

Spark技术及其应用开发

吴建军

目录

- Spark 简介
- RDD 介绍
- Spark 核心机制
- SparkUI 简介

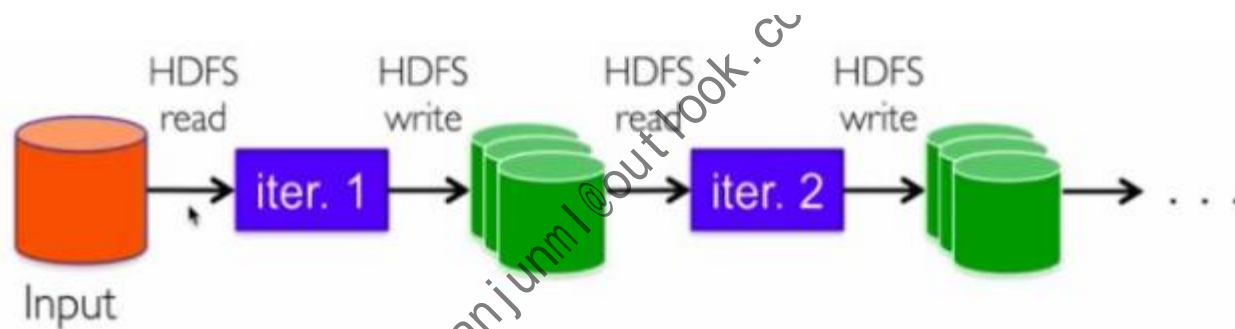
wujianjunml@outlook.cn

Spark简介

wujianjunml@outlook.com

Hadoop MapReduce的缺点

- 一个job只能有Map和Reduce两个阶段，
- job之间基于磁盘进行数据交换，不适合迭代计算，



- 框架多样，组合使用麻烦，离线用MR/hive/pig，实时用Storm。

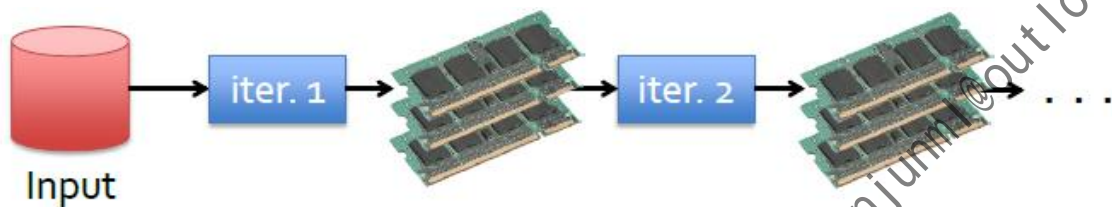
Spark发展历史

- 2009年始于UC Berkeley AMPLab,
- 2010开源,
- 2013年移交给Apache ,
- 2014成为Apache顶级项目并发布1.0,
- 2016年发布1.6,
- 2018年发布2.3,



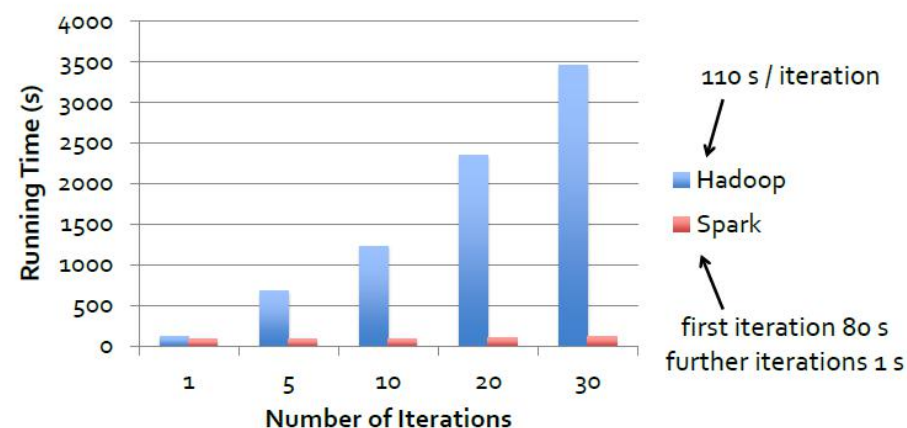
Spark优势

- 速度快:
 - 基于内存数据交换,
 - DAG描述处理过程, 整体优化,



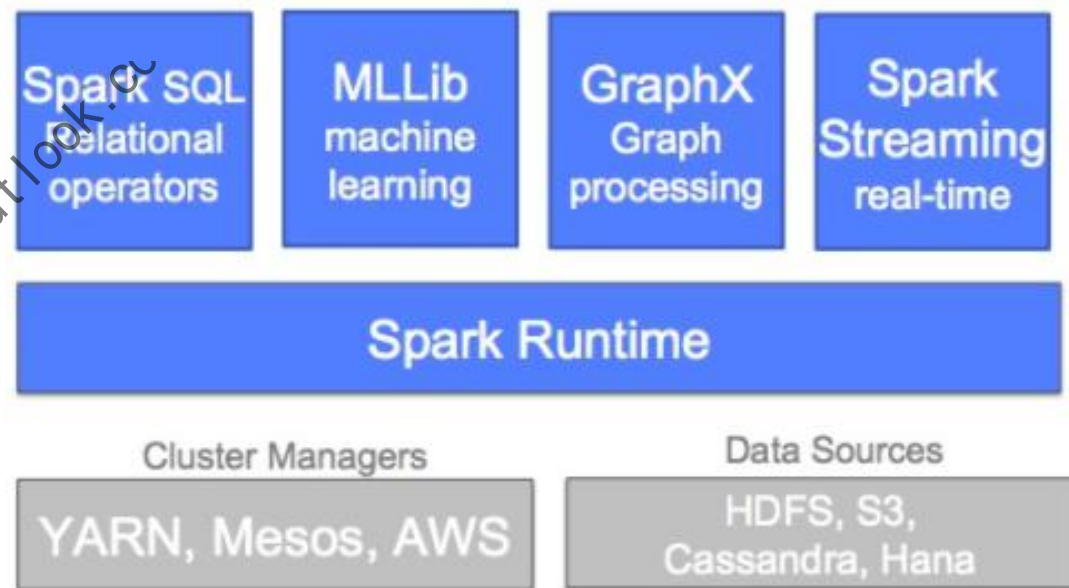
- 易于编程:
 - 支持 scala, java, python语言,
 - 函数式编程,
 - 80多个高级算子,

Logistic Regression Performance



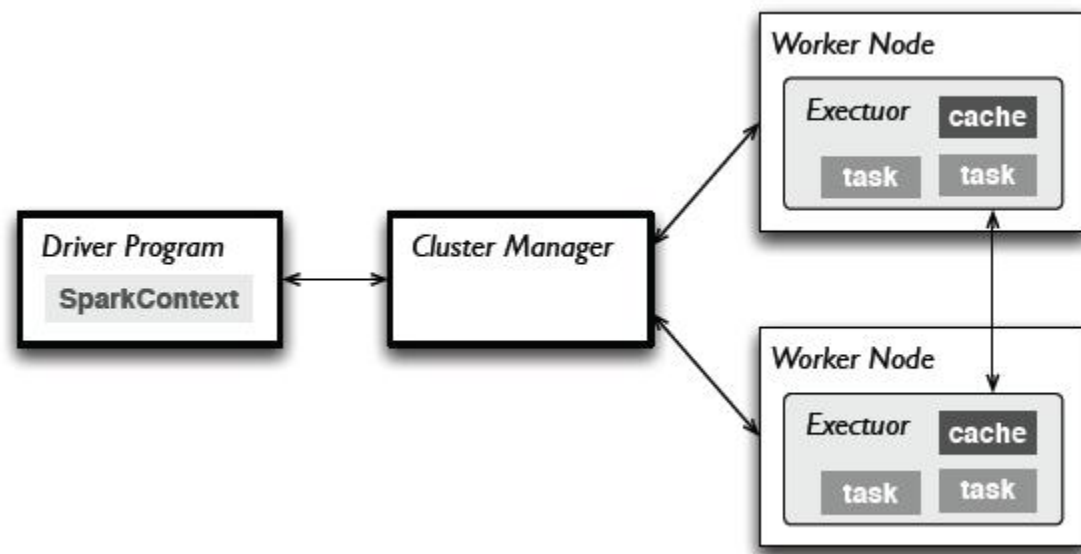
Spark优势

- 组件丰富，可以一站式解决不同场景：
 - Spark SQL 提供SQL功能，
 - Spark Streaming提供近实时计算，
 - MLLib提供机器学习库，
 - GraphX提供图计算，
- 兼容多种环境：
 - 可以运行于多种资源调度系统，比如standalone, YARN, Mesos, ...
 - 可以读写多种存储系统，比如HDFS, Hive, HBase,...



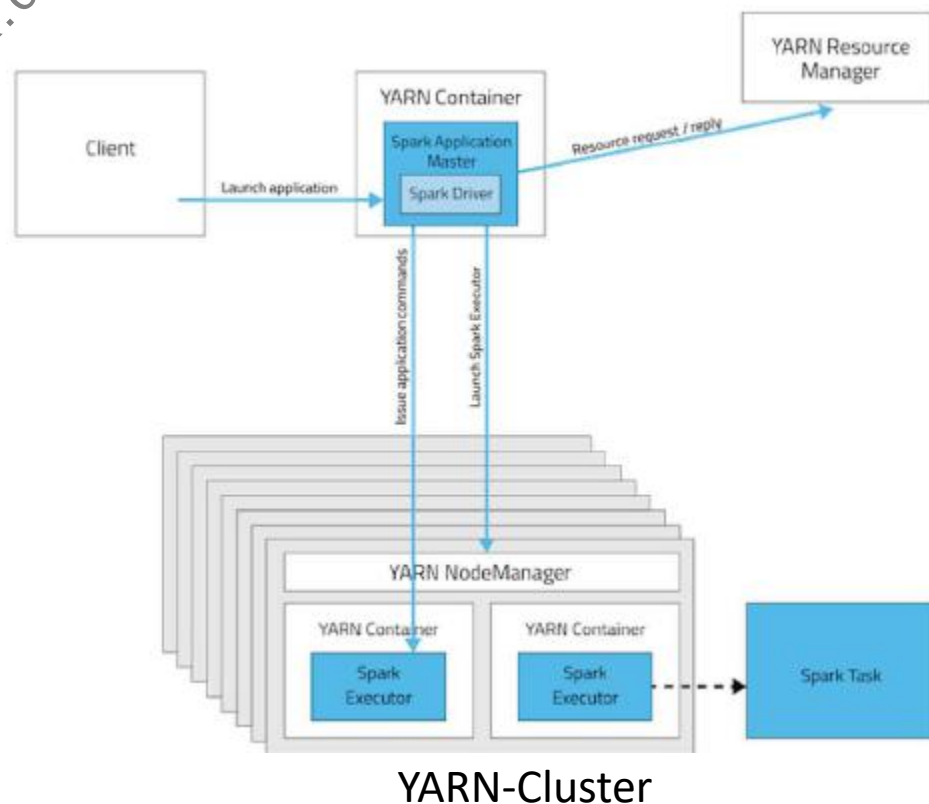
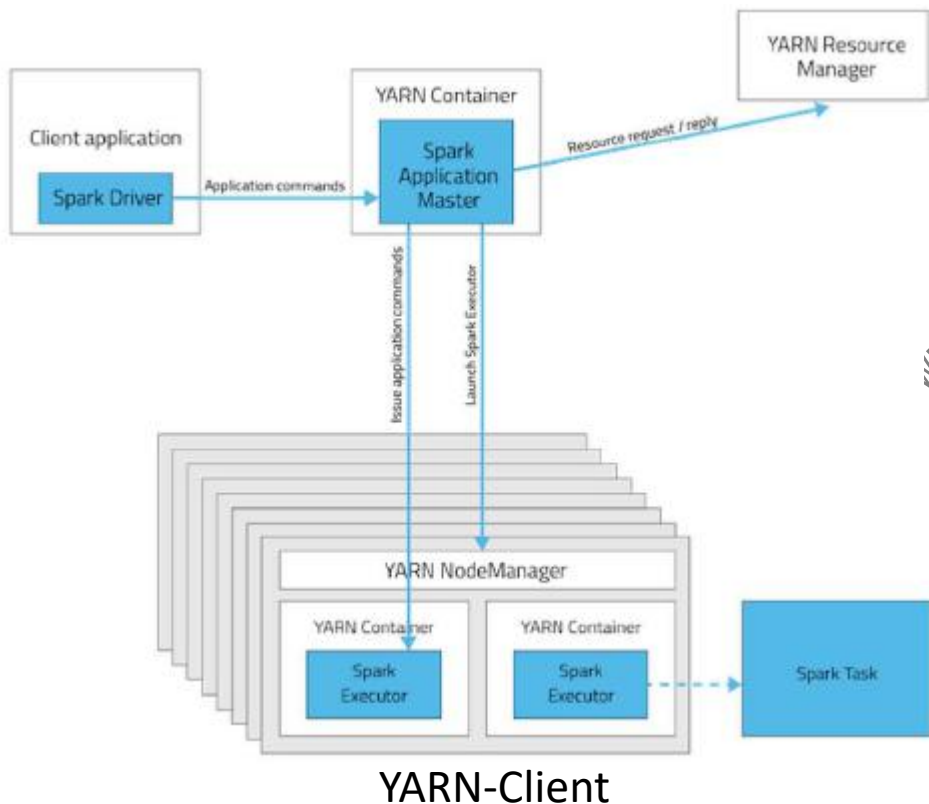
Spark架构简介

- Spark整个架构中有三个角色,
 - Driver 负责调度任务,
 - Cluster Manager 负责分配资源(executor),
 - Executor 负责执行计算任务(Task)的进程,
- 任务执行过程:
 - 1. 用户在driver提交任务(application),
 - 2. cluster manager分配资源(executor),
 - 3. 每个executor启动task执行任务,



Spark架构简介

- Spark 可以运行在Yarn上，并且支持两种模式：
 - YARN-Client模式，Driver在客户端本地运行，
 - YARN-Cluster模式，Driver运行在Yarn的Application Master中，



RDD详解

RDD定义

- RDD全称Resilient Distributed Datasets。
- RDD是对分布式数据集及其操作的抽象。
- Spark中所有操作都基于RDD。

Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing

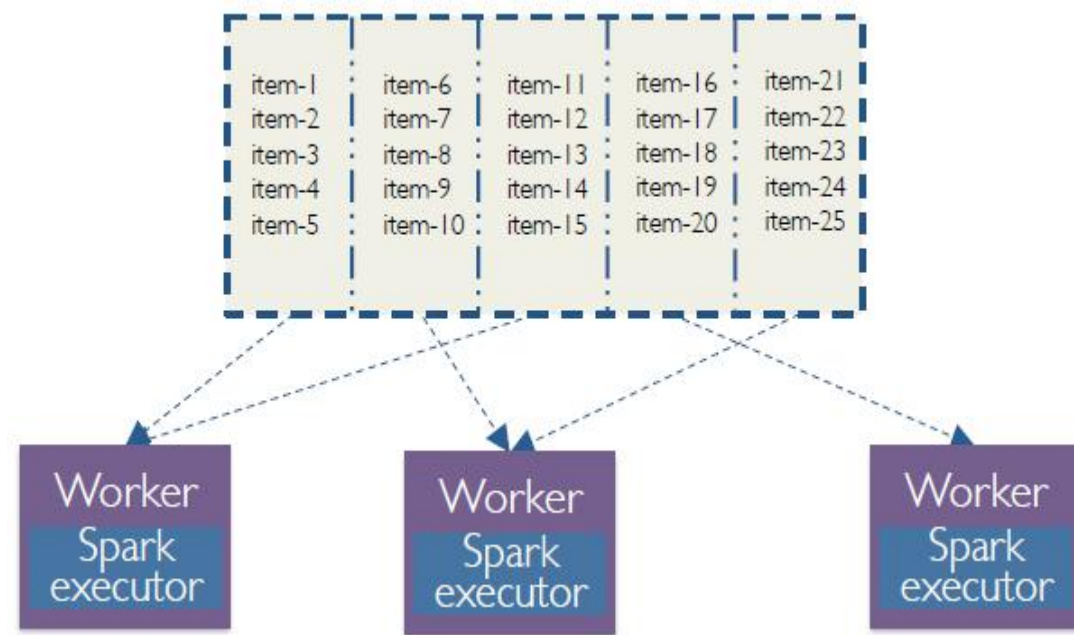
[M Zaharia](#), [M Chowdhury](#), [T Das](#), [A Dave](#), [J Ma...](#) - Proceedings of the 9th ..., 2012 - dl.acm.org

Abstract We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on ...

☆ 99 被引用次数 : 3027 相关文章 所有 80 个版本

数据分区

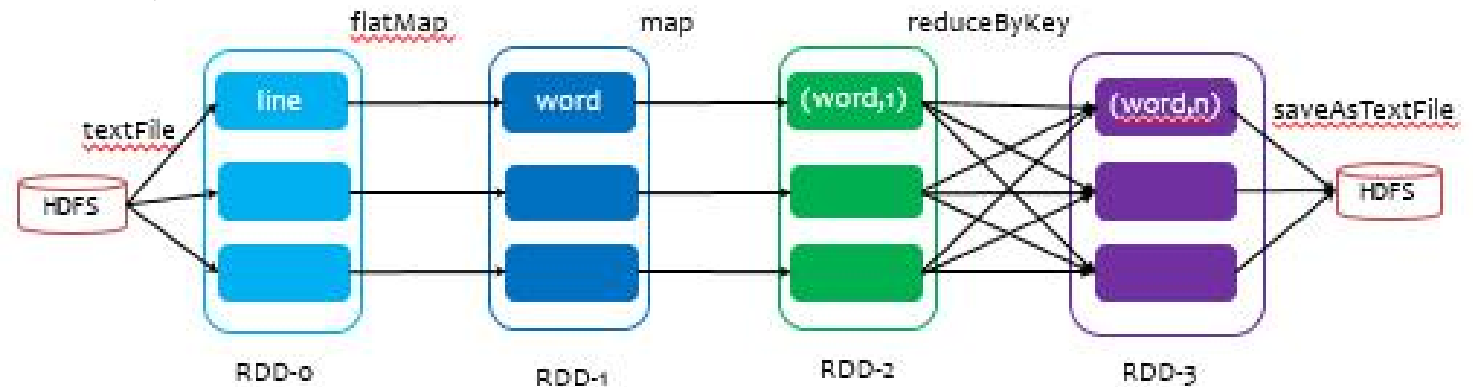
- 整个数据集划分为成一个个分区。
- 不同分区可以存储在不同节点上。
- 每个分区由多条数据记录组成。
- 从HDFS读入数据时，每个block对应一个分区。
- 可以通过Spark更改数据分区布局。
- 数据分区对性能至关重要。



Hello World

```
import org.apache.spark.{SparkContext, SparkConf}
```

```
object WordCount {  
  def main(args: Array[String]) {  
    if (args.length < 2) {  
      System.err.println("Usage: WordCount <inputfile> <outputfile>")  
      System.exit(1)  
    }  
    val conf = new SparkConf().setAppName("WordCount")  
    val sc = new SparkContext(conf)  
    val result = sc.textFile(args(0))  
      .flatMap(line => line.split("\\s+"))  
      .map(word => (word, 1))  
      .reduceByKey(_ + _)  
    result.saveAsTextFile(args(1))  
    sc.stop()  
  }  
}
```

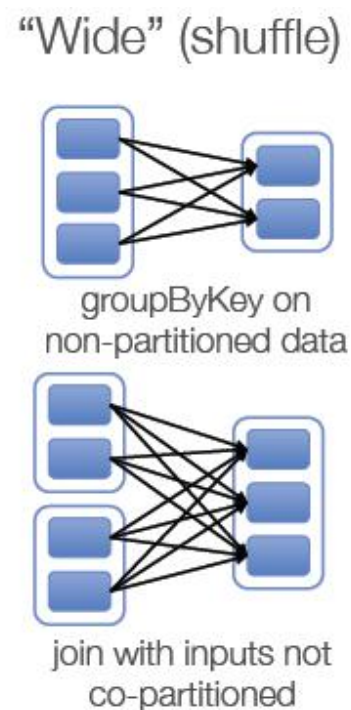
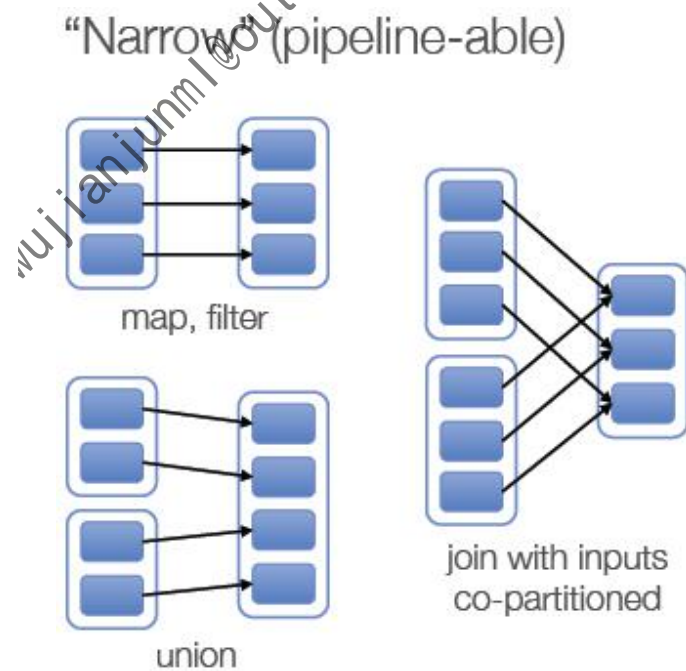


RDD特点

- 每个RDD由5部分组成：
 - partitions(数据分区列表),
 - dependencies(依赖的父RDD),
 - compute(每个分区的计算函数),
 - partitioner(分区函数, 可选),
 - preferred locations(每个分区的本地化计算节点, 可选),
- 创建后不可变,
- 支持 transformation和action两类操作,
- 跟踪血缘(lineage)关系以便失败重算,
- 可将数据缓存于内存或磁盘,

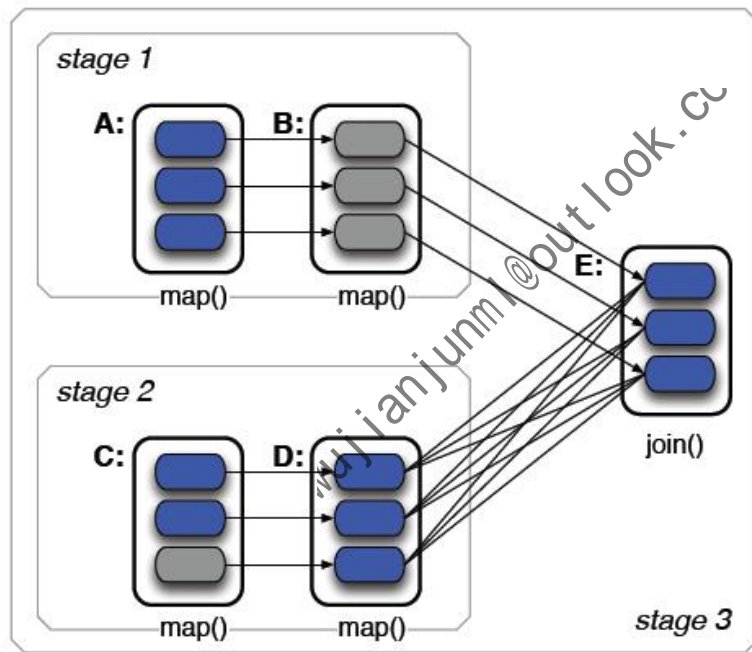
RDD依赖

- 每个RDD的transformation都会生产一个新的RDD，进而形成依赖关系。
- 依赖有两种类型：
 - 窄依赖：每个父RDD的分区最多被一个子RDD的分区使用。
 - 宽依赖：每个父RDD的分区会被多个子RDD的分区使用。
- 窄依赖的特点：
 - 窄依赖计算无需移动数据，可以在一个计算节点上完成。
 - 窄依赖计算并行化和容错容易。
- 宽依赖的特点：
 - 跨依赖计算需要执行Shuffle，在节点之间移动数据。
 - 宽依赖计算的并行和容错代价高。



RDD依赖

- 根据RDD之间是否为宽依赖将计算依赖划分成Stage:



- 每个RDD的action才会触发计算被执行, transformation仅仅是记录操作。

RDD API

- 典型API如下:

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

广播变量与累加器变量

- 广播变量：

- 有些数据，可能会被多个分区的计算函数读入，并且很小，可以采用广播变量来修饰此类数据。Spark运行时会把广播变量的内容发到各个节点，并保存下来。

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

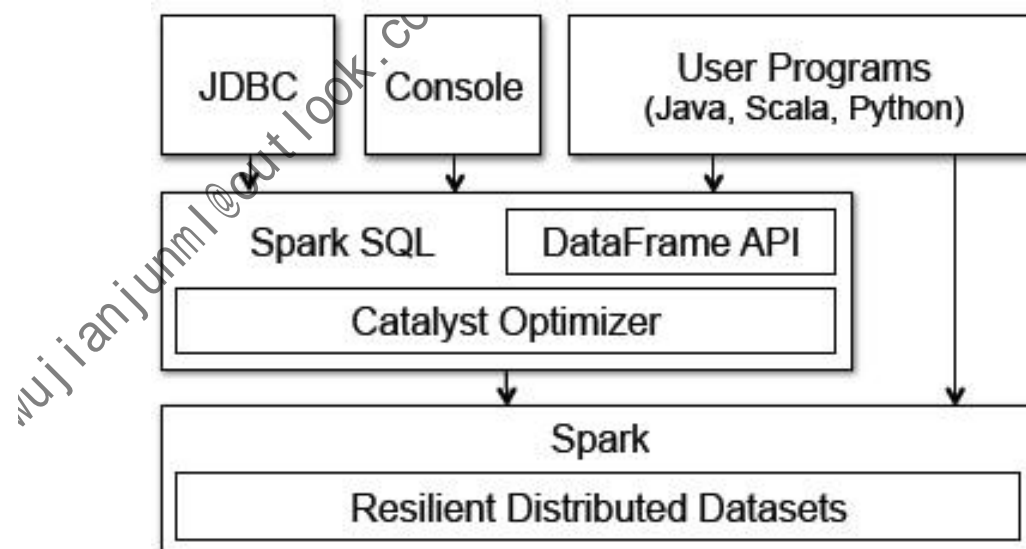
- 累加器变量：

- Spark也会把累加器发送到各个节点，它能保证+=正确地并行计算。每一个节点只能访问和操作其自己本地的累加器，全局累加器则只允许driver访问。

```
scala> val accum = sc.accumulator(0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
scala> accum.value
res2: Int = 10
```

RDD 扩展

- RDD有两个重要的扩展：
 - DataFrame: RDD加上schema, 类似关系数据库中的表, 支持SQL类API, 是Spark SQL的重要组成部分。



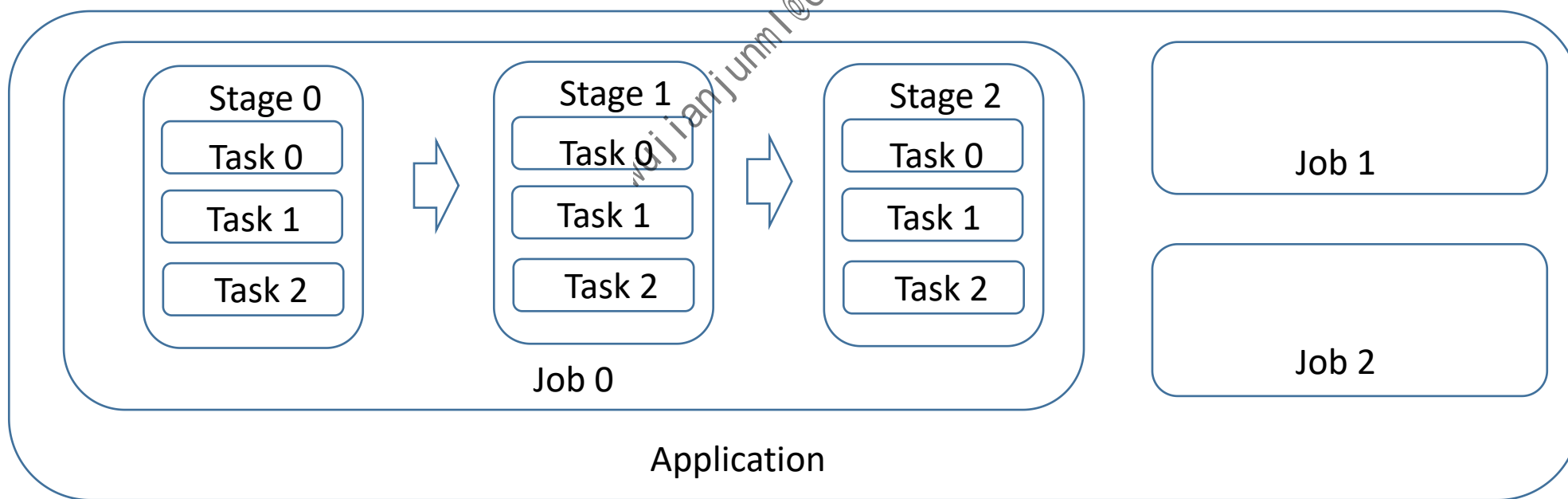
- Datasets: 2.0开始提供, 扩展自DataFrame, 强类型, 支持无须反序列即可排序和Suffle。

Spark 核心机制

wujianjunm1@cs.hk.hk

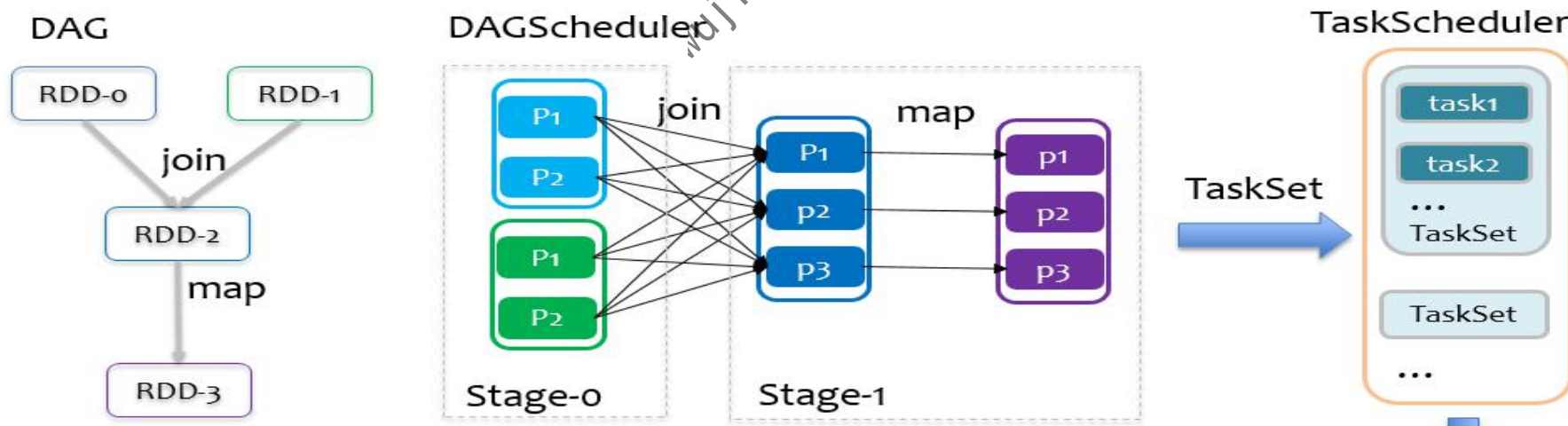
Spark作业调度

- 一个Spark任务分4个层次：
 - Application：指的是用户编写的整个Spark应用程序，由多个job组成。
 - Job：每个RDD action操作触发一个job，包含该action所依赖的全部操作。
 - Stage：每个Job根据Shuffle被拆分成多个Stage。
 - Task：任务最小执行单元，一般来说，RDD有多少个分区就会有多个Task。



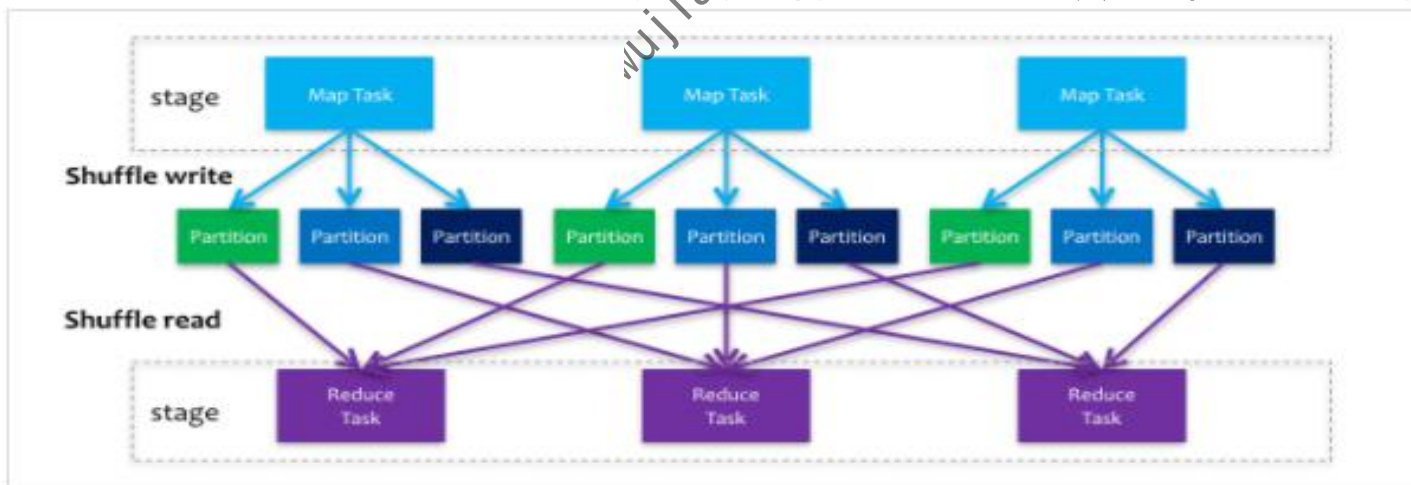
Spark作业调度

- Spark作业调度有两个核心类:
 - DAGScheduler:
 - 分析RDD之间的依赖关系(DAG), 拆分Stage。
 - 生成Task, 并提交给TaskScheduler。
 - 监控Task的执行状态。
 - TaskScheduler:
 - 根据策略分配执行Task的资源。
 - 将Task的执行状态传给DAGScheduler。



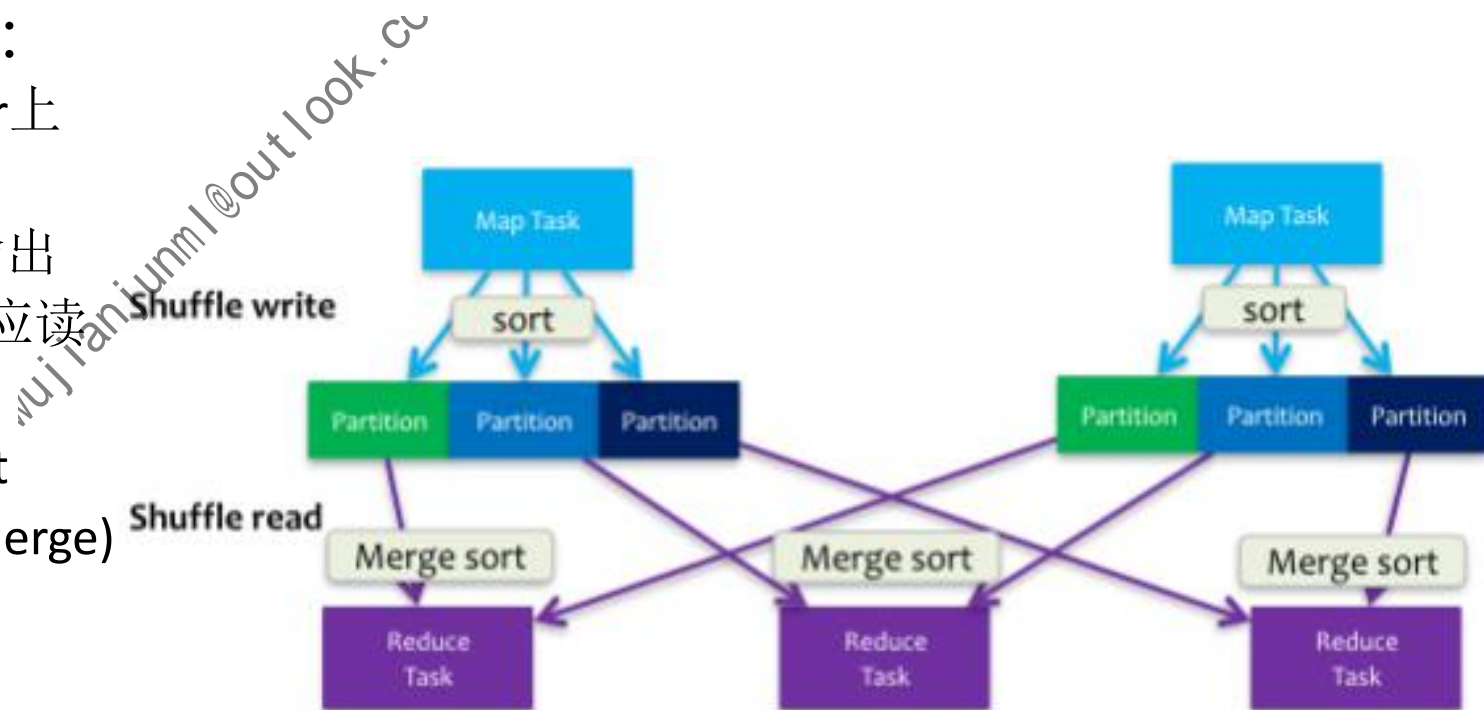
Spark Shuffle机制简介

- shuffle是指将不同分区上的数据按照key重新划分的过程，结果往往是key相同的记录被划分在同一个分区中。由于shuffle涉及到序列化反序列化、跨节点网络IO以及磁盘读写IO等，shuffle的性能高低直接影响了整个程序的性能。Spark的Shuffle实现大致如下图所示：
 - 在DAG中以shuffle为界，划分stage，上游stage做map，下游stage做reduce。
 - 每个mapper将结果数据分成多份，每一份对应一个reducer，该过程称shuffle write。
 - 每个reducer通过网络拉取上游所有分给本reducer数据，该过程称shuffle read。



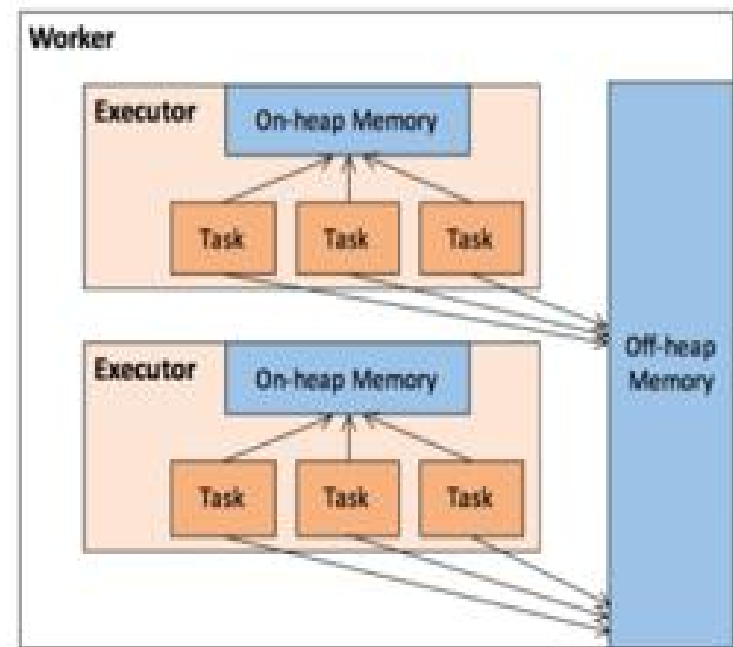
Spark Shuffle机制简介

- Spark有多种shuffle实现：hash，sort，tungsten。目前统一到Sort Shuffle。
- Sort Shuffle的主要特点如下：
 - shuffle write阶段同一个executor上的mapper的输出被统一排序后，写入同一个data文件，另外还输出一个index文件表明每个reducer应读入的数据。
 - 自动识别是否采用Tungsten-Sort Shuffle(无须反序列即可sort和merge)



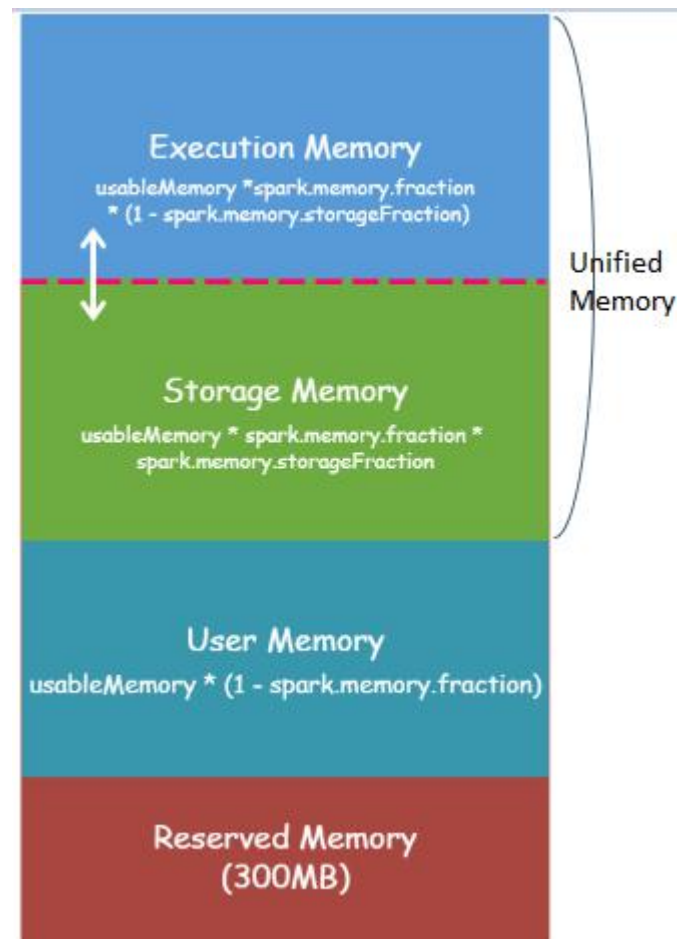
Spark内存管理简介

- Spark立足内存运算，对内存需求大，又运行于JVM平台，受到回收机制(GC)影响，所以内存管理是其关键问题。
- Driver 的内存管理相对来说较为简单，主要讨论 Executor 的内存管理。
- 自1.6版本spark的内存分为堆内内存(On-heap)和堆外内存(Off-heap)。
 - 堆内内存的大小由executor-memory参数指定。
 - 可通过 spark.memory.offHeap.enabled 参数启用堆外内存，并由 spark.memory.offHeap.size 参数大小。



Spark内存管理简介

- 堆内内存分三部分：
 - Reserved Memory: 预留内存，不可使用。
 - User Memory: 用户直接控制的内存空间，存储用户临时对象。
 - Unified Memory: 系统框架所需要使用的空间，又分为两个部分：
 - Storage Memory: 主要存储cache的数据。
 - Execution Memory: 执行内存，如shuffle用内存。
- Storage和Execution之间存在动态互相占用的机制：
 - 双方的空间都不足时，则存储到硬盘。
 - 若己方空间不足而对方空余时，可借用对方的空间。
 - 执行内存的空间被对方占用后，可让"归还"空间。
 - 存储内存的空间被对方占用后，无法让对方"归还"。
- 堆外的空间分配较为简单，只有Storage和Execution。



Spark性能调优简介

- Spark性能调优的空间很大，细致的调优将获得：
 - 更短的执行时间。
 - 更少的资源消耗。
- Spark性能调优常见方法有：
 - 正确使用API：不要使用groupByKey，不要使用distinct，尽早filter，etc。
 - 使用Kryo序列化：

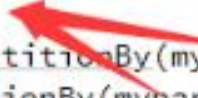
```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

- 广播大变量，否则每个task都会从driver端拷贝。
- RDD被多次使用时，需要将之cache，不再使用时需要将之uncache。

Spark性能调优简介

- 避免数据倾斜，过大的分区拖慢整个job，且容易导致OOM。
- 避免过多分区：过多的分区将加大shuffle的通信代价，也将加大driver端的内存消耗。
 - 避免很多小文件，
 - 主动收缩分区数，
- 避免不必要的shuffle，避免不了则尽量减少shuffle的数据量。
- 一个很大的RDD与一个非常大的RDD进行join之前首先用相同的partitioner划分使得两个RDD相同key都在一个executor上，然后再join。

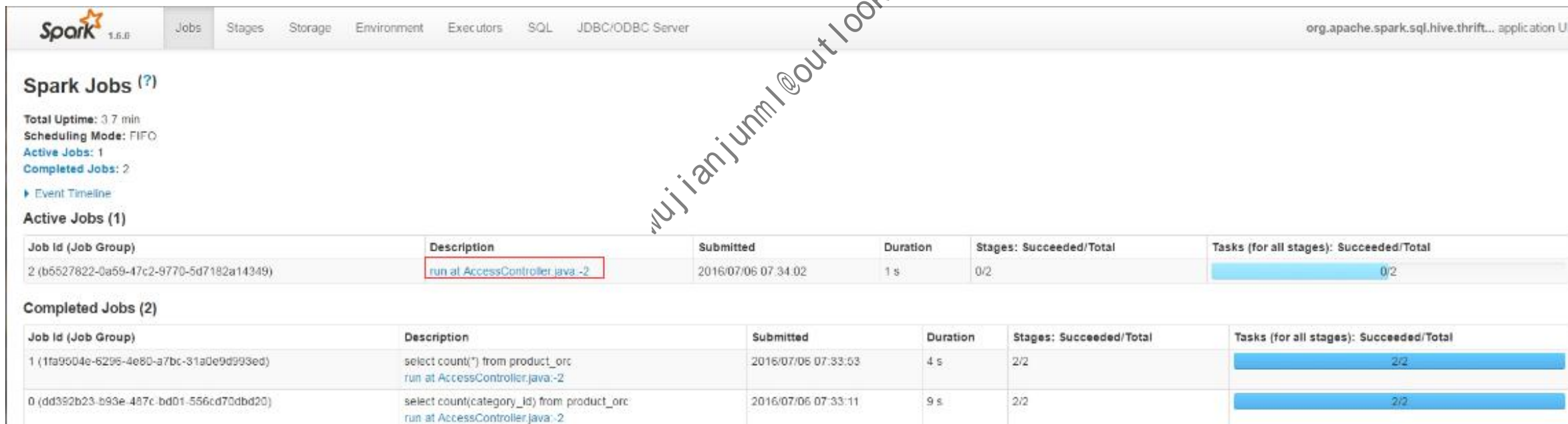
```
println("join adtaget and rule_data.....")
case class tmpTargetPoint(mid:String,point_str:String)
val mypartiotioner = new org.apache.spark.HashPartitioner(100)
val join_adtaget = adtaget.map(item=>(item.target_id,item)).partitionBy(mypartiotioner) //10亿记录,10
val join_rule = rule_data.map(item=>(item.rule_id,item)).partitionBy(mypartiotioner) //16万记录,2个分l
val adtarget_rule = join_adtaget.join(join_rule)
                                .map(item=>(item._2._1.mid,item._2._2.point_str))//(mid,point_str)
```



Spark UI

SparkUI

- SparkUI用于分析和诊断Spark任务运行细节，是性能调优的主要依据：
 - Jobs页面显示一个application的全部job以及每个job的进度。
 - 点击job的名字可以查看每个job的详细情况。



The screenshot displays the SparkUI interface for a Spark application. The top navigation bar includes tabs for Jobs, Stages, Storage, Environment, Executors, SQL, and JDBC/ODBC Server. The main content area is titled "Spark Jobs (?)".

Active Jobs (1)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2 (b5527822-0a59-47c2-9770-5d7182a14349)	run at AccessController.java:-2	2016/07/06 07:34:02	1 s	0/2	0/2

Completed Jobs (2)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1 (1fa9504e-6295-4e80-a7bc-31a0e9d993ed)	select count(*) from product_orc run at AccessController.java:-2	2016/07/06 07:33:53	4 s	2/2	2/2
0 (dd392b23-b93e-487c-bd01-556cd70dbd20)	select count(category_id) from product_orc run at AccessController.java:-2	2016/07/06 07:33:11	9 s	2/2	2/2

SparkUI

- 每个job的详情包括两方面：
 - Job有哪些Stage，点击Stage可以查看每个task的信息：

Details for Job 2

Status: SUCCEEDED

Job Group: b5627822-0a59-47c2-9770-5d7182a14349

Completed Stages: 2

▶ Event Timeline

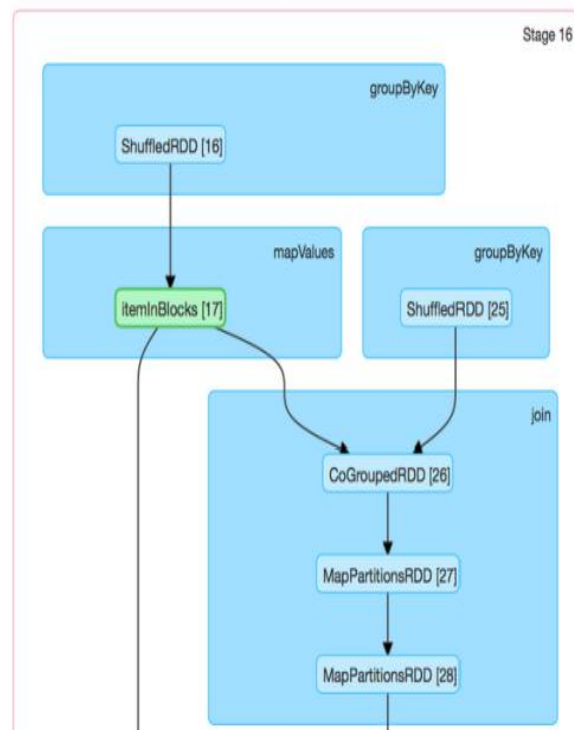
▶ DAG Visualization

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	select count(*) from product_orc run at AccessController.java-2	2016/07/06 07:34:06	4 s	1/1			42.0 B	
4	select count(*) from product_orc run at AccessController.java-2	2016/07/06 07:34:02	4 s	1/1	7.1 MB			42.0 B

- job的DAG，显示了该job的RDD依赖关系：

▼ DAG Visualization



SparkUI

- 每个task的运行信息是性能调优的重点分析对象，他包括：
 - GC试过过长,
 - 数据是否倾斜,
 - 读取是否耗时,
 - etc, ...

Spark 2.0.0-SNAPSHOT Jobs Stages Storage Environment Executors SQL Spark shell application UI

Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 48 ms
Locality Level Summary: Process local: 4
Shuffle Write: 506.0 B / 11

[DAG Visualization](#)
[Show Additional Metrics](#)
[Event Timeline](#)

Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 ms	13 ms	13 ms	13 ms	13 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	92.0 B / 2	138.0 B / 3	138.0 B / 3	138.0 B / 3	138.0 B / 3

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Write Size / Records
driver	192.168.1.9:65297	70 ms	4	0	0	4	506.0 B / 11

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Write Time	Shuffle Write Size / Records	Errors
0	16	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	9 ms		1 ms	92.0 B / 2	
1	17	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
2	18	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
3	19	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	

SparkUI

- 点击Executor页面，可以查看每个executor的详情：



Spark 1.6.0 Jobs Stages Storage Environment **Executors** SQL JDBC/ODBC Server

Executors (5)

Memory: 0.0 B Used (5.7 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Completed Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	m001:47325	0	0.0 B / 1247.6 MB	0.0 B	0	0	0	2	4.2 s	7.1 MB	0.0 B	42.0 B	stdout stderr	Thread Dump
2	m002:50225	0	0.0 B / 1247.6 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	m001:43087	0	0.0 B / 1247.6 MB	0.0 B	0	0	2	2	9.3 s	68.0 KB	0.0 B	42.0 B	stdout stderr	Thread Dump
4	m002:58652	0	0.0 B / 1247.6 MB	0.0 B	0	0	2	2	4.3 s	7.1 MB	0.0 B	42.0 B	stdout stderr	Thread Dump
driver	192.168.1.88:39796	0	0.0 B / 879.0 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

- 另外还有：
 - Storage页面，用于显示RDD缓存情况。
 - Enviroment页面，显示参数配置和运行环境情况。

Q&A

wujianjunm1@outlook.com