并行计算
Parallel Computing

主讲 孙经纬

2024年 春季学期

概要

- 第三篇 并行编程
  - 第十三章 并行程序设计基础
  - 第十四章 共享存储系统并行编程
  - 第十五章 分布存储系统并行编程
  - 补充章节1 GPU并行编程
  - **补充章节2 关于并行编程的更多话题**

关于并行编程的更多话题

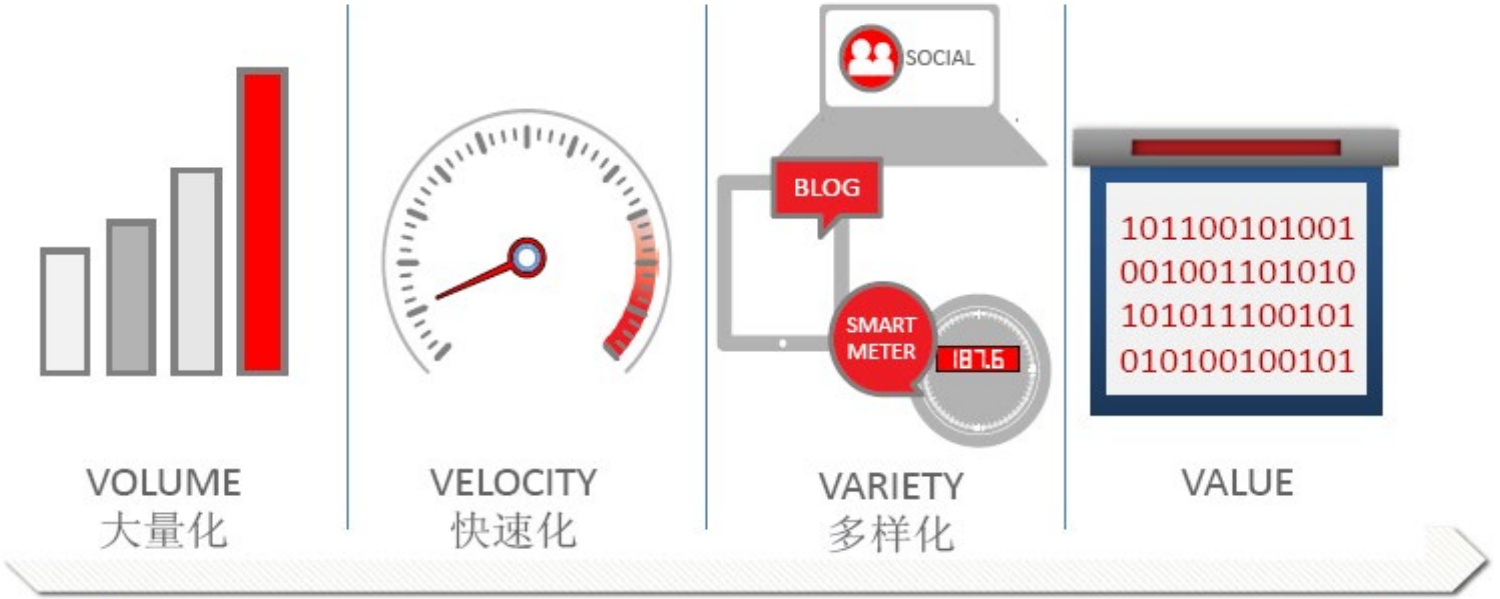
- 数据依赖与自动并行
- MapReduce
- 并行程序性能优化
- 案例：图与矩阵算法的GPU优化

MapReduce

- **背景介绍**
- MapReduce
- Hadoop
- Spark

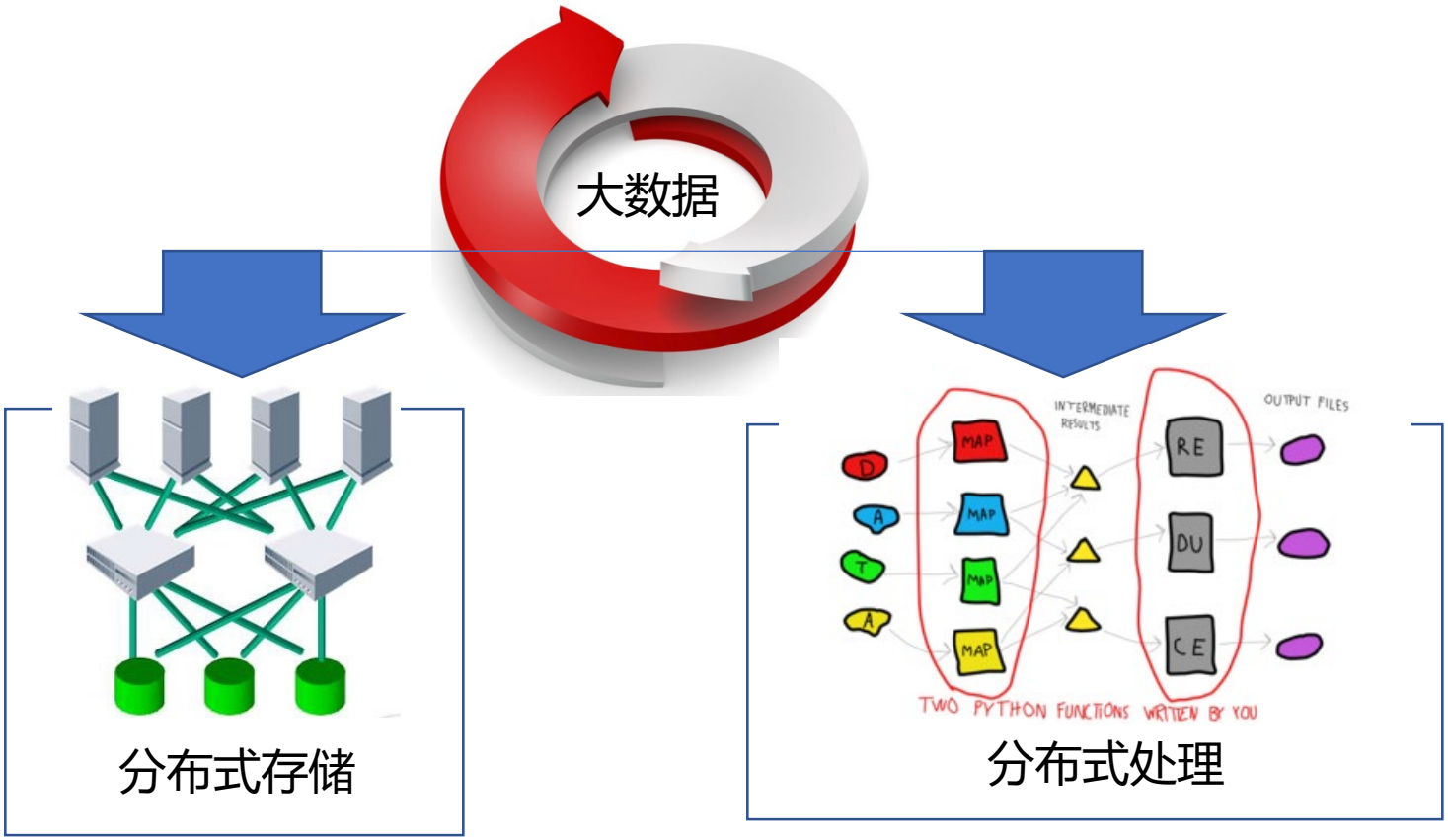
参考了厦门大学林子雨老师《大数据原理与应用》课件

大数据



大数据不仅仅是数据的“大量化”，而是包含“快速化”、“多样化”和“价值化”等多重属性。

大数据



GFS\HDFS
BigTable\HBase
NoSQL（键值、列族、图形、文档数据库）
NewSQL（如：SQL Azure）

MapReduce

分布式处理

- 如何让程序在成千上万的节点上高效运行?
- 当某个节点崩溃，如何保证程序仍然能正常运行?
- 如何便捷地编程实现?

- Google（在2004年）给出的答案:

MapReduce

分布式处理

- 在MapReduce出现之前，已经有像MPI这样非常成熟的并行编程模型了，那么为什么Google还需要MapReduce？MapReduce相较于传统的并行计算框架有什么优势？

	传统并行计算框架	MapReduce
集群架构/容错性	共享式(共享内存/共享存储)，容错性差	非共享式，容错性好
硬件/价格/扩展性	刀片服务器、高速网、SAN，价格贵，扩展性差	普通PC机，便宜，扩展性好
编程/学习难度	what-how，难	what，简单
适用场景	实时、细粒度计算、计算密集型	批处理、非实时、数据密集型

MapReduce

- 背景介绍
- **MapReduce**
- Hadoop
- Spark

MapReduce

- MapReduce是一种编程模型
- 假设所有计算任务接收一组key-value对作为输入，产生一组key-value对作为输出。
- 计算过程通过Map和Reduce两个函数实现
  - Map：对数据做预处理（筛选、排序等），产生中间结果
  - Reduce：对中间结果进行计算（计数、求和等），产生最终结果
- Map和Reduce是高阶函数，需要用户提供实际使用的函数

Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150

MapReduce

MapReduce可以很好地应用于多种计算问题

- 关系代数运算（选择、投影、并、交、差、连接）
- 分组与聚合运算

- 矩阵-向量乘法
- 矩阵乘法

MapReduce

用MapReduce实现关系的自然连接

雇员

Name	EmpId	DeptName
Harry	3415	财务
Sally	2241	销售
George	3401	财务
Harriet	2202	销售

部门

DeptName	Manager
财务	George
销售	Harriet
生产	Charles

雇员 ⋈ 部门

Name	EmpId	DeptName	Manager
Harry	3415	财务	George
Sally	2241	销售	Harriet
George	3401	财务	George
Harriet	2202	销售	Harriet

假设有关系R(A，B)和S(B,C)，对二者进行自然连接操作:

- 使用Map过程，把来自R的每个元组<a,b>转换成一个键值对<b, <R,a>>，其中的键就是属性B的值。把关系R包含到值中，使得可以在Reduce阶段，只把那些来自R的元组和来自S的元组进行匹配。类似地，使用Map过程，把来自S的每个元组<b,c>，转换成一个键值对<b,<S,c>>
- 所有具有相同B值的元组被发送到同一个Reduce进程中，Reduce进程的任务是，把来自关系R和S的、具有相同属性B值的元组进行合并
- Reduce进程的输出则是连接后的元组<a,b,c>，输出被写到一个单独的输出文件中

MapReduce

用MapReduce实现关系的自然连接

Order

Orderid	Account	Date
1	a	d1
2	a	d2
3	b	d3

Map

Key	Value
1	“Order”,(a,d1)
2	“Order”,(a,d2)
3	“Order”,(b,d3)

Item

Orderid	Itemid	Num
1	10	1
1	20	3
2	10	5
2	50	100
3	20	1

Map

Key	Value
1	“Item”,(10,1)
1	“Item”,(20,3)
2	“Item”,(10,5)
2	“Item”,(50,100)
3	“Item”,(20,1)

Reduce

(1,a,d1,10,1)

(1,a,d1,20,3)

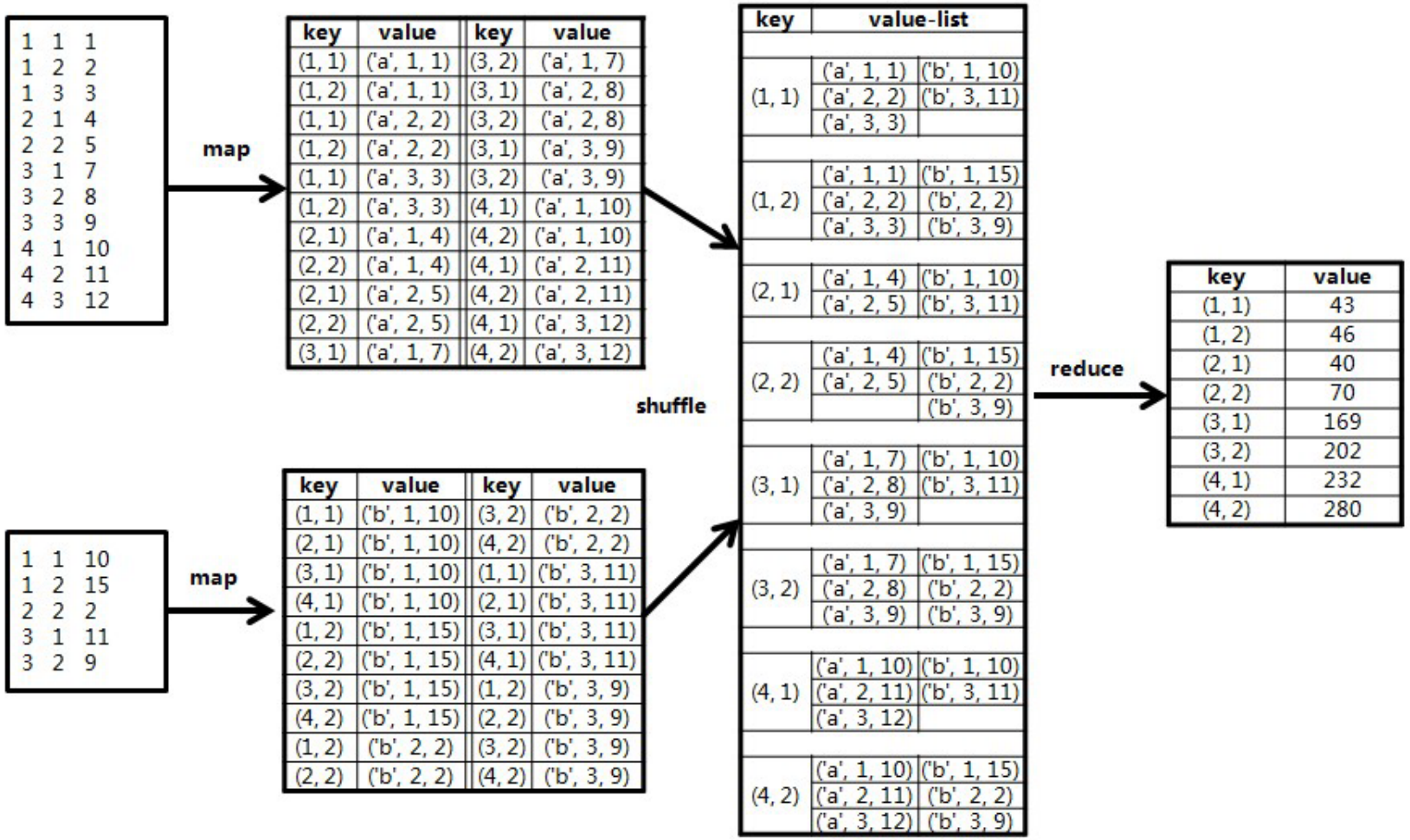
(2,a,d2,10,5)

(2,a,d2,50,100)

(3,b,d3,20,1)

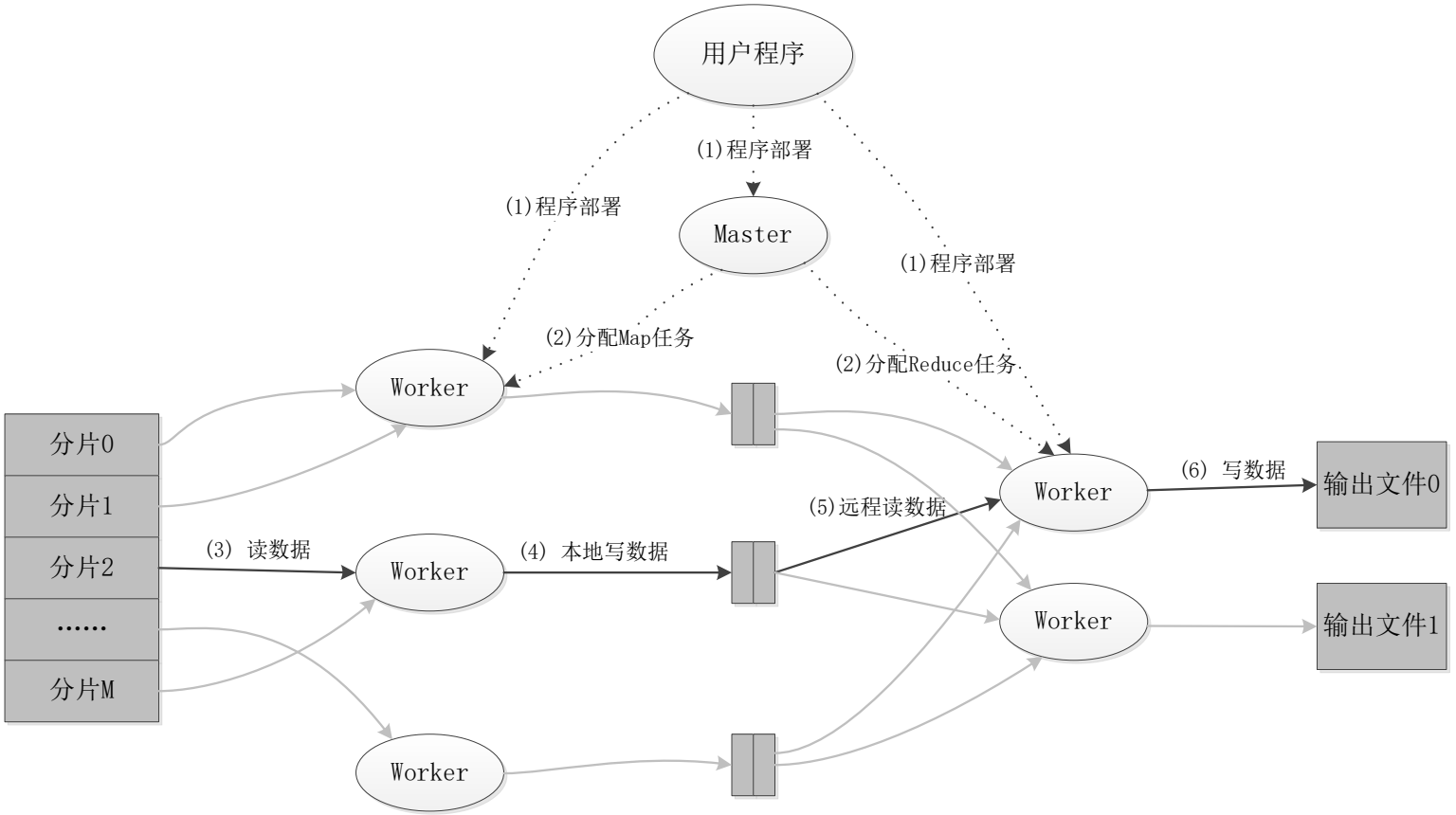
MapReduce

用MapReduce实现矩阵乘法

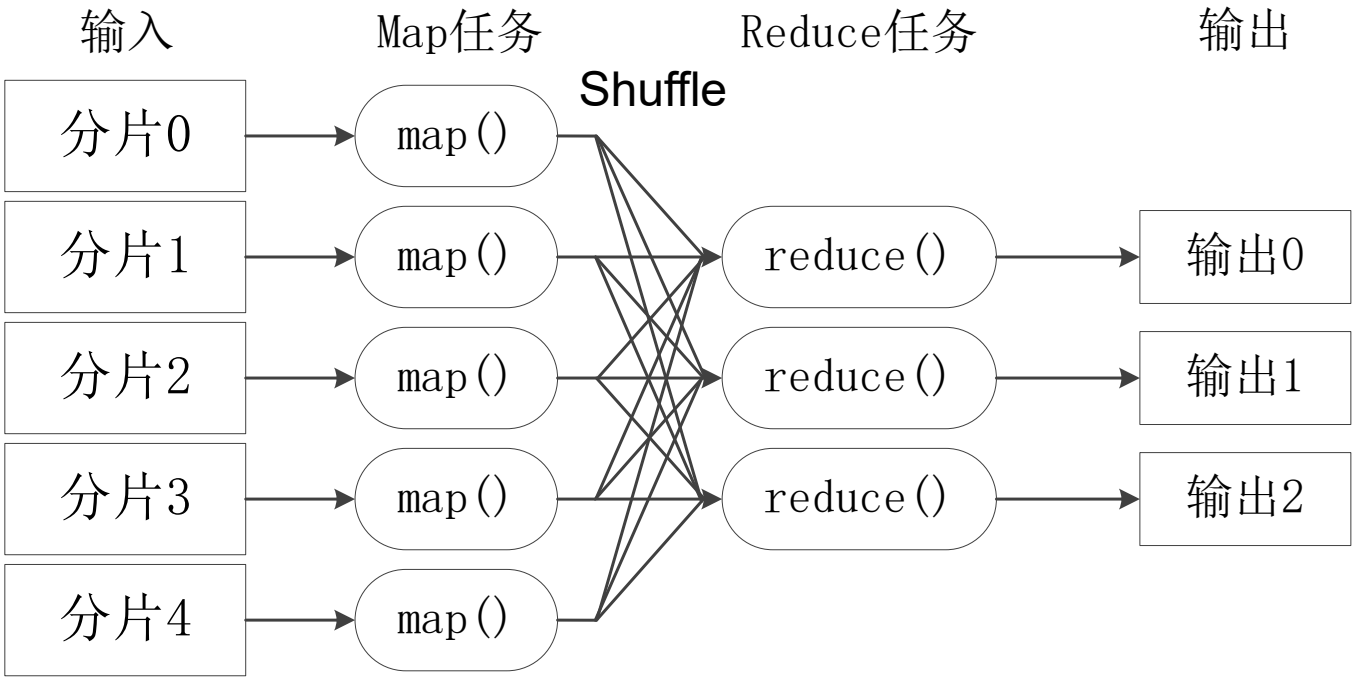


MapReduce

- MapReduce也指代基于该编程模型设计的分布式处理系统

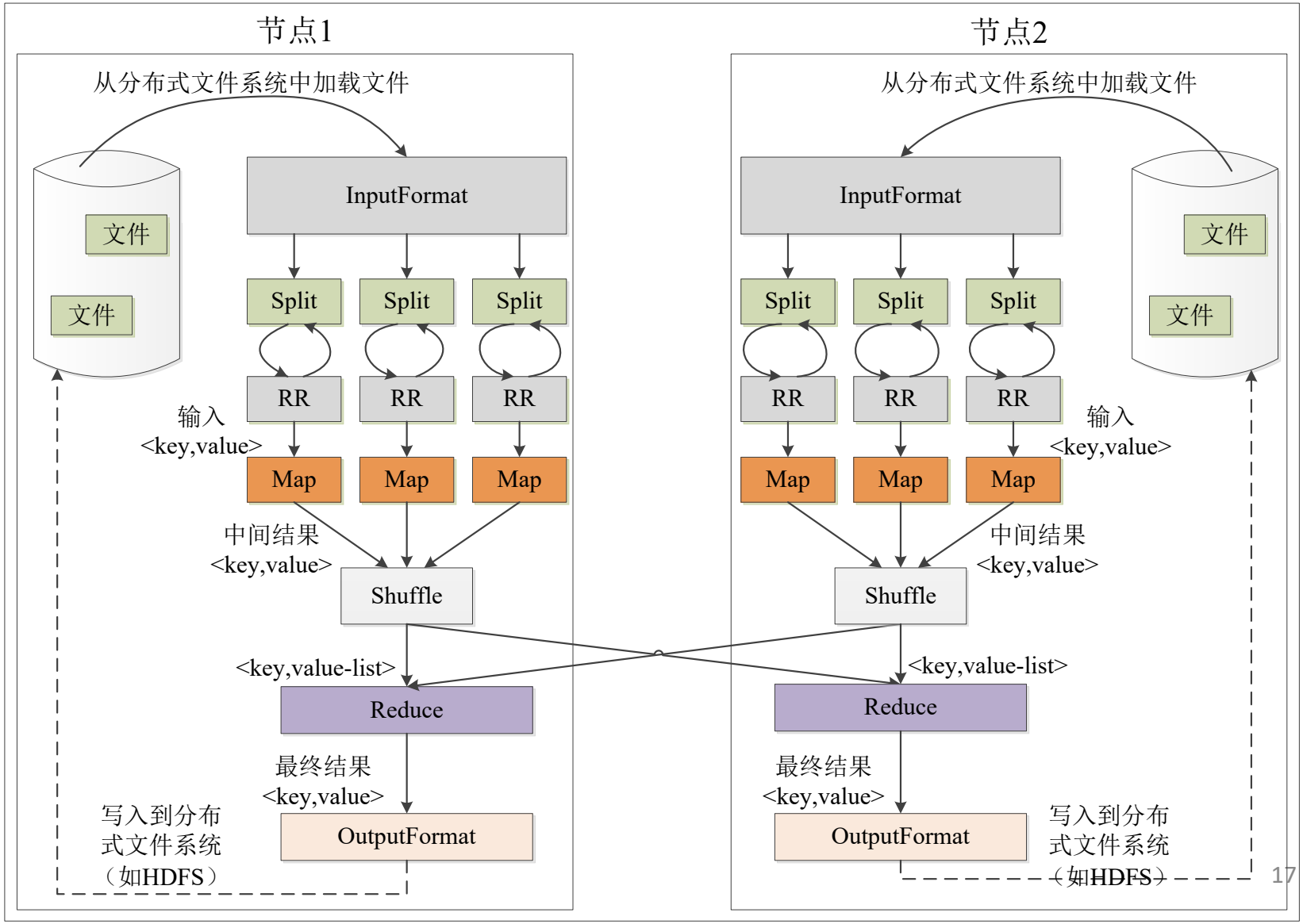


MapReduce

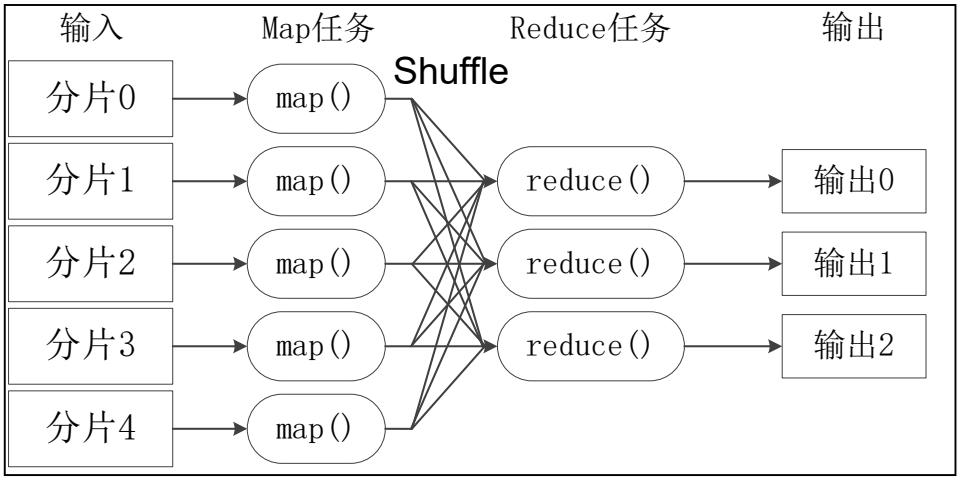


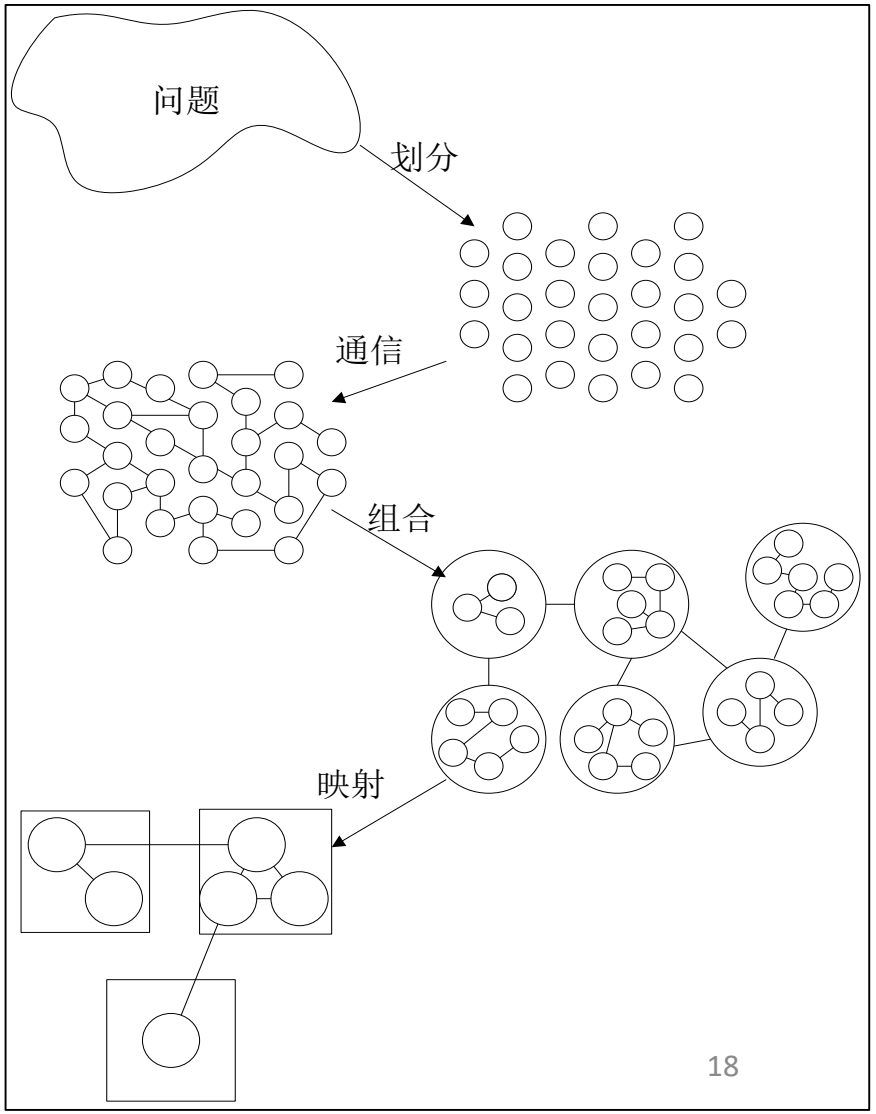
- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- 用户不能显式地从一个worker向另一个worker发送消息
- Shuffle过程保证一个key关联的所有value在同一个worker上

MapReduce


节点1
从分布式文件系统中加载文件
文件
文件
InputFormat
Split
Split
Split
RR
RR
RR
输入
<key,value>
Map
Map
Map
中间结果
<key,value>
Shuffle
<key,value-list>
Reduce
最终结果
<key,value>
写入到分布式文件系统（如HDFS）
OutputFormat
节点2
从分布式文件系统中加载文件
InputFormat
Split
Split
Split
RR
RR
RR
输入
<key,value>
Map
Map
Map
中间结果
<key,value>
Shuffle
<key,value-list>
Reduce
最终结果
<key,value>
OutputFormat
文件
文件
写入到分布式文件系统（如HDFS）

MapReduce vs. PCAM





MapReduce

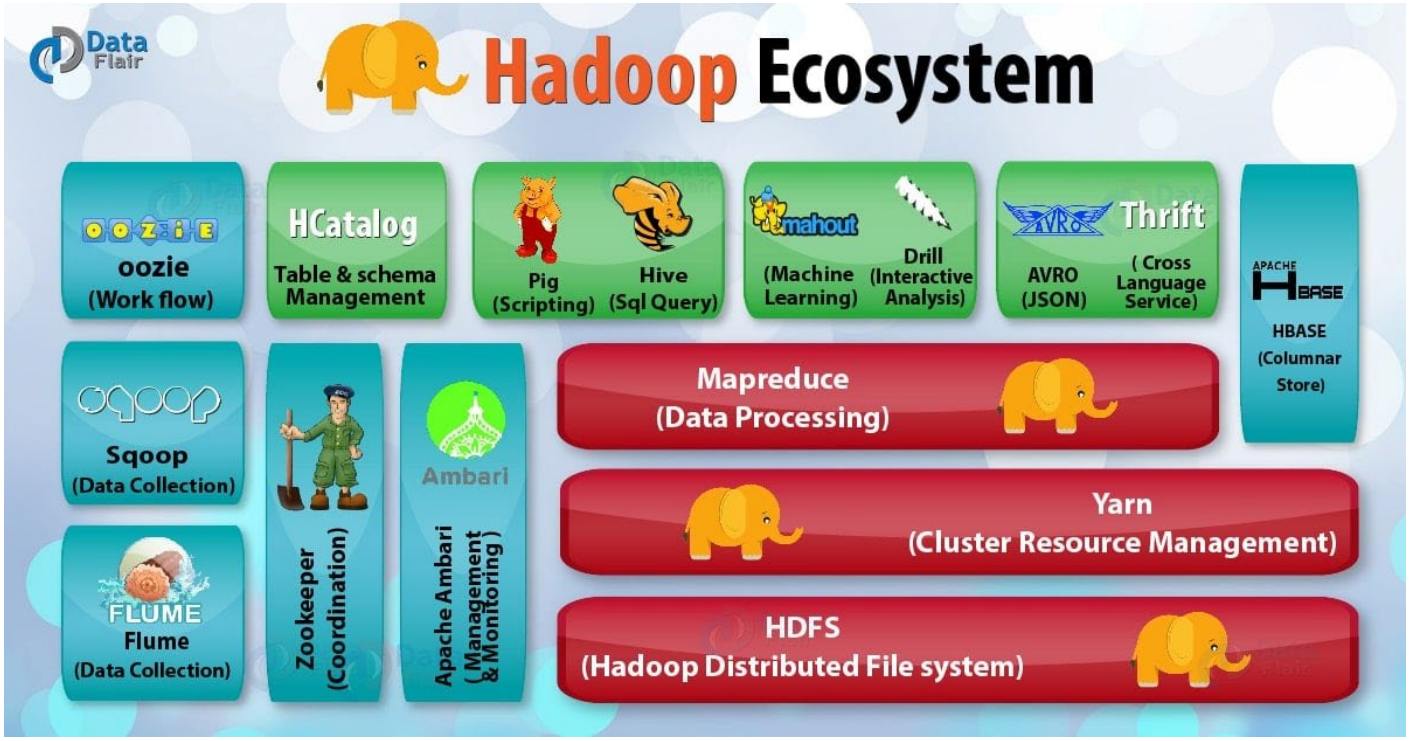
- 背景介绍
- MapReduce
- **Hadoop**
- Spark

Hadoop

https://hadoop.apache.org/

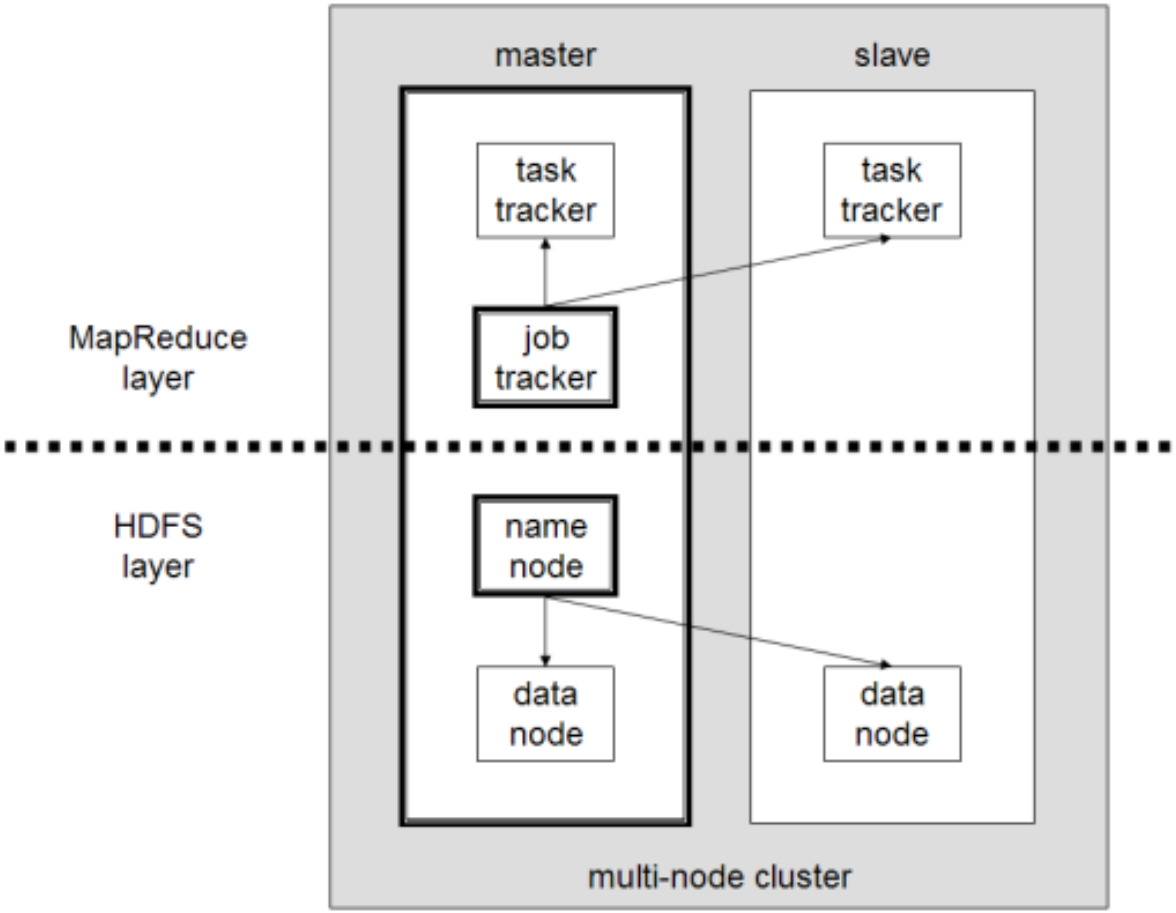
- Hadoop是MapReduce的一种开源实现
- 早期的Hadoop主要包括MapReduce引擎和HDFS文件系统（Hadoop Distributed File System ）

目前已发展成了一系列的工具组合



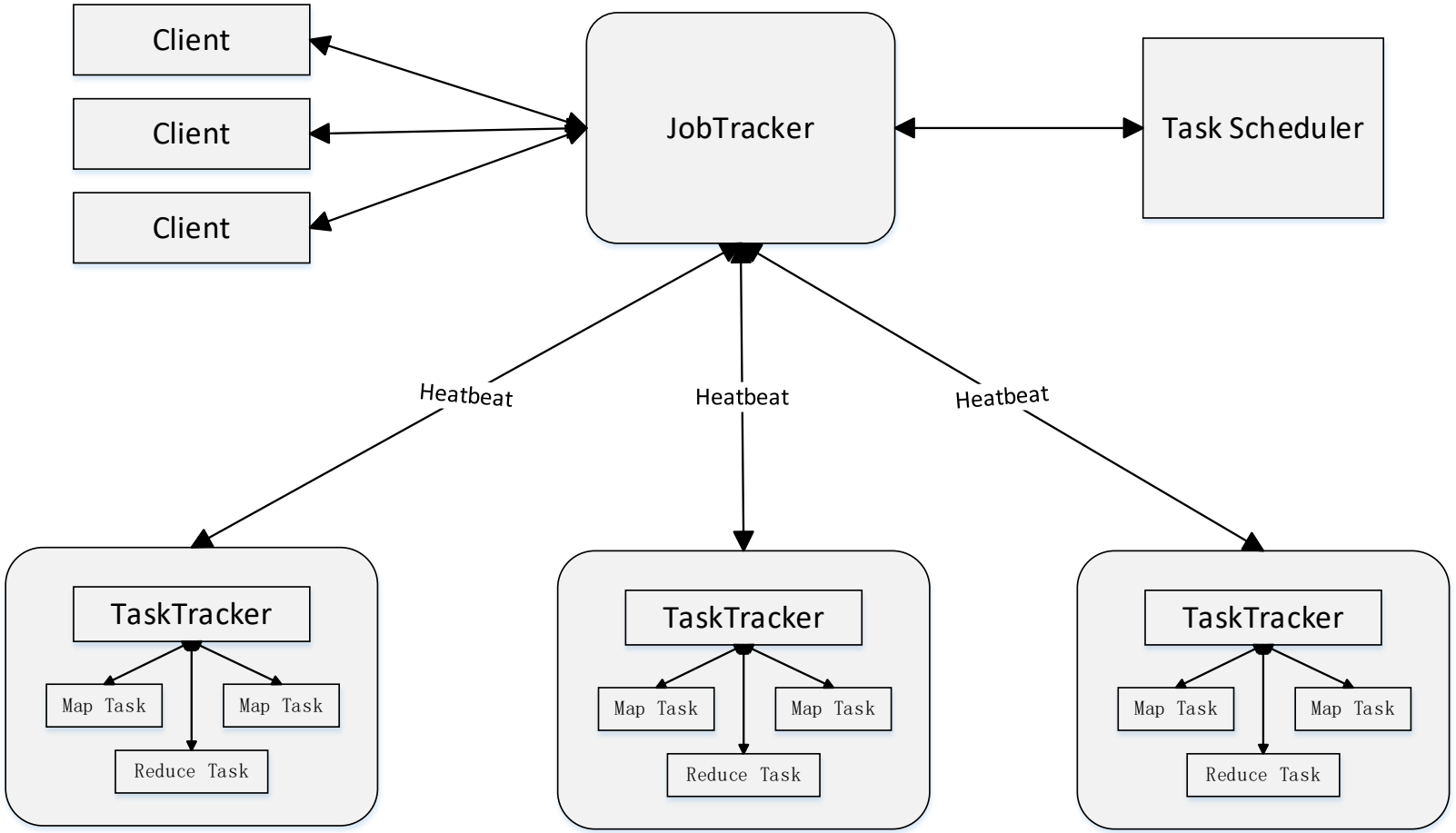
Hadoop

- Hadoop组织架构



Hadoop

- MapReduce



Hadoop

- MapReduce

1. Client
   - 用户编写的MapReduce程序通过Client提交到JobTracker端
   - 用户可通过Client提供的一些接口查看作业运行状态
2. JobTracker
   - JobTracker负责资源监控和作业调度
   - JobTracker 监控所有TaskTracker与Job的健康状况，一旦发现失败，就将相应的任务转移到其他节点
   - JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器（TaskScheduler），而调度器会在资源出现空闲时，选择合适的任务去使用这些资源

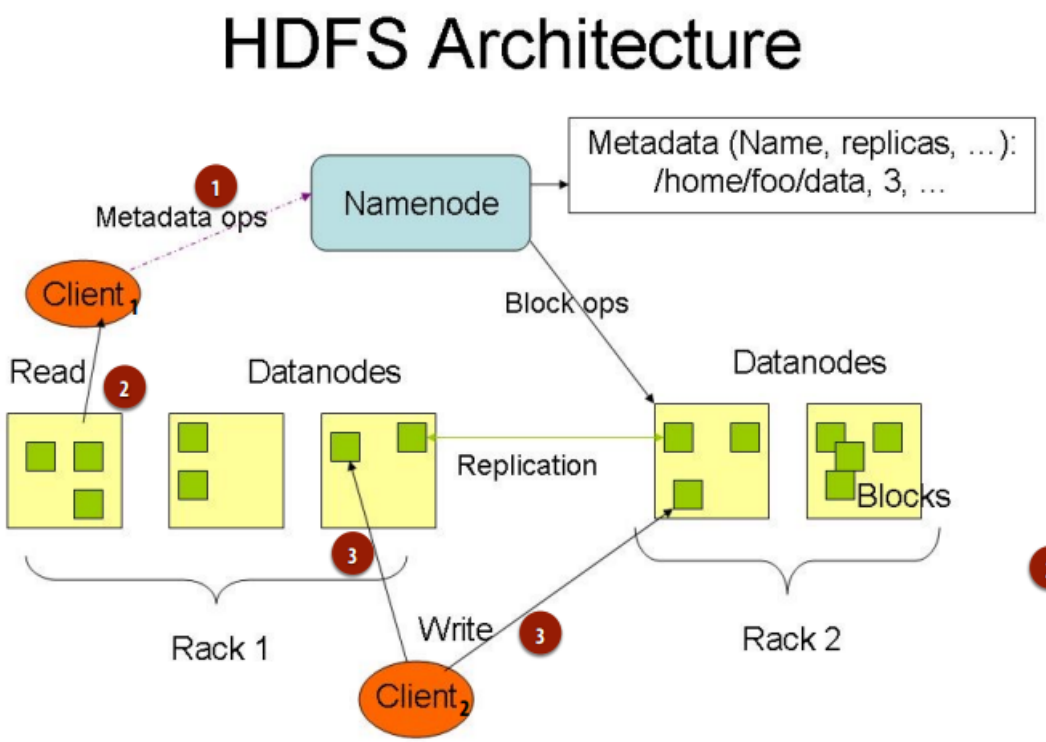
Hadoop

- MapReduce

3. TaskTracker
   - TaskTracker 会周期性地通过“Heatbeat”将本节点上资源的使用情况和任务的运行进度汇报给JobTracker，同时接收JobTracker 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）
   - TaskTracker 使用“slot”等量划分本节点上的资源量（CPU、内存等）。一个Task 获取到一个slot 后才有机会运行，调度器的作用就是将各个TaskTracker上的空闲slot分配给Task使用。slot 分为Map slot 和Reduce slot，分别供MapTask 和Reduce Task 使用。
4. Task
   - Task 分为Map Task 和Reduce Task 两种，均由TaskTracker 启动

Hadoop

- HDFS文件系统



- **Global namespace**
- **Files broken into blocks**
  - **Typically 256 MB each**
  - **Each block replicated on multiple DataNodes**
- **Intelligent Client**
  - (1) **Client finds locations of blocks from NameNode**
  - (2) (3) **Client accesses data directly from DataNode**

Hadoop

- 案例：WordCount

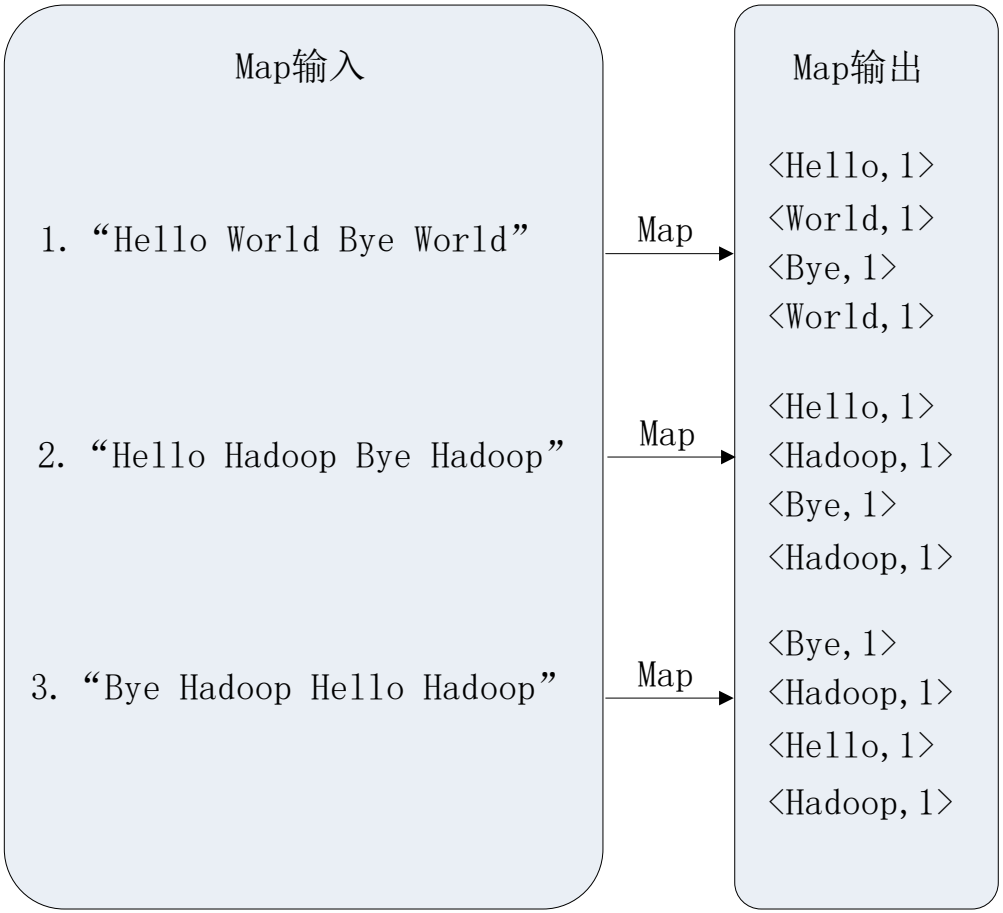
程序	WordCount
输入	一个包含大量单词的文本文件
输出	文件中每个单词及其出现次数（频数），并按照单词字母顺序排序，每个单词和其频数占一行，单词和频数之间有间隔

输入	输出
Hello World Hello Hadoop Hello MapReduce	Hadoop 1 Hello 3 MapReduce 1 World 1

示例代码（Java）https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

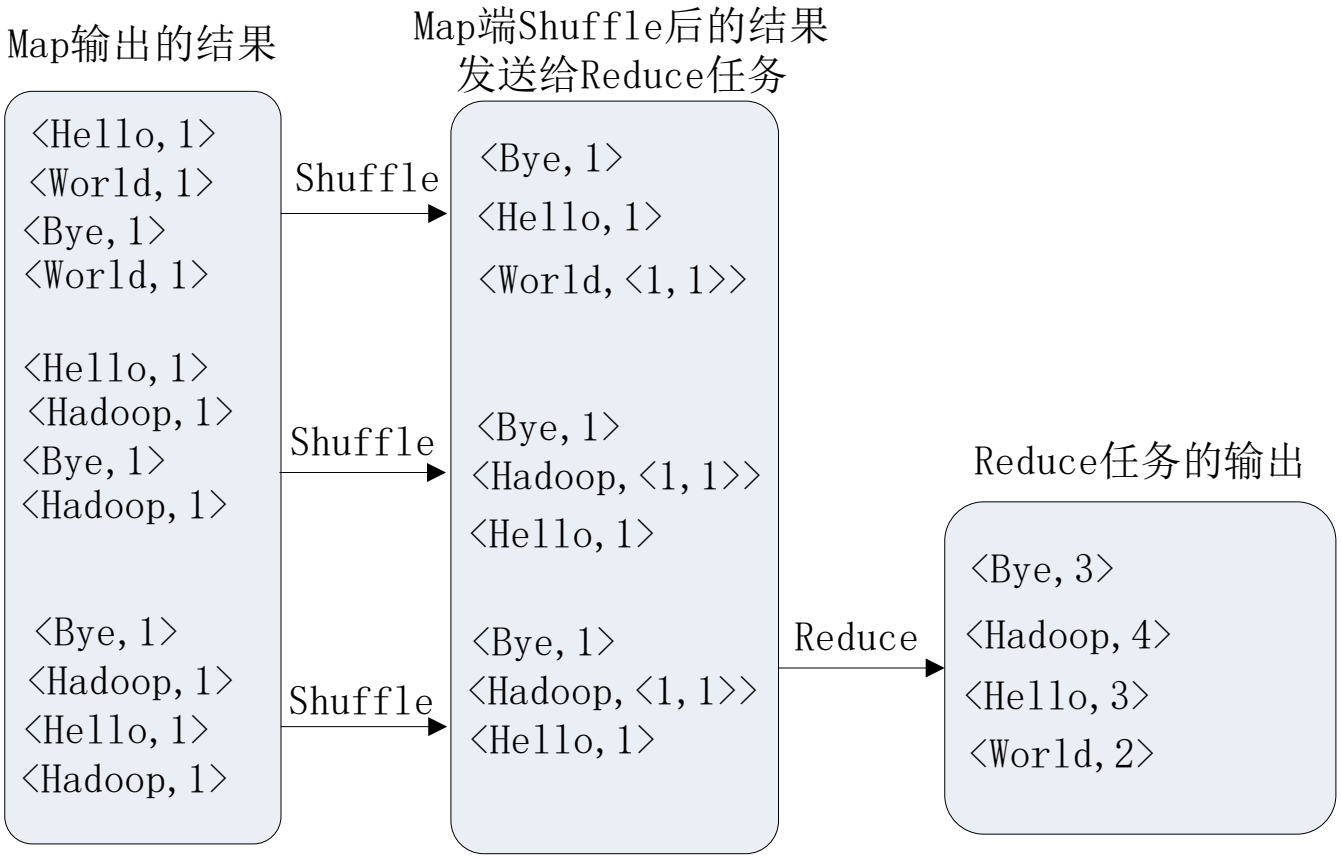
Hadoop

- 案例：WordCount



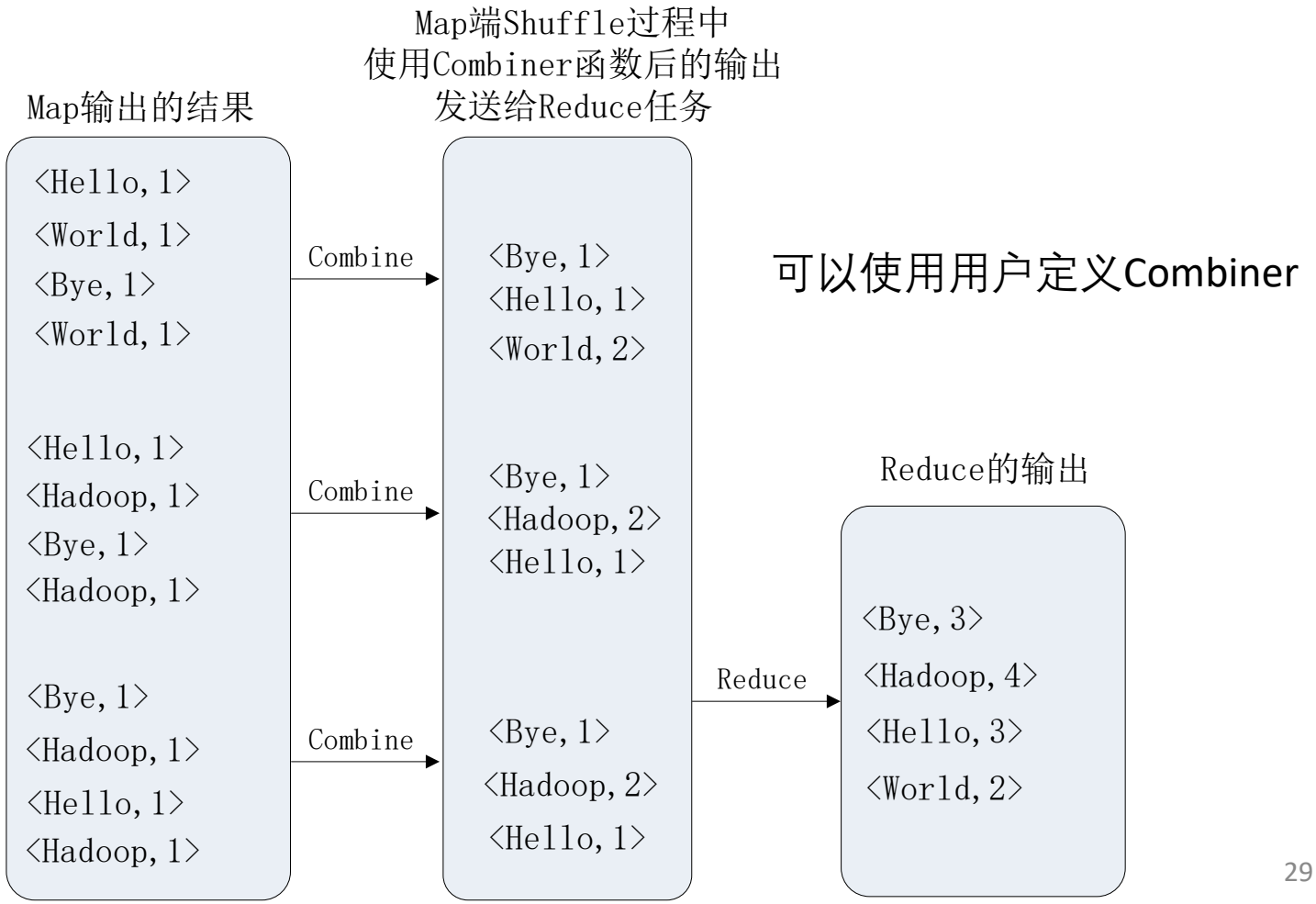
Hadoop

- 案例：WordCount



Hadoop

- 案例：WordCount



MapReduce

- 背景介绍
- MapReduce
- Hadoop
- **Spark**

Spark

Hadoop存在如下一些缺点：

- 表达能力有限
- 磁盘IO开销大
- 延迟高
  - 任务之间的衔接涉及IO开销
  - 在前一个任务执行完成之前，其他任务就无法开始，难以胜任复杂、多阶段的计算任务

Spark

https://spark.apache.org/

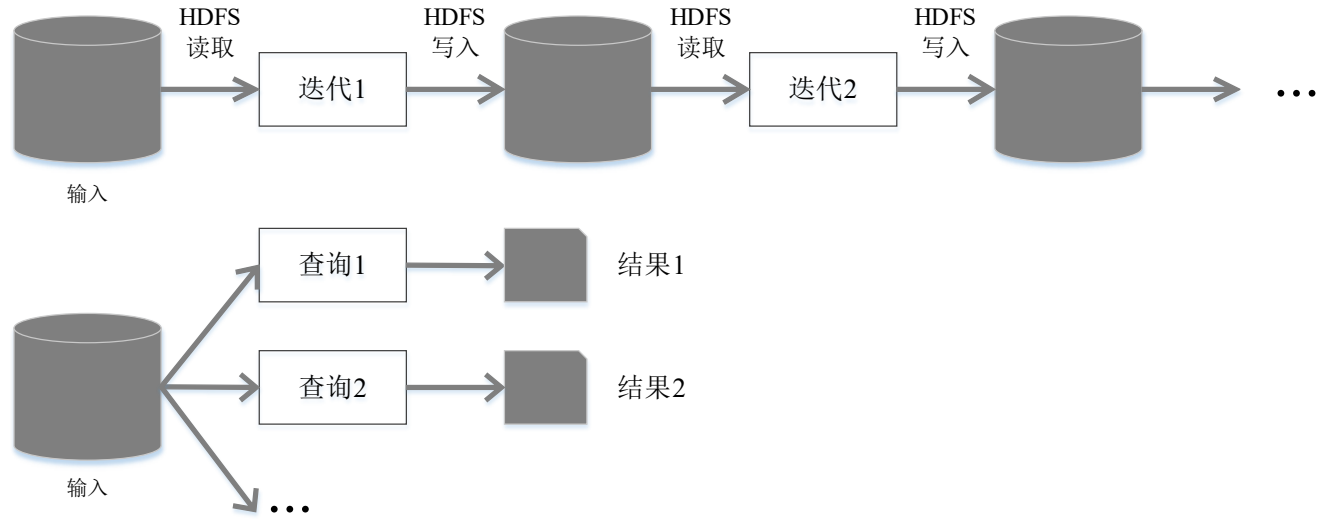
Spark在借鉴Hadoop MapReduce优点的同时做出了改进：

- Spark的计算模式也属于MapReduce，但不局限于Map和Reduce操作，还提供了多种数据集操作类型，比MapReduce更灵活
- 提供内存计算，可将中间结果放到内存中，迭代运算效率更高
- Spark基于DAG的任务调度执行机制，要优于Hadoop MapReduce的迭代执行机制

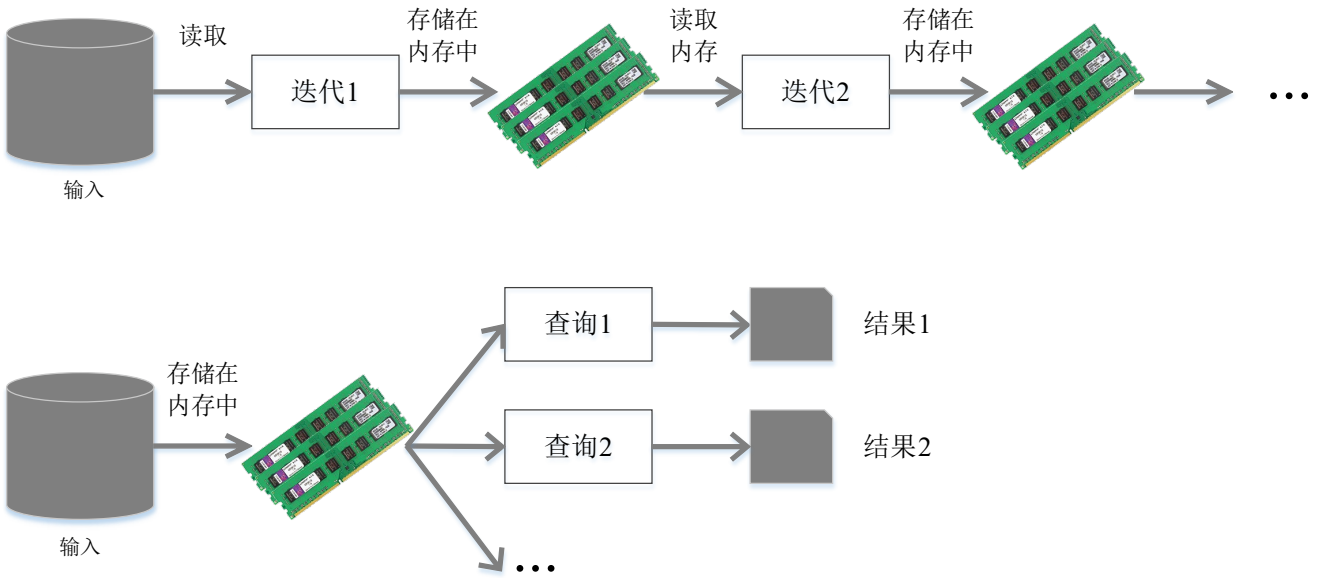
- Spark在2014年打破了Hadoop保持的基准排序纪录
  - Spark/206个节点/23分钟/100TB数据
  - Hadoop/2000个节点/72分钟/100TB数据
- Spark用十分之一的计算资源，获得了Hadoop 3倍的速度

Spark

Hadoop与Spark
的执行流程对比



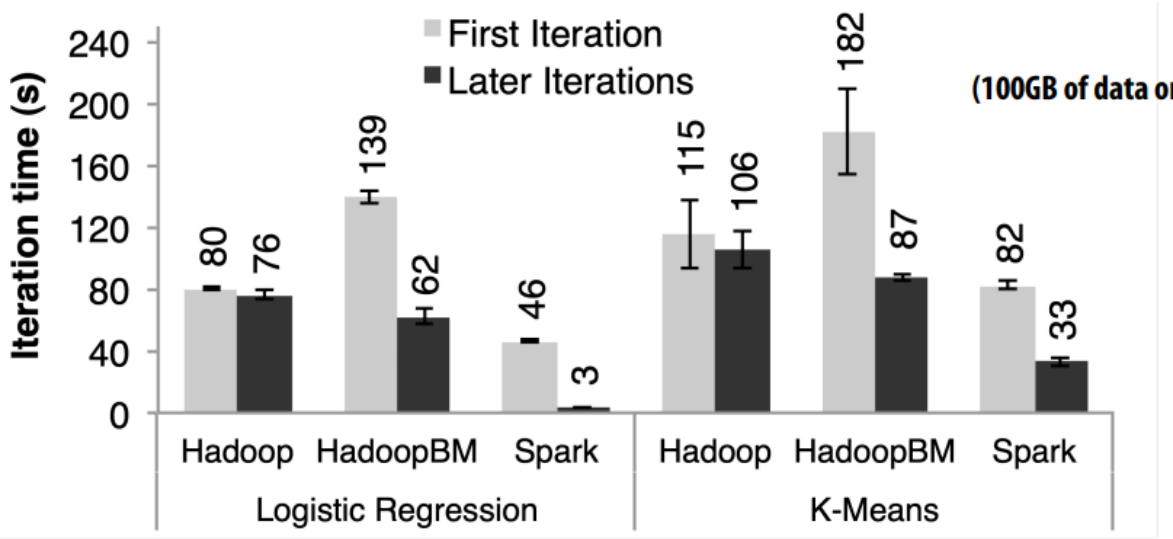
(a) Hadoop MapReduce执行流程



(b) Spark执行流程

Spark

Spark performance



HadoopBM = Hadoop Binary In-Memory (convert text input to binary, store in in-memory version of HDFS)

- Spark对Hadoop的优化，并不仅仅是把硬盘换成内存这么简单
- 可以看HadoopBM的性能，仅仅利用内存，带来的提升并没有非常高
- 第一轮迭代的速度会比较慢，因为要做很多初始化操作

Spark

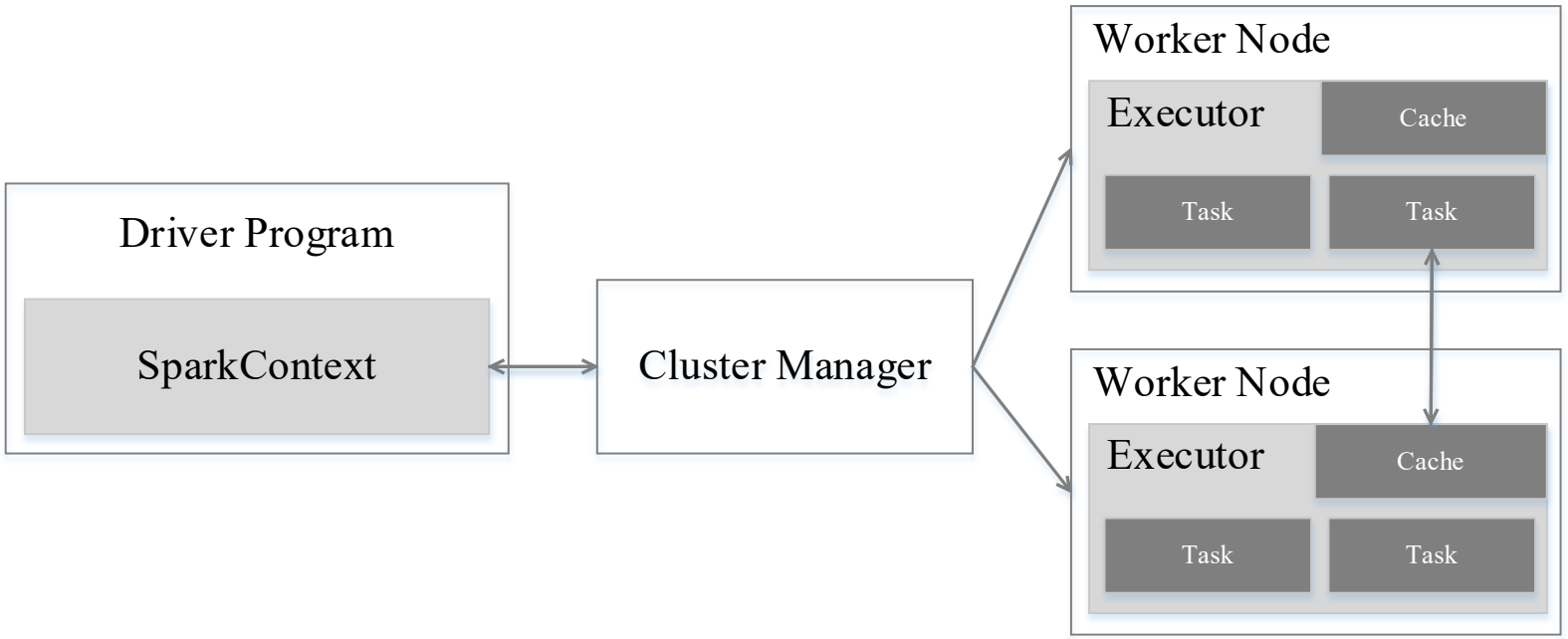
基本概念

- RDD：Resillient Distributed Dataset（弹性分布式数据集），是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型
- DAG：是Directed Acyclic Graph（有向无环图）的简称，反映RDD之间的依赖关系
- Executor：运行在工作节点（WorkerNode）的进程，负责运行Task
- Application：用户编写的Spark应用程序
- Task：运行在Executor上的工作单元
- Job：一个Job包含多个RDD及作用于相应RDD上的各种操作
- Stage：是Job的基本调度单位，一个Job会分为多组Task，每组Task被称为Stage，或者也被称为TaskSet，代表了一组关联的、相互之间没有Shuffle依赖关系的任务组成的任务集

Spark

基本概念

- Spark运行架构包括集群资源管理器（Cluster Manager）、运行作业任务的工作节点（Worker Node）、每个应用的任务控制节点（Driver）和每个工作节点上负责具体任务的执行进程（Executor）



Spark

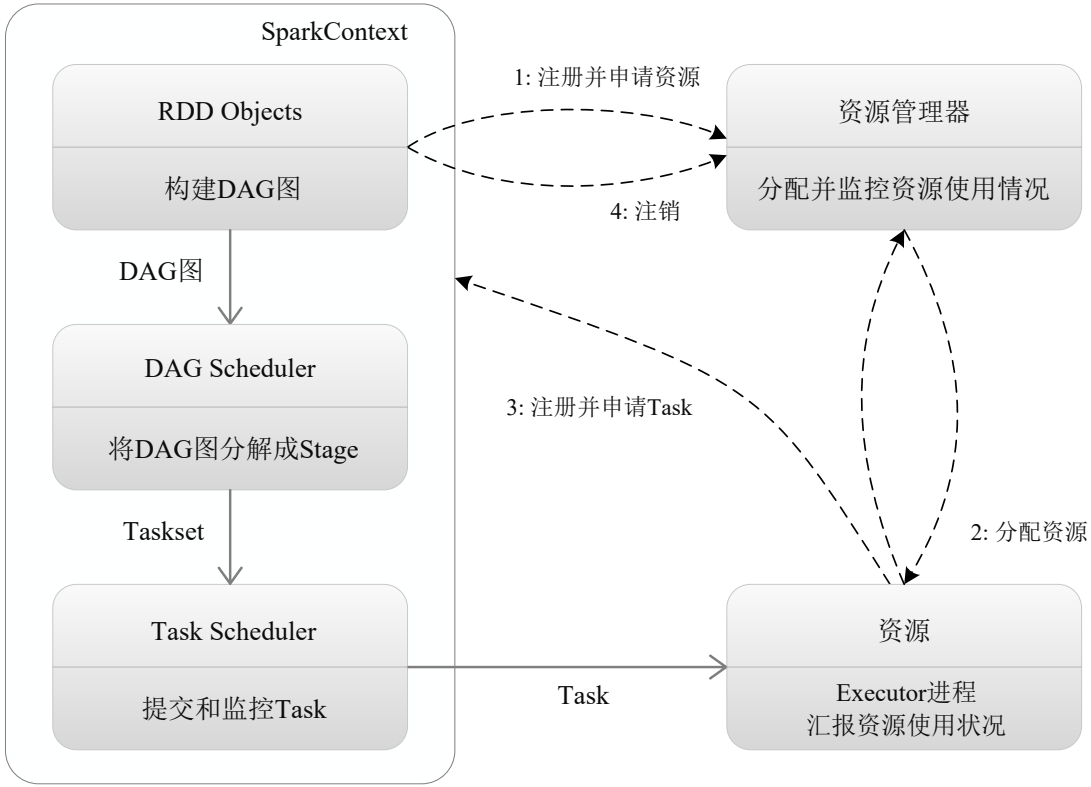
运行流程

（1）首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，进行资源的申请、任务的分配和监控

（2）资源管理器为Executor分配资源，并启动Executor进程

（3）SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAGScheduler解析成Stage，然后把一个个TaskSet提交给底层调度器TaskScheduler处理；Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，并提供应用程序代码

（4）Task在Executor上运行，把执行结果反馈给TaskScheduler，然后反馈给DAGScheduler，运行完毕后写入数据并释放所有资源



Spark

RDD设计背景

- 许多迭代式算法（比如机器学习、图算法等）和交互式数据挖掘工具，共同之处是，不同计算阶段之间会重用中间结果
- Hadoop把中间结果写入到HDFS中，带来了大量的数据复制、磁盘IO和序列化开销
- RDD就是为了满足这种需求而出现的，它提供了一个抽象的数据架构，我们不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，不同RDD之间的转换操作形成依赖关系，可以实现流水线化，避免中间数据存储

Spark

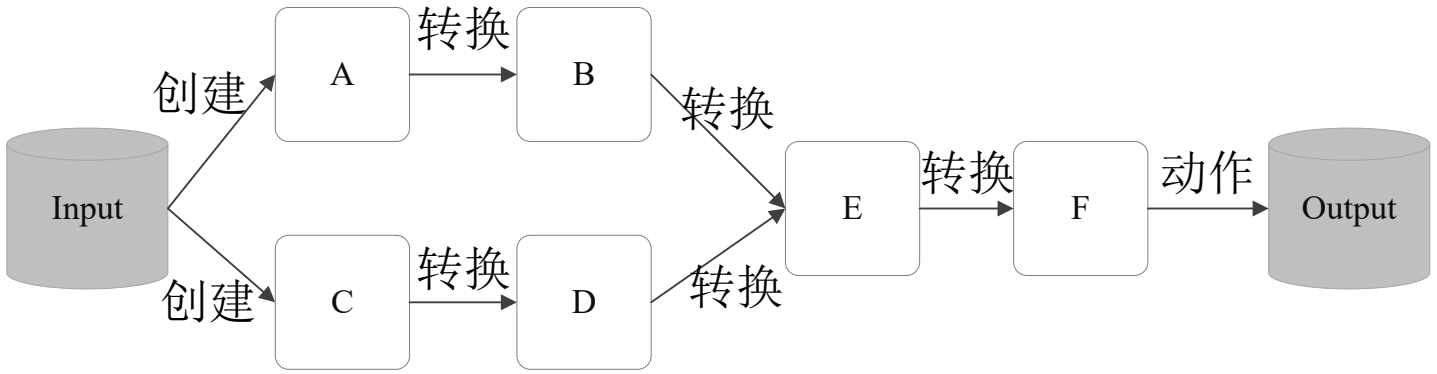
RDD概念

- 一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合，每个RDD可分成多个分区，每个分区就是一个数据集片段，并且一个RDD的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算
- RDD提供了一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，不能直接修改，只能基于稳定的物理存储中的数据集创建RDD，或者通过在其他RDD上执行确定的转换操作（如map、join和group by）而创建得到新的RDD

Spark

RDD概念

- RDD提供了一组丰富的操作以支持常见的数据运算，分为“动作”（Action）和“转换”（Transformation）两种类型
- RDD提供的运算非常简单，都是类似map、reduce、filter、groupBy、join等粗粒度的数据操作，而不是针对某个数据项的细粒度修改
- Spark的执行可以看做是将输入RDD经过一系列“转换” 和“动作” 处理为输出的过程。这一系列处理称为一个Lineage（血缘关系）



Spark

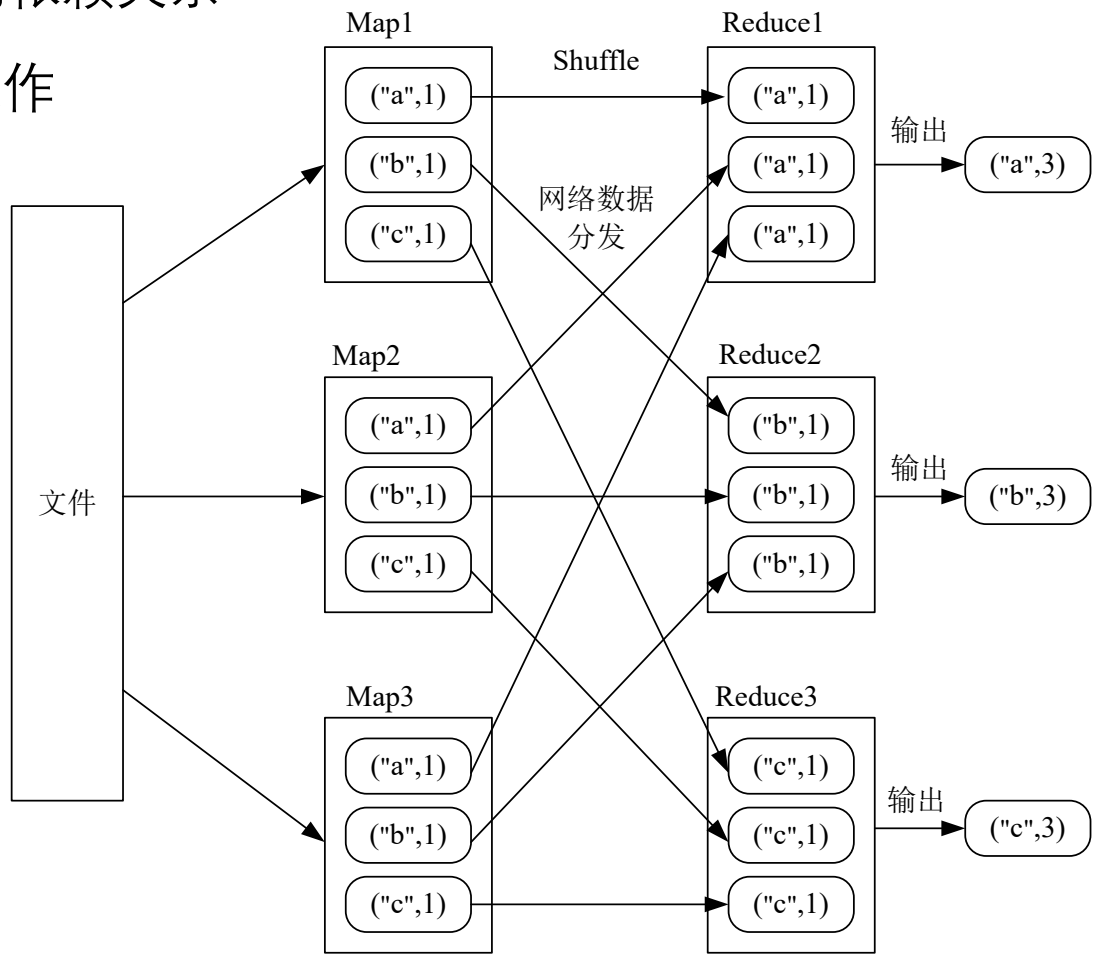
RDD之间的依赖关系

- 窄依赖（Narrow Dependency）
  - 窄依赖指的是每个父 RDD 的分区最多被一个子 RDD 的分区所使用。这意味着子 RDD 的每个分区只依赖于父 RDD 的一个或少数几个分区。
  - 窄依赖的优点是，由于依赖性较小，它通常不需要在节点间进行大规模数据传输。即使发生节点故障，任务的重新计算也只需涉及有限的数据重新获取。
- 宽依赖（Wide Dependency）
  - 宽依赖，也称为 Shuffle 依赖，是指父 RDD 的分区被多个子 RDD 的分区所依赖。这种类型的依赖需要将不同分区的数据通过网络进行混合（Shuffle），以便重新组合成新的分区。
  - 主要缺点是对性能的影响。Shuffle 操作涉及广泛的数据传输、磁盘 I/O 和网络 I/O，这可能成为整个 Spark 应用的瓶颈。此外，在 Shuffle 过程中，如果任何节点失败，可能需要重新计算多个分区的数据，从而增加恢复时间。

Spark

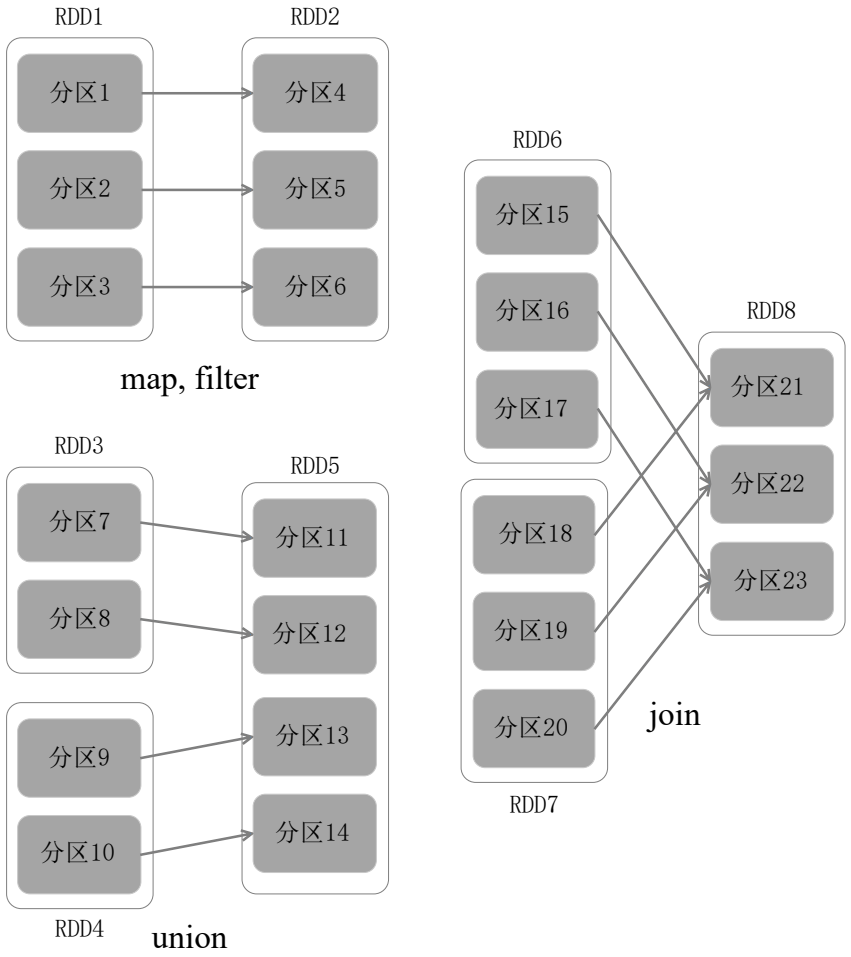
RDD之间的依赖关系

- Shuffle操作

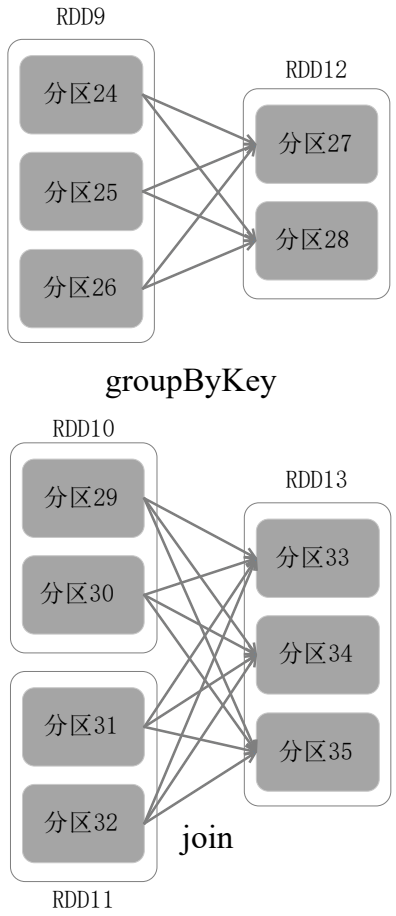


Spark

- 窄依赖指的是每个父 RDD 的分区最多被一个子 RDD 的分区所使用
- 宽依赖是指父 RDD 的分区被多个子 RDD 的分区所依赖



(a)窄依赖



(b)宽依赖

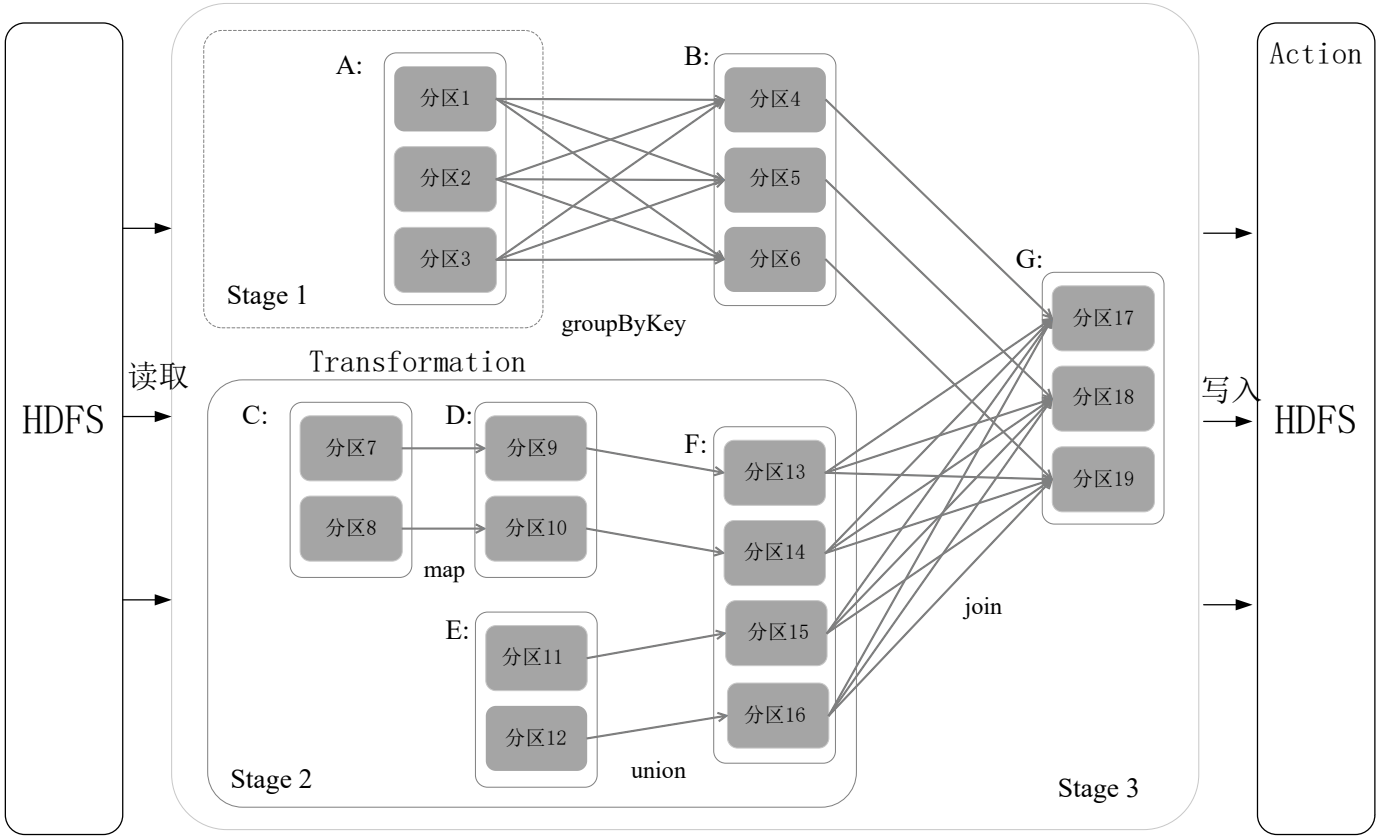
Spark

RDD阶段划分

- Spark 根据DAG 图中的RDD 依赖关系，把一个作业分成多个Stage。Stage划分的依据是窄依赖和宽依赖。窄依赖对于作业的优化很有利，宽依赖无法优化
- 逻辑上，每个RDD 操作都是一个fork/join（一种用于并行执行任务的框架），把计算fork 到每个RDD 分区，完成计算后对各个分区得到的结果进行join 操作，然后fork/join下一个RDD 操作
- 具体划分方法是
  - 在DAG中按依赖方向反向解析，遇到宽依赖就断开
  - 遇到窄依赖就把当前的RDD加入到Stage中
  - 将窄依赖尽量划分在同一个Stage中，可以实现流水线计算

Spark

RDD阶段划分

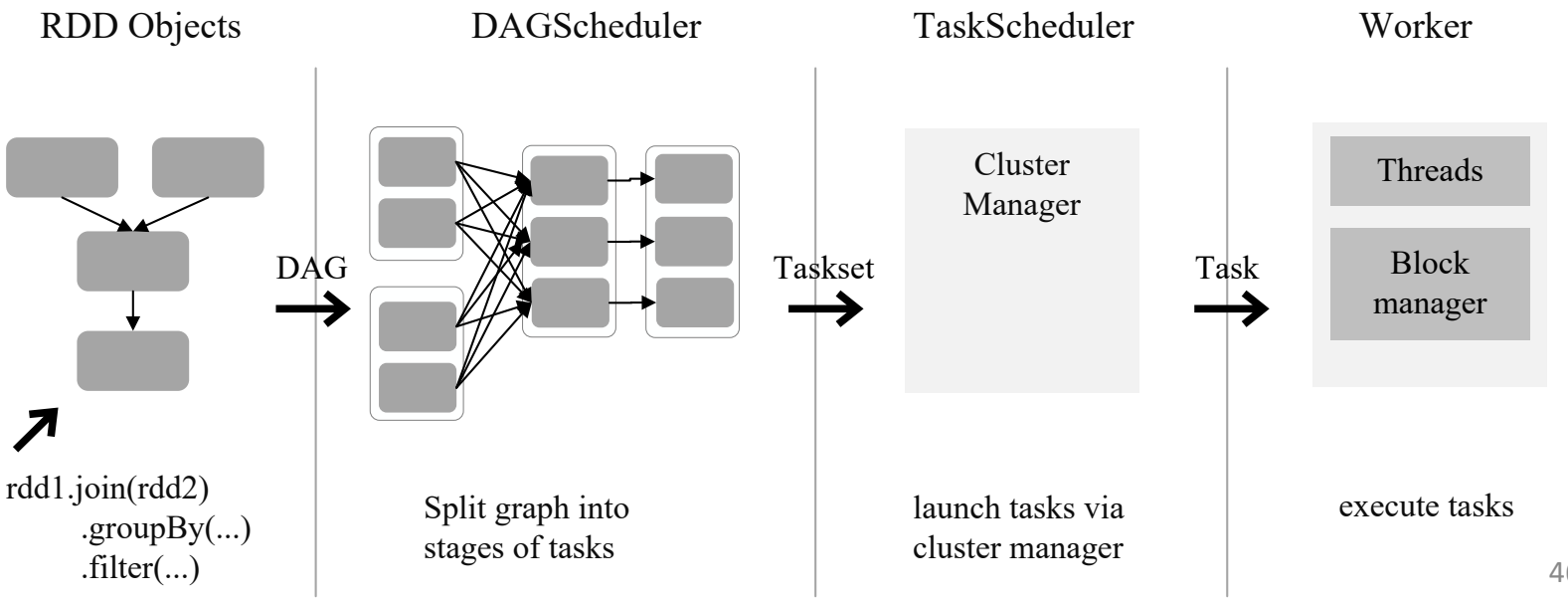


流水线优化示例

DAG被分成三个Stage。在Stage2中，分区7通过map操作生成的分区9，可以不用等待分区8到分区10这个map操作的计算结束，而是继续进行union操作，得到分区13。Map和union可以流水线执行。

Spark

- 从RDD的角度再次梳理Spark的运行过程
  1. 创建RDD对象；
  2. SparkContext负责计算RDD之间的依赖关系，构建DAG；
  3. DAGScheduler把DAG图分解成多个Stage，每个Stage中包含了多个Task，每个Task会被TaskScheduler分发给各个WorkerNode上的Executor执行。



Spark

Spark采用RDD以后能够实现高效计算的原因主要在于：

- 高效的容错性
  - 现有容错机制：数据复制或者记录日志
  - RDD：血缘关系、重新计算丢失分区、无需回滚系统、重算过程在不同节点之间并行、只记录粗粒度的操作
- 中间结果持久化到内存，数据在内存中的多个RDD操作之间进行传递，避免了不必要的读写磁盘开销
- 存放的数据可以是Java对象，避免了不必要的对象序列化和反序列化

Spark

案例：WordCount

```python
from pyspark.sql import SparkSession

# Initialize a SparkSession
spark = SparkSession.builder.appName("WordCount").getOrCreate()

# Read the input text file into an RDD (Resilient Distributed Dataset)
text_file = spark.sparkContext.textFile("path/to/input.txt")

# Split each line into words, flatten the result, and map each word to a tuple (word, 1)
counts = (text_file.flatMap(lambda line: line.split(" "))
                   .map(lambda word: (word, 1))
                   .reduceByKey(lambda a, b: a + b))

# Collect the counts and print them
for word, count in counts.collect():
    print(f"{word}: {count}")

# Save the counts to a file
counts.saveAsTextFile("path/to/output")

# Stop the Spark session
spark.stop()
```