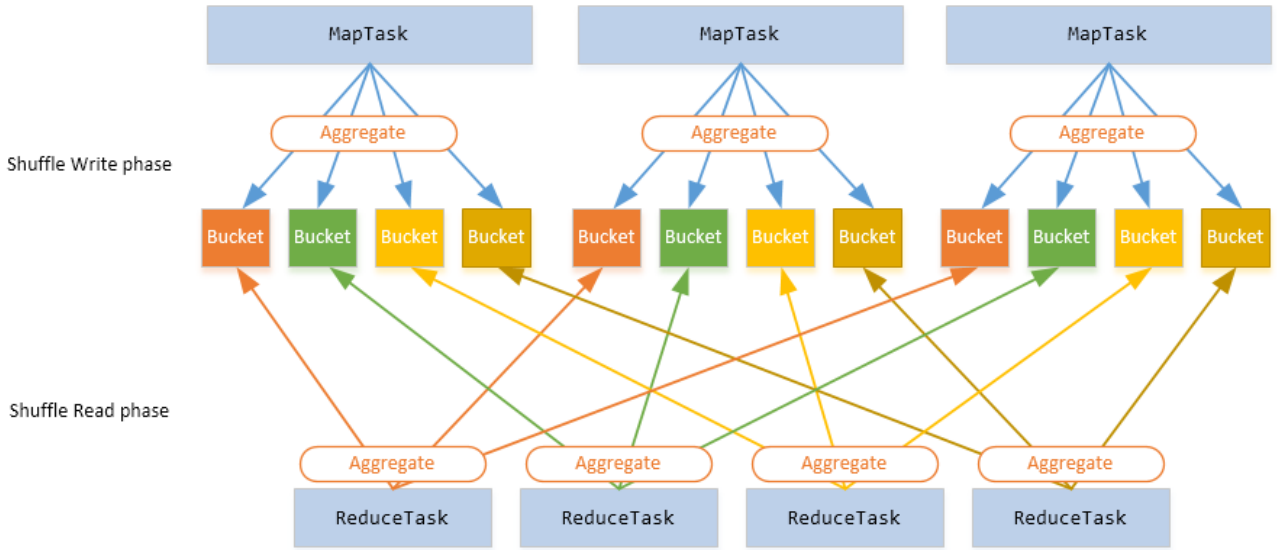
- **Executors run two kinds of tasks**
  - `ResultTask`: apply the action on the RDD, once it has been computed, alongside all its dependencies
    `Line 19`
  - `ShuffleTask`: use the Block Manager to store shuffle output using the `ShuffleWriter`
    `Lines 23,24`
  - The `ShuffleRead` component depends on the type of the RDD, which is determined by the compute function and the transformation applied to it

Data Shuffling

The Spark Shuffle Mechanism

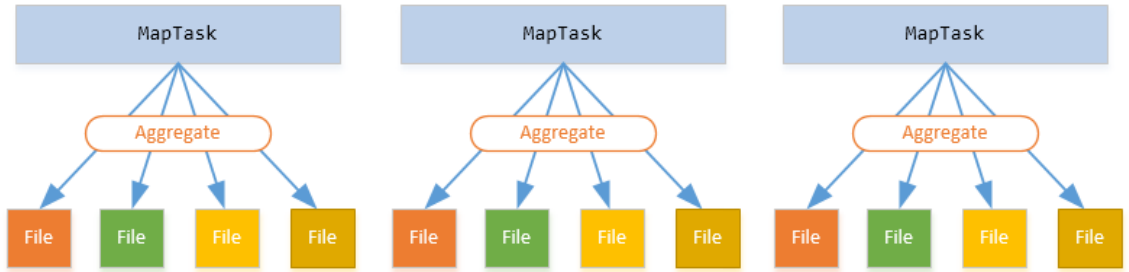
- **Same concept as for Hadoop MapReduce, involving:**
  - Storage of “intermediate” results on the local file-system
  - Partitioning of “intermediate” data
  - Serialization / De-serialization
  - Pulling data over the network
- **Transformations requiring a shuffle phase**
  - `groupByKey()`, `reduceByKey()`, `sortByKey()`, `distinct()`
- **Various types of Shuffle**
  - *Hash Shuffle*
  - *Consolidate Hash Shuffle*
  - *Sort-based Shuffle*

The Spark Shuffle Mechanism: an Illustration



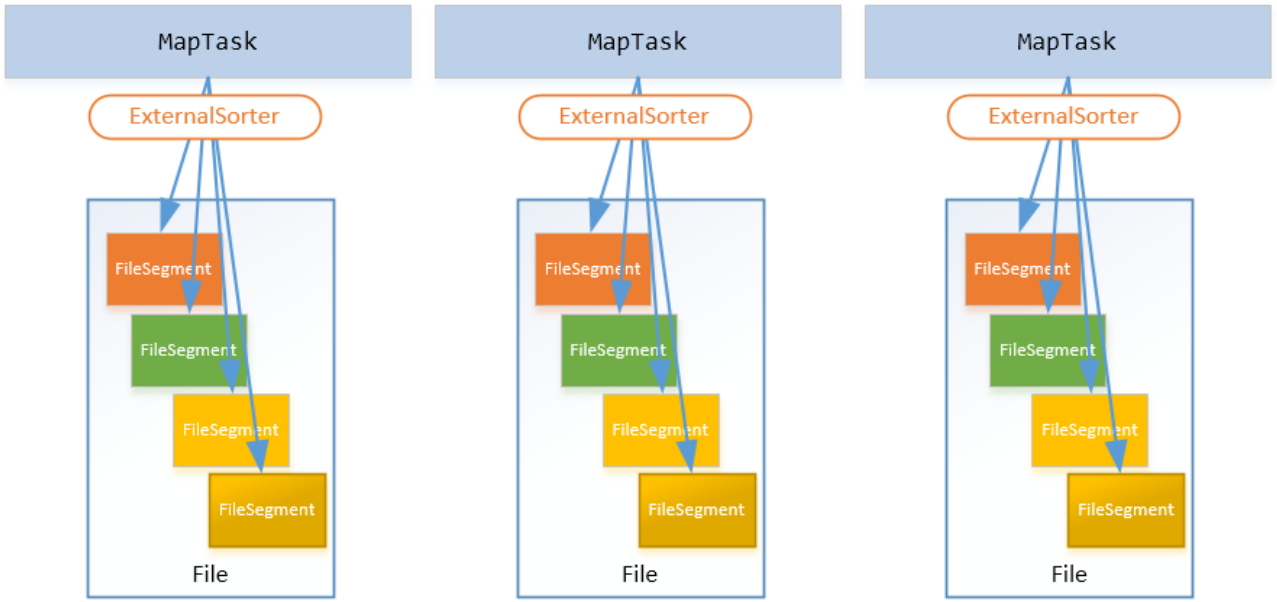
- **Data Aggregation**
  - Defined on `ShuffleMapTask`
  - Two methods available:
    - `AppendOnlyMap`: in-memory hash table combiner
    - `ExternalAppendOnlyMap`: memory + disk hash table combiner
- **Batching disk writes to increase throughput**

The Hash Shuffle Mechanism

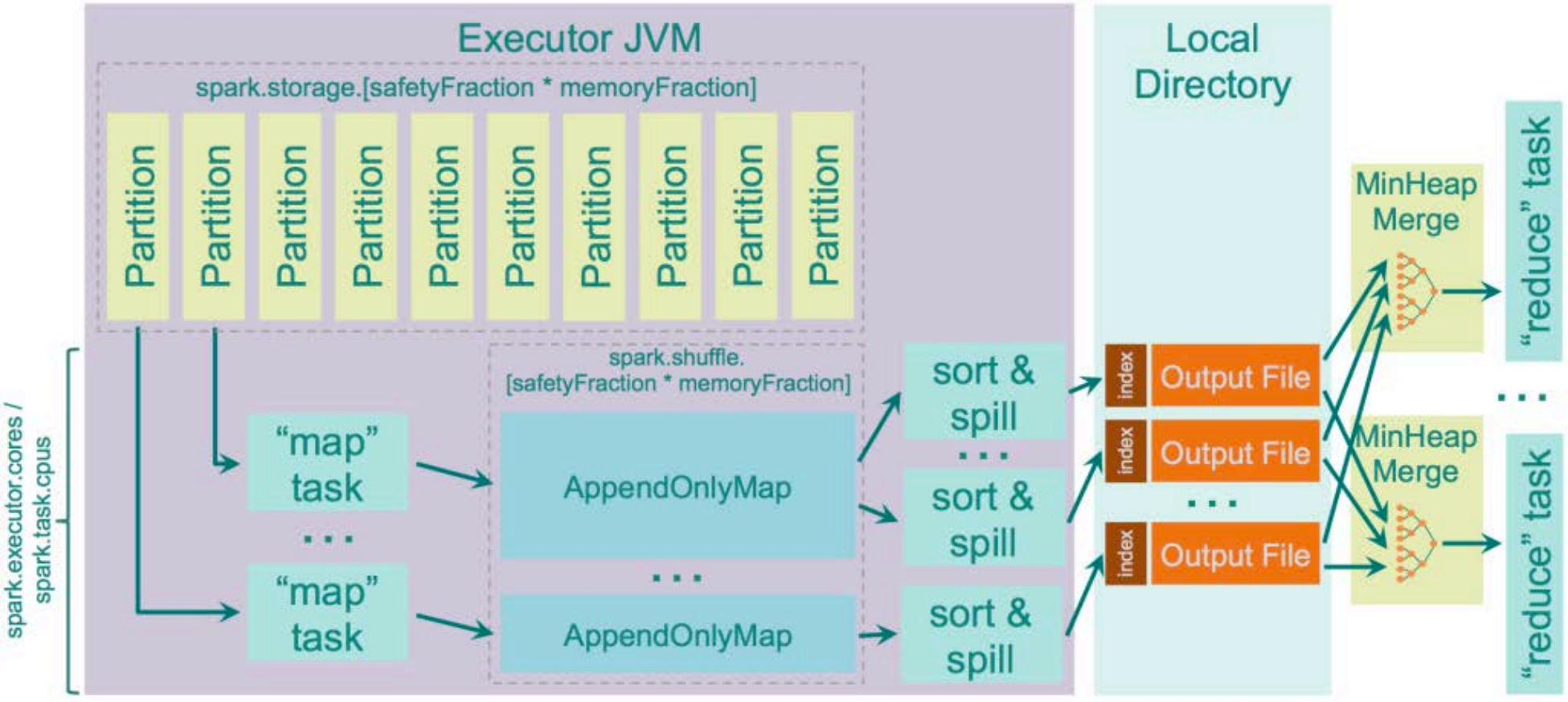


- **Map Tasks write output to multiple files**
  - Assume: m map tasks and r reduce tasks
  - Then: $m \times r$ shuffle files as well as in-memory buffers (for batching writes)
- **Be careful on storage space requirements!**
  - Buffer size must not be too big with many tasks
  - Buffer size must not be too small, for otherwise throughput decreases

The Sort-based Shuffle Mechanism



- **Implements the Hadoop Shuffle mechanism**
  - Single shuffle file, plus an index file to find “buckets”
  - Very beneficial for write throughput, as more disk writes can be batched
- **Sorting mechanism**
  - Pluggable external sorter
  - Degenerates to Hash Shuffle if no sorting is required



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