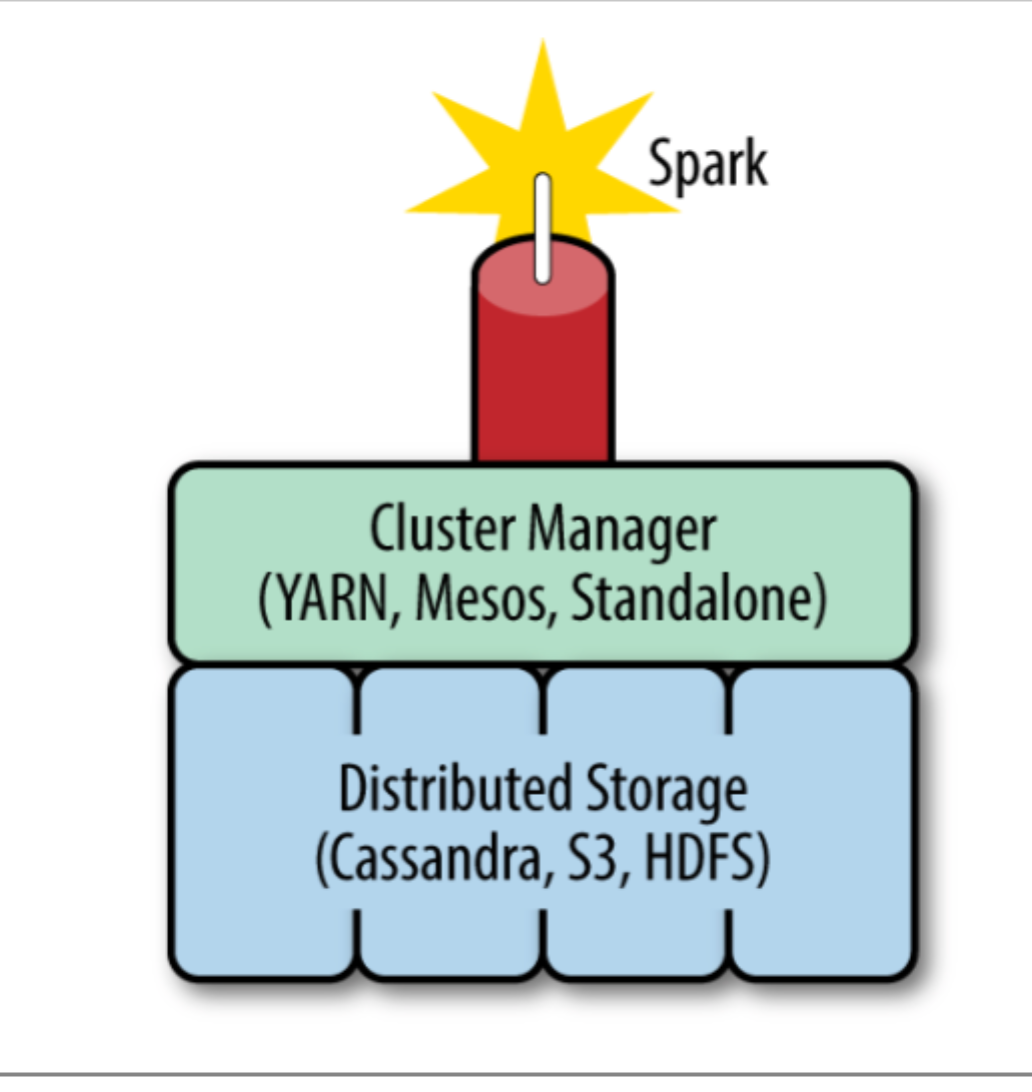
# High Performance Spark

# Chapter 2 How Spark Works

Spark：依托于分布式存储系统集群管理器之上的分布式通用计算框架



## Spark Components

基于抽象数据集RDD：惰性预估计、静态类型、分布式集合，具有tansformatins 操作函数

1. spark SQL
2. spark ML
3. spark streaming

## Spark Model of Parallel Computing: RDDs

以不可变、分布式数据集对象RDDs来标识数据集，RDDs的partitions可以分布在不同的节点上

RDDs是lazy的，只有最终的RDD(通常为写存储，collect 一个聚合结果到driver)数据需要计算时，才会计算RDD transfromations

RDD是在内存内部生成并复用、是immutable类型所以transforming 将产生新的RDD，而不是在原有的RDD上直接操作，这也提示我们，尽量复用RDD，尽量避免不必要的trans操作

### Lazy Evaluation

RDD纯lazy，直到有action操作才会计算分区，返回类型：

* RDD
* 返回给Driver的结果（operations like count or collect）
* 写入外部存储系统的结构（such as copyToHadoop）的结果

Actions触发scheduler，创建基于RDD transformations依赖的DAG。即，spark反向定义一个计算步骤（DAG）来产生最终分布式数据集（每一个分区）中的每一个object，

利用这个计算步骤（执行计划），scheduler会为每个stage计算其缺失的partitions。

【备注】并非所有的transformations都是100%lazy的，例如sortByKey需要预估RDD来决定数据的range，因此它既是一个transformations也是一个action。

### Performance and usability advantages of lazy evaluation

（1）高效性：Lazy特性使得不需要和driver互通的一些operation(一般one-to-one dependences 类型的transformation)可以合并，避免重复多次传输数据，例如：

对于相同RDD的map和filter可以一起执行，避免多次访问数据计算。

（2）简化计算逻辑：避免了向MapReduce一样由开发者代码consolidate maping操作，lazy 允许链式执行窄依赖，spark完成consolidate

### Lazy evaluation and fault tolerance

不是采用传统的维护更新日志方式来跟踪RDD，RDD自身的每个分区会记录自己的血缘lineage，优势：快速并行重计算恢复数据

### Lazy evaluation and debugging

Lazy为debug一些线索，spark程序只在action的点fail（stack traces 反馈于action的点，尽管逻辑上可能fail的点在前面的某个transformation）

### [待补]In-Memory Persistence and Memory Management

### [待补]Immutability and the RDD Interface

### Types of RDDs

RDD中定义的 toDebugString 函数返回RDD类型和父RDDs的List

### Functions on RDDs: Transformations Versus Actions

两种functions：*actions* 、*transformations*

1. Actions :返回有副作用的非 RDD
2. Transformations :返回另外一个 RDD.

每一个spark程序必须包含一个action（bring information to driver or write to stable storage），Action 触发计算。

（1）拉取数据至driver的actions有：collect、count、collectAsMap、sample、reduce、take。【备注】由于driver的内存有限，最好不用collect 或sample

（2）写入存储的actions有：saveAsTextFile, saveAsSequenceFile, and saveAsObjectFile .存入Hadoop的actions只能是K-V pairs RDDS（由PairRDDFunctons class和NewHadoopRDD class 产生），其他的诸如saveAsTextFile and saveAsObjectFile 可作用于任何RDD，主要是通过加入一个隐式的NULL key（会在saving level被忽略）到每一条记录。

【备注】返回nothing（void for java or Unit for scala）的函数，例如foreach也是action，也会触发job

### Wide Versus Narrow Dependencies

Transformations 分为两类：

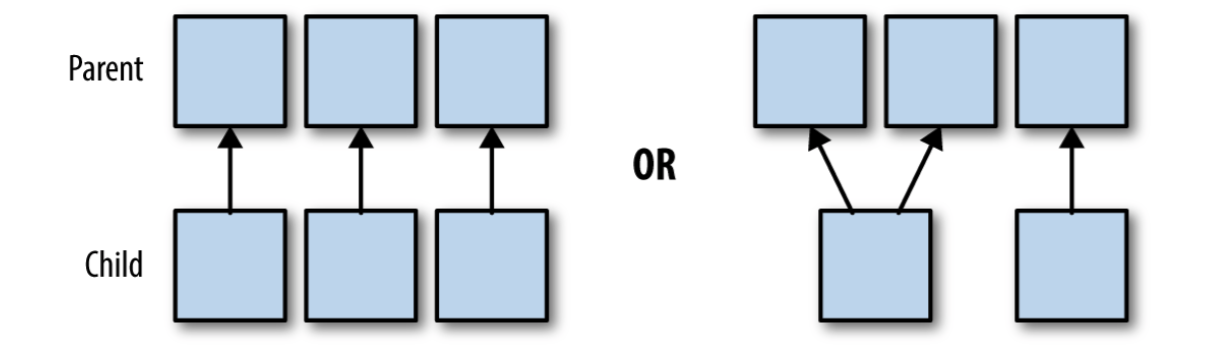
1. transformations with *narrow dependencies*
2. transformations with *wide dependencies*

narrow dependencies

child RDD的每个partition只简单、有限的依赖于parent RDD的partitions， parent RDD的每个partition至多只有一个child partition

一般的trans为：

map, filter, mapPartitions, and flatmap



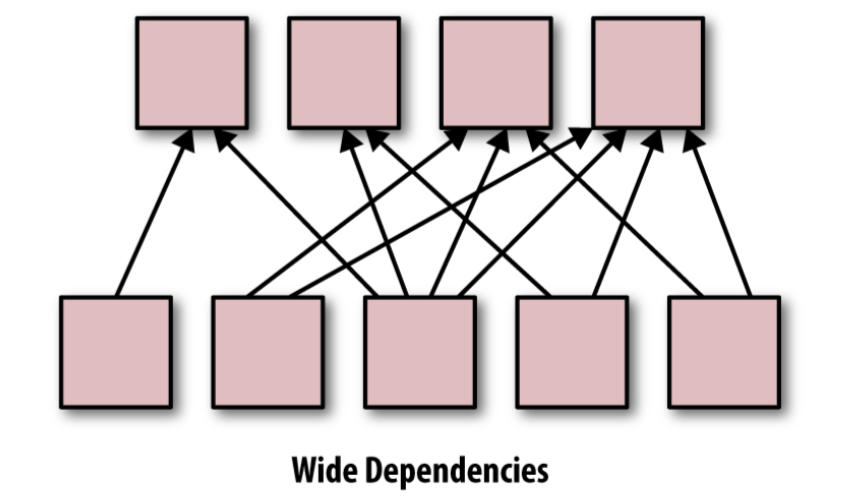
child partitions 依赖parent 的任意partions，在parent数据计算完成前不可完全确定对parent partions的具体依赖。

一般的trans为：

groupByKey, reduceByKey, sort, sortByKey

【备注】

join functions 有点复杂，根据两个parent RDD的具体分区情况，可能是窄依赖，也可能是宽依赖



## Spark Job Scheduling

Application包含一个driver process,由集群调度器根据SparkContext调度

driver process:由RD写高层spark logic，一系列的executor processes(分发到不同集群节点执行)

Spark program 在driver node 运行，向executors 分发instructions

Applications

Job由RDD中的action触发

### Resource Allocation Across Applications

static allocation

dynamic allocation

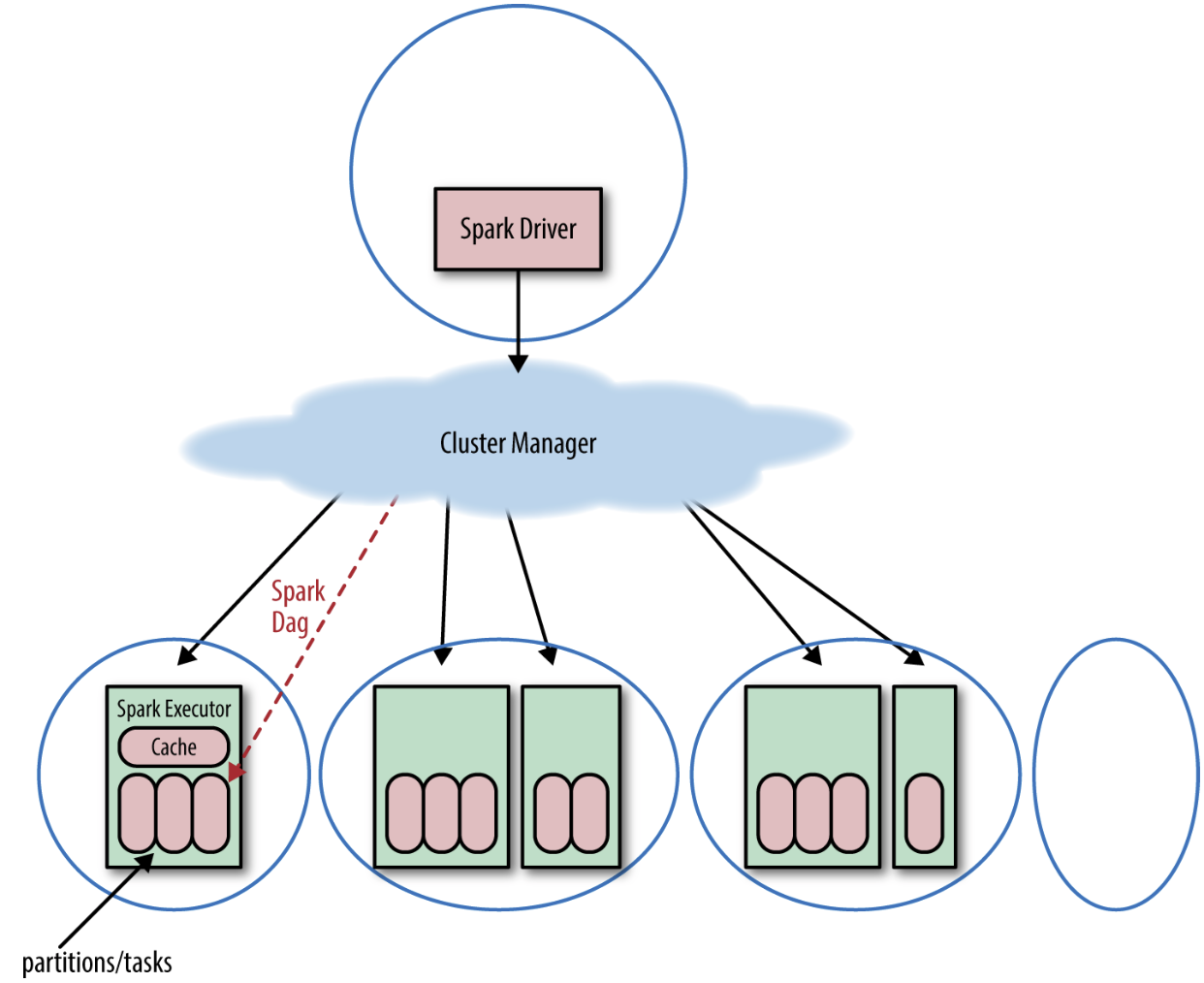
### The Spark Application

1. SparkContext启动，然后Spark application启动，driver和一系列的executors在clusters中启动。
2. 每个executor拥有自身的JVM，executor 不可跨多节点，而一个节点可有
3. SparkContext决定每个executor的资源分配
4. Job启动后每个相关executor会运行相应的tasks来计算RDD
5. applications间的RDDs不共享

SparkContext启动，driver program pings cluster manager，cluster manager 在worker nodes中申请下发 Spark executors (JVMs shown as black boxes in the diagram)

RDD 在executors中计算（RDD的分区partition分布在不同的节点）

每个executor可以有多个partitions，但是一个partition不能跨多executors



### Default Spark Scheduler

默认情况下， Spark按照先入先出的决策调度jobs，支持FairScheduler round-robin形式，分批调度所有的任务，任务尽优的获取利用集群资源,Spark application 找找SparkContext调起的相关的actions的顺序，调起jobs

## The Anatomy of a Spark Job

Spark的lazy型计算范式中，Spark application 在driver program 调用到action前不会干任何工作

Spark scheduler 为每个action的job构建一个DAG然后触发job

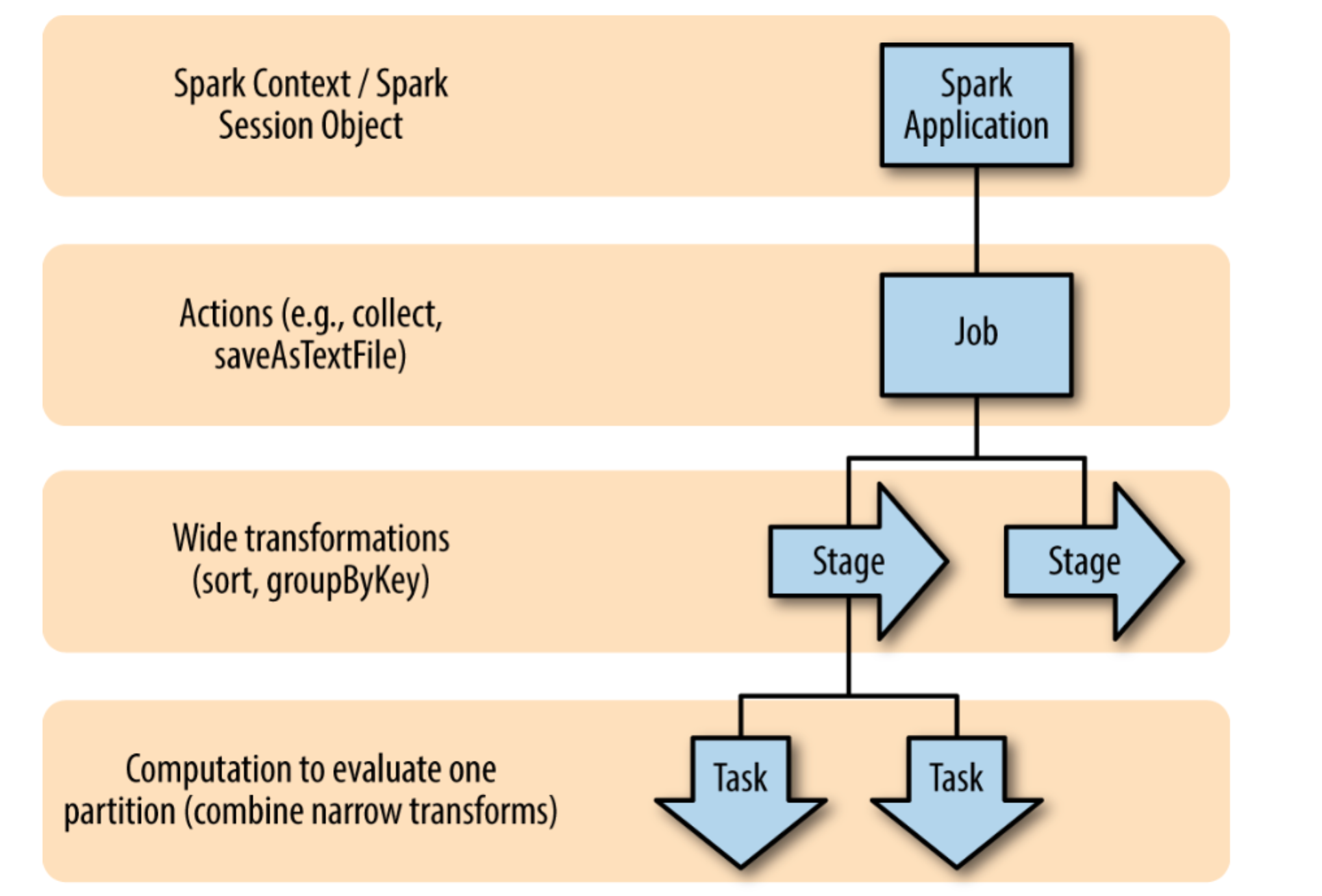
job分为不同的stages,每个stage包含一个task（代表每一个并行中计算）集，task于executors中执行。

Application:针对一个RDD action可能有多个jobs，

Job:根据wide transformation【map shuffle stage 或 result shuffle stage】划分为多个stages

Stage:可能包含多个tasks，这取决于stage计算中的parallelizable unit单元（通常也就是一个task，通常task数==partition数），

Task:运行于一个partition



### DAG

Spark高层调度器（DAG Scheduler)利用RDD的dependencies为job中的stages构建DAG（Directed Acylic Graph）

DAG是stage中RDD的依赖执行图，DAG基于RDD transformations中partitions间依赖关系构建DAG的边线

DAG决定这job的执行，从另一个角度看DAG 为job构建了stages的构成图，决定task运行的locations，将这些信息传递给TaskScheduler

TaskScheduler负责集群中task的运行，TaskScheduler在partitions间根据dependencies创建一个图graph

DAG 的边线是基于RDD transformations中partitions间依赖关系构建的，因此，若一个operation返回的不是RDD，则无children，依据图论，称其为叶子“leaf”，一个复杂的transformations集，将和一个执行图映射在在一起，一旦action调起，不能再向图中添加其他节点

### Jobs

Job是 execution层高级成员，每个job与一个action有关，Action触发job。

job is defined by calling an action

### Stages

（1）Lazily transformations 只有在action调起是才会执行

（2）Job由action定义触发

（3）Action可包含多个transformations,其中宽transformations是job中stages的切分点，也就是说stage和wide transformation产生的shuffle dependency有关。

（4）从另一个角度看，Stage可以看做是可独立在一个executor执行的计算集（computations\tasks），独立是和其他executor或者driver间无communication.

话一句话说，stage 在任何worker间需要网络通信的点发生，例如wide transformations 产生shuffle,由shuffle产生的stage边缘，存在ShuffleDependencies。例如groupByKey ，需要分布在不同partitions中的数据。

（5）对于窄依赖的transformations可以在同一个stage内执行。

例如word count中flatMap, map, filter操作在同一个stage内

### Tasks

stage由tasks组成，tasks执行层的最小单元，每一个task代表一个本地计算过程。一个stage中的所有tasks在不同的数据片上执行相同的代码code. 每个task只能在一个excutor上执行。但是excutor可以分片slot执行多个tasks。每个stage中的Task数，与该stage输出RDD的分区数有关

**def** simpleSparkProgram(rdd **: RDD**[**Double**])**: Long** ={ *//stage1*

rdd.filter(**\_**< 1000.0) .map(x **=>** (x, x) )

*//stage2*

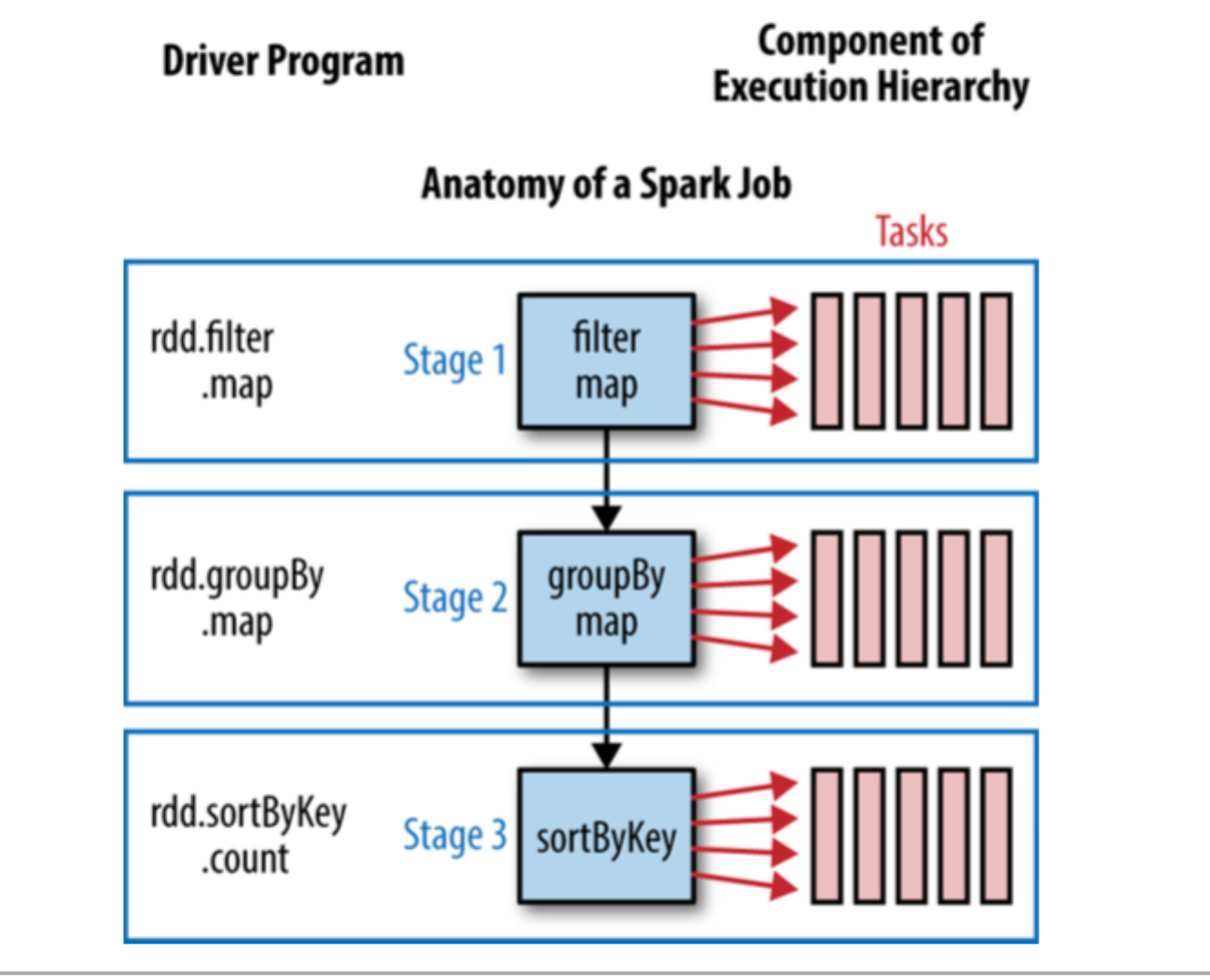
.groupByKey()

.map{ **case**(value, groups) **=>** (groups.sum, value)}

*//stage 3*

.sortByKey()

.count() }



stages是shuffle操作（groupBykey,sortByKey）的边界,每一个stage包含多个tasks，每一个task对应于trans结果RDD的每一个分区，tasks通常并行的。

1. 集群没不能为所有stage并行执行每个task
2. 每个executor存在有限个cores，executor cores的配置在application层，同时也和集群物理cores数有关。
3. Spark并行执行的task数不多于分配给application的executor cores的数目。
4. 可依据Spark Conf中的配置如下计算task的并行数：
5. executor cores总数 = 每个executor的cores \* executor数
6. 若分区数partitions（tasks数）多于可运行tasks数，则只有等待运行中的task执行完毕才可执行新的计算。

spark执行模型小结：

* Job：计算一个最终结果所需要的一组RDD transformations集合
* Stage：一个work任务的分片segment，可脱离（无通信）driver执行，换一句话说，无partitions间的数据move过程的一组计算
* Tasks：stage中作用于每个分区的work任务的执行单元

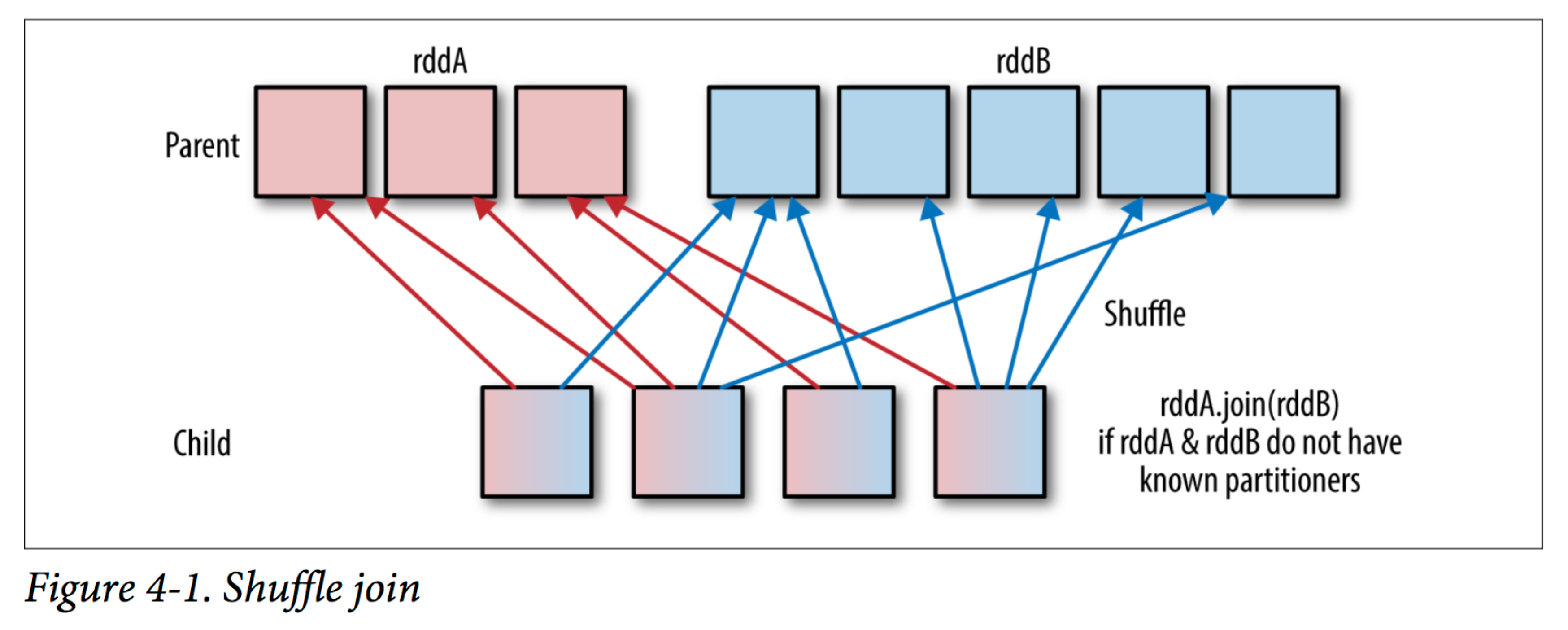
# Chapter 4 Joins (SQL and Core)

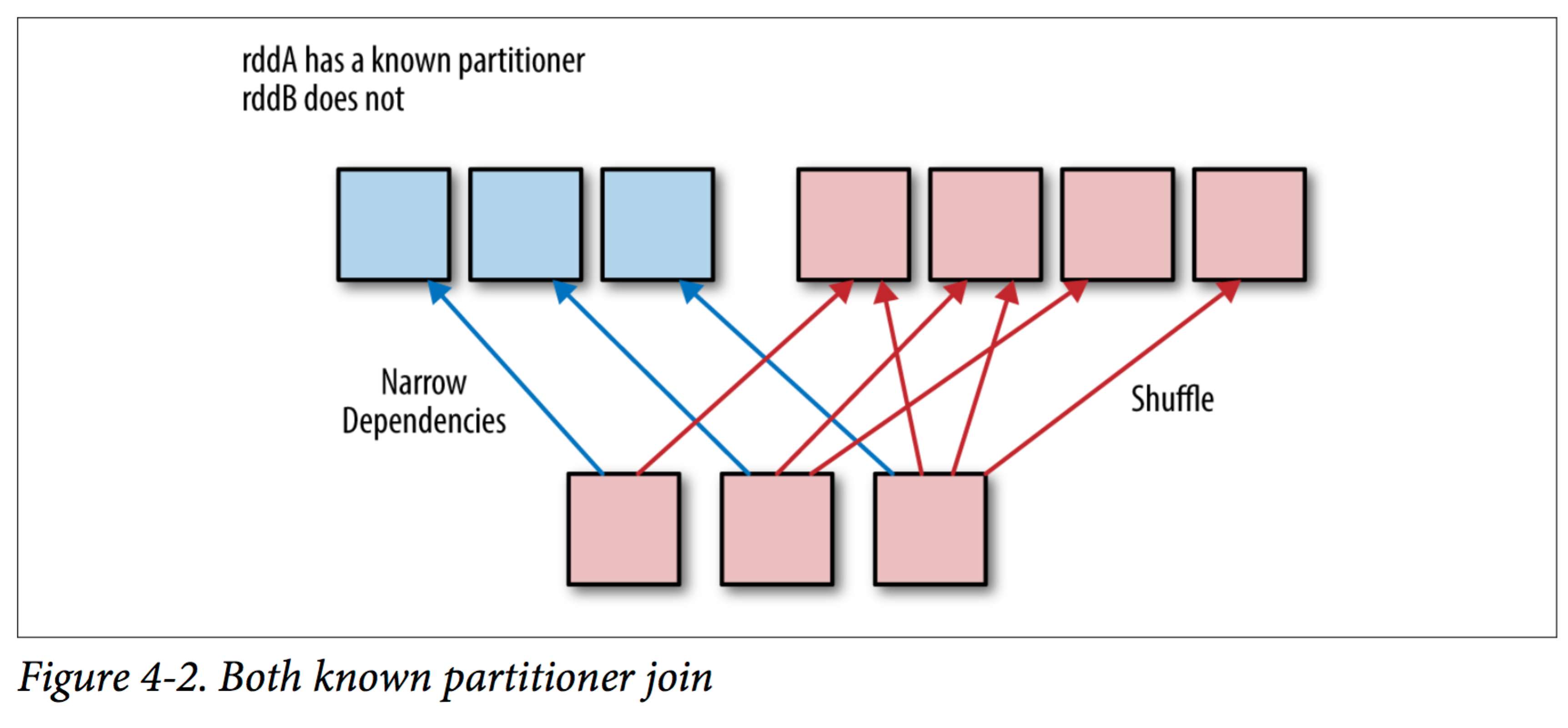
## Core Spark Joins

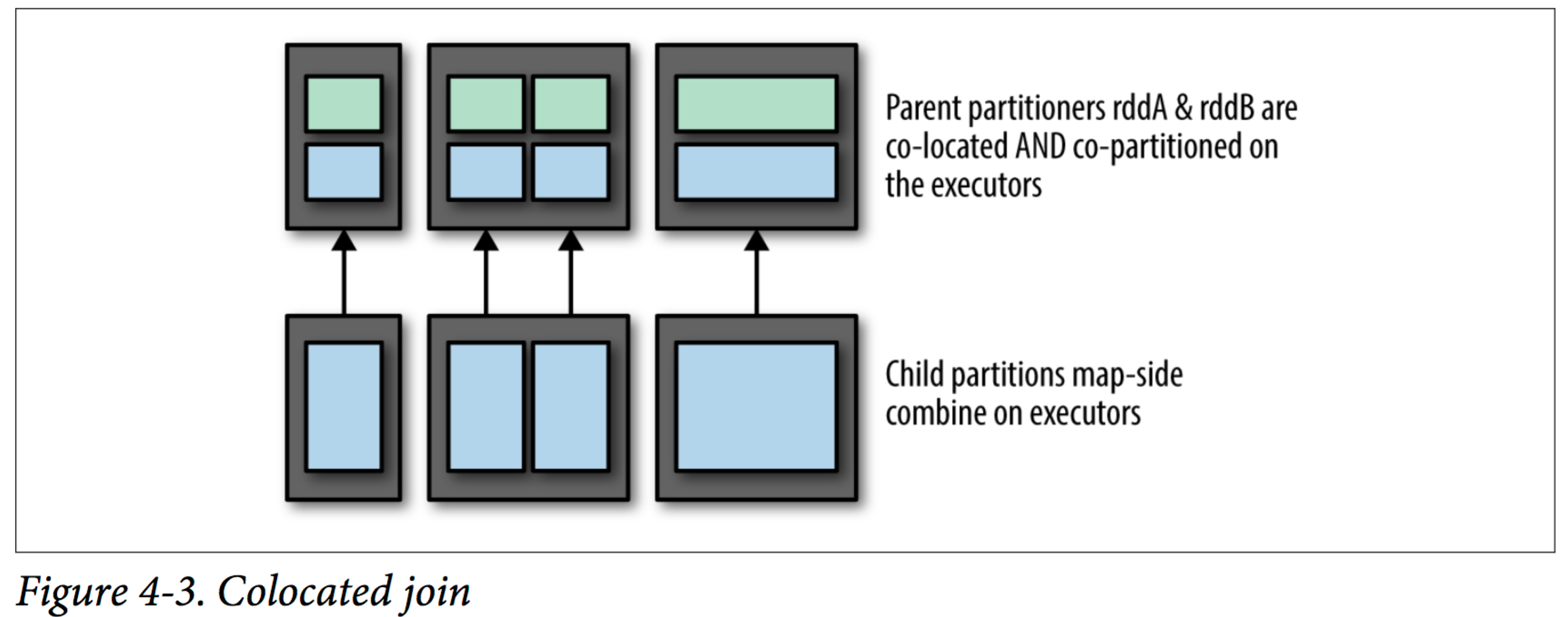
为实现本地聚合，不同RDD相同key的数据要located到相同的partion中。

1. 若没有known partitions，通过shuffle实现
2. 若数据位于相同的partitioner中，则通过简单的colocated实现
3. 若一方或者双方有known partitioner，则只有窄依赖创建

同大多数K-V操作一样，join的cost代价与keys的数目和记录传输到准确分区中的距离有关







### Choosing a Join Type

1. 避免重复数据，distinct、combineByKey、congroup后进行join
2. 支持join和leftOuterJoin、RightOuterJoin

左右关联时，有值返回Some（‘x’）否则Some（None）

1. 过滤容易产生数据倾斜的值（有shuffle）

下面是Join相关API的简单应用测试：

|  |
| --- |
| **package** com.dt.spark.main.RDDLearn.PairRDDJoinFunAPI **import** org.apache.spark.{SparkConf, SparkContext} */\*\*  \* Created by on 16/12/06.  \*/ //========================================== /\*  PairRDD间关联API,注意返回值类型  def join[W](other :Tuple2[K, W]) : Tuple2[K, Tuple2[V, W]]  def join[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[V, W]]] = { /\* compiled code \*/ }   def leftOuterJoin[W](other :Tuple2[K, W]) : Tuple2[K, Tuple2[V, Option[W]]  def leftOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[V, scala.Option[W]]]] = { /\* compiled code \*/ }   def rightOuterJoin[W](other : Tuple2[K, W]) : Tuple2[K, Tuple2[Option[V], W]]  def rightOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[scala.Option[V], W]]] = { /\* compiled code \*/ }   def fullOuterJoin[W](other : Tuple2[K, W]) : Tuple2[K, scala.Tuple2[Option[V], Option[W]]  def fullOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[scala.Option[V], scala.Option[W]]]] = { /\* compiled code \*/ }  \*/* **object** PairRDDJoinFunAPI {  **def** main(args: Array[String]) {  **val** conf = **new** SparkConf()  conf.setAppName(**"test"**)  conf.setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  *//==========================================  /\*  def join[W](other :Tuple2[K, W]) : Tuple2[K, Tuple2[V, W]]  def join[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[V, W]]] = { /\* compiled code \*/ }   def leftOuterJoin[W](other :Tuple2[K, W]) : Tuple2[K, Tuple2[V, Option[W]]  def leftOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[V, scala.Option[W]]]] = { /\* compiled code \*/ }   def rightOuterJoin[W](other : Tuple2[K, W]) : Tuple2[K, Tuple2[Option[V], W]]  def rightOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[scala.Option[V], W]]] = { /\* compiled code \*/ }   def fullOuterJoin[W](other : Tuple2[K, W]) : Tuple2[K, scala.Tuple2[Option[V], Option[W]]  def fullOuterJoin[W](other : org.apache.spark.rdd.RDD[scala.Tuple2[K, W]], numPartitions : scala.Int) : org.apache.spark.rdd.RDD[scala.Tuple2[K, scala.Tuple2[scala.Option[V], scala.Option[W]]]] = { /\* compiled code \*/ }  \*/* **val** KVPairRDD1 = sc.parallelize(*List*((**"beijing"**, **"京"**), (**"shaanxi"**, **"陕"**),(**"jiangsu"**, **"苏"**),(**"shandong"**, **"鲁"**),(**"guangxi"**,**"桂"**)))  **val** KVPairRDD2 = sc.parallelize(*List*( (**"beijing"**, **"北京"**), (**"shaanxi"**, **"西安"**),(**"jiangsu"**, **"南京"**),(**"shandong"**, **"济南"**)))  **def** getVauleOfOption(in:Option[String]):String = in **match** {  **case** *Some*(x) => x  **case** None => **"NULL"** }  **val** res = KVPairRDD1.fullOuterJoin(KVPairRDD2).map(a=>(a.\_1,getVauleOfOption(a.\_2.\_1),getVauleOfOption(a.\_2.\_2)))  res.foreach(*println*(\_))  *// (guangxi,桂,NULL)  // (shaanxi,陕,西安)  // (shandong,鲁,济南)  // (beijing,京,北京)  // (jiangsu,苏,南京)* sc.stop()  ] ] |

### Choosing an Execution Plan

Join默认要求相同的key在同一分区，因此默认是shuffle hash join，为避免shuffle：

1. 两个RDD有已知的分区（known partition）
2. 一方数据量很少可以直接放于内存，可以使用hash join

|  |
| --- |
| **package** com.dt.spark.main.RDDLearn.PairRDDJoinFunAPI **import** org.apache.spark.rdd.RDD **import** org.apache.spark.{HashPartitioner, SparkContext, SparkConf}  */\*\*  \* Created by hjw on 17/9/16.  \*/* **object** KnownPartitionJoin {  */\*  有数据  (id,score)和(id,address)  得到得分最高的地址  \*/* **def** joinScoresWithAddress1( scoreRDD : RDD[(Long, Double)],  addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {  **val** joinedRDD = scoreRDD.join(addressRDD)  joinedRDD.reduceByKey( (x, y) => **if**(x.\_1 > y.\_1) x **else** y )  }  */\*  减少join的数据量,想reduceBykey  \*/* **def** joinScoresWithAddress2(scoreRDD : RDD[(Long, Double)],  addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {  **val** bestScoreData = scoreRDD.reduceByKey((x, y) => **if**(x > y) x **else** y)  bestScoreData.join(addressRDD)  }  */\*  避免缺失,用leftOuterJoin  \*/* **def** outerJoinScoresWithAddress(scoreRDD : RDD[(Long, Double)],  addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, Option[String]))]= {  **val** joinedRDD = scoreRDD.leftOuterJoin(addressRDD)  joinedRDD.reduceByKey( (x, y) => **if**(x.\_1 > y.\_1) x **else** y )  }  */\*  用known parttion 避免shuffle join  \*/* **def** joinScoresWithAddress3(scoreRDD: RDD[(Long, Double)], addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {  *// If addressRDD has a known partitioner we should use that,  // otherwise it has a default hash parttioner, which we can reconstruct by  // getting the number of partitions.* **val** addressDataPartitioner = addressRDD.*partitioner* **match** {  **case** (*Some*(p)) => p  **case** (None) => **new** HashPartitioner(addressRDD.partitions.length) }  **val** bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner, (x, y) => **if**(x > y) x **else** y)  bestScoreData.join(addressRDD)  }   **def** main(args: Array[String]) {  **val** conf = **new** SparkConf()  conf.setAppName(**"test"**)  conf.setMaster(**"local"**)  **val** sc = **new** SparkContext(conf)  **val** coreRDD = sc.parallelize(*List*((1L,1.0),(2L,3.0),(3L,90.0),(4L,100.0)),2)  **val** addressRDD = sc.parallelize(*List*((1L,**"Japan"**),(2L,**"USA"**),(3L,**"Indian"**),(4L,**"China"**)),2)  **val** resRDD = *joinScoresWithAddress3*(coreRDD,addressRDD)  **val** res = resRDD.collect().apply(0)  *println*(res.\_1 + **"->("** + res.\_2.\_1 + **" , "** + res.\_2.\_2 + **")"** )  *//4->(100.0 , China)* **while**(**true**){;}  sc.stop()  }  } |

### Speeding up joins by assigning a known partitioner

|  |
| --- |
| **def** joinScoresWithAddress3(scoreRDD: RDD[(Long, Double)], addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {  *// If addressRDD has a known partitioner we should use that,  // otherwise it has a default hash parttioner, which we can reconstruct by  // getting the number of partitions.* **val** addressDataPartitioner = addressRDD.*partitioner* **match** {  **case** (*Some*(p)) => p  **case** (None) => **new** HashPartitioner(addressRDD.partitions.length) }  **val** bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner, (x, y) => **if**(x > y) x **else** y)  bestScoreData.join(addressRDD)  } |



### Speeding up joins using a broadcast hash join

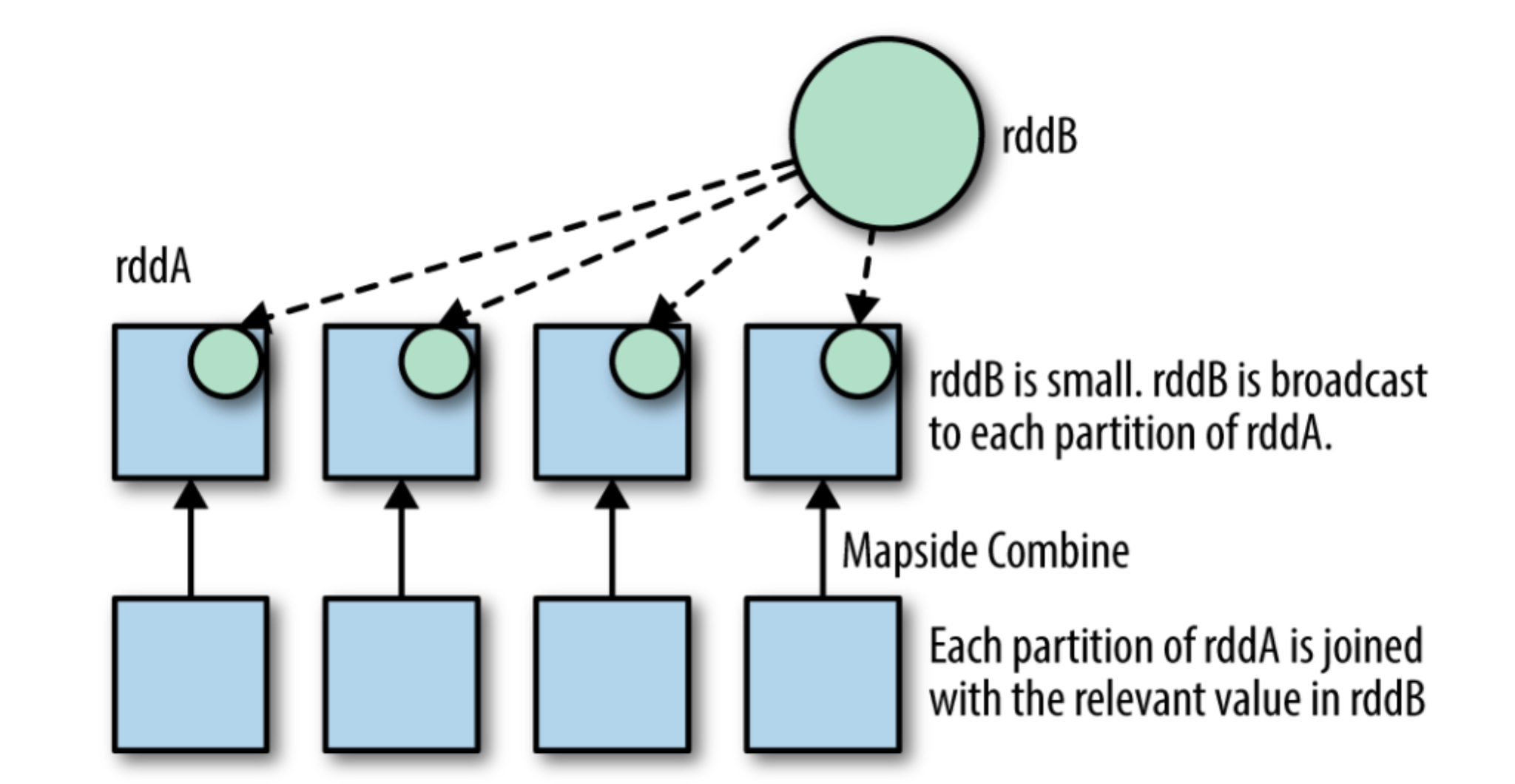
broadcast hash join :将小的RDD（内存可容纳）的RDD传输到每个工作节点上去，和大RDD进行类似于hiveSQL的map-side combine。

SparkSQL可以根据以下两个参数控制：

1. spark.sql.autoBroadcastJoinThreshold
2. spark.sql.broadcastTimeout

Spark Core没有相关实现，但可利用map实现：

1. collectAsMap 小的RDD值driver
2. broadcast
3. mapPartions 合并元素



|  |
| --- |
| **def** manualBroadCastHashJoin[K : Ordering : ClassTag, V1 : ClassTag, V2 : ClassTag] (bigRDD : RDD[(K, V1)], smallRDD : RDD[(K, V2)])= {  **val** smallRDDLocal: Map[K, V2] = smallRDD.collectAsMap()  **val** smallRDDLocalBcast = bigRDD.sparkContext.broadcast(smallRDDLocal)  bigRDD.mapPartitions(iter => {  iter.flatMap{  **case** (k,v1 ) =>  smallRDDLocalBcast.value.get(k) **match** {  **case** None => *Seq*.empty[(K, (V1, V2))]  **case** *Some*(v2) => *Seq*((k, (v1, v2)))  }  }  }, preservesPartitioning = **true**) } |

### Partial manual broadcast hash join

当RDD大于内存限制，可将较大RDD中高负载（数目最多一些如topN,countByKeyApprox或reduceByKey + sort取topN）keys，在小RDD中将这些key过滤出来，作为可存于内存的中间RDD分发做map-join，和剩余部分的普通join做union

# Chapter 5 Effective Transformations

## Narrow Versus Wide Transformations

从child的角度（DAG的构建是反向的，RD定义）

（1）wide transformations

有shuffle, child RDD的每个partition可依赖于多个parent RDD的partitions

（2）narrow transformations

child RDD的每个partition只简单、有限的依赖于parent RDD的partitions， parent RDD的每个partition至多只有一个child partition

从parent的角度（程序开发是顺向的，creator定义）

（1）wide dependencies

多个child partition可依赖于parent partition

（2）narrow dependencies

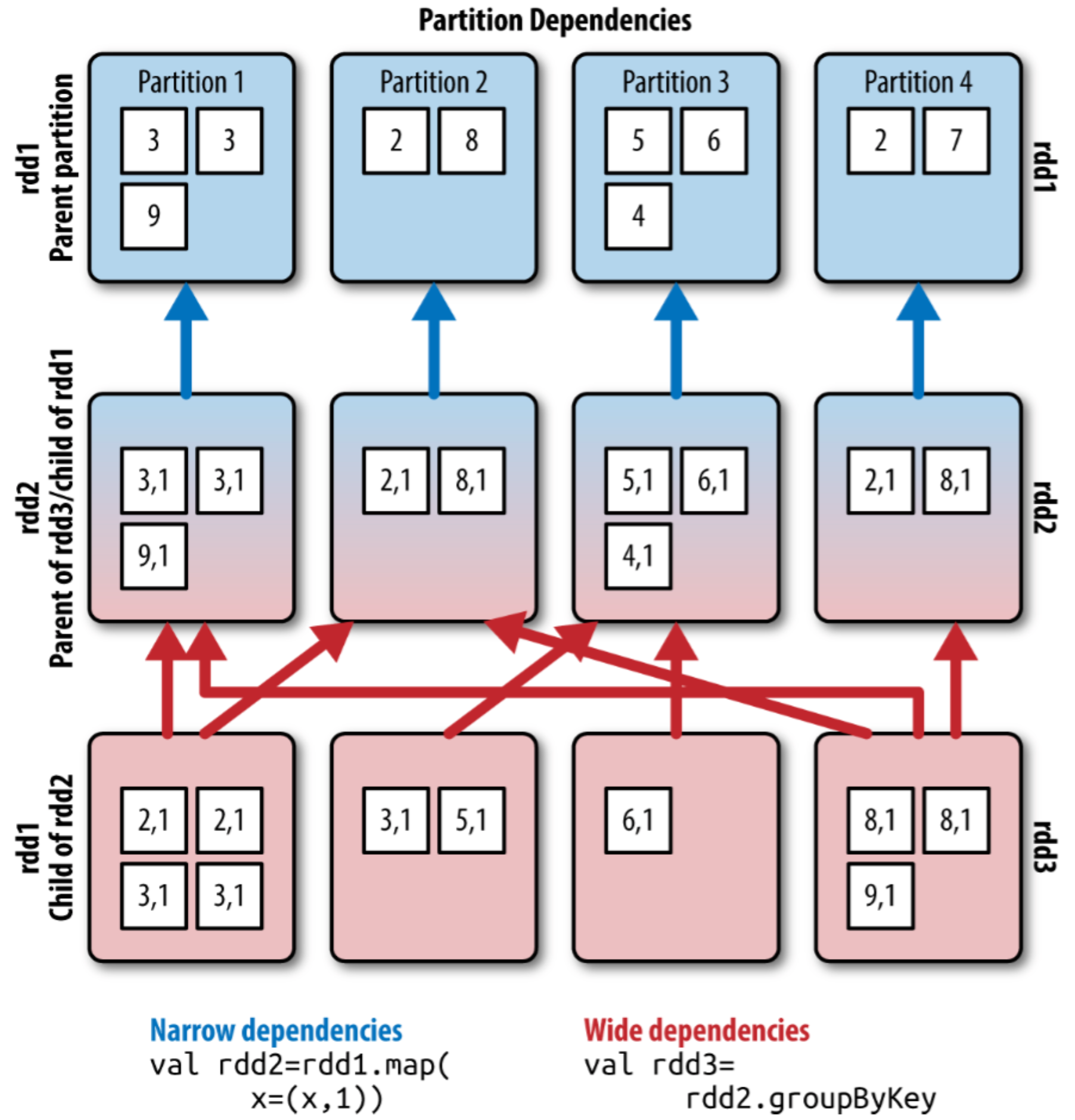
parent RDD的partition只能至多被一个子RDD的partition依赖

【备注】

从parent的角度，可以很好的解释为什么tasks的数目和输出RDD output partition的数目有关，因为task需要在child partitions上计算transformation

图解实例

|  |
| --- |
| *//Narrow dependency. Map the rdd to tuples of (x, 1)*  **val** rdd2 **=** rdd1.map(x **=>** (x, 1))  *//wide dependency groupByKey*  **val** rdd3 **=** rdd2.groupByKey() |



### Implications for Performance

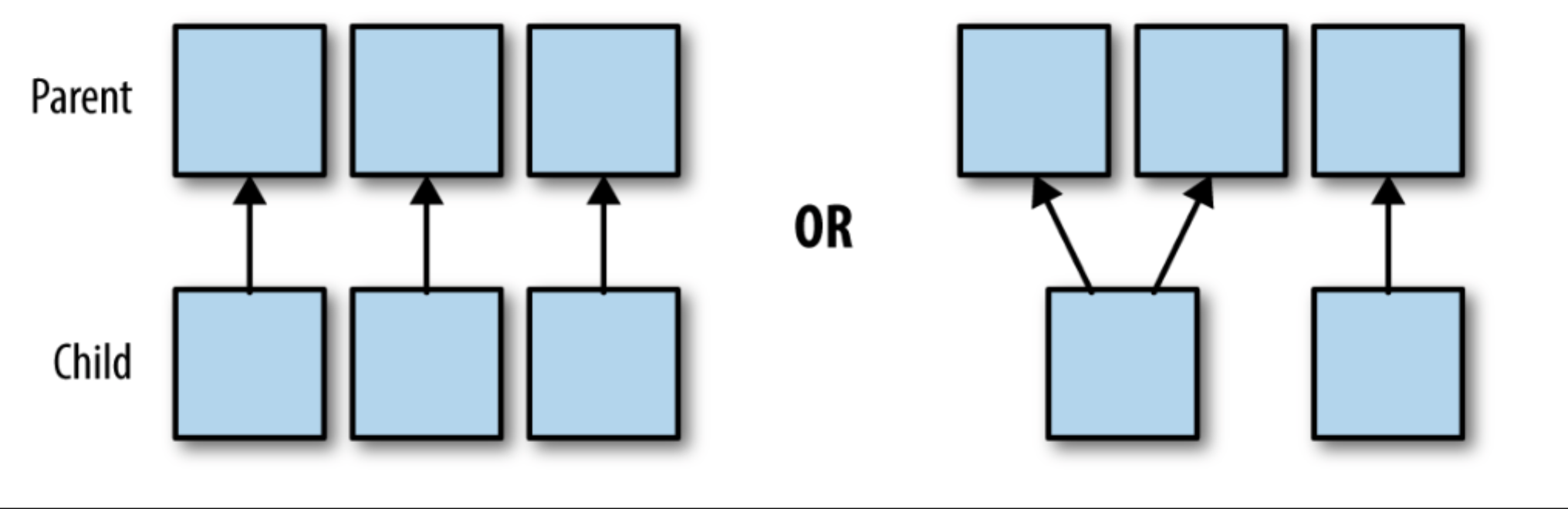
narrow transformations 在同一stage中并发执行

wide transformations 由shuffle切分stage，上行的tasks执行完，下行的tasks才可执行

### Implications for Performance

由于依赖关系，narrow transformations在父partition失败，只有一个子RDD需要重计算，而wide transformations中父partition失败，最坏情况下所有子RDD需要重计算

### The Special Case of coalesce



coalesce用于重分区，当减少分区时，parent partition只被一个child parent依赖，属于窄依赖（虽然改变了RDD的partitions数）。正因task在child的partition中执行，task数和结果集RDD的partition数相等。

当增加分区时恰好相反

## What Type of RDD Does Your Transformation Return?

注意RDD的数据类型，不同Trans操作只能用于对应数据类型的RDD之上，否则有会异常抛出

## Minimizing Object Creation

Garbage collection or “GC” errors

* 导致任务出错（如shullle）
* 即便不会导致job直接挂掉，由于GC需要额外的序列化时间，也会增加任务的时间。

减少object，减小object的size

复用已有的object，利用data structures（例如private type）

### Reusing Existing Objects

Reduce（聚合aggregate）操作，fold折叠操作（foldLeft, fold, foldRight）可在对象复用中得益

【备注】对于其他的例如窄依赖的操作，利用可变对象，修改前面的对象可能会有序列化和结果数据不准确的问题，因为lazy操作，前面的数据可能被后面的计算多次访问利用

示例：

（K-V）对数据，K为instructors，V为pupile的report记录，对于每个instructor计算最长用的word，每个report计算的平均word数，word数中“happy

”单次出现的次数

|  |
| --- |
| **class** MetricsCalculator(  **val** totalWords : Int,  **val** longestWord: Int,  **val** happyMentions : Int,  **val** numberReportCards: Int) **extends** Serializable {   **def** sequenceOp(reportCardContent : String) : MetricsCalculator = {  **val** words = reportCardContent.split(**" "**)  **val** tW = words.length  **val** lW = words.map( w => w.length).max  **val** hM = words.count(w => w.toLowerCase.equals(**"happy"**))   **new** MetricsCalculator(  tW + totalWords,  Math.*max*(longestWord, lW),  hM + happyMentions,  numberReportCards + 1)  }   **def** compOp(other : MetricsCalculator) : MetricsCalculator = {  **new** MetricsCalculator(  **this**.totalWords + other.totalWords,  Math.*max*(**this**.longestWord, other.longestWord),  **this**.happyMentions + other.happyMentions,  **this**.numberReportCards + other.numberReportCards)  }   **def** toReportCardMetrics =  *ReportCardMetrics*(  longestWord,  happyMentions,  totalWords.toDouble/numberReportCards) }  **case class** ReportCardMetrics(  longestWord : Int,  happyMentions : Int,  averageWords : Double)  **def** calculateReportCardStatistics(rdd : RDD[(String, String)] ): RDD[(String, ReportCardMetrics)] ={  rdd.aggregateByKey(**new** MetricsCalculator(totalWords = 0,  longestWord = 0, happyMentions = 0, numberReportCards = 0))(  seqOp = ((reportCardMetrics, reportCardText) =>  reportCardMetrics.sequenceOp(reportCardText)),  combOp = (x, y) => x.compOp(y))  .mapValues(\_.toReportCardMetrics) } |

复用对像

|  |
| --- |
| **class MetricsCalculatorReuseObjects**( **var** totalWords **: Int**, **var** longestWord**: Int**, **var** happyMentions **: Int**,  **var** numberReportCards**: Int**) **extends Serializable** {  **def** sequenceOp(reportCardContent **: String**) **: this.type** = { **val** words **=** reportCardContent.split(" ") totalWords += words.length longestWord **= Math**.max(longestWord, words.map( w **=>** w.length).max) happyMentions += words.count(w **=>** w.toLowerCase.equals("happy")) numberReportCards +=1  **this**  }  **def** compOp(other **: MetricsCalculatorReuseObjects**) **: this.type** = { totalWords += other.totalWords longestWord **= Math**.max(**this**.longestWord, other.longestWord) happyMentions += other.happyMentions  numberReportCards += other.numberReportCards  **this**  }  **def** toReportCardMetrics **= ReportCardMetrics**(  longestWord,  happyMentions,  totalWords.toDouble/numberReportCards)  } |

### Using Smaller Data Structures

Spark内存独占，job中基本类型要比自定义类和对象更少的引起GC，同时Array要比case class 和tuple更有效，array是集合中最有效的，底层是java的array

|  |
| --- |
| **object MetricsCalculator\_Arrays extends Serializable** {  **val** totalWordIndex **=** 0  **val** longestWordIndex **=** 1  **val** happyMentionsIndex **=** 2  **val** numberReportCardsIndex **=** 3  **def** sequenceOp(reportCardMetrics **: Array**[**Int**],  reportCardContent **: String**) **: Array**[**Int**] **=** {  **val** words **=** reportCardContent.split(" ") *//modify each of the elements in the array* reportCardMetrics(totalWordIndex) += words.length reportCardMetrics(longestWordIndex) **= Math**.max(  reportCardMetrics(longestWordIndex),  words.map(w **=>** w.length).max) reportCardMetrics(happyMentionsIndex) += words.count(  w **=>** w.toLowerCase.equals("happy")) reportCardMetrics(numberReportCardsIndex) +=1 reportCardMetrics  }  **def** compOp(x **: Array**[**Int**], y **: Array**[**Int**]) **: Array**[**Int**] **=** { *//combine the first and second arrays by modifying the elements // in the first array*x(totalWordIndex) += y(totalWordIndex) x(longestWordIndex) **= Math**.max(x(longestWordIndex), y(longestWordIndex)) x(happyMentionsIndex) += y(happyMentionsIndex) x(numberReportCardsIndex) += y(numberReportCardsIndex) x  }  **def** toReportCardMetrics(ar **: Array**[**Int**]) **: ReportCardMetrics** = **ReportCardMetrics**(  ar(longestWordIndex),  ar(happyMentionsIndex),  ar(totalWordIndex)/ar(numberReportCardsIndex)  ) }  **def** calculateReportCardStatisticsWithArrays(rdd **: RDD**[(**String**, **String**)] )**: RDD**[(**String**, **ReportCardMetrics**)] **=** {  rdd.aggregateByKey( *//the zero value is a four element array of zeros* **Array**.fill[**Int**](4)(0)  )(  *//seqOp adds the relevant values to the array*  seqOp **=** (reportCardMetrics, reportCardText) **=> MetricsCalculator\_Arrays**.sequenceOp(reportCardMetrics, reportCardText),  *//combo defines how the arrays should be combined*  combOp **=** (x, y) **=> MetricsCalculator\_Arrays**.compOp(x, y)) .mapValues(**MetricsCalculator\_Arrays**.toReportCardMetrics)  } |

## Iterator-to-Iterator Transformations with mapPartitions

mapPartitions iterator已一个partition的records（迭代器代表整个分区中的记录而非单条记录）为单位操作于给给定函数，其中操作函数可以是简单的如string parsing，也可以是复杂的数据操作如二次排序或用户自定义聚合，其他诸如ilter, map, and flatMap可以嵌入mapPartitions一起使用。

为使Spark在将records灵活向瓷片spill，要注意mapPartitions中的函数不能将整个分区中的数据loading（隐式的转为list加载到内存）到内存。

Iterators中有许多方法允许我们写出函数式的transformations，同时允许我们构建自定义iterator来扩充Iterator interface。

Iterator-to-iterator transformation：a transformation directly takes and returns an iterator without forcing it through another collection,

#### What Is an Iterator-to-Iterator Transformation?

Scala iterator 对象不是一个collection，是可one-by-one访问获取（获取但不可改变元素（immutable），只可访问获取一次）集合collection中元素过程的方法。

从某些角度看iterator方法可以定义为RDD的transformations 或者 actions，iterator实际是一个计算指令（evaluation instructions）集而非存储状态集。

例如iterator的next, size, foreach方法访问并计算，更像action,而map、flatMap返回一个新的iterator，实际是一个evaluation instructions集，更像RDD的transformations（返回一个新的RDD）。

但是不同于Spark transformations，iterator transformations是线性执行的，例如，需要将已有的records存储，可以利用iterator之上的filter或者map函数。

#### Space and Time Advantages

Spark流程中使用iterator-to-iterator transformations的主要有益点为：允许spark选择性的spill数据到磁盘

概念上，iterator-to-iterator transformation是一次访问一个元素的计算处理过程。因此，Spark在处理这个procedure时可以批处理记录而不是将整个分区中的数据读到内存亦或是在内存中将整个结果集构建成一个collection而后才能返回它（这个目标结果集），所以这使得spark可以在单executor中处理超大partitions（内存加载不了整个partition）而不会出现内存溢出问题。利用iterators之上定义的methods同时也避免了中介数据结构，减少了垃圾回收

Spark Streaming mapPartitions API以及groupByKey仍多依赖受限于java的Iterable迭代器，因此需要整个数据分区的加载，所以，在groupByKey之后尽量减少iterator-to-iterator transformation的使用

“Goldilocks problem”

a sorted RDD of (value, columnIndex), count) tuples

a list of rank statistics on this partition

返回(value, colum nIndex) pairs

|  |
| --- |
| **package** com.dt.spark.main.HighPormenceSark.ch5  **import** org.apache.spark.{SparkContext, SparkConf}  **import** scala.collection.mutable.ArrayBuffer **import** scala.collection.mutable.HashMap  */\*\*  \* Created by hjw on 17/9/24.  \*/* **object** MapPartionsIter2Iter {   */\*\*  \* This sub routine returns an Iterator of (columnIndex, value) that correspond  \* to one of the desired rank statistics on this partition.  \*  \* Because in the original iterator, the pairs are distinct  \* and include the count, one row of the original iterator could map to multiple  \* elements in the output.  \*  \* i.e. if we were looking for the 2nd and 3rd element in column index 4 on  \* this partition. And the head of this partition is  \* ((3249.0, 4), 23)//这个是原始数聚合好的  \* (i.e. the element 3249.0 in the 4 th column appears 23 times),  \* then we would output (4, 3249.0) twice in the final iterator.  \* Once because 3249.0 is the 2nd element and once because it is the third  \* element on that partition for that column index and we are looking for both the  \* second and third element.  \*  \** **@param valueColumnPairsIter** *passed in from the mapPartitions function.  \* An iterator of the sorted:  \* ((value, columnIndex), count) tupples.  \** **@param targetsInThisPart** *- (columnIndex, index-on-partition pairs). In the above  \* example this would include (4, 2) and (4,3) since we  \* desire the 2nd element for column index 4 on this  \* partition and the 3rd element.  \** **@return** *All of the rank statistics that live in this partition as an iterator  \* of (columnIndex, value pairs)  \*/   // MapPartitions example without an iterator-to-iterator transformation  // [1] loop through the iterator,  // [2] store the running totals in a hashMap,  // [3] and build a new collection of the elements we want to keep using an array buffer  // [4] then convert the array buffer to an iterator, shown later.* **def** withArrayBuffer(valueColumnPairsIter : Iterator[((Double, Int), Long)], targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {  **val** columnsRelativeIndex: Predef.Map[Int, List[Long]] = targetsInThisPart.groupBy(\_.\_1).mapValues(\_.map(\_.\_2))  *// The column indices of the pairs that are desired rank statistics that live in  // this partition.* **val** columnsInThisPart: List[Int] = targetsInThisPart.map(\_.\_1).distinct  *// A HashMap with the running totals of each column index. As we loop through  // the iterator, we will update the hashmap as we see elements of each  // column index.* **val** runningTotals : HashMap[Int, Long]= **new** HashMap()  runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap  *//we use an array buffer to build the resulting iterator* **val** result: ArrayBuffer[(Int, Double)] = **new** ArrayBuffer()  valueColumnPairsIter.foreach {  **case** ((value, colIndex), count) =>  **if** (columnsInThisPart contains colIndex) {  **val** total = runningTotals(colIndex)  *//the ranks that are contained by this element of the input iterator.  //get by filtering the* **val** ranksPresent = columnsRelativeIndex(colIndex)  .filter(index => (index <= count + total) && (index > total))  ranksPresent.foreach(r => result += ((colIndex, value)))  *//update the running totals.* runningTotals.update(colIndex, total + count)  } }   *//convert* result.toIterator  }   **def** main(args: Array[String]) {  **val** conf = **new** SparkConf()   conf.setAppName(**"test"**)  conf.setMaster(**"local"**)   **val** sc = **new** SparkContext(conf)   *//==========================================  /\*  构建RDD  [1]从scala数据集构建RDD: parallelize()  \*/* **val** KVPairRDD = sc.parallelize(*List*(  (1.0, 1), (4.0, 2),  (1.0, 1), (5.0, 2),  (1.0, 1), (6.0, 2),  (2.0, 1), (7.0, 2),  (3.0, 1), (8.0, 2)  )  )  **val** mapRDD = KVPairRDD.map(iter => (iter,1L)).reduceByKey(\_ + \_).sortBy(\_.\_1.\_1)  mapRDD.foreach(wordNumberPair => *println*(wordNumberPair.\_1 + **" : "** + wordNumberPair.\_2))  *// (1.0,1) : 3  // (2.0,1) : 1  // (3.0,1) : 1  // (4.0,2) : 1  // (5.0,2) : 1  // (6.0,2) : 1  // (7.0,2) : 1  // (8.0,2) : 1* **val** iterRDD = mapRDD.toLocalIterator  **val** iter = *withArrayBuffer*(iterRDD,*List*((1,1L),(1,2L),(1,4L),(1,5L)))  **for**(it <- iter ){  *println*(it.\_1 + **"--"** + it.\_2)  }  *// 1--1.0  // 1--1.0  // 1--2.0  // 1--3.0* sc.stop()  }  } |

## Set Operations

由于RDD中数据可以有重复，intersection和subtract 数据上的概念不一致

如 A中有重复值，A. subtract(B)以后A的size 可能少于两者的size差值

|  |
| --- |
| **val** a **= Array**(1, 2, 3, 4, 4, 4, 4)  **val** b **= Array**(3, 4)  **val** rddA **=** sc.parallelize(a)  **val** rddB **=** sc.parallelize(b)  **val** rddC **=** rddA.subtract(rddB) assert(rddC.count() < rddA.count() - rddB.count()) |

## Reducing Setup Overhead

当需要每个worker或每个partition构建初识化时（构建数据库链接，构建随机数生成器），对于transformation 可以用mapPartitions

例如：在每个分区产生伪随机生成器

|  |
| --- |
| rdd.mapPartitions{iter=>  //每个partiton中只构建一个RNG（randomNumberGenerator）  val r = new Random()  itr.filter(x=>r.nextInt(10) == 0)  } |

### Shared Variables

两种共享变量：

broadcast variables 广播变量

accumulator 累加器

1. broadcast variables：在driver program中被写，在executors中被读取
2. accumulator在executors中被写，在driver中被读取

### Broadcast Variables

broadcast variables是将driver端的一个本地local 值已只读read-only副本的形式分发到不同的机器上machine,而不是在每个task中搬运一个新的副本。Map join 的实现是其中一个很多的例子。

1. broadcast variables必须是本地的local、可序列化的serializable的值，而不是RDDs或其他distributed data structres
2. 当broadcast variables 不再需要时可以显式地去除unpersist()

例1：将一个非法ids的hashset分发，在不同节点直接本地化过滤非法值

|  |
| --- |
| **val** invalid = HashSet() ++ invalidPandas **val** invalidBroadcast= sc.broadcast(invalid) input.filter{panda => !invalidBroadcast.value.contains(panda.id)} |

例2：在每个worker上构建一个随机数生成器

|  |
| --- |
| **class** LazyPrng{  @transient **lazy val** *r* = **new** Random() } **def** customSampleBroadcast[T:ClassTag](sc:SparkContext,rdd:RDD[T]):RDD[T] = { **val** bcastprng = sc.broadcast(**new** LazyPrng()) rdd.filter(x=>bcastprng.value.r.nextInt(10)==0) } |

### Accumulators [待补]

## Reusing RDDs

三种方式:Persiting、caching、checkpoint

会打断pipelining，所以要权衡与重计算间的代价

### Iterative computations

其中validationSet在map中多次应用

|  |
| --- |
| **val** testSet: Array[RDD[(Double, Int)]] =  *Array*(validationSet.mapValues(\_ + 1), validationSet.mapValues(\_ + 2), validationSet)  validationSet.persist() *//persist since we are using this RDD several times* **val** errors = testSet.map( rdd => {  Math.rmse(rdd.join(validationSet).values) }) |

【备注】

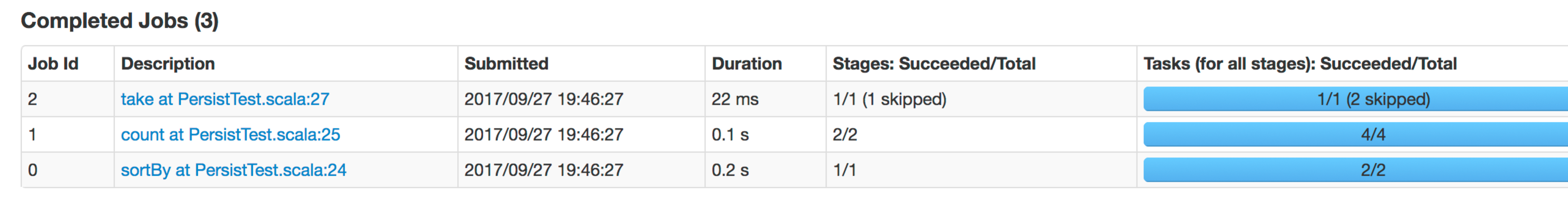
Ckeckpoint是将RDD写到外部存储中，打破了RDD间的依赖，单分区关系不变。

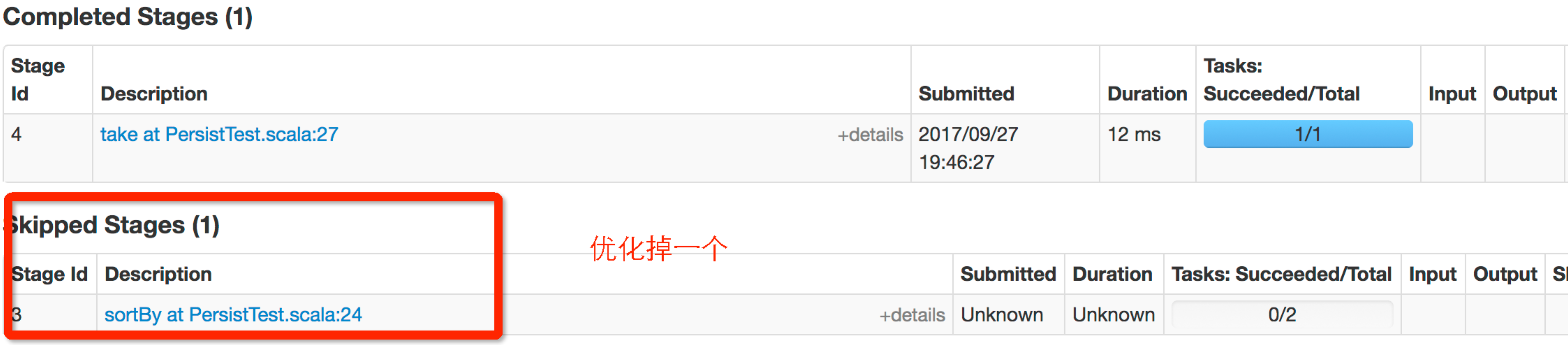
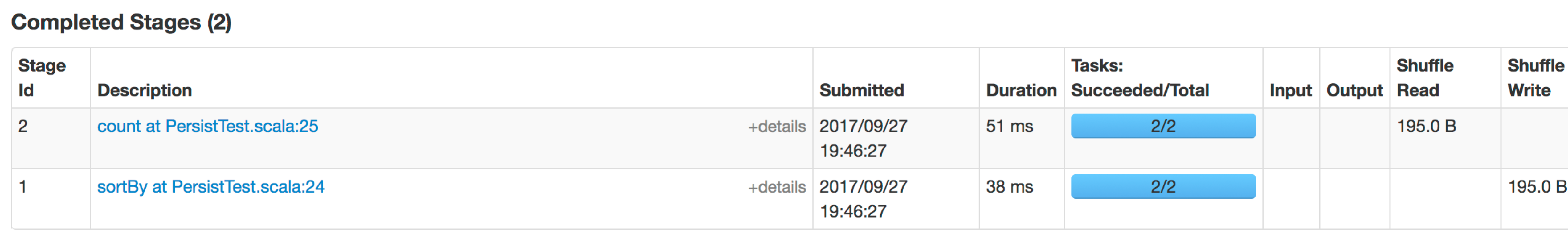
Persisted只能在同一个application中应用

在applications间复用，可用用同一个directory中的 checkpointing

### Multiple actions on the same RDD

|  |
| --- |
| **val** sorted **=** rddA.sortByKey() //触发一次job  **val** count **=** sorted.count() //触发一个job，调用sort  **val** sample**: Long** = count / 10  sorted.take(sample.toInt) //触发一个job，调用sort |





|  |
| --- |
| **val** sortedRDD = listRDDExample.sortBy(f=>f)*//触发一个job sortedRDD.persist(StorageLevel.MEMORY\_ONLY)* **val** cnt = sortedRDD.count() **val** sample:Long = cnt/10 sortedRDD.take(sample.toInt) |

### If the cost to compute each partition is very high

在窄依赖trans序列较长的场景中，即便不存在RDD复用，但由于操作线过长，增加了GC fail的概率，在关键点persist中间结果，有时对稳定性有益，但是一般情况下不建议打断窄依赖trans，这打断了pipline

### Deciding if Recompute Is Inexpensive Enough

Perisity在内存中要考虑内存OOM（和正常的计算公用JVM），写外部有存储空间和磁盘I/O的性能问题需要考虑，经验是对于相对简单的操作重计算优于复用RDD，在防止触发重复read操作（如读外部文件，大结果集RDD）、防止大量迭代运算时有效

1. 加上persist与不加persist对比，是否加速了程序的运行
2. 程序GC或者OOM错，尝试添加checkpointing or persisting off\_heap ,可能会使程序正常运行。
3. 相反如果程序程序GC或者OOM错，尝试去除 persisting切换成checkpointing or off\_heap
4. 建议将persistence level 设置成变量，便于调试

### Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files

#### Persist and cache

storage levels

useDisk, useMemory, useOfHeap, deserialized, and replication

useDisk:

是在标识中有DISK，例如MEMORY\_AND\_DISK ,当内存有限时选用，注意读写磁盘的时间消耗与重计算的权衡

useMemory

cache() 是persist() 相等的形式，即无参数（default storage-level MEM ORY\_ONLY ）

useOfHeap

off\_heap主要是外存，不常用

deserialized

序列化的java对象形式存储

replication

备份数

#### Checkpointing

Checkpointing writes the RDD to an external storage system such as HDFS or S3, and—in contrast to persisting—forgets the RDD’s lineage

总的来说，

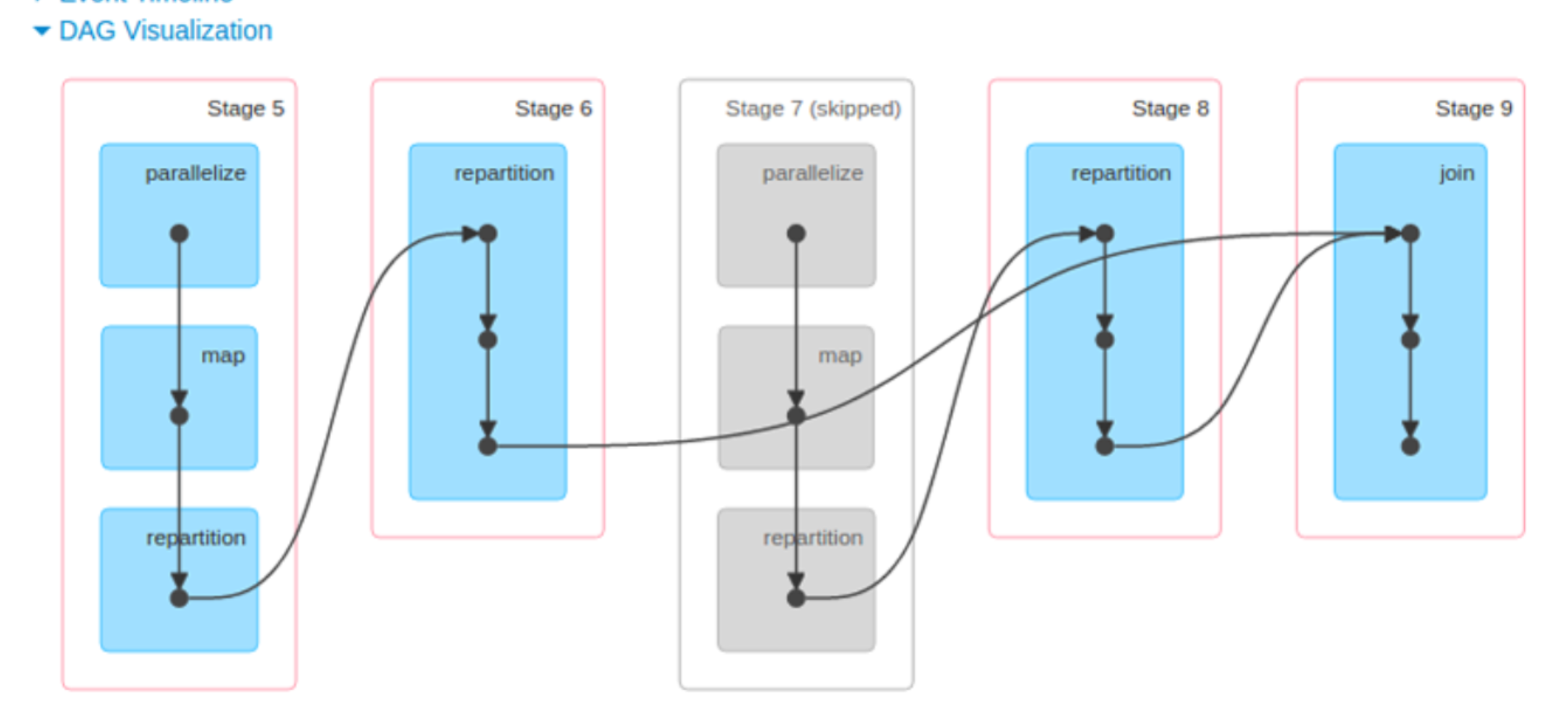
1. 当job慢时建议用persist
2. 当jod因OOM失败时，checkpoint减少失败而不消耗executor的内存
3. 当job因network错误和抢占（noise cluster）失败，checkpoint将long-running的job切成小segments减少失败的概率

|  |
| --- |
| def findQuantilesWithCustomStorage(valPairs: RDD[((Double, Int), Long)], colIndexList: List[Int],  targetRanks: List[Long],  storageLevel: StorageLevel = StorageLevel.MEMORY\_AND\_DISK,  checkPoint : Boolean, directory : String = "")  : Map[Int, Iterable[Double]] = {  val n = colIndexList.last + 1  val sorted = valPairs.sortByKey()  //将sorted 消耗大的排序RDD 存储  //合理利用参数便于调试  if (storageLevel != StorageLevel.NONE) {  sorted.persist(storageLevel)  }  //合理利用参数便于调试  if (checkPoint) {  sorted.sparkContext.setCheckpointDir(directory) sorted.checkpoint()  }  val partitionColumnsFreq = getColumnsFreqPerPartition(sorted, n)  val ranksLocations = getRanksLocationsWithinEachPart(  targetRanks, partitionColumnsFreq, n)  val targetRanksValues = findTargetRanksIteratively(sorted, ranksLocations) targetRanksValues.groupByKey().collectAsMap()  } |

#### Shuffle files

Regardless of a persist or checkpoint call, Spark does write some data to disk dur‐ ing a shuffle. These files are called “shuffle files” and they usually contain all of the records in each input partition sorted by mapper

一些shuffle的中间文件也会被缓存在work中，当重计算时直接利用，这些不能认为的决定，单从UI中可以看出哪些stage被忽略不计算了



# Chapter 6 Working with Key/Value Data

PairRDDFunctions :join和聚合

OrderedRDDFunctions: sort

KV pairRDD一般涉及wide Trans，所以值得仔细学习优化

KV pairRDD一般会导致：

[1]Out-of-memory errors in the driver

[2]Out-of-memory errors on the executor nodes

[3]Shuffle failures

[4]“Straggler tasks” or partitions, which are especially slow to compute

其中：

memory errors in the driver  一般是由action导致，在会面章节细聊

out of memory on the executors, shuffles, and straggler tasks  一般是wide Transde 的PairRDDFunctons或者OrderedRDDFunctions的shuffles导致，对此从两方面优化“shuffle less”和“shuffle better”

Shuffle less often

[1] preserve partitioning across narrow transformations to avoid reshuffling data

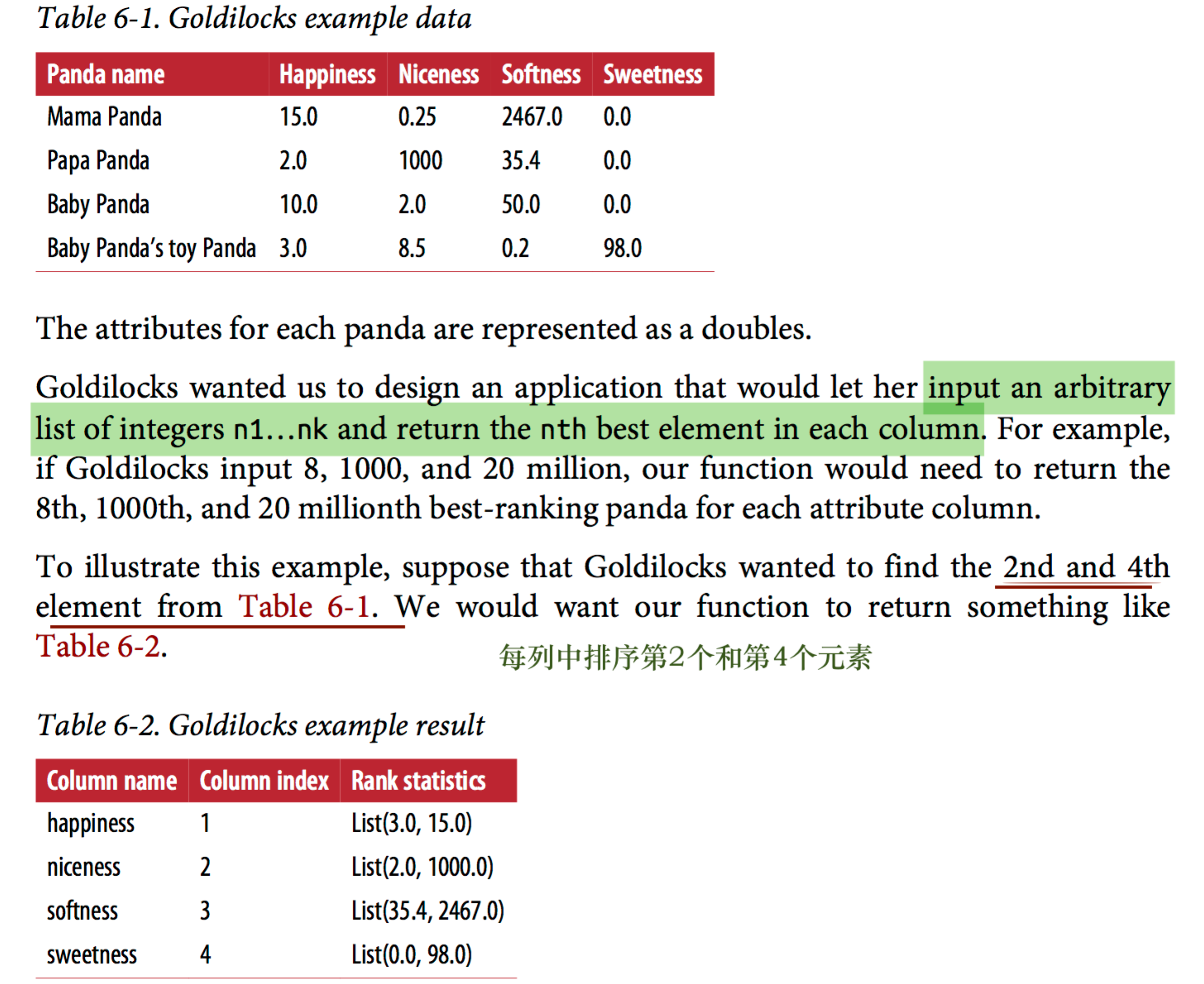
[2] use the same partitioner on a sequence of wide transforma‐ tions. This can be particularly useful to avoid shuffles during joins and to reduce the number of shuffles required to compute a sequence of wide transformations

[3] leveraging custom partitioners  to distribute the data most effectively for down‐ stream computations as well as how to push computational work into the shuffle stage to make a complicated computation more efficient

Shuffle better

reduceByKey and aggregateByKey 是**map-side**的reductions,不需要将一个key的所有记录loading到内存中，以此避免executors中的内存错误，加快wide transformations,尤其是aggregation类操作

## The Goldilocks Example



用SQL 不支持 需要UDF和UDAF 很累赘，不好实现

### Goldilocks Version 0: Iterative Solution

对每一列排序打标

|  |
| --- |
| **package** com.dt.spark.main.HighPerformenceSpark.ch6 **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.rdd.RDD **import** org.apache.spark.sql.DataFrame **import** org.apache.spark.sql.hive.HiveContext **import** org.apache.spark.{SparkContext, SparkConf} */\*\*  \* Created by hjw on 17/9/30.  \*/* **object** GoldilocksV0 {  *//每列不依赖其他的列,可以设法并行计算  //将数据转化成一个List[K-V],K表示column对每个key并行计算* **def** findRandStatics(  dataFram: DataFrame,  ranks: List[Long]  ): Map[Int, Iterable[Double]] = {  *require*(ranks.forall(\_ > 0))  **val** numOfColumns = dataFram.schema.length  **var** i = 0  **var** resultMap = *Map*[Int,Iterable[Double]]()  *//对每列排序打标过滤出目标记录* **while**(i < numOfColumns){  *//取出第列* **val** col = dataFram.*rdd*.map(row=>row.getDouble(i))  **val** sortedColWithIdex:RDD[(Double,Long)] = col.sortBy(v=>v).zipWithIndex()  *//过滤出目标记录* **val** ranksOnly = sortedColWithIdex.filter{  *//zipWithIndex的序号是从0开始的* **case** (colVaule,index)=>ranks.contains(index + 1)  }.keys  **val** list = ranksOnly.collect()  resultMap += (i+1 ->list)  i += 1  }  resultMap  }   **def** main(args: Array[String]) {   **val** conf = **new** SparkConf()  conf.setAppName(**"test"**)  conf.setMaster(**"local"**)   **val** sc = **new** SparkContext(conf)   *//设置日志级别* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*WARN*)  Logger.*getLogger*(**"org.apache.spark.sql"**).setLevel(Level.*WARN*)   **val** sqlContext = **new** HiveContext(sc)  *//加载json文件* **val** table = sqlContext.read.json(**"./src/main/scala/com/dt/spark/main/HighPerformenceSpark/srcFile/GoldilocksData.json"**).  select(**"Happiness"**, **"Niceness"**, **"Softness"**, **"Sweetness"**)   *//table.show()  // +---------+--------+---------+--------+---------+  // |Happiness|Niceness|Pandaname|Softness|Sweetness|  // +---------+--------+---------+--------+---------+  // | 15.0| 0.25| Mama| 2467.0| 0.0|  // | 2.0| 1000.0| Papa| 35.4| 0.0|  // | 10.0| 2.0| Baby| 50.0| 0.0|  // | 3.0| 8.5| Cacy| 0.2| 98.0|  // +---------+--------+---------+--------+---------+* **val** res = *findRandStatics*(table,*List*(2,4))  res.foreach(*println*(\_))  *// (1,WrappedArray(3.0, 15.0))  // (2,WrappedArray(2.0, 1000.0))  // (3,WrappedArray(35.4, 2467.0))  // (4,WrappedArray(0.0, 98.0))* sc.stop()  } } |

### How to Use PairsRDDFunctions

#### 