Spark 内核

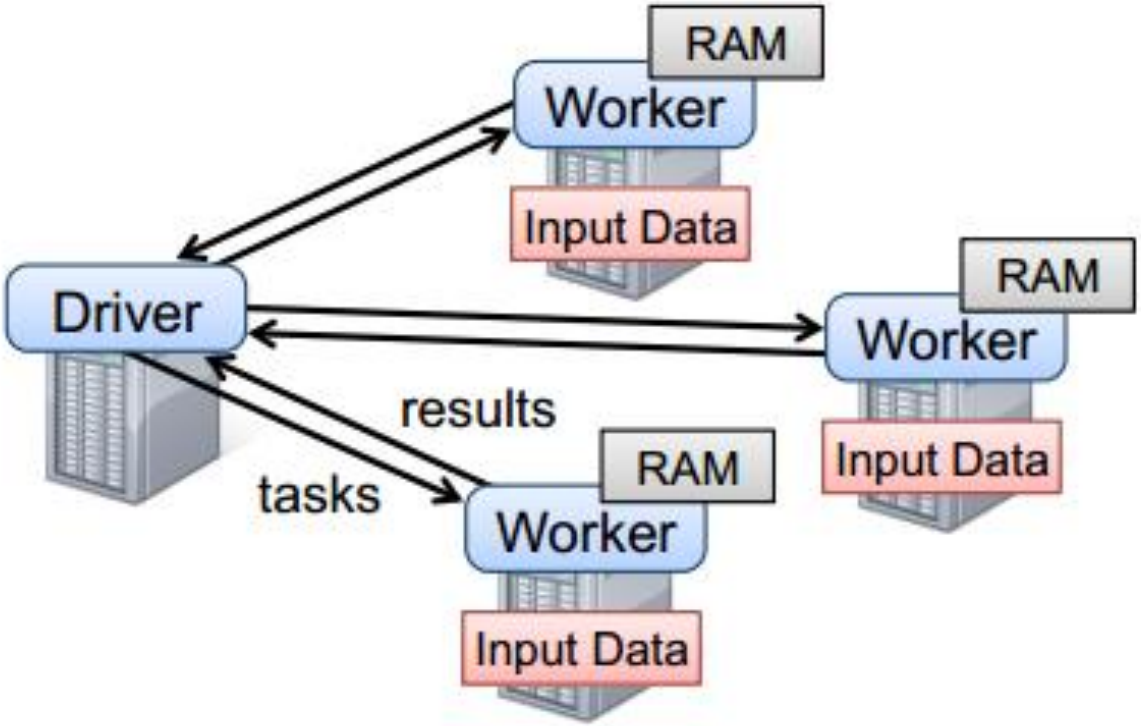
讲师：陈博

RDD是基础

- Resilient Distributed Dataset
- 弹性分布式数据集

- 五大特性：
- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Partitioner for key-value RDDs
- Optionally, a list of preferred locations to compute each split on

Spark运行时


RAM
Worker
Input Data
Driver
RAM
Worker
Input Data
results
tasks
RAM
Worker
Input Data

流程示意

- 分布式文件系统（File system）--加载数据集
- transformations延迟执行--针对RDD的操作
- Action触发执行

代码示例

- lines = sc.textFile(“hdfs://...”)
- 加载进来成为RDD
- errors = lines.filter(_.startsWith(“ERROR”))
- Transformation转换
- errors.persist()
- 缓存RDD
- Mysql_errors = errors.filter(_.contain(“MySQL”)).count
- Action执行
- http_errors = errors.filter(_.contain(“Http”)).count
- Action执行

缓存策略

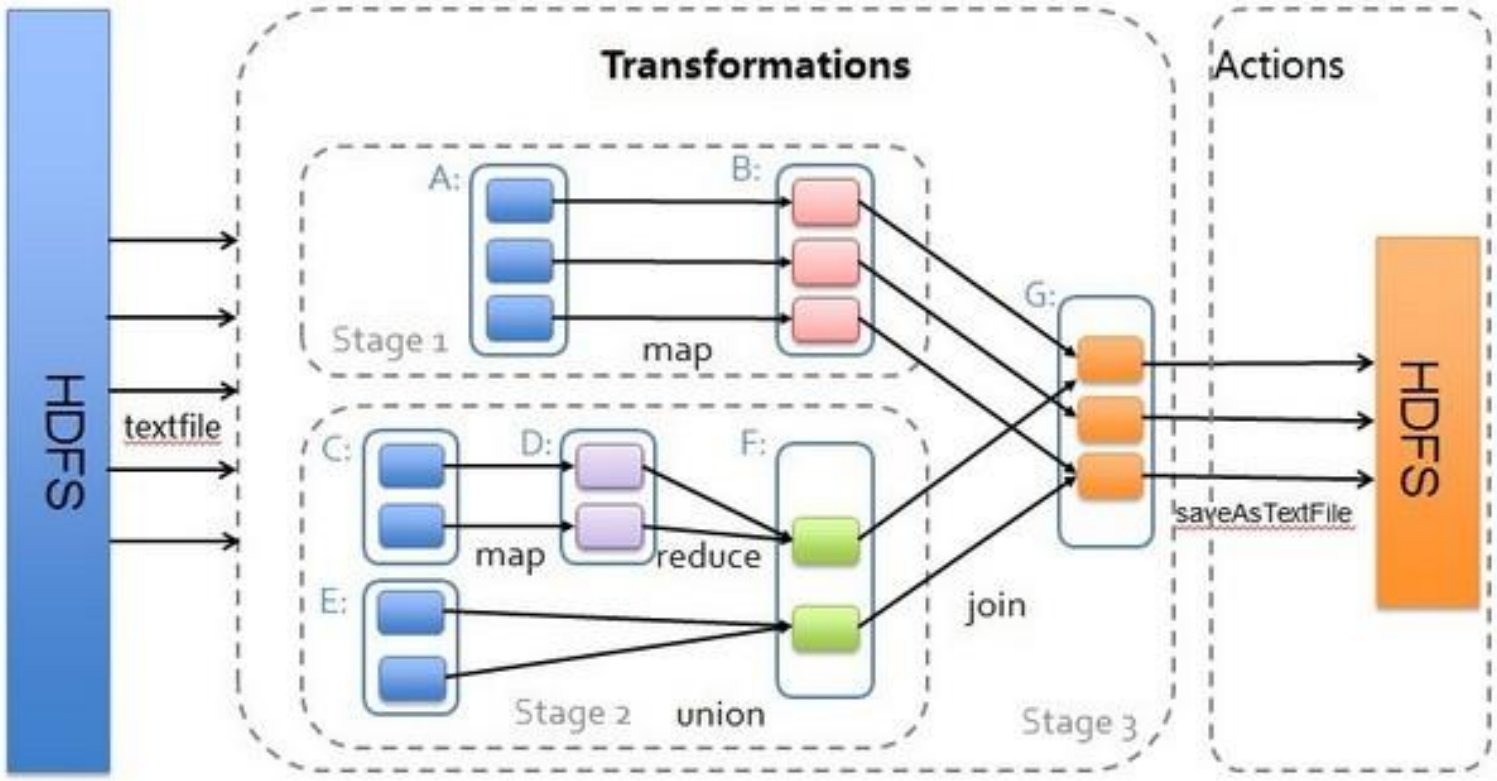

```
38  class StorageLevel private(
39      private var _useDisk: Boolean,
40      private var _useMemory: Boolean,
41      private var _useOffHeap: Boolean,
42      private var _deserialized: Boolean,
43      private var _replication: Int = 1)
```

```
val NONE = new StorageLevel(false, false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, false, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
val OFF_HEAP = new StorageLevel(false, false, true, false)
```

转换算子 操作算子

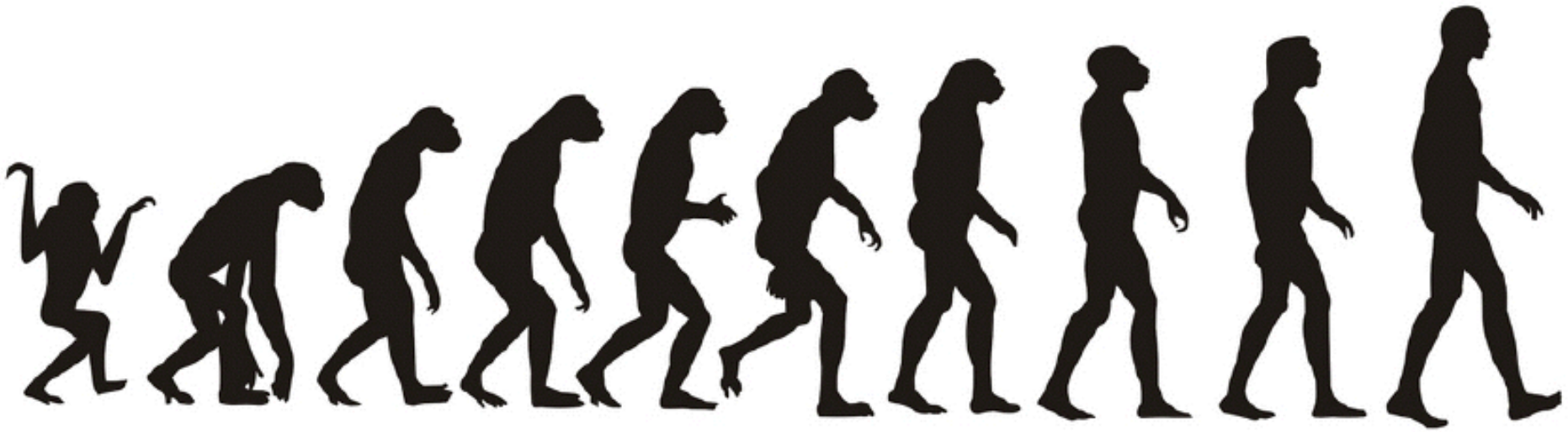
Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark:Transformations & Actions



血统

- Lineage
- 每个看做一个RDD



Rdd 容错

- Lineage（血统）
- 利用内存加快数据加载,在众多的其它的In-Memory类数据库或Cache类系统中也有实现，Spark的主要区别在于它处理分布式运算环境下的数据容错性（节点实效/数据丢失）问题时采用的方案。为了保证RDD中数据的鲁棒性，RDD数据集通过所谓的血统关系(Lineage)记住了它是如何从其它RDD中演变过来的。相比其它系统的细颗粒度的内存数据更新级别的备份或者LOG机制，RDD的Lineage记录的是粗颗粒度的特定数据转换（Transformation）操作（filter, map, join etc.)行为。当这个RDD的部分分区数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区。这种粗颗粒的数据模型，限制了Spark的运用场合，但同时相比细颗粒度的数据模型，也带来了性能的提升。
- RDD在Lineage依赖方面分为两种Narrow Dependencies与Wide Dependencies用来解决数据容错的高效性。Narrow Dependencies是指父RDD的每一个分区最多被一个子RDD的分区所用，表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区，也就是说一个父RDD的一个分区不可能对应一个子RDD的多个分区。Wide Dependencies是指子RDD的分区依赖于父RDD的多个分区或所有分区，也就是说存在一个父RDD的一个分区对应一个子RDD的多个分区。对与Wide Dependencies，这种计算的输入和输出在不同的节点上，lineage方法对与输入节点完好，而输出节点宕机时，通过重新计算，这种情况下，这种方法容错是有效的，否则无效，因为无法重试，需要向上其祖先追溯看是否可以重试（这就是lineage，血统的意思），Narrow Dependencies对于数据的重算开销要远小于Wide Dependencies的数据重算开销。
- 容错
- 在RDD计算，通过checkpoint进行容错，做checkpoint有两种方式，一个是checkpoint data，一个是logging the updates。用户可以控制采用哪种方式来实现容错，默认是logging the updates方式，通过记录跟踪所有生成RDD的转换（transformations）也就是记录每个RDD的lineage（血统）来重新计算生成丢失的分区数据。

容错

- val logs = sc.textFile(...).filter(_.contains(“spark”)).map(_.split(‘\t’)(1))

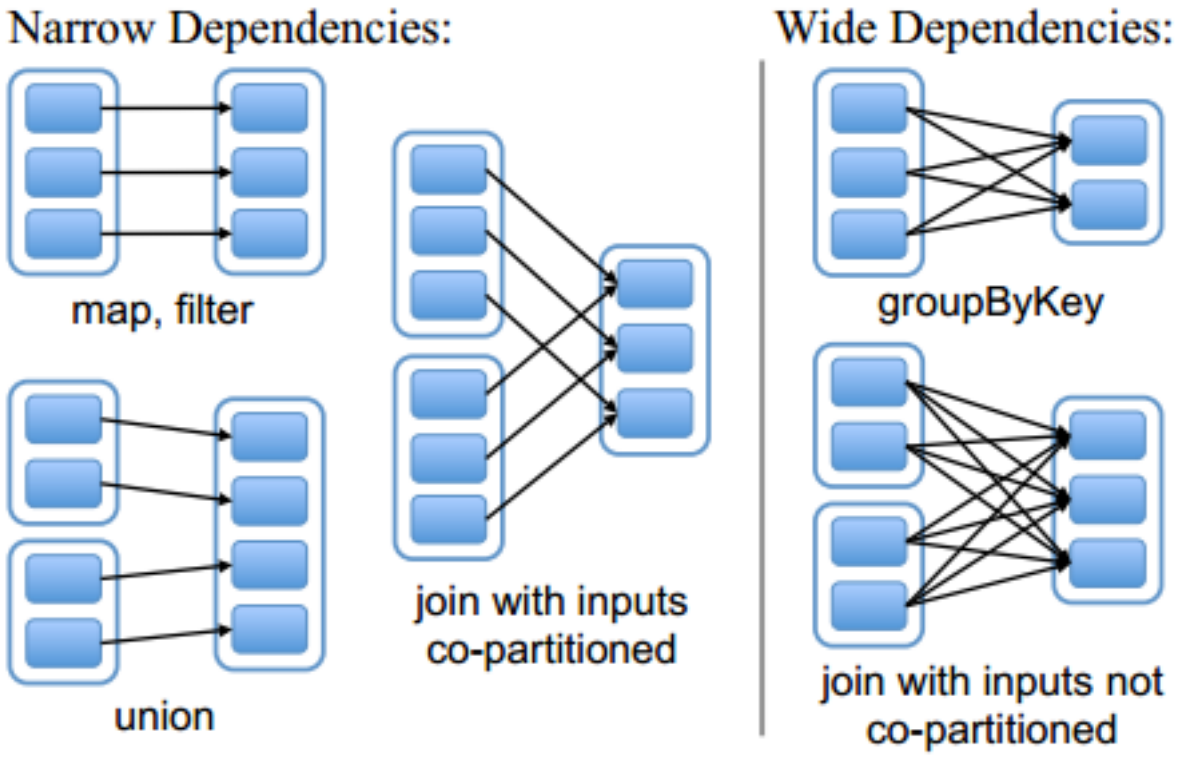
- 上面代码对应
- HadoopRDD　　sc.textFile(...)
- FilterRDD　　_.contains(...)
- MappedRDD　　_.split(...)

- 每个RDD都会记录自己依赖与哪个或哪些RDD，万一某个RDD的某些partition挂了，可以通过其它RDD并行计算迅速恢复出来

checkpoint

- Lineage过长
- 对rdd做doCheckpoint()
  - SparkContext.setCheckPointDir()
  - 设置数据存路径

窄依赖和宽依赖的例子。（方框表示RDD，实心矩形表示分区）



术语解释

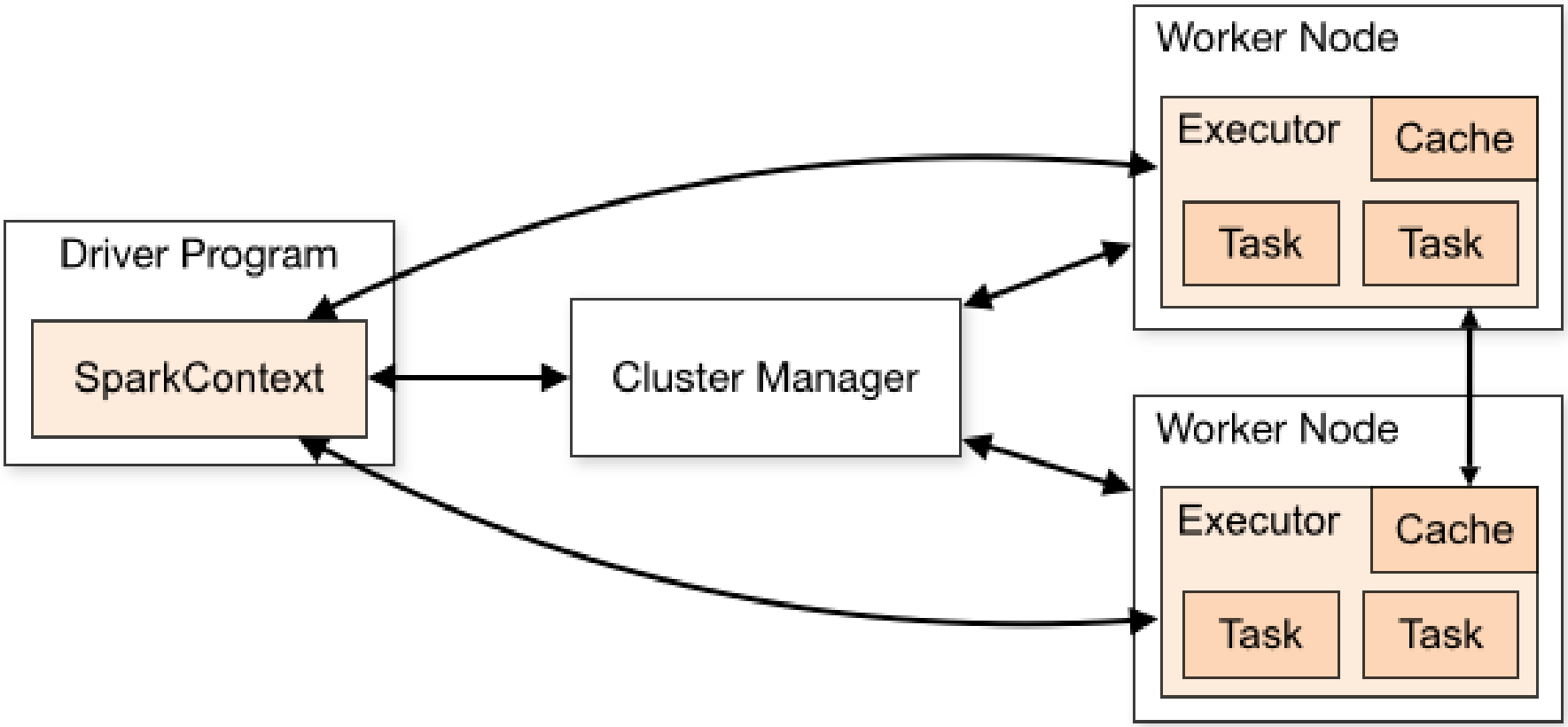
- Application 基于Spark的用用户程序，包含了driver程序和集群上的executor
- Driver Program 运行行main函数并且新建SparkContext的程序
- Cluster Manager 在集群上获取资源的外部服务(例如standalone,Mesos,Yarn)

- Worker Node 集群中任何可以运行行应用用代码的节点
- Executor是在一一个worker node上为某应用用启动的一一个进程，该进程负责运行行任务，并且负责将数据存在内存或者磁盘上。每个应用用都有各自自独立立的executors
- Task 被送到某个executor上的工工作单元

- Job 包含很多任务的并行行计算，可以看做和Spark的action对应

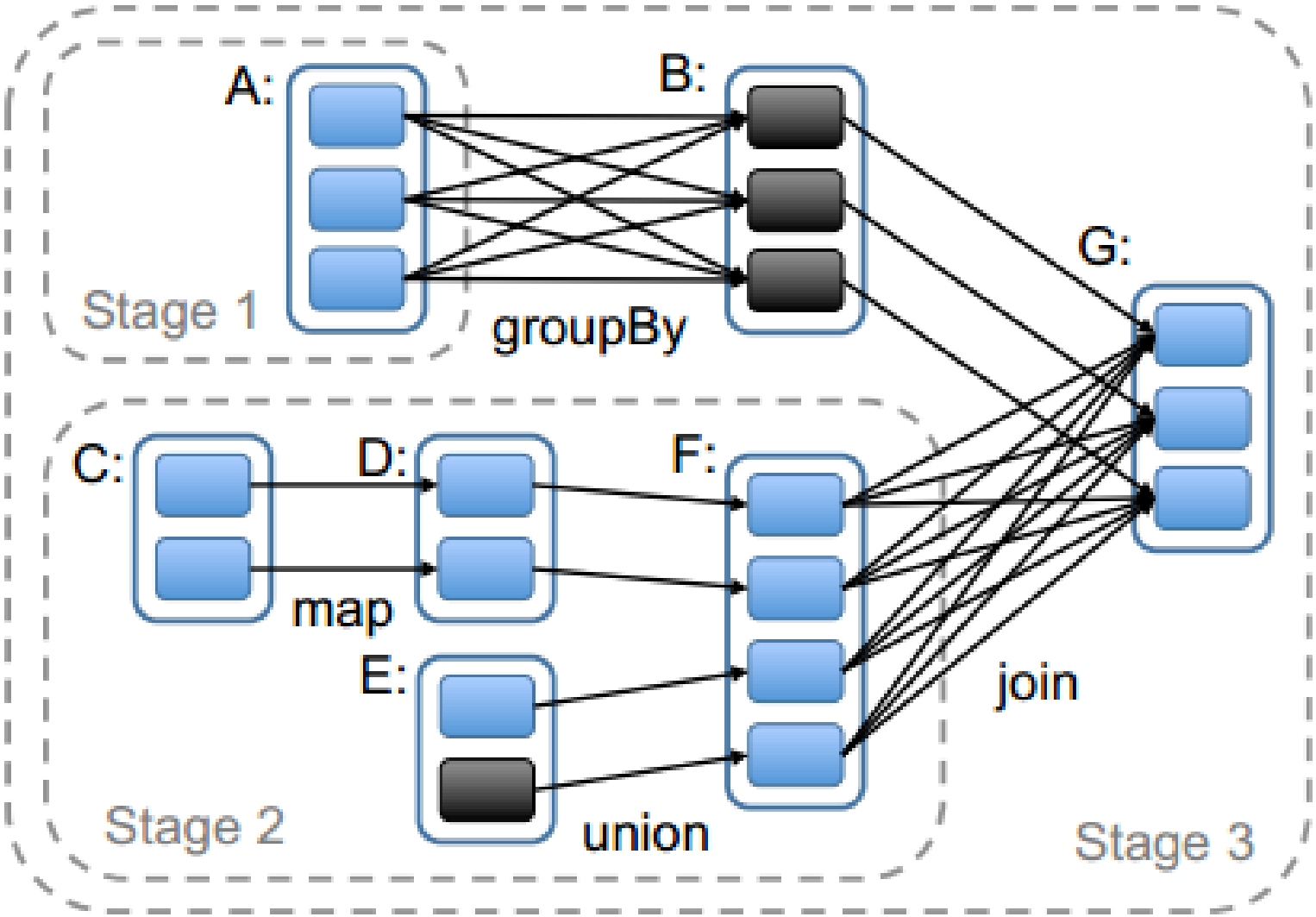
- Stage一一个Job会被拆分很多组任务，每组任务被称为Stage(就像Mapreduce分map任务和reduce任务一一样)

Cluster Overview



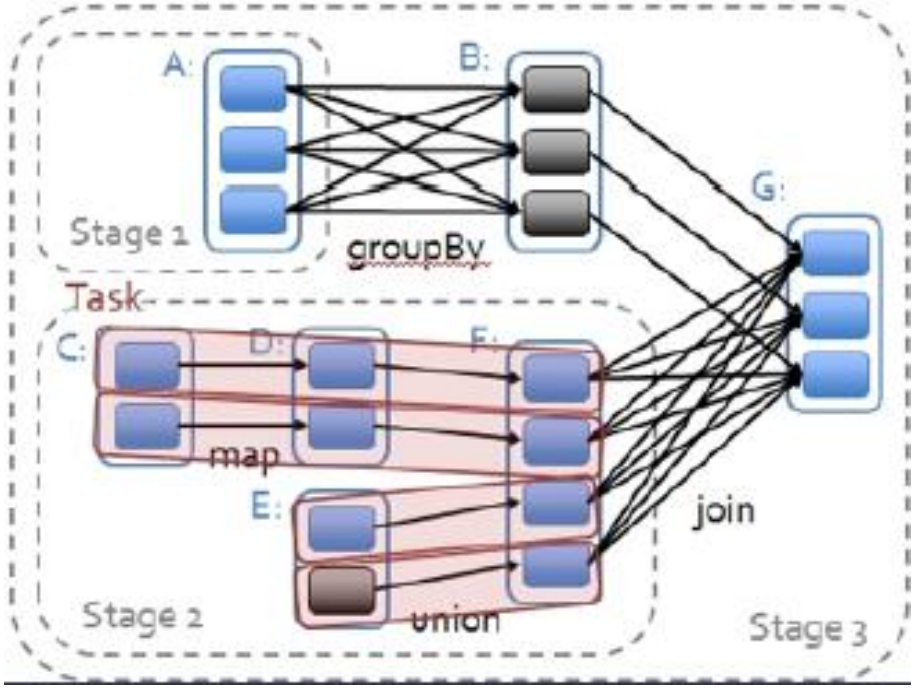
Spark任务调度器

- 调度器根据RDD的结构信息为每个动作确定有效的执行计划。调度器的接口是runJob函数，参数为RDD及其分区集，和一个RDD分区上的函数。该接口足以表示Spark中的所有动作（即count、collect、save等）。
- 总的来说，我们的调度器跟Dryad类似，但我们还考虑了哪些RDD分区是缓存在内存中的。调度器根据目标RDD的Lineage图创建一个由 stage构成的无回路有向图（DAG）。每个stage内部尽可能多地包含一组具有窄依赖关系的转换，并将它们流水线并行化（pipeline）。 stage的边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区，它可以缩短父RDD的计算过程。例如图6。父RDD完成计算后，可以在stage内启动一组任务计算丢失的分区。

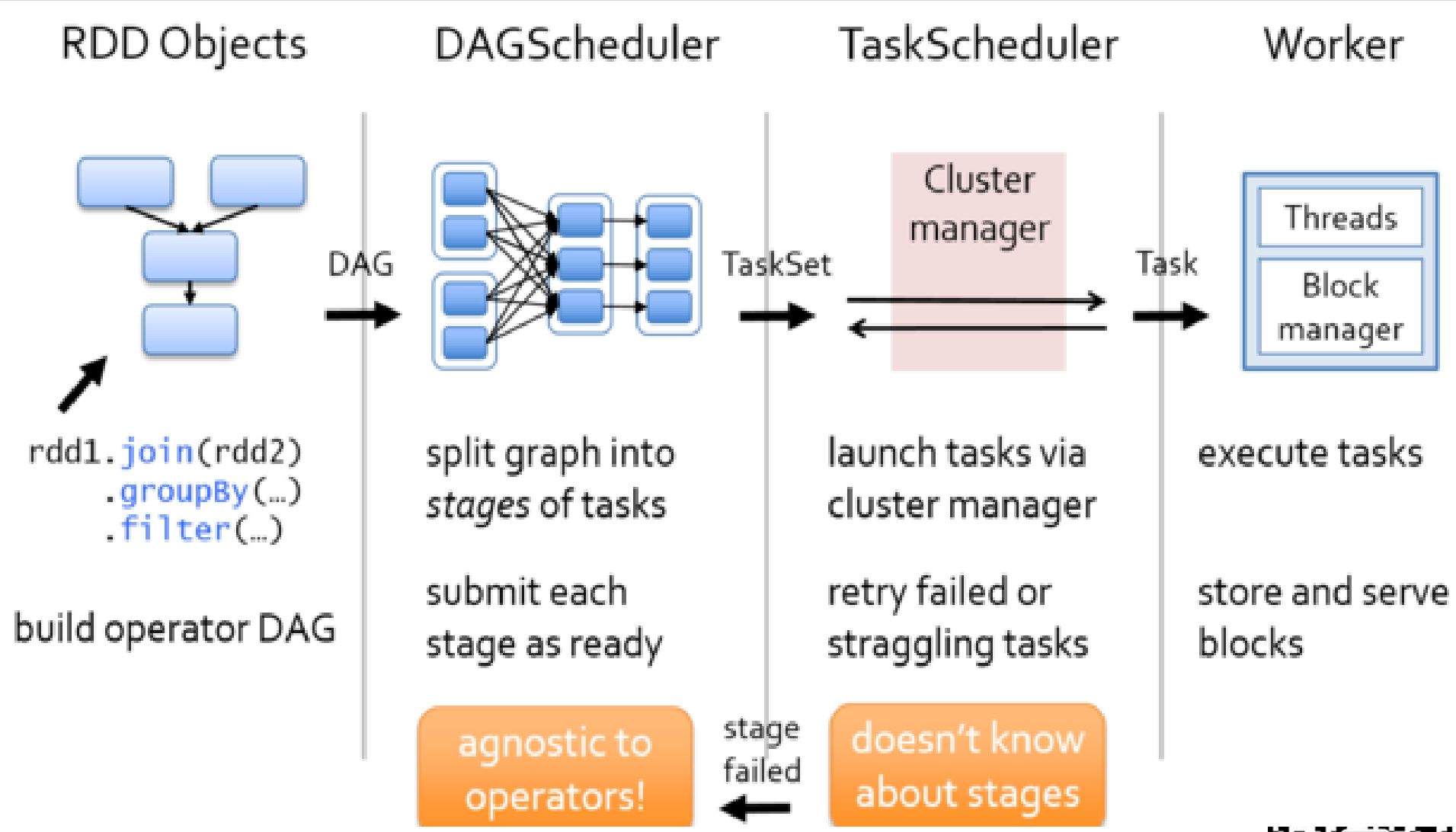

A:
B:
G:
Stage 1
groupBy
C:
D:
F:
map
E:
join
Stage 2
union
Stage 3

一个stage内的窄依赖进行pipeline操作

- 1+1+1+1=4
- 1+1=2; 2+1=3; 3+1=4



任务调度



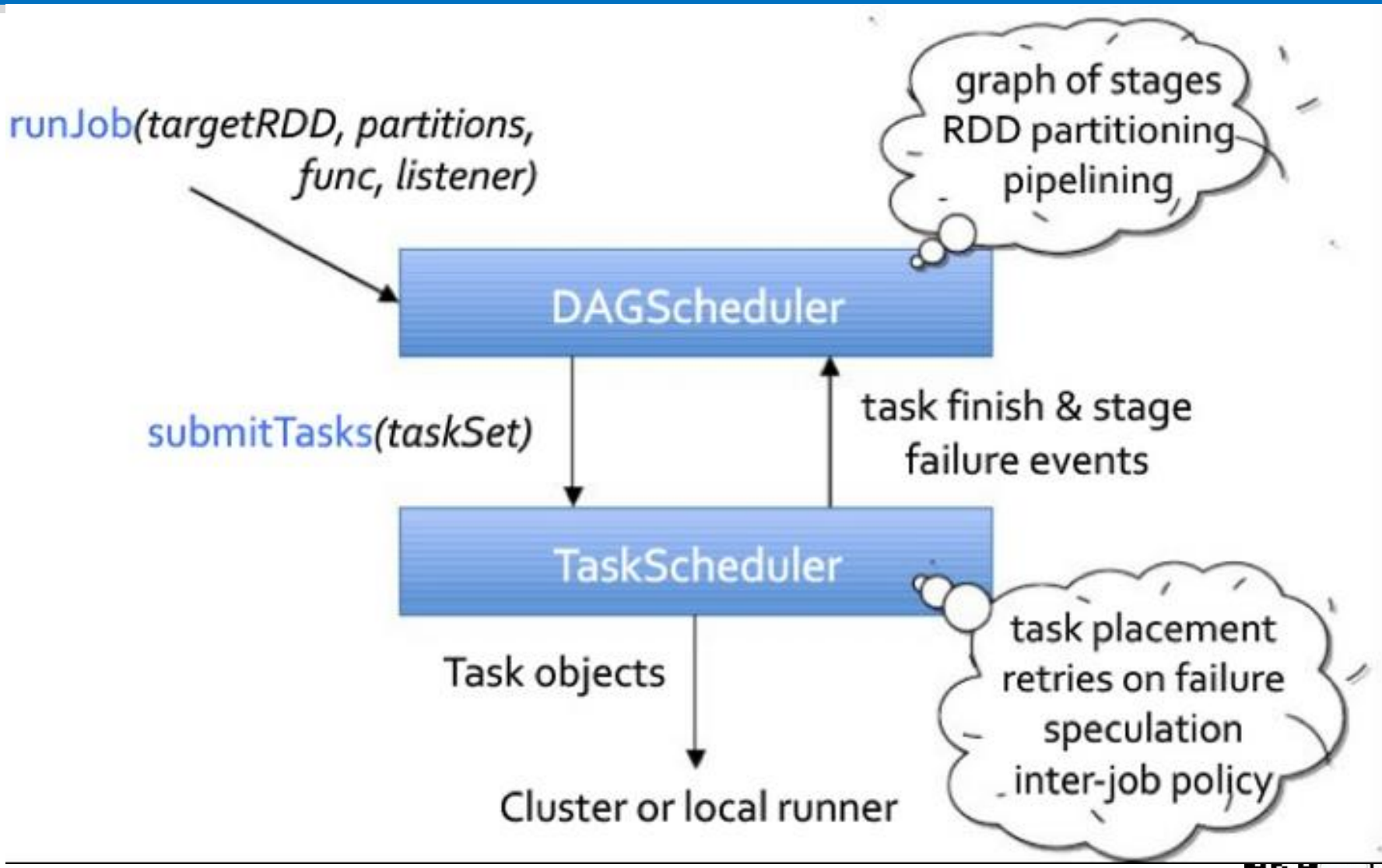
DAG Scheduler

- 基于Stage构建DAG，决定每个任务的最佳位置
- 记录哪个RDD或者Stage输出被物化
- 将taskset传给底层调度器TaskScheduler
- 重新提交shuffle输出丢失的stage

Task Scheduler

- 提交taskset(一一组task)到集群运行行并汇报结果
- 出现shuffle输出lost要报告fetch failed错误
- 碰到straggle任务需要放到别的节点上重试
- 为每一一个TaskSet维护一一个TaskSetManager(追踪本地性及错误信息)

Job调度流程


runJob(targetRDD, partitions, func, listener)
graph of stages
RDD partitioning
pipelining
DAGScheduler
submitTasks(taskSet)
task finish & stage failure events
TaskScheduler
task placement
retries on failure
speculation
inter-job policy
Task objects
Cluster or local runner

wordcount

- 我们在sparkshell中运行一下最简单的例子，在spark-shell中输入如下代码

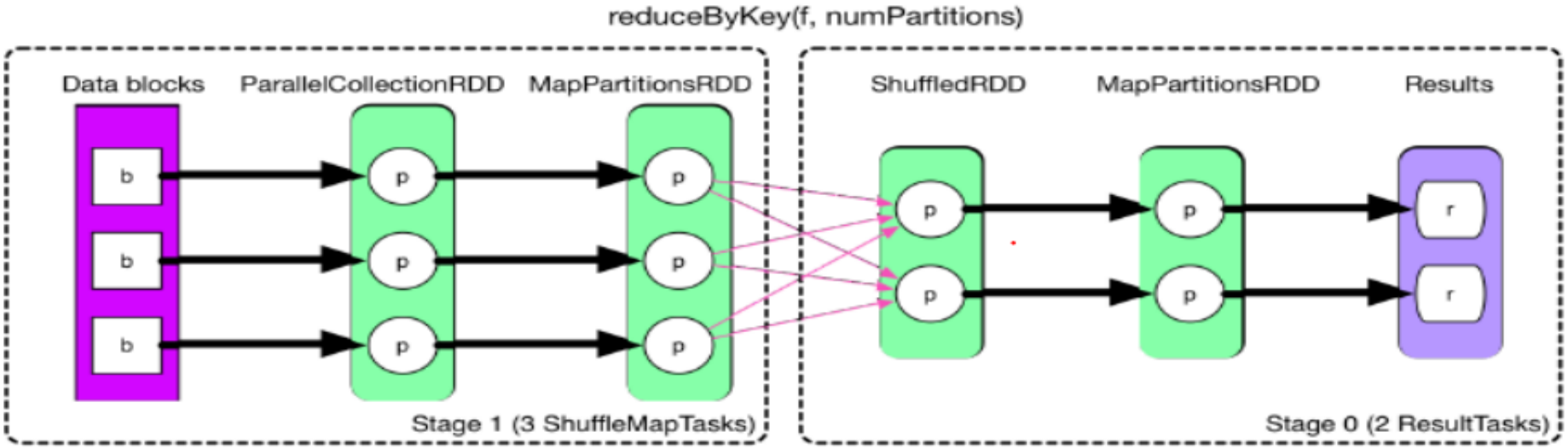
- scala>sc.textFile("README.md").filter(_.contains("Spark")).count

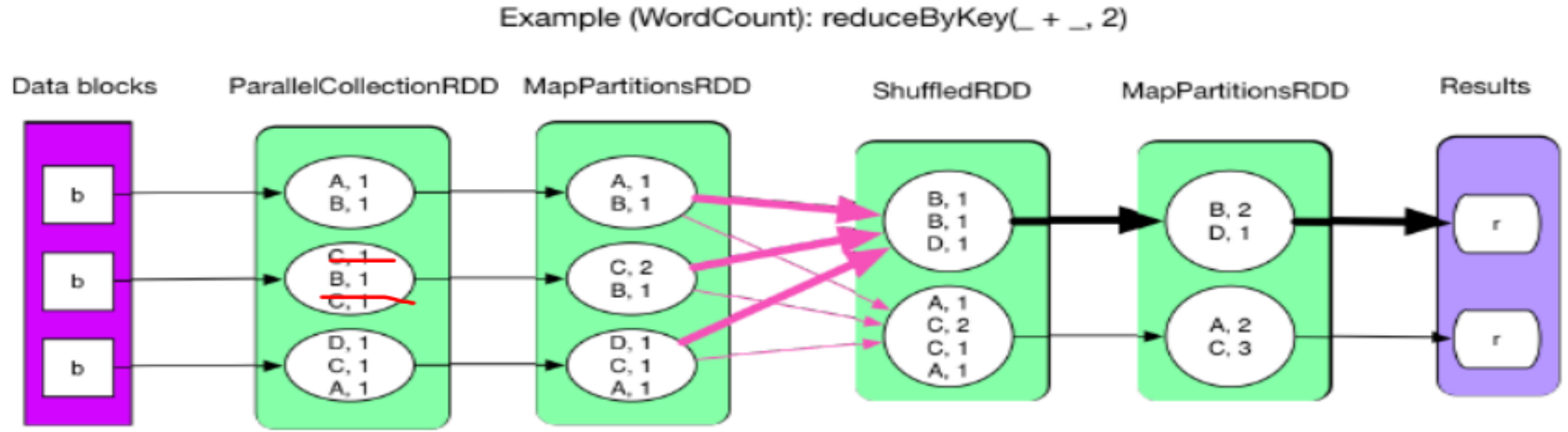
- 上述代码统计在README.md中含有Spark的行数有多少

wordcount

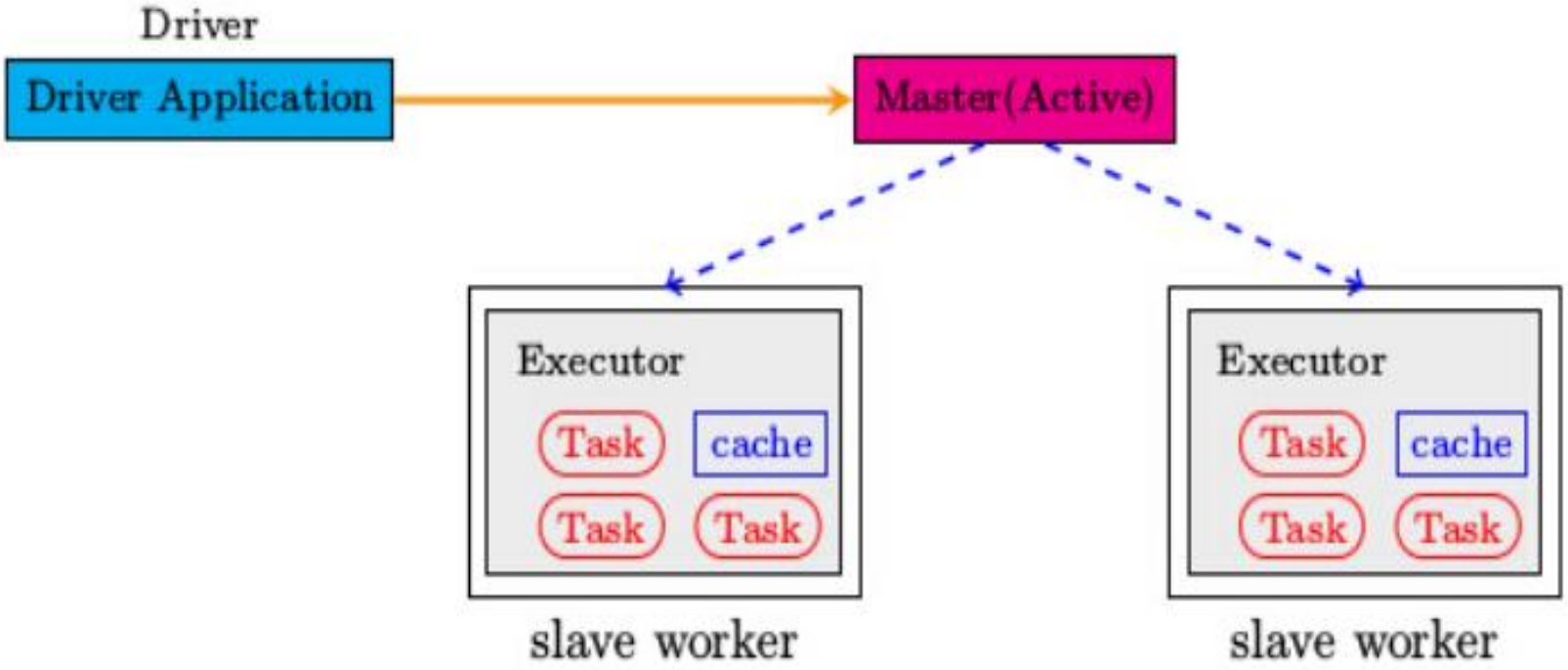
```
scala> val wordcount = rdd1.flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_)
wordcount: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[11] at reduceByKey at <console>:14

scala> wordcount.toDebugString
res2: String =
MapPartitionsRDD[11] at reduceByKey at <console>:14 (2 partitions)
  ShuffledRDD[10] at reduceByKey at <console>:14 (2 partitions)
    MapPartitionsRDD[9] at reduceByKey at <console>:14 (2 partitions)
      MappedRDD[8] at map at <console>:14 (2 partitions)
        FlatMappedRDD[7] at flatMap at <console>:14 (2 partitions)
          MappedRDD[1] at textFile at <console>:12 (2 partitions)
            HadoopRDD[0] at textFile at <console>:12 (2 partitions)

scala>
```

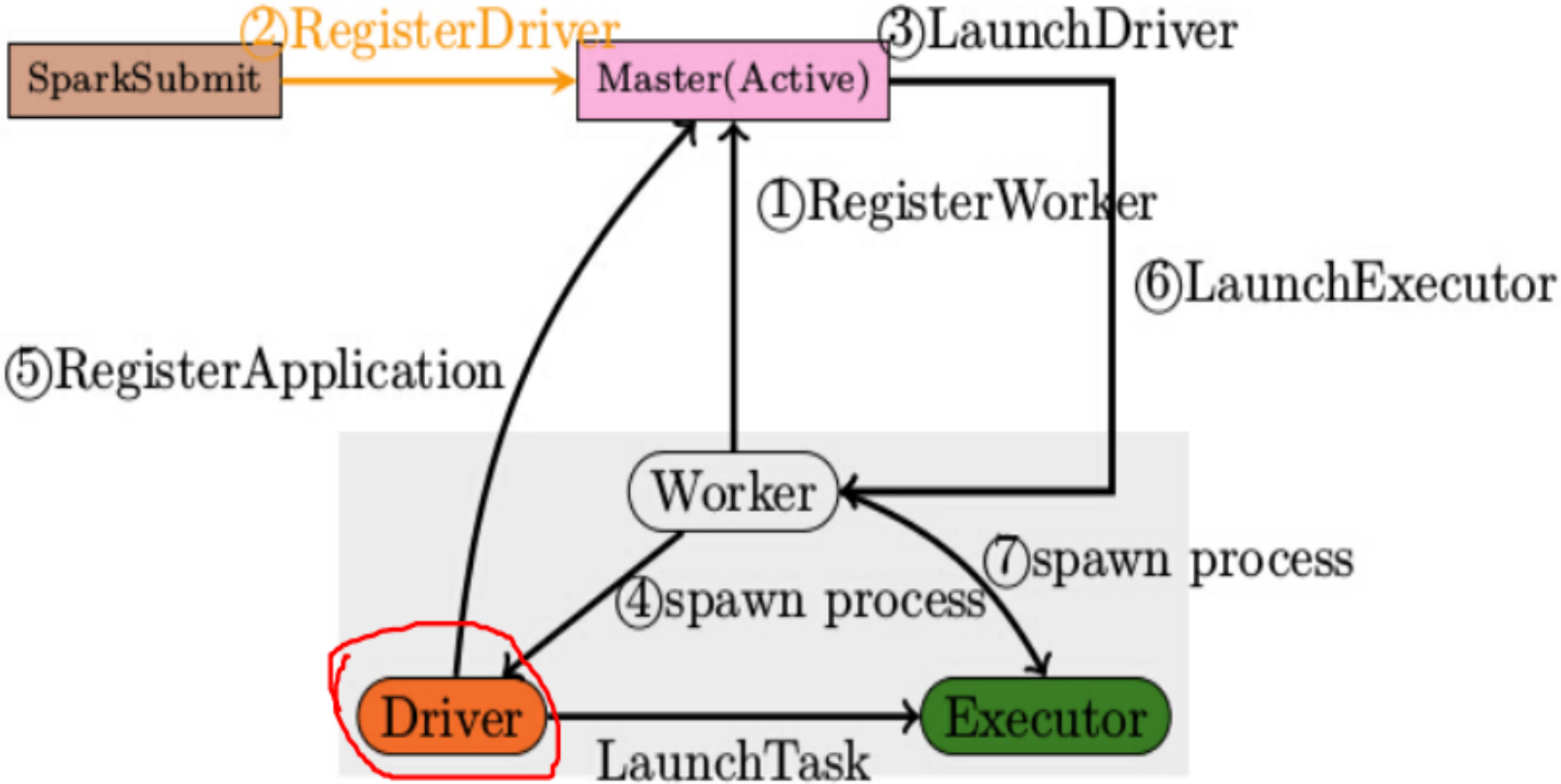
wordcount


reduceByKey(f, numPartitions)
Data blocks
ParallelCollectionRDD
MapPartitionsRDD
ShuffledRDD
MapPartitionsRDD
Results
b
b
b
p
p
p
p
p
p
p
p
p
r
r
Stage 1 (3 ShuffleMapTasks)
Stage 0 (2 ResultTasks)


Example (WordCount): reduceByKey(_ + _, 2)
Data blocks
ParallelCollectionRDD
MapPartitionsRDD
ShuffledRDD
MapPartitionsRDD
Results
b
b
b
A, 1
B, 1
C, 1
B, 1
C, 1
D, 1
C, 1
A, 1
A, 1
B, 1
C, 2
B, 1
D, 1
C, 1
A, 1
B, 1
B, 1
D, 1
A, 1
C, 2
C, 1
A, 1
B, 2
D, 1
A, 2
C, 3
r
r

client


Driver
Driver Application
Master(Active)
Executor
Task
cache
Task
Task
slave worker
Executor
Task
cache
Task
Task
slave worker


②RegisterDriver
③LaunchDriver
SparkSubmit
Master(Active)
①RegisterWorker
⑥LaunchExecutor
⑤RegisterApplication
Worker
⑦spawn process
④spawn process
Driver
Executor
LaunchTask

性能优化

- 问题: val rdd = data.filter(f1).filter(f2).reduceBy...经过以上语句会有很多空任务或者小小任务

- 解决: 使用用coalesce或者repartition去减少RDD中partition数量

性能优化

- 问题: 每个记录的开销太大大
rdd.map{x=>conn=getDBConn;conn.write(x.toString);conn.close}

- 解决：rdd.mapPartitions(records => conn.getDBConn;for(item <- records))write(item.toString); conn.close)

性能优化

- 问题: 任务执行行速度倾斜
- 解决:

- 1、数据倾斜(一一般是partition key取的不好)
- 考虑其它的并行行处理方方式 中间可以加入入一一步aggregation

- 2、Worker倾斜(在某些worker上的executor不给力力)
- 设置spark.speculation=true 把那些持续不给力力的node去掉

性能优化

- 问题:

  不设置spark.local.dir 这是spark写shuffle输出的地方

- 解决: 设置一一组磁盘
- spark.local.dir=/mn1/spark, /mnt2/spar, /mnt3/spark

- 增加IO即加快速度

性能优化

- 问题: reducer数量不合适
- 解决: 需要按照实际情况调整
- 太多的reducer,造成很多的小小任务,以此产生生很多启动任务的开销。
- 太少的reducer,任务执行行慢!!
- reduce的任务数还会影响到内存

性能优化

- 问题：collect输出大大量结果慢，审视源码
- 解决：直接输出到分布式文文件系统

性能优化

- 问题:序列化
- Spark默认使用用JDK自自带的ObjectOutputStream
  - 兼容性好，体积大，速度慢

- 解决: 使用用Kryo serialization
  - 体积小，速度快