Hadoop平台

BIG DATA

Hadoop平台

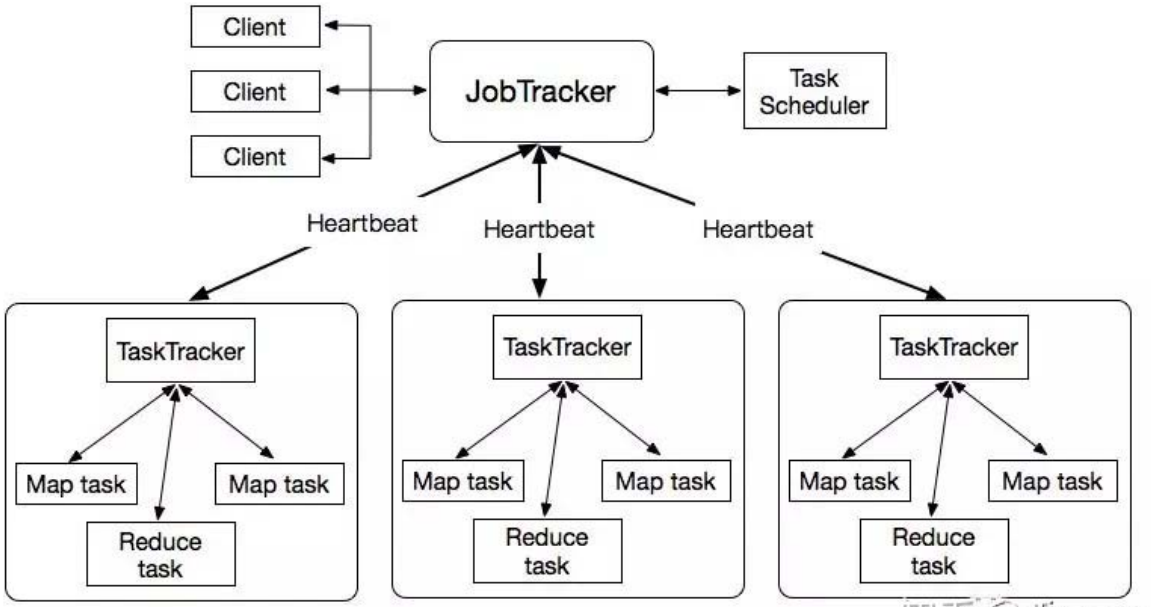
CONTENTS

3.1 YARN（Yet Another Resource Negotiator） 又一个资源调度器

■ YARN的产生背景

- MapReduce作为Hadoop中的分布式并行计算框架，不可避免地涉及资源调度
- 在Hadoop 1.x版本中，资源调度主要由NameNode中的JobTracker守护进程来完成，如下图所示：

- NameNode上运行JobTracker ，通过调度TaskTrakcer上运行的Map、Reduce任务来协同完成任务
- DataNode上运行TaskTracker，负责执行任务并将执行进度报告给JobTracker
- JobTracker同时负责作业调度（如任务下发等）和任务进度监控（任务跟踪，重启失败的任务，记录任务流水等）



3.1 YARN（Yet Another Resource Negotiator）

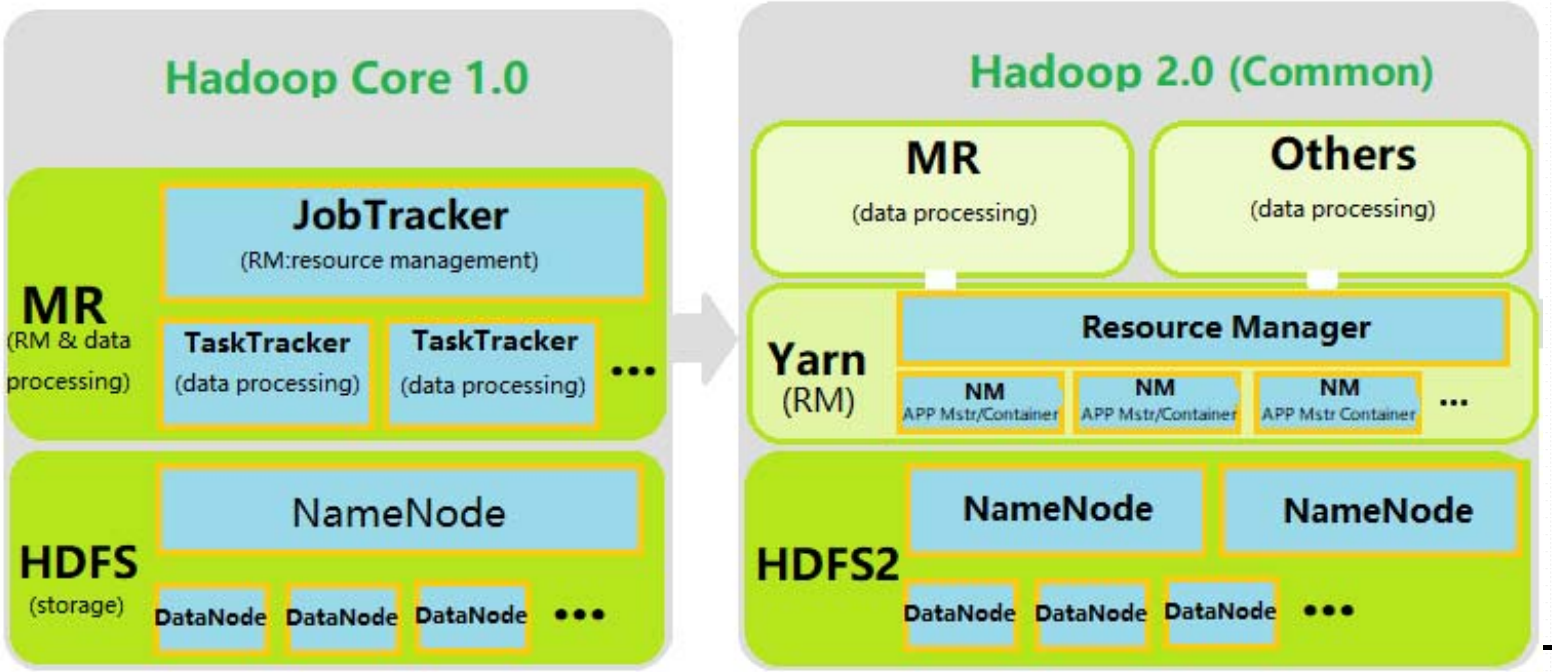
■ Hadoop 1.x调度存在的问题

- **单点故障问题**：JobTracker是MapReduce的集中处理点，任务繁重，既要进行资源调度，又要进行任务调度和进度监控，容易单点故障，系统可靠性较低
- **性能问题**：当集群规模很大， Map-Reduce job非常多时，会造成JobTrakcer很大的内存开销，潜在地增加了JobTracker fail风险。业界总结经验表明，Hadoop 1.x的Map-Reduce只能支持4000节点主机的上限
- **资源描述过于简单**：在TaskTracker端，仅以Map/Reduce task数目作为资源的表示过于简单，没有考虑到CPU/内存占用情况，如果两个内存消耗大的task被调度到一起，容易出现内存溢出
- **资源利用率问题**：在TaskTracker端，把资源强制划分为Map task slot和Reduce task slot，造成资源浪费
- **框架兼容性问题**：不支持MapReduce之外的计算框架，如Spark/Strom/flink等

3.1 YARN（Yet Another Resource Negotiator）

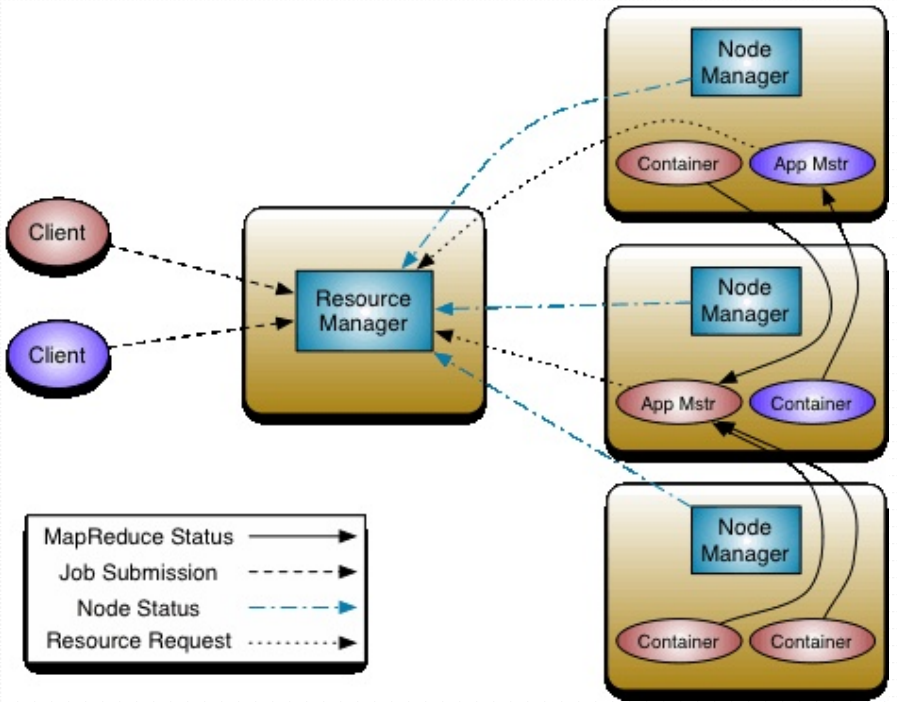
■ YARN的设计原则

- 将资源调度与任务调度分离，资源中考虑了CPU和内存的情况
- 提供更高层的API，向用户隐藏资源管理细节，从而支持更多并行计算框架（如Spark）



3.2 YARN的基本组件

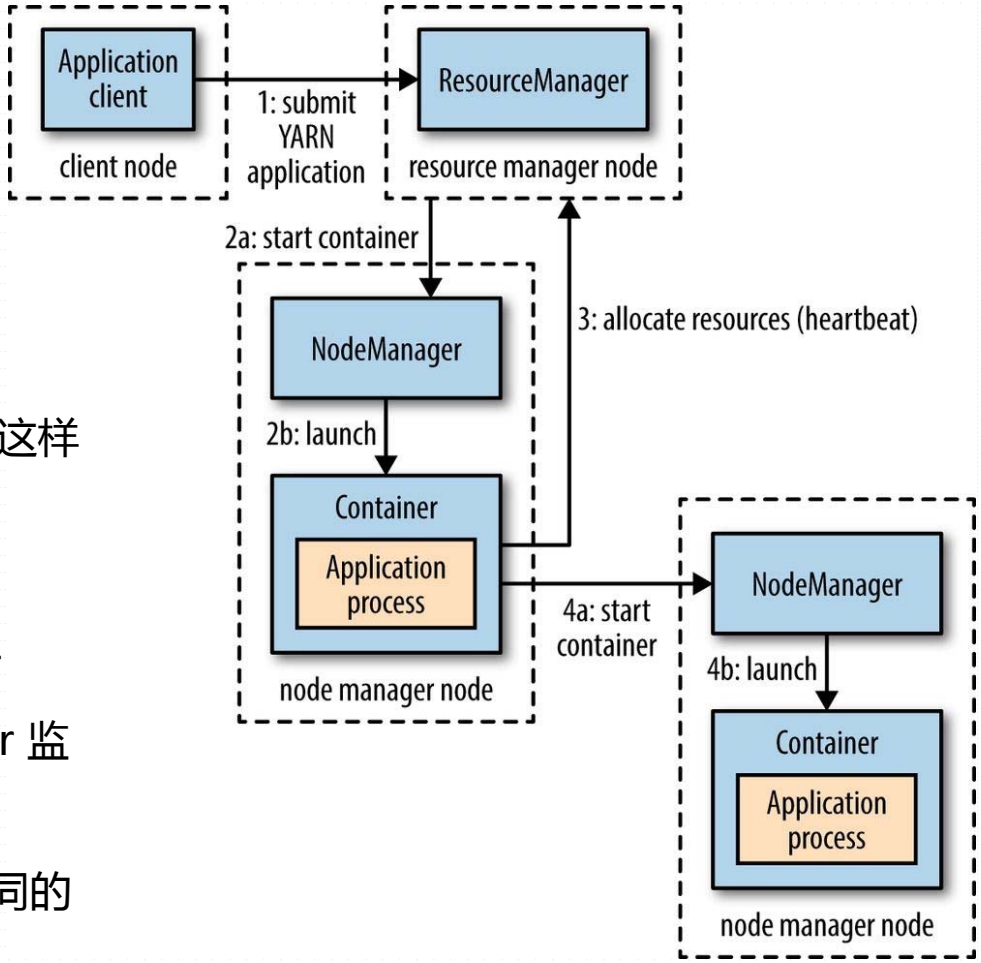
- **Resource Manager**：全局的资源管理者，负责集群的资源统一管理和分配调度，1个集群只有1个
  - 处理客户端的请求
  - 启动监控 ApplicationMaster
  - 监控 NodeManager
  - 资源分配调度
- **NodeManager**：负责节点自身的资源管理使用，有多个
  - 定时向RM汇报本节点的资源情况
  - 单个节点上的资源任务管理
  - 执行 NodeManager 的命令：启动 Container
  - 协助 ApplicationMaster处理任务
- **ApplicationMaster**：
  - 每当Client 提交一个 Application 时候，就会新建一个 ApplicationMaster 。由这个 ApplicationMaster 去与 ResourceManager 申请容器资源，获得资源后会将要运行的程序发送到容器上启动，然后进行分布式计算。
- **Container**：资源抽象，封装了该节点上的多维度资源
  - 封装了 CPU、Memory 等资源的一个容器
  - 是一个任务运行环境的抽象，且该任务只能使用该 container 中描述的资源



3.3 YARN的运行机制

1. 客户端向yarn提交作业，首先找 RM 分配资源；
2. RM 接收到作业以后，会与对应的 NM 建立通信；
3. RM 要求 NM 创建一个 Container 来运行 ApplicationMaster 实例；
4. ApplicationMaster 会向 RM 注册并申请所需资源，这样 Client 就可以通过 RM 获知作业运行情况；
5. RM 分配给 ApplicationMaster 所需资源，ApplicationMaster 在对应的 NM 上启动 Container
6. Container 启动后开始执行任务，ApplicationMaster 监控各个任务的执行情况并反馈给 RM

- 其中 ApplicationMaster 是可插拔的，可以替换为不同的应用程序。



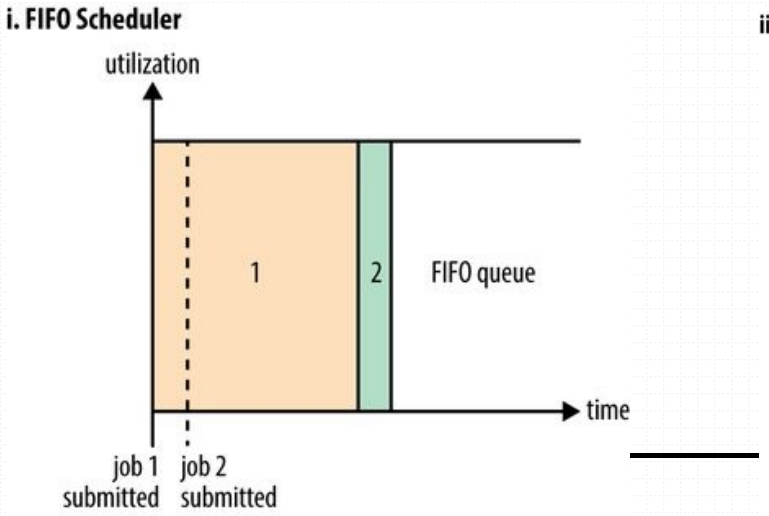
3.4 Hadoop 1.x中的MapReduce 1与YARN对比

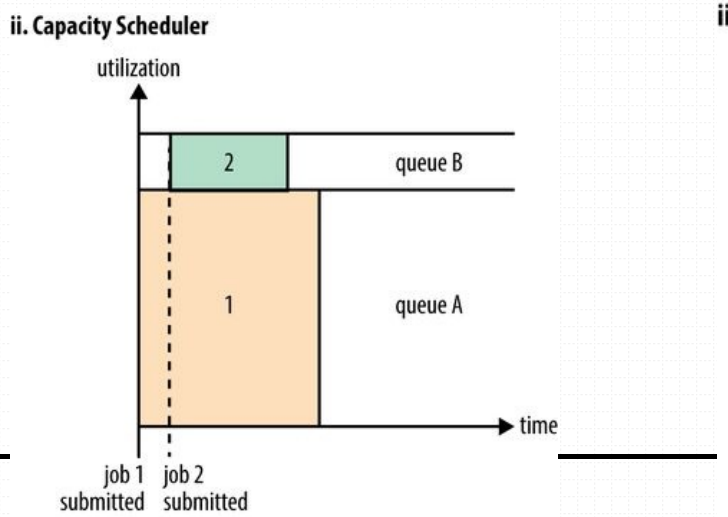
对比项	MapReduce 1	YARN
NameNode节点	JobTracker	ResourceManager、Application Master
DataNode节点	TaskTracker	节点管理器
资源封装	Map/Reduce Slot	容器Container
可扩展性	最大支持4000个节点和4万个任务	可扩展到1万个节点和10万个任务
可用性	较低	较高
利用率	Slot静态分布，资源利用率低	通过资源池管理，利用率高
框架兼容性	仅支持MapReduce	支持MapReduce、Spark、Storm等计算框架

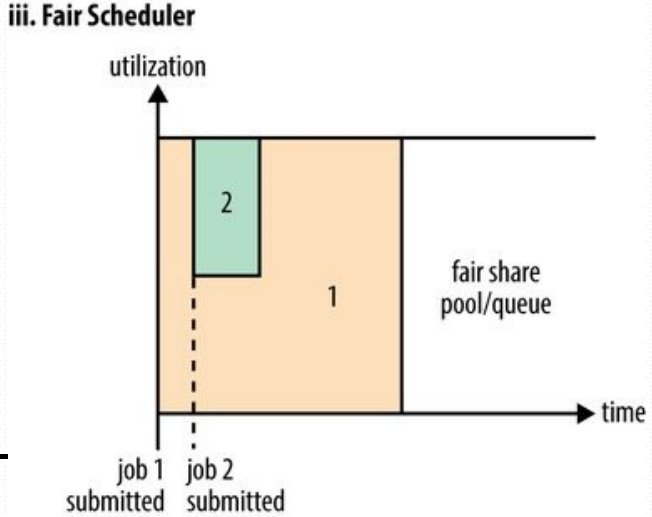
3.5 YARN中的调度策略

- 当应用发出的资源请求不能马上得到满足时，需要使用一定的调度策略

调度器	FIFO调度器	Capacity调度器	公平调度器
基本思想	将应用放置在一个队列中，按照提交顺序依次运行应用	预留一个独立的专门队列保证小作业一提交就可以启动	调度器在所有运行的作业之间动态平衡资源
优点	简单易懂，不需要任何配置	能保证小作业的顺利执行	即得到了较高的集群利用率，又能保证小作业能及时完成
缺点	不适合共享集群，大的应用会长时间占用集群中所有资源，小作业容易被阻塞	以牺牲整个集群的利用率为代价，大作业执行时间比FIFO更长。	调度规则更复杂







Hadoop平台

CONTENTS

4.1 分布式并行编程

■ 摩尔定律

- CPU性能大约每隔18~24个月翻一番
- 从2005年开始摩尔定律逐渐失效 ，需要处理的数据量快速增加，人们开始借助于分布式并行编程来提高程序性能

■ 分布式处理

- 分布式程序运行在大规模计算机集群上，可以并行执行大规模数据处理任务，从而获得海量的计算能力
- 谷歌公司最先提出了分布式并行编程模型MapReduce，Hadoop MapReduce是它的开源实现，后者比前者使用门槛低很多

4.1 分布式并行编程

- 为什么需要MapReduce？

表1：传统计算框架和MapReduce对比

	传统并行计算框架	MapReduce
集群架构/容错性	共享式，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，价格高，扩展性差	普通PC机，便宜，扩展好
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

4.1 分布式并行编程

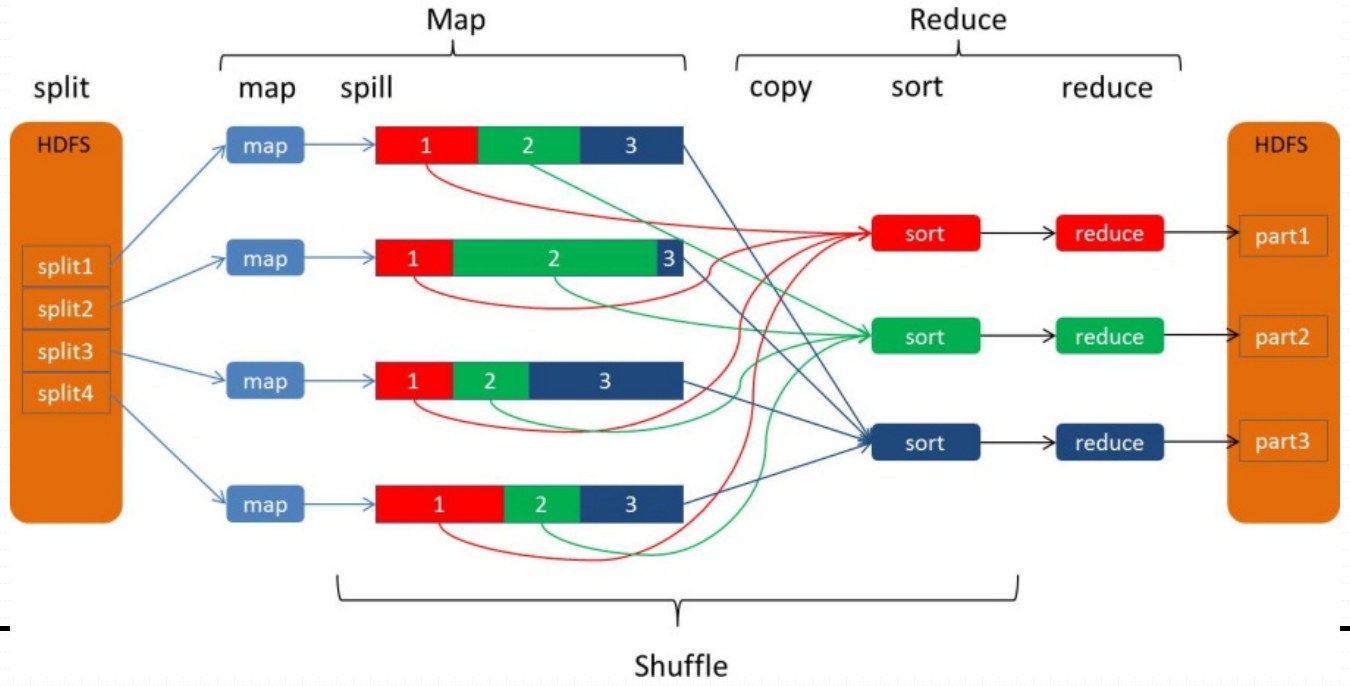
- MapReduce将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：Map和Reduce
- Hadoop框架是用Java实现的，但是，MapReduce应用程序则不一定要用Java来写

函数	输入	输出	说明
Map	**<k_1,v_1>** **如：** **<行号, “a b c”>**	**List(<k_2,v_2>)** **如：** **<“a”,1>** **<“b”,1>** **<“c”,1>**	1. 将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理 2. 每一个输入的<k_1,v_1>会输出一批<k_2,v_2>的列表，<k_2,v_2>是计算的中间结果
Reduce	**<k_2, List(v_2)>** **如：** **<“a”,<1,1,1>>**	**<k_3,v_3>** **<“a”,3>**	输入的中间结果<k_2,List(v_2)>中的List(v_2)表示是一批属于同一个k_2的value的列表

在Map和Reduce之间，存在一个**Shuffle**操作，会根据Map输出的List(<k_2,v_2>)中的k_2进行合并，即将相同k_2的value值合并到List中，并按k_2排序，即将**List(<k2,v2>)转换为<k2, List(v2)>**

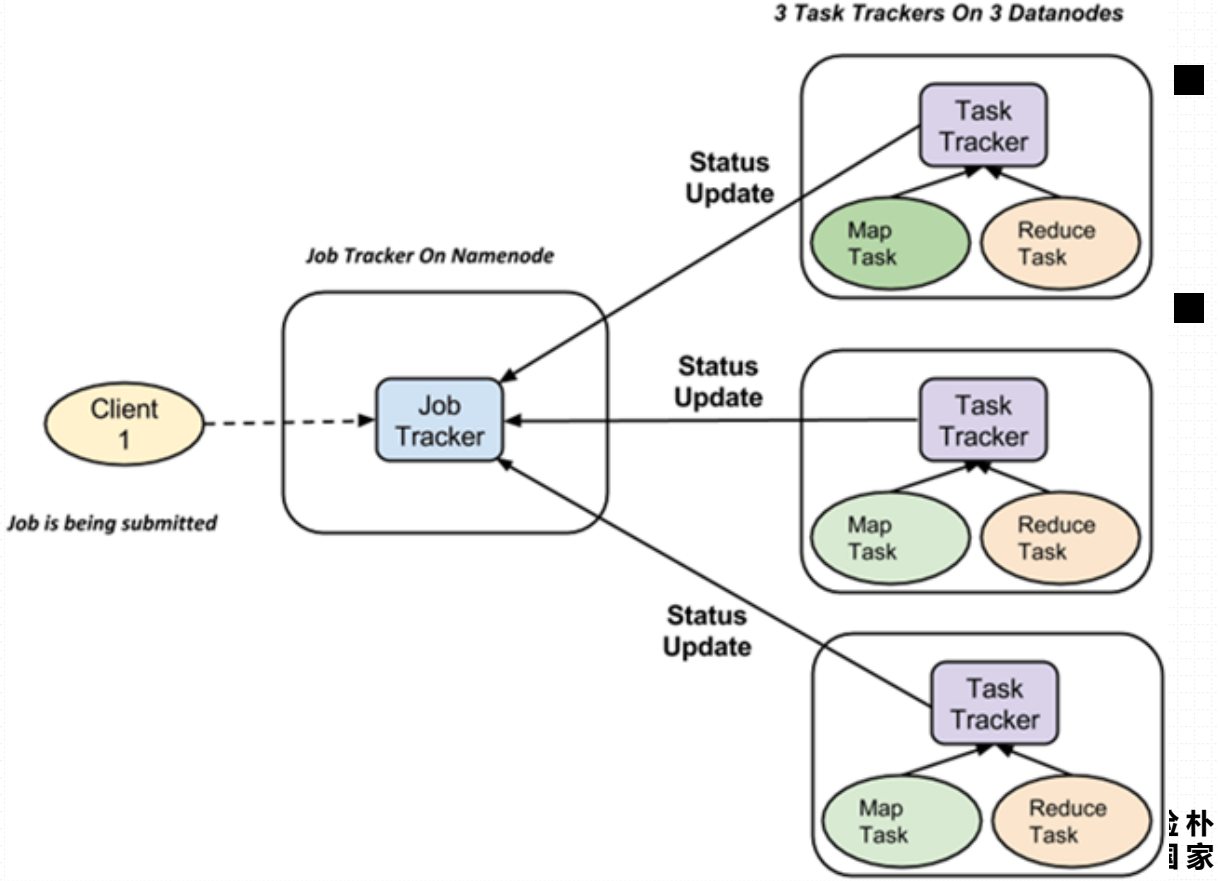
4.1 分布式并行编程

- MapReduce采用“分而治之”策略，一个存储在分布式文件系统中的大规模数据集，会被切分成许多独立的分片（split），这些分片可以被多个Map任务并行处理
- MapReduce设计的一个理念就是“计算向数据靠拢”，而不是“数据向计算靠拢”。因为，移动数据需要大量的网络传输开销



4.2 MapReduce体系结构（Hadoop 1.x版本）

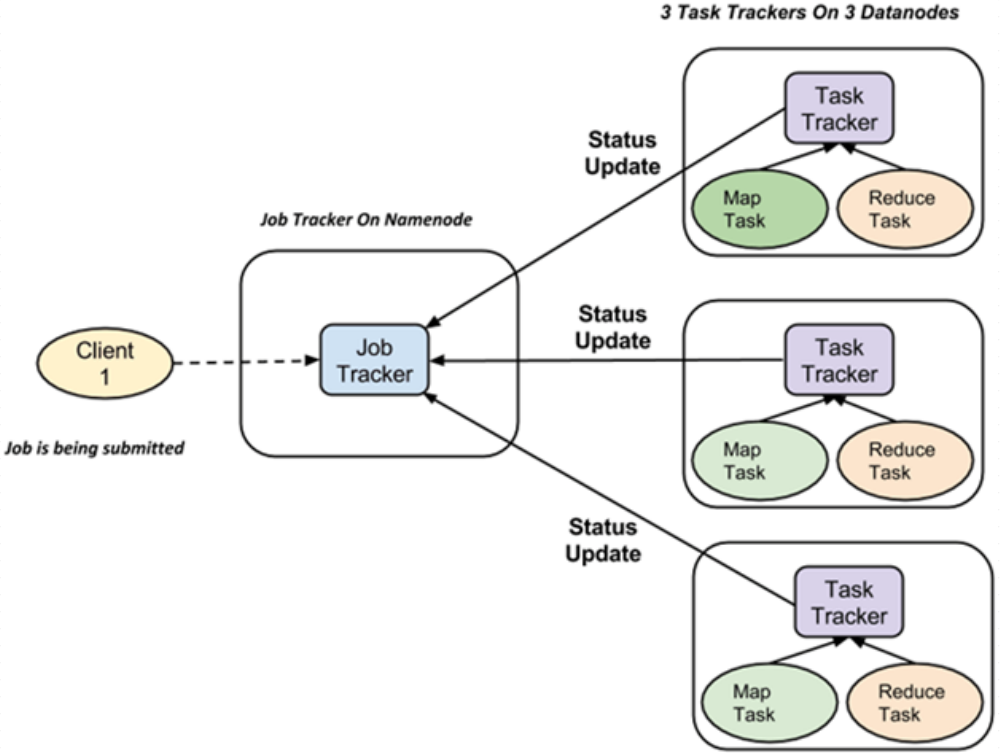
MapReduce体系结构由 Client、JobTracker、TaskTracker以及Task四部分组成



- Client
  - MapReduce程序通过Client提交到JobTracker端
  - 用户通过Client提供的接口查看作业运行状态
- JobTracker
  - JobTracker负责资源监控和作业调度
  - JobTracker 监控所有TaskTracker与Job的状况，一旦失败，就将任务转移到其他节点
  - JobTracker 跟踪任务执行进度、资源使用量等，并将这些信息告诉任务调度器（TaskScheduler）
  - TaskScheduler在资源出现空闲时，选择合适的任务去使用这些资源

4.2 MapReduce体系结构（Hadoop 1.x版本）

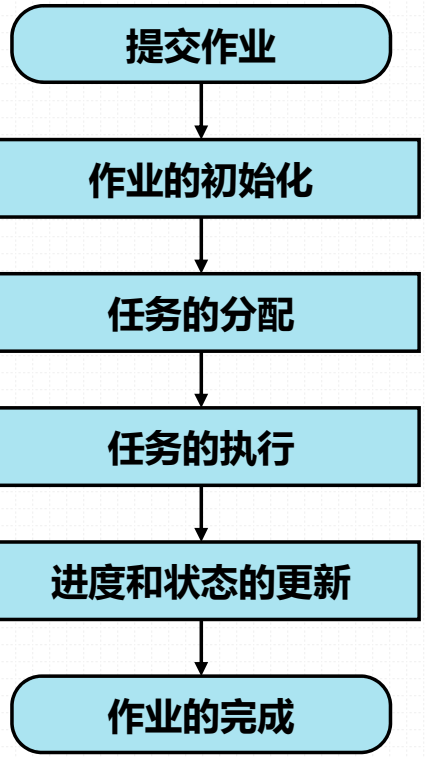
MapReduce体系结构主要由 Client、JobTracker、TaskTracker以及Task四部分组成

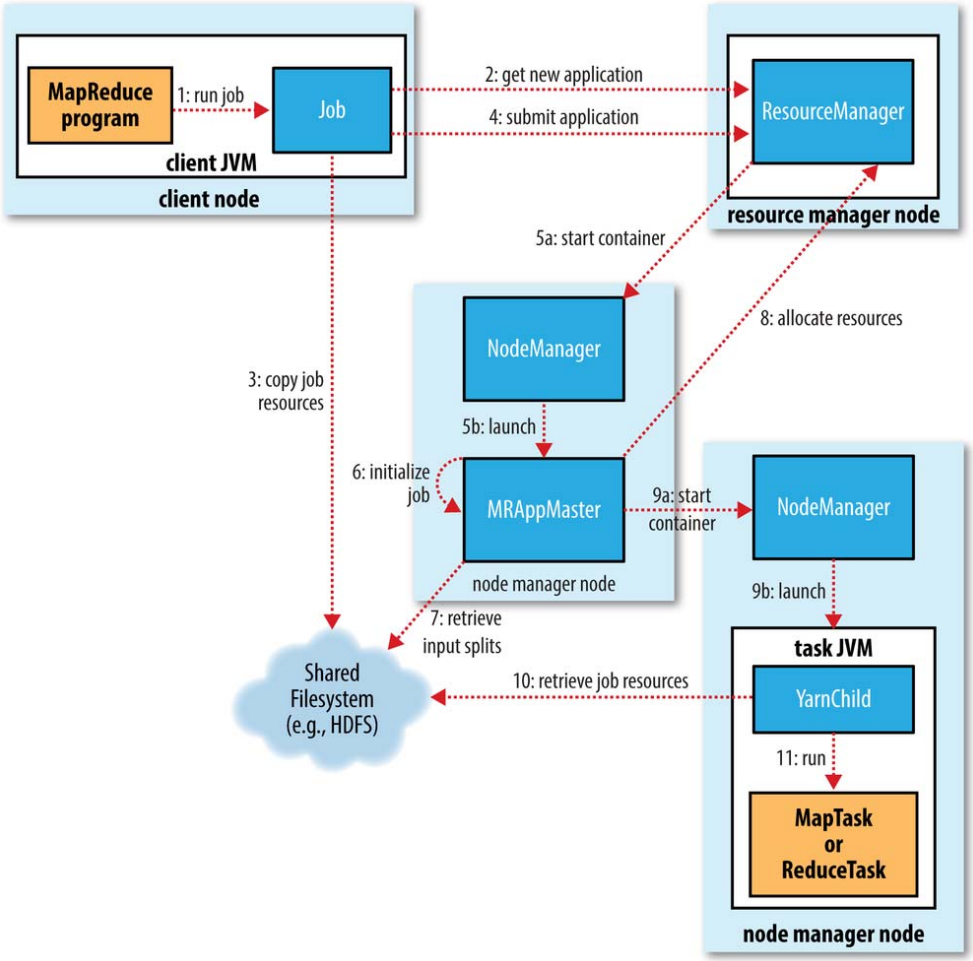


- TaskTracker
  - 周期性地通过“心跳”将本节点上资源的使用情况和任务进度汇报给JobTracker，同时接收JobTracker 命令并执行相应操作
  - 使用“slot”等量划分本节点资源（CPU、内存等）。一个Task 获取到一个slot 后才有机会运行，而Hadoop调度器将各个TaskTracker上的空闲slot分配Task使用, slot 分为Map slot 和Reduce slot ，分别供MapTask 和Reduce Task 使用
- Task
  - Task 分为Map Task 和Reduce Task 两种，均由TaskTracker 启动

4.2 MapReduce工作机制（Hadoop 2.x版本）

基本流程如下：

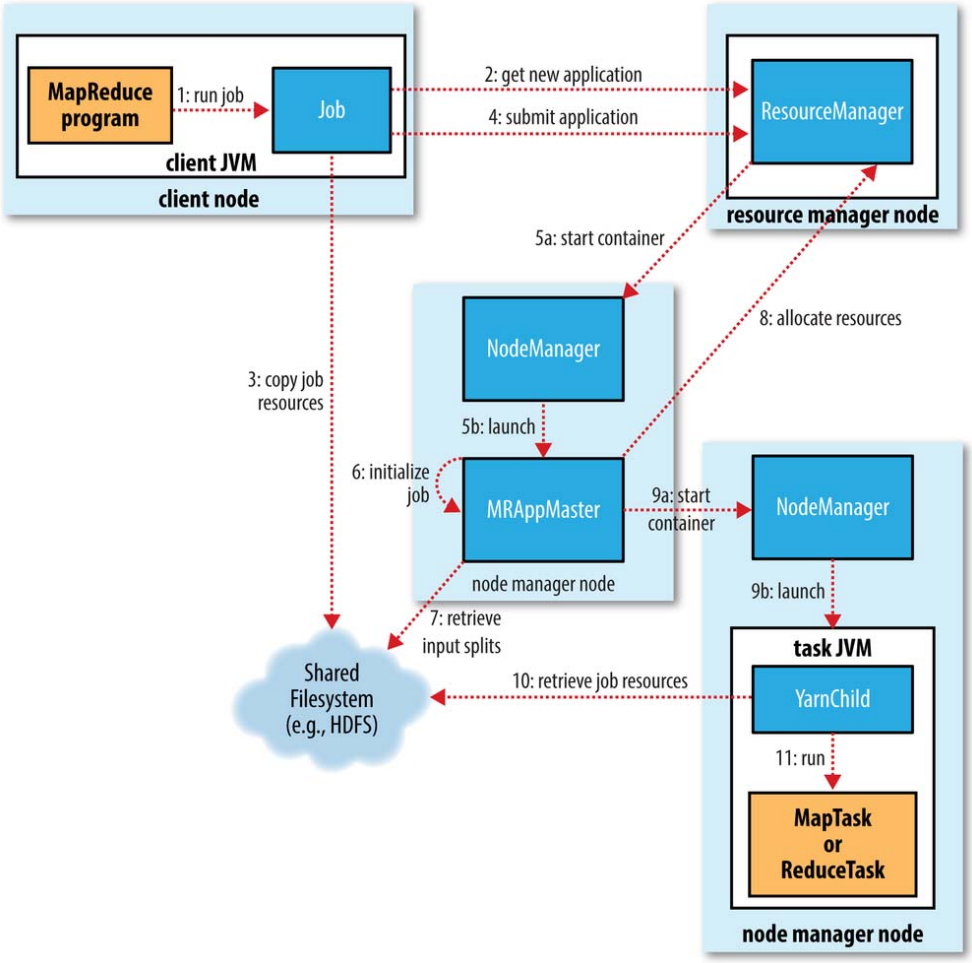




4.2 MapReduce工作机制（Hadoop 2.x版本）

类比关系：MapReduce vs 企业项目管理

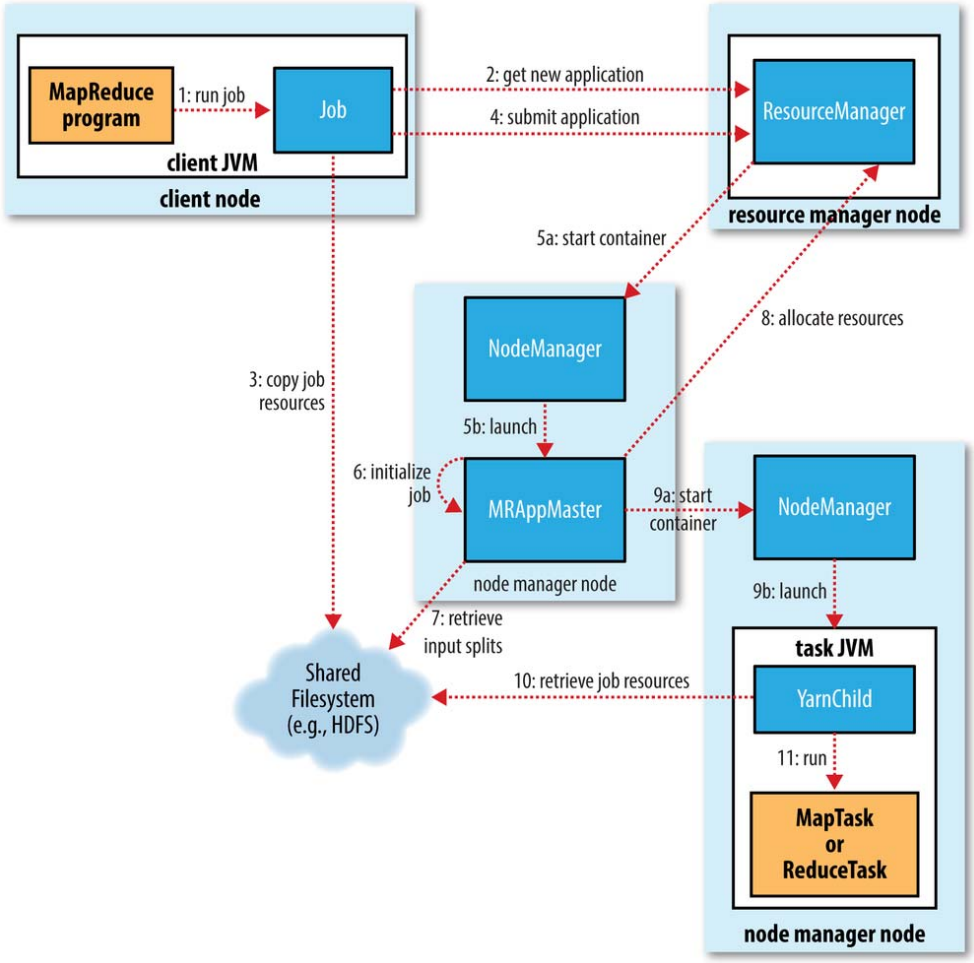
Resource Manager	公司CEO，负责协调资源
Application Master	项目经理，负责项目管理
Job	项目
Node Manager	任务负责人，管理任务
Task	任务
客户端	客户
容器Container	完成任务所需的资源
JVM	运行任务的通用工具
YARN	公司的管理制度



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程一：作业的提交

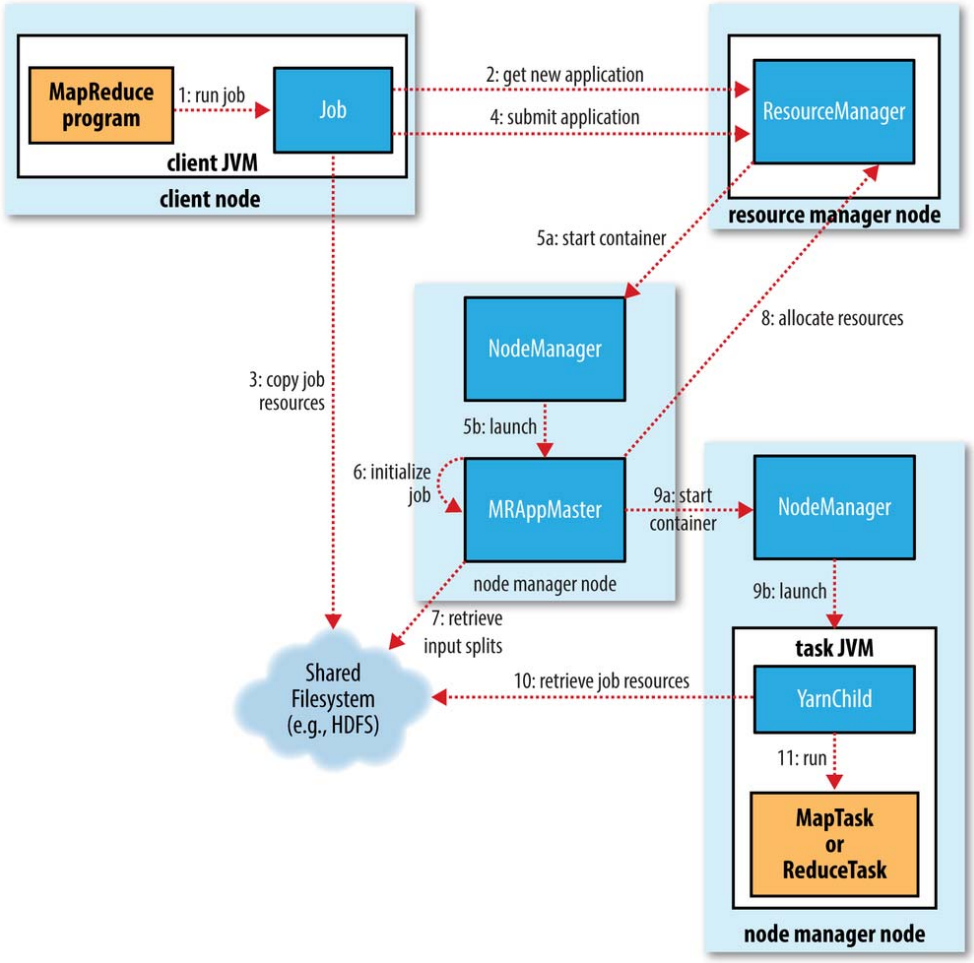
- 客户端通过Job对象的submit()方法创建一个JobSubmitter实例，该实例向资源管理器请求一个新应用ID作为MapReduce作业ID
- 检查作业的输出说明，若没有指定输出目录或输出目录已经存在，则返回错误
- 计算作业的输入分片，若输入分片无法计算，则返回错误
- 将运行作业所需资源（包括JAR文件、配置文件和输入分片）复制到以作业ID命名的指定目录
- 通过调用资源管理器的submitApplication()方法提交作业



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程二：作业的初始化

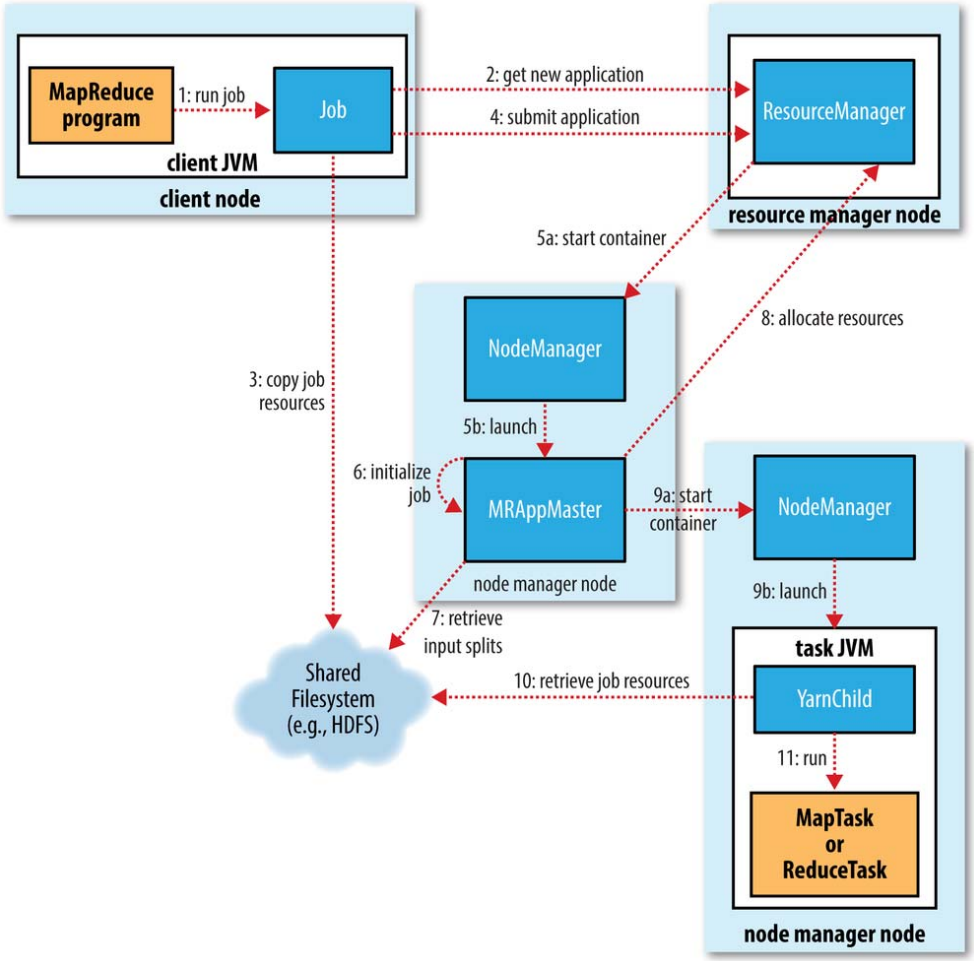
- 资源管理器收到submit Application()消息后，将请求传递给YARN调度器
- 调度器分配一个容器，然后资源管理器在节点管理器的管理下在容器中启动application master进程
- application master接受输入分片，并对每一个分片创建一个map任务对象以及多个reduce任务对象。任务ID在此时分配。
- application master根据作业大小决定任务运行方式
- 最后，在运行任务之前，application master调用setup Job()函数设置OutputCommitter指定输出方式，默认为FileOutputCommitter，即输出到文件目录。



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程三：任务的分配

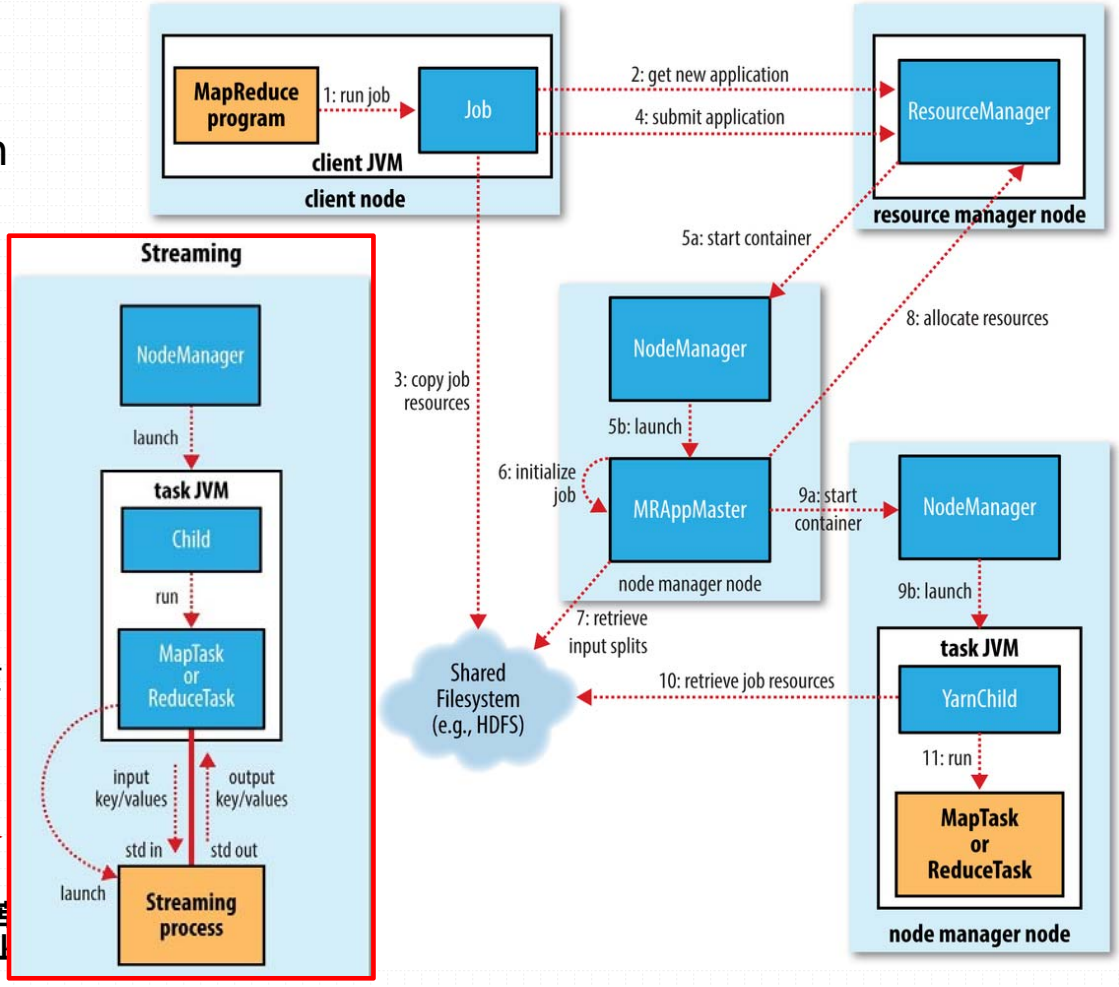
- application master为作业中的所有map和reduce任务向资源管理器请求容器（先请求map任务的容器，当有5%的map任务完成时，再请求reduce任务的容器）
- reduce任务可以在集群中的任意位置运行，map任务则尽可能满足数据的“本地化”限制，包括数据本地化（data local，分片数据和任务在同一节点）和机架本地化（rack local，数据和任务在同一机架的不同节点）
- 请求容器时，为任务指定内存和CPU需求，默认每个map和reduce任务分配1024MB内存和1个虚拟内核。



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程四：任务的执行

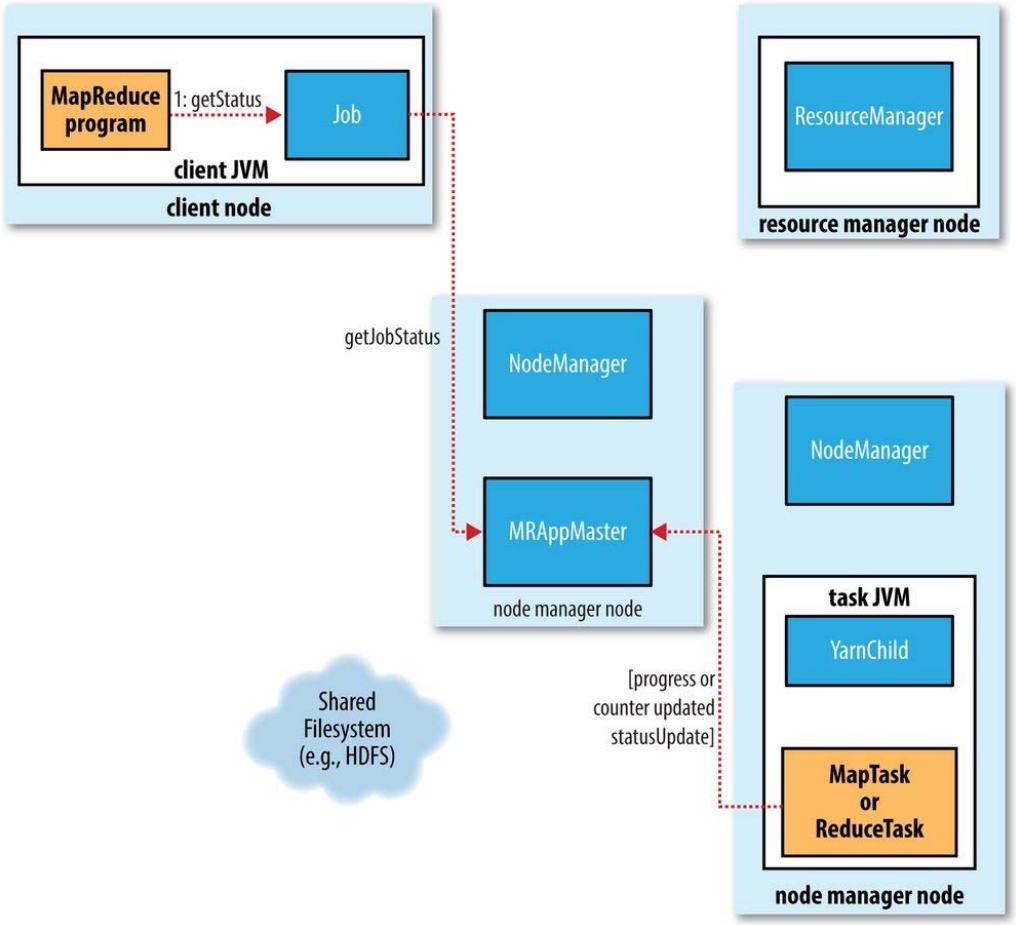
- 一旦为任务分配了某个节点上的容器，application master与节点管理器通信来启动容器
- 任务的执行是通过主类为YarnChild的Java程序在指定的JVM中运行map或reduce任务
- 在运行任务之前，将任务需要的资源本地化，包括作业的配置、JAR文件和所有来自分布式缓存的文件
- **Streaming**：允许用户用任何可执行程序和脚本作为mapper和reducer来完成Map/Reduce任务（如通过标准输入和输出流）。



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程五：进度和状态更新

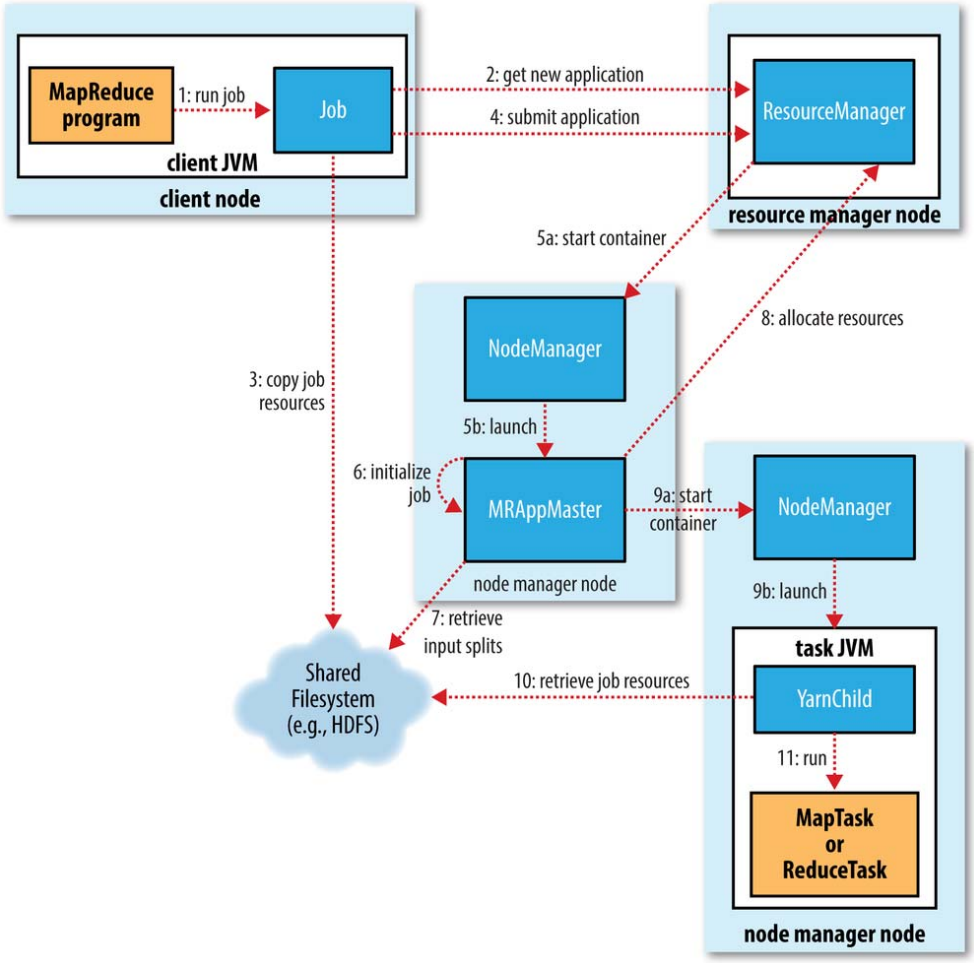
- **状态信息包括**：任务和作业的状态（运行中、成功完成、失败）、map和reduce进度、作业计数器的值、状态消息描述
- **进度跟踪**：对于map任务，其进度为已处理的输入数据的比例，对于reduce任务，情况更复杂一些，会估算已处理的reduce输入比例。
- 当map或reduce任务运行时，子进程每隔3秒钟通过接口向父进程application master报告进度和状态（包括计数器），application master形成作业的汇聚视图
- 作业期间，客户端每秒钟轮一次application master以查看作业的最新状态。



4.2 MapReduce工作机制（Hadoop 2.x版本）

流程六：作业的完成

- 当application master收到最后一个任务已完成通知后，便将作业状态设置为“成功”
- Job轮询状态时，发现任务已成功完成，打印一条消息告知用户，并输出Job的统计信息和计数值到控制台
- 通过HTTP发送通知：客户端可以通过在Hadoop中进行参数配置让Job结果以HTTP的方式发送给用户
- application master和任务容器清理其工作状态（中间输出将被删除），并调用OutputCommitter的commitJob()方法对作业结果信息进行存档，以便用户日后查询。



4.2 MapReduce故障处理机制（Hadoop 2.x版本）

在实际应用中，由于**用户代码错误**、**进程崩溃**、**机器故障**等原因，都可能导致MapReduce运行失败。Hadoop的强大之处在于，拥有一套完整的失败处理机制。

情况1：任务运行失败

- **map或reduce任务中的用户代码抛出异常**（最常见）：任务的JVM在退出前向其父进程application master发送错误报告，错误报告被记录到用户日志；application master将此次任务标记为失败，并释放容器以便资源可以被其他任务使用
- **任务JVM突然退出**：可能是由于JVM软件缺陷等特殊原因导致，此时节点管理器会注意到进程已经退出，并通知application master将此次任务标记为失败
- **任务挂起**：一旦application master在一段时间（通常为10分钟，可通过timeout参数设置）内没有收到任务的进度更新，则将任务标记为失败，并杀死任务JVM进程（timeout=0时永远不会杀死挂起的进程，应避免这样的设置，会降低集群效率）

4.2 MapReduce故障处理机制（Hadoop 2.x版本）

情况1：任务运行失败

- **任务重启**：当一个任务尝试失败后， application master将重新调度该任务的执行，并避免选择在之前失败过的节点上执行，若一个任务失败超过4次（可通过maxattempts参数设置），则整个作业失败

- **其他说明**：一些应用程序中，允许少数任务失败而照样输出结果，此时，可通过maxpercent参数设置作业允许的失败任务最大百分比

思考：通常具备什么特点的作业会允许一定比例的任务失败？

4.2 MapReduce故障处理机制（Hadoop 2.x版本）

情况2：application master运行失败

- **失败检测**：application master由YARN调度器进行管理，YARN在application master运行失败之后会进行几次尝试，次数由max-attempts参数指定，默认为2，当失败次数超过2次时，作业失败。

- **恢复过程**：正常情况下，application master向资源管理器发送周期性心跳，当application master失败时，资源管理器将检测到该失败并在一个新的容器中开始一个新的application master实例，使用作业历史来恢复失败的应用程序所运行任务的状态，使其不必重新运行。

- **客户端**：客户端会定期向application master发送请求得到其更新状态，若请求超时，则客户端会向资源管理器请求新的application master地址。

4.2 MapReduce故障处理机制（Hadoop 2.x版本）

情况3：节点管理器运行失败

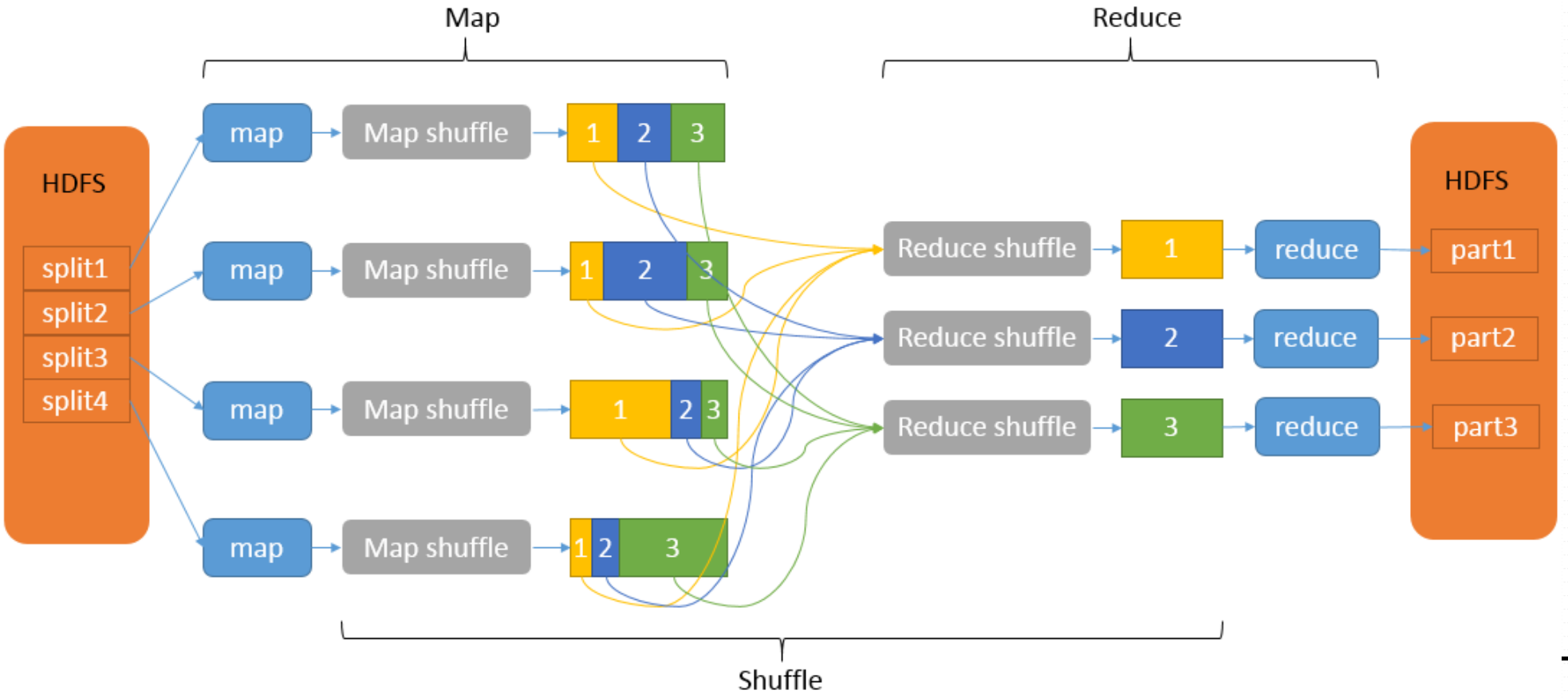
- **失败检测**：节点管理器会定期向资源管理器发送心跳信息，若资源管理器超过10分钟（可通过参数指定）没有收到心跳信息，则通知该节点管理器，将其从自己的节点池中移除
- **节点拉黑机制**：如果一个节点管理器上有超过三个任务失败，该节点将被“拉黑”，application master会尽量将任务调度到其他节点上

情况4：资源管理器运行失败

- 资源管理器失败是非常严重的问题，没有资源管理器，作业和任务容器将无法启动。因此，在默认的Hadoop配置中，资源管理器存在单点故障的问题。
- **解决方案**：可采用双机热备机制，运行一对资源管理器，当主资源管理器失效时，让备用资源管理器接替，且客户端不会感到明显的中断。

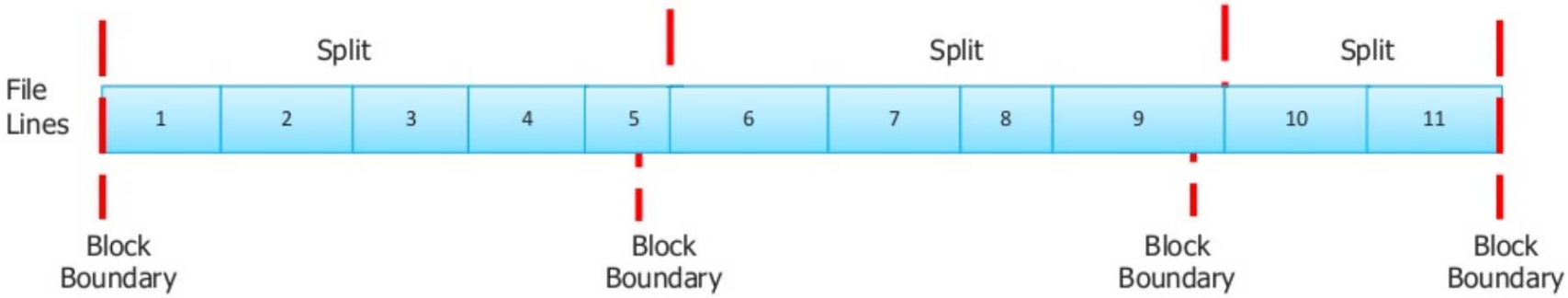
4.3 MapReduce工作流程

MapReduce整体工作流程如下，后面将展示归并排序&词频统计实例。



4.3.1 MapReduce--Split

- HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。split 是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。



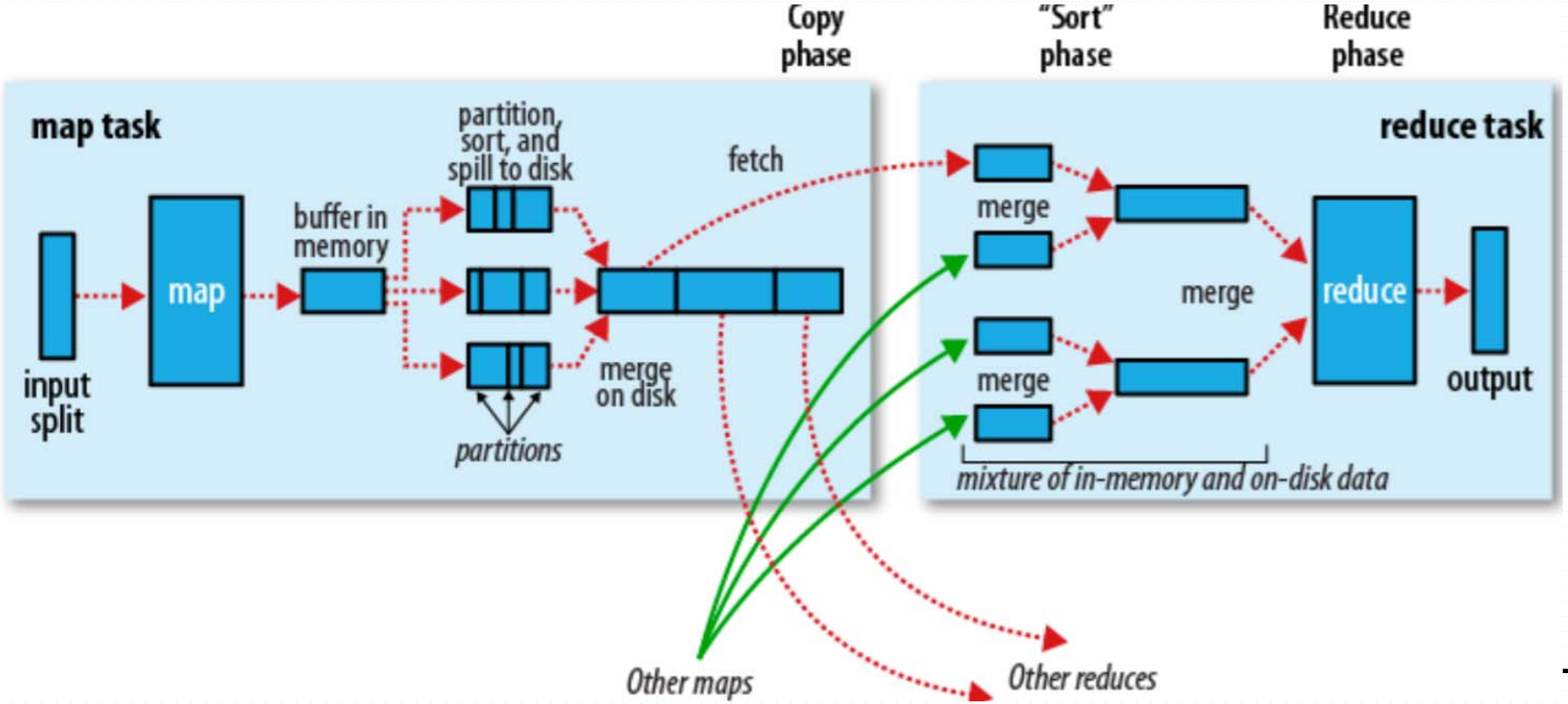
→ Logical records do not fit neatly into the HDFS blocks.

→ Logical records are lines that cross the boundary of the blocks.

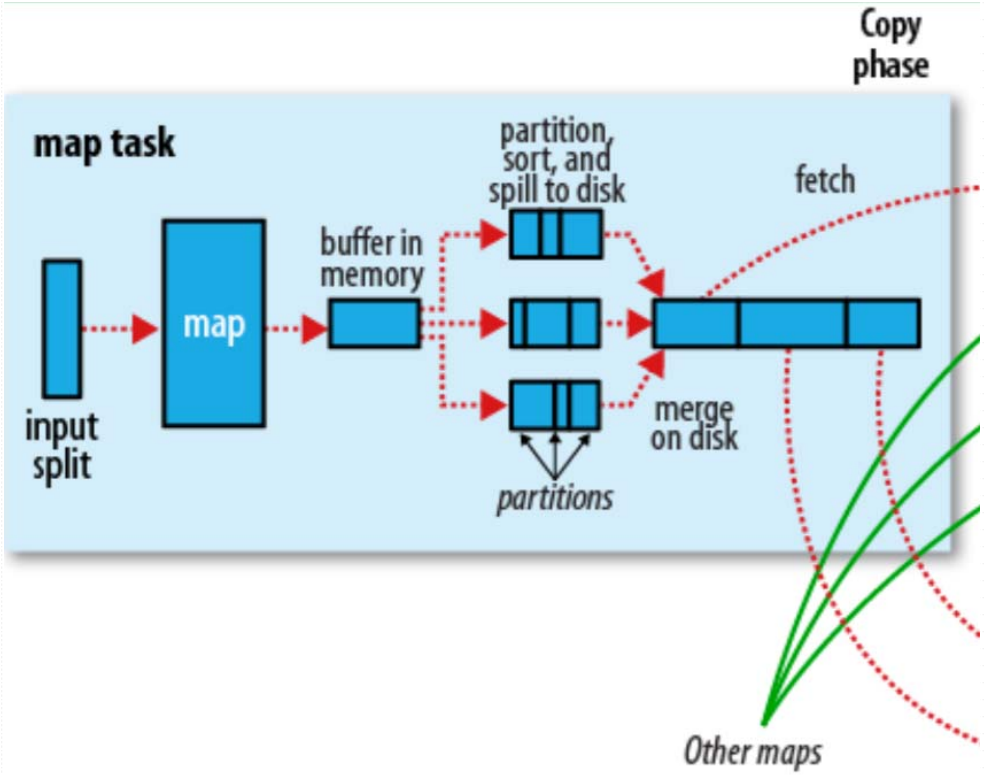
→ First split contains line 5 although it spans across blocks.

4.3.2 MapReduce--Shuffle

- Map函数开始产生输出时，并不会立刻写到磁盘上，而是利用缓冲的方式写到内存并进行预排序。当缓冲区内容达到阈值时（如80%），Map会被阻塞，同时将缓冲区内容溢出（Spill）到磁盘上。

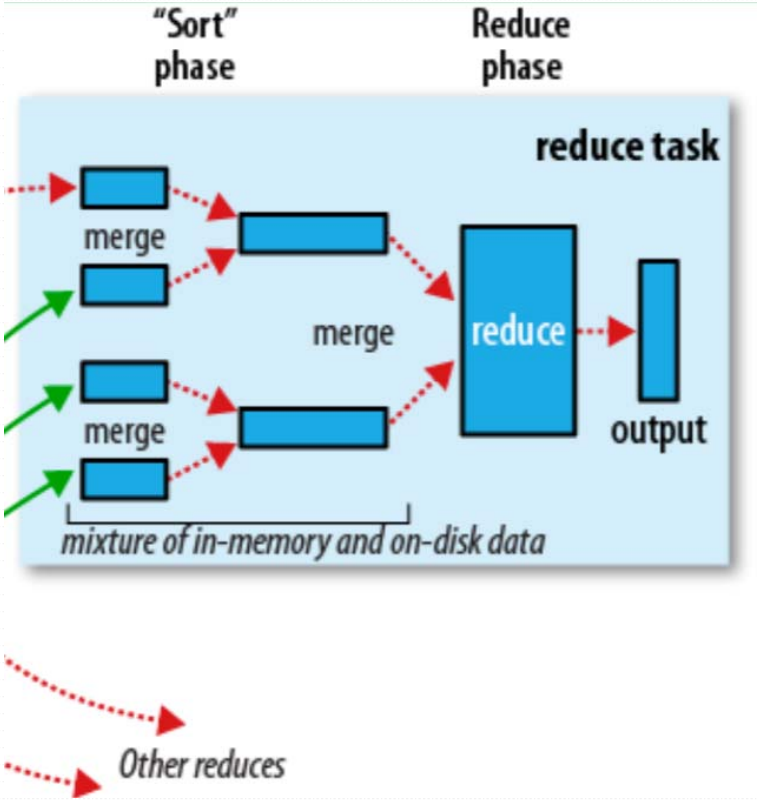


4.3.2 MapReduce--Shuffle



- 每个Map任务分配一个缓存,MapReduce默认100MB缓存
  - 设置溢写比例0.8
  - 分区默认采用哈希函数
  - 排序是默认操作，且排序后进行合并（Combine）
  - 排序后的合并不会改变最终结果
- Map任务全部结束前进行归并，归并得到一个大文件，放在本地磁盘。文件归并时，如溢写文件数量大于预定值（默认3）则可再次启动Combiner，以减少文件数量
- JobTracker会一直监测Map任务的执行，并通知Reduce任务来领取数据

4.3.2 MapReduce--Shuffle



- Reduce任务通过RPC询问Map任务是否已经完成，若完成，则通过HTTP领取数据
- Reduce领取数据先放入缓存，来自不同Map机器，先归并，再合并，写入磁盘
- 多个溢写文件归并成一个或多个大文件，文件中的键值对是排序的
- 当数据很少时，不需要溢写到磁盘，直接在缓存中归并，然后输出给Reduce

4.4 实例1 -- WordCount

算法思想

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
see spot run run spot run see the cat	cat 1 run 3 see 2 spot 2 the 1

4.4 Map Reduce

算法思想

如何快速数出一摞牌中有多少张黑桃？

MapReduce方法

1. 给在座的所有玩家中分配这摞牌
2. 让每个玩家数自己手中的牌有几张是黑桃，然后把这个数目汇报给你
3. 你把所有玩家告诉你的数字加起来，得到最后的结论

4.4 Map Reduce

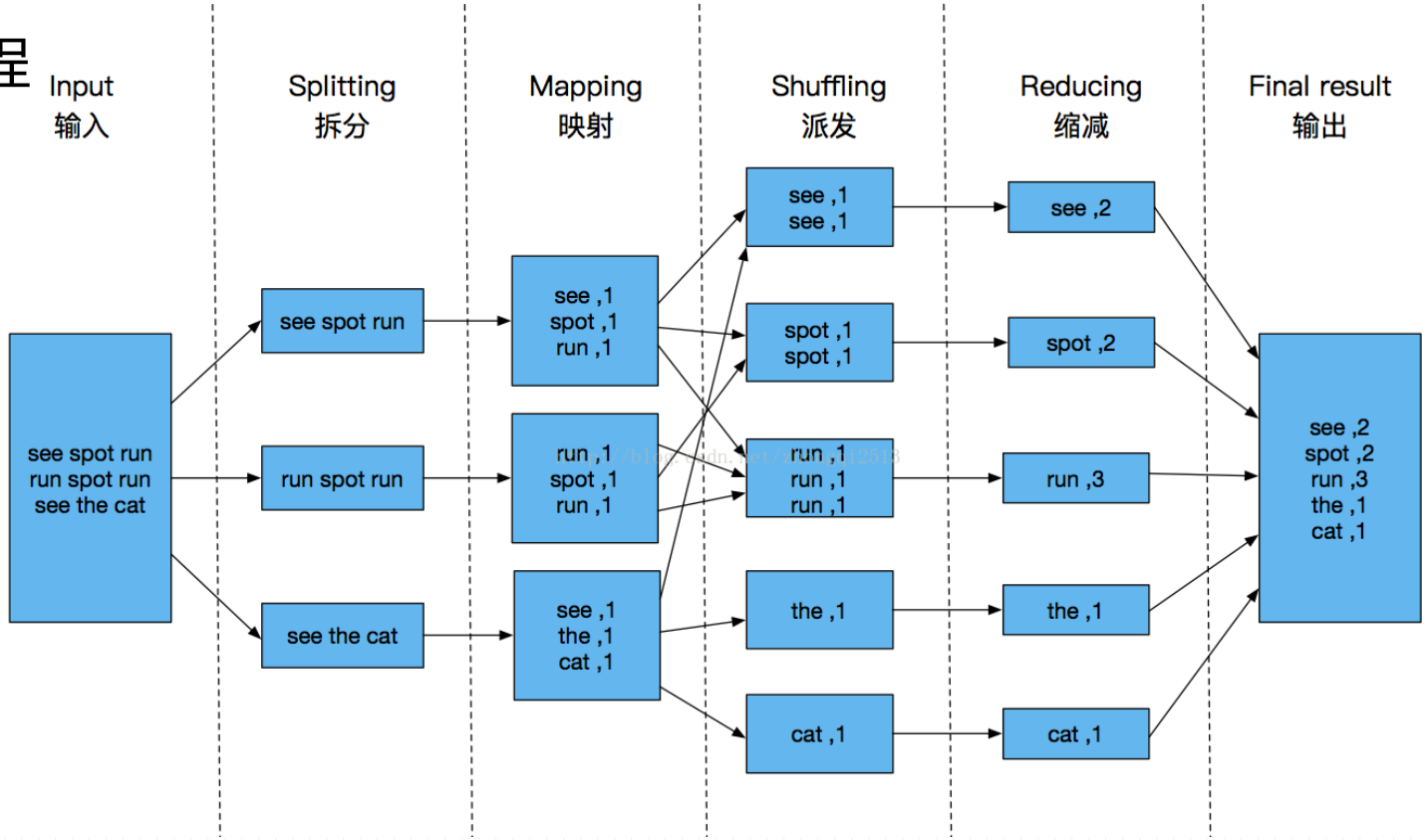
算法流程

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

4.4 实例1 -- WordCount

算法流程



实例详解：WordCount

- 如果想统计过去10年计算机论文中出现次数最多的几个单词，看看大家都在研究些什么，可以大致采用以下几种方法：
- 1)　写一个小程序，把所有论文按顺序遍历一遍，统计每一个遇到的单词的出现次数，最后就可以知道哪几个单词最热门了（适合于数据集比较小，且非常有效的、实现最简单）。
- 2)　写一个多线程程序，并发遍历论文。理论上是可以高度并发的，因为统计一个文件时不会影响统计另一个文件。使用多核机器时比方法一高效。但是，写一个多线程程序要复杂得多。
- 3)　把作业交给多个计算机去完成。可以使用方法一的程序，部署到N台机器上去，然后把论文集分成N份，一台机器跑一个作业。这个方法跑得足够快，但是部署起来很麻烦，既要人工把论文集分开，复制到各台机器，还把N个运行结果进行整合。
- 4)　使用MapReduce。MapReduce本质上就是方法三，但如何拆分文件集，如何复制分发程序，如何整合结果这些都是框架定义好的。我们只要定义好这个任务（用户程序），其它都交给MapReduce。

WordCount程序任务

重庆大学 CHONGQING UNIVERSITY

WordCount程序任务

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

一个WordCount的输入和输出实例

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

WordCount设计思路

- 每个拿到原始数据的机器只要将输入数据切分成单词就可以了，所以可以在map阶段完成单词切分的任务。另外，相同单词的频数计算也可以并行化处理，可以将相同的单词交给一台机器来计算频数，然后输出最终结果，这个过程可以交给reduce阶段完成。至于将中间结果根据不同单词分组再发送给reduce机器，这正好是MapReduce过程中的shuffle能够完成的。

- 因此，WordCount的整个过程可表示为：
- 1. Map阶段完成由输入数据到单词切分的工作。
- 2. Shuffle阶段完成相同单词的聚集和分发工作（这个过程是MapReduce的默认过程，不用具体配置）。
- 3. Reduce阶段完成接收所有单词并计算其频数的工作。

WordCount设计思路

首先，需要检查WordCount程序任务是否可以采用MapReduce来实现。

其次，确定MapReduce程序的设计思路。

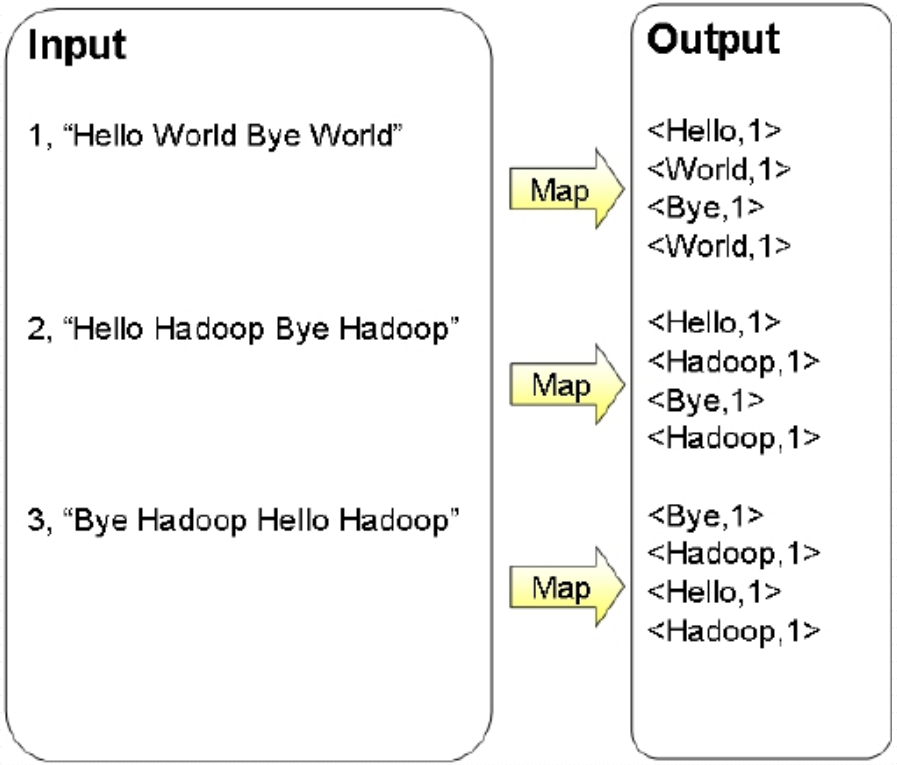
最后，确定MapReduce程序的执行过程。

- MapReduce中传递的数据都是<key, value>形式的，并且shuffle排序聚集分发是按照key值进行的，所以，将map的输出设计成由word作为key，1作为value的形式，它表示单词出现了1次（map的输入采用Hadoop默认的输入方式，即文件的一行作为value，行号作为key）。
- Reduce的输入是map输出聚集后的结果，即<key, value-list>，具体到这个实例就是<word, {1,1,1,1,...}>，reduce的输出会设计成与map输出相同的形式，只是后面的数值不再是固定的1，而是具体算出的word所对应的频数。

WordCount过程解释

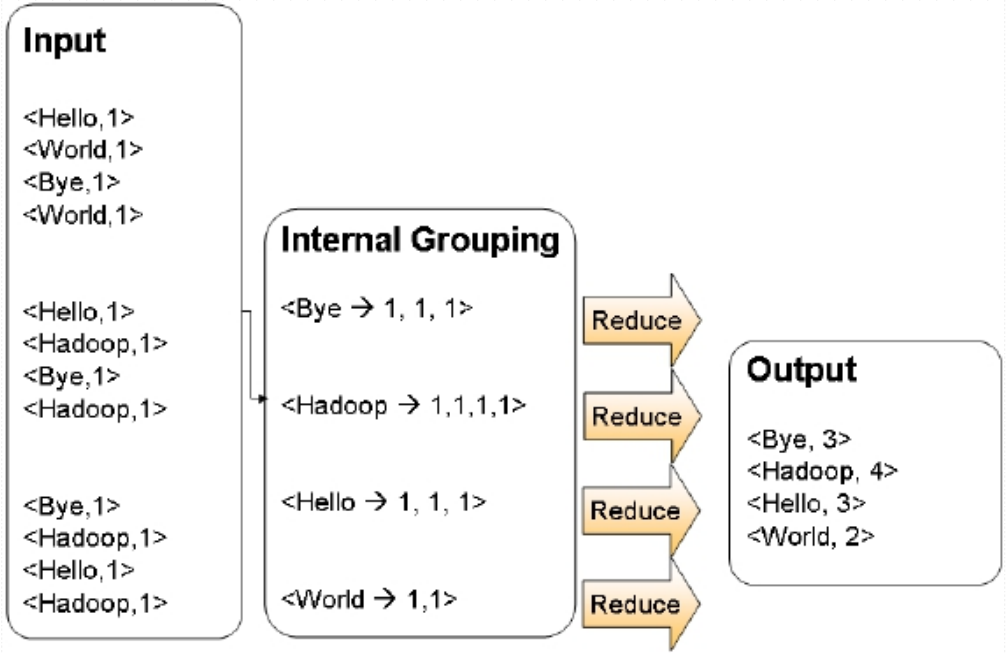
- Map操作的输入是<key, value>形式，其中，key是文档中某行的行号，value是该行的内容。

  Map操作会将输入文档中每一个单词的出现输出到中间文件中去。



WordCount过程解释

- Reduce操作的输入是单词和出现次数的序列，如<“Hello”，[1,1,1]>，<“World”，[1,1]>，<“Bye”，[1,1,1]>，<“Hadoop”，[1,1,1,1]>等。然后根据每个单词，算出总的出现次数。
- 最后输出排序后的最终结果就会是：<“Bye”，3>，<“Hadoop”，4>，<“Hello”，3>，<“World”，2>。



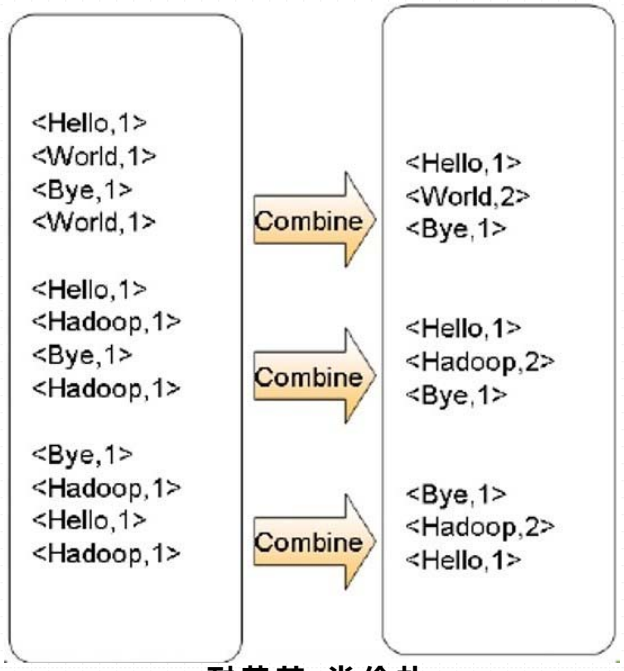
WordCount过程解释

整个MapReduce过程实际的执行顺序是：

- 1.MapReduce Library将Input分成M份。
- 2.Master将M份Job分给空闲状态的M个worker来处理。
- 3.对于输入中的每一个<key， value> 进行Map操作，将中间结果缓冲在内存里。
- 4.定期地（或者根据内存状态）将缓冲区中的中间信息刷写到本地磁盘上，并且把文件信息传回给Master（Master需要把这些信息发送给Reduce worker）。
- 5.R个Reduce worker开始工作，从不同的Map worker的Partition那里拿到数据，用key进行排序。
- 6.Reduce worker遍历中间数据，对每一个唯一Key，执行Reduce函数（参数是这个key以及相对应的一系列Value）。
- 7.执行完毕后，唤醒用户程序，返回结果（最后应该有R份Output，每个Reduce Worker一个）。

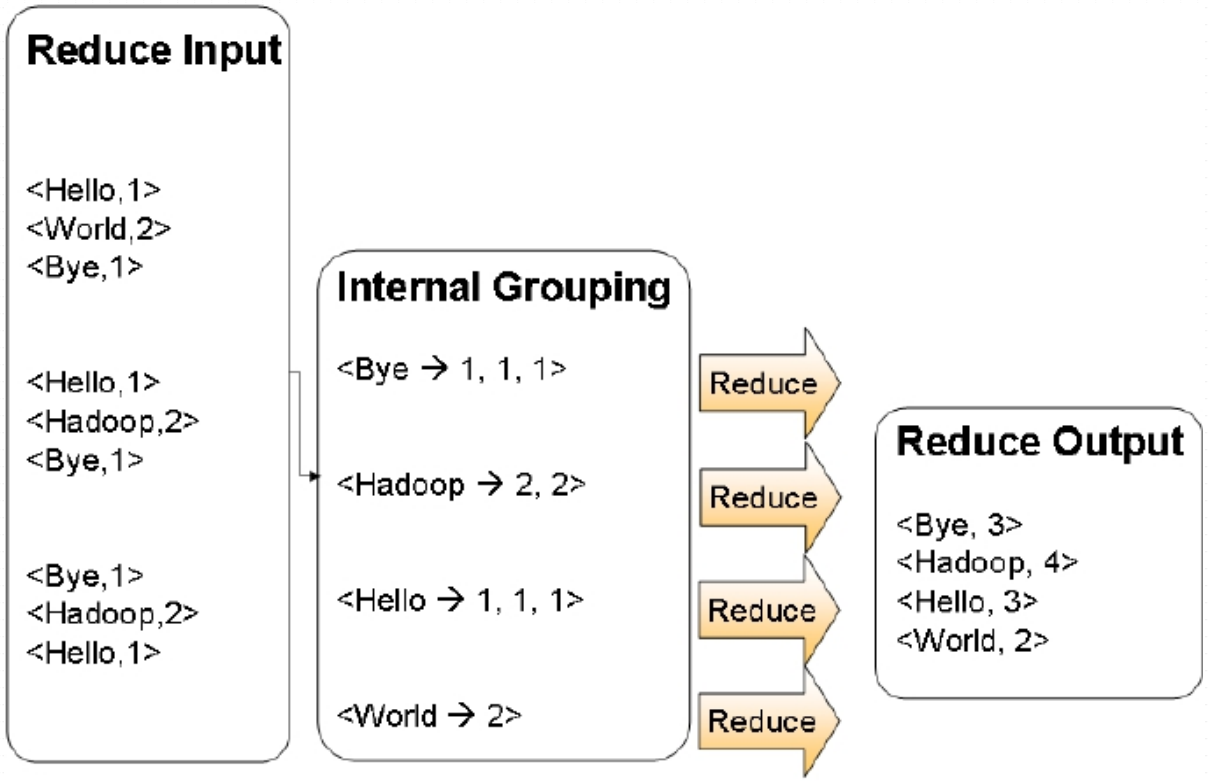
WordCount过程解释

- 对于上面的例子来说，每个文档中都可能会出现成千上万的 (“the”，1)这样的中间结果，琐碎的中间文件必然导致传输上的损失。
- 因此，MapReduce还支持用户提供Combiner函数。这个函数通常与Reduce函数有相同的实现，不同点在于Reduce函数的输出是最终结果，而Combiner函数的输出是Reduce函数某一个输入的中间文件。

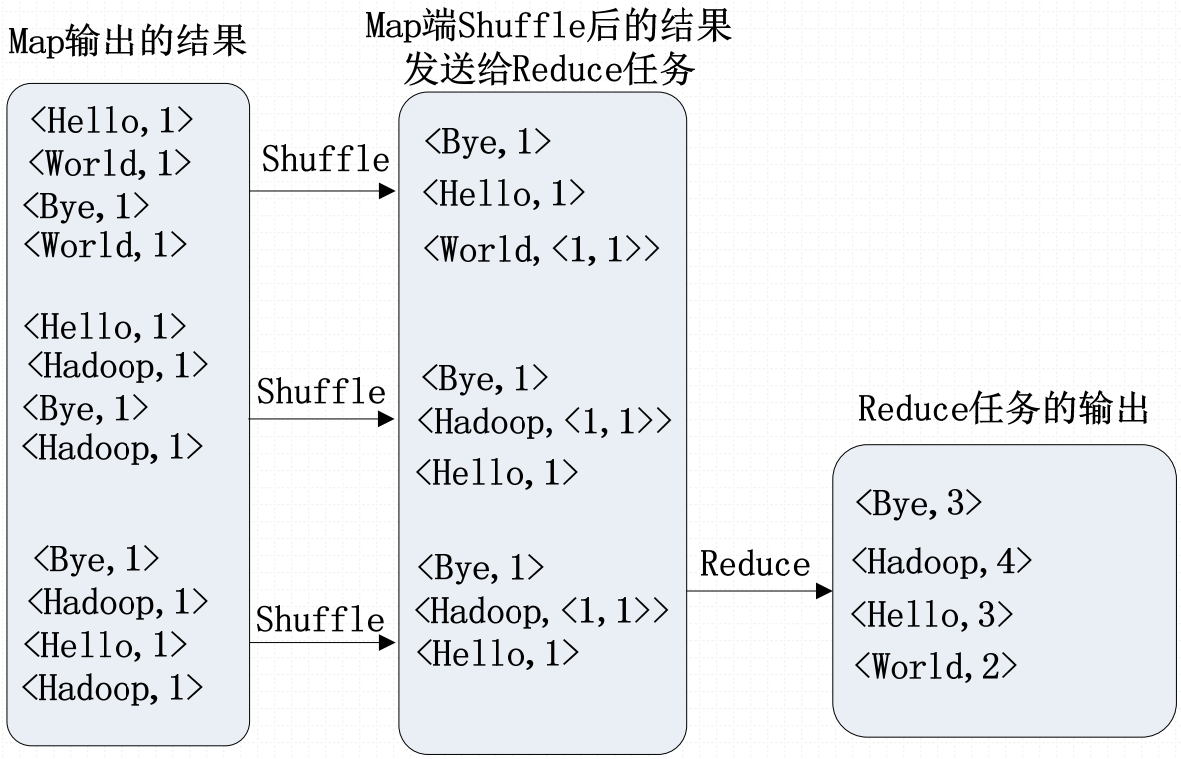


WordCount过程解释

- 下图是以Combine输出结果作为输入的Reduce过程示意图

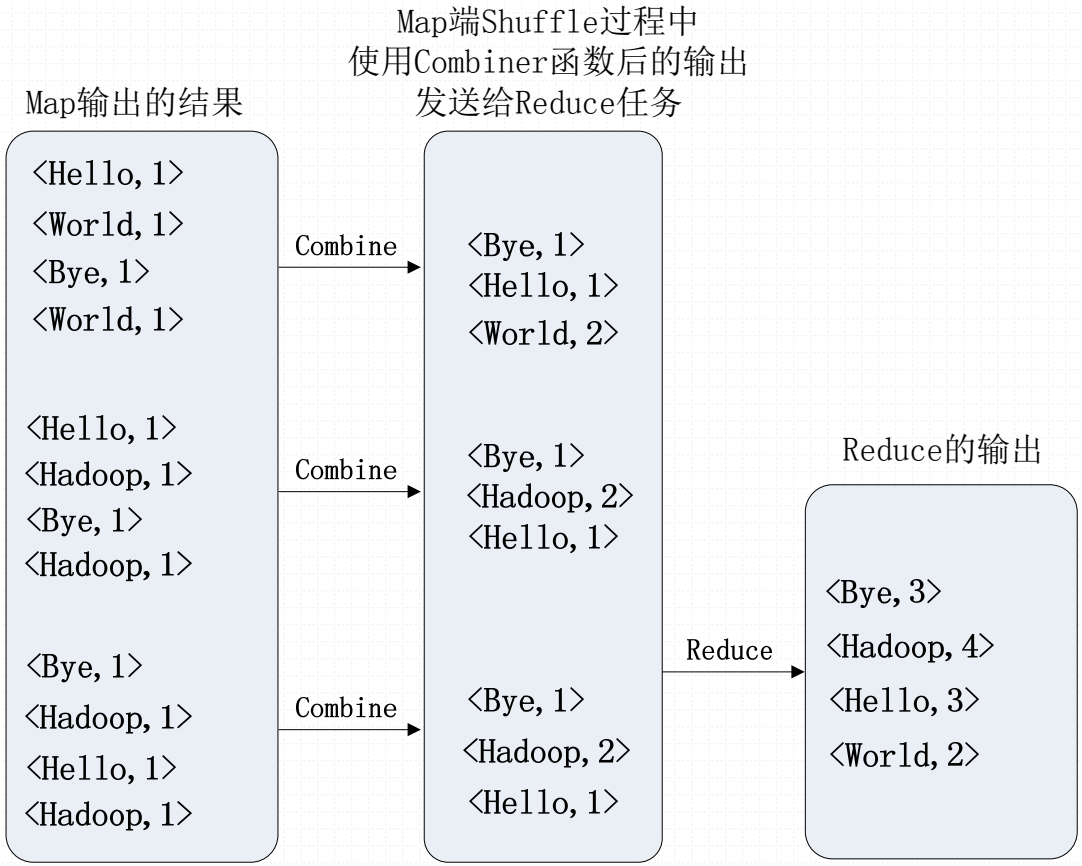


WordCount过程解释



用户没有定义Combiner时的Reduce过程示意图

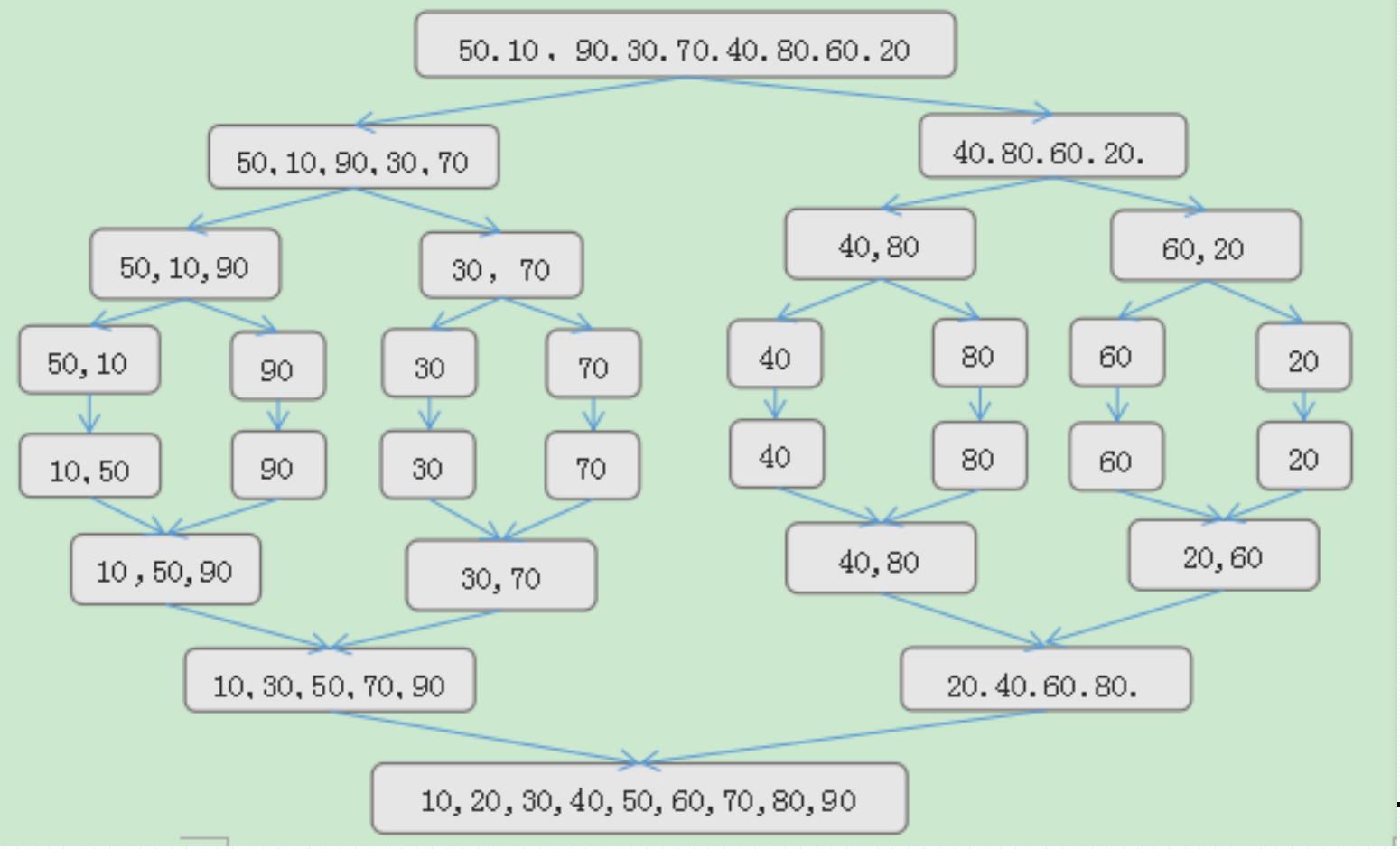
WordCount过程解释



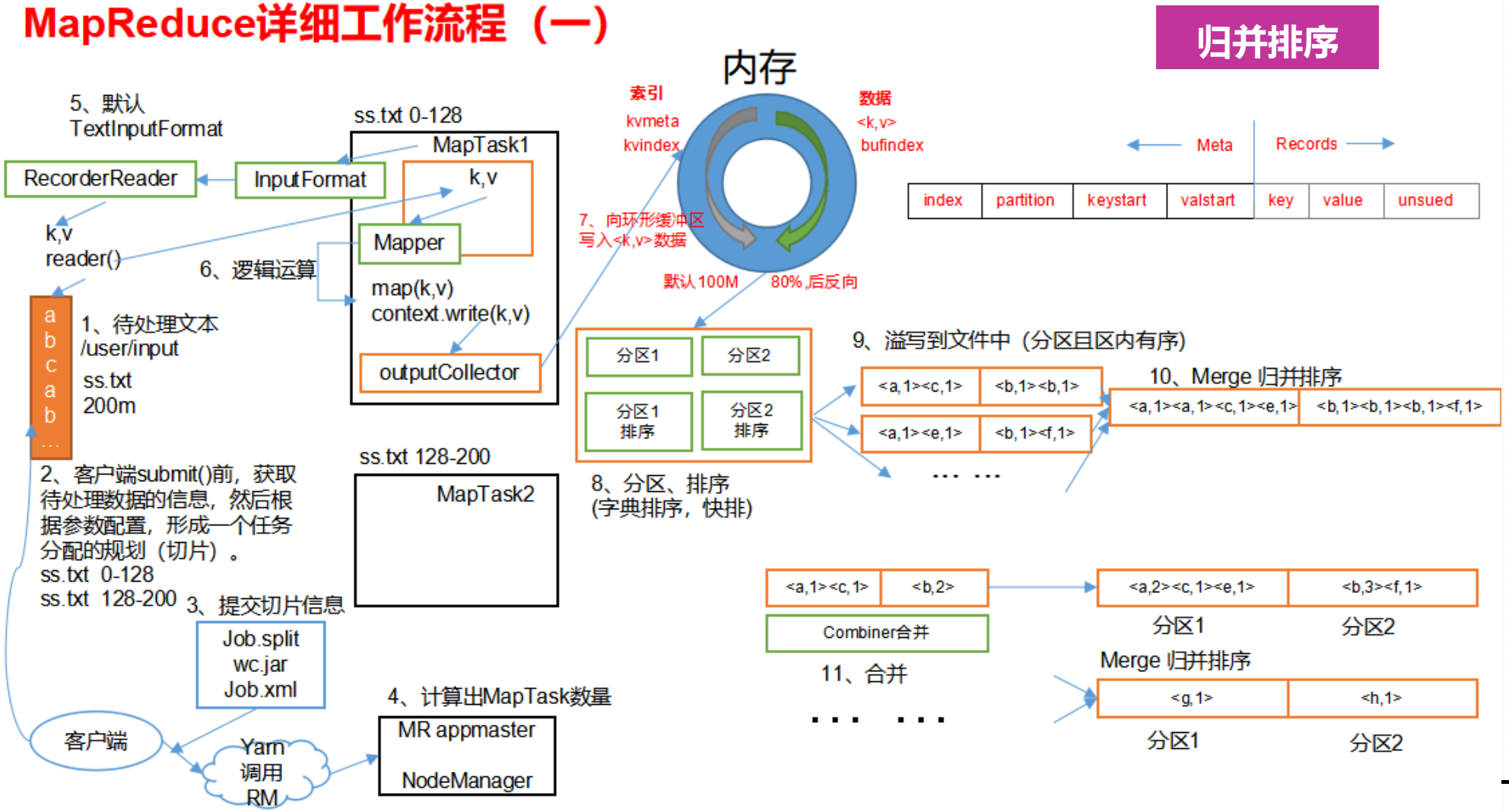
用户有定义Combiner时的Reduce过程示意图

2.4 实例2 -- 归并排序

归并排序基本原理

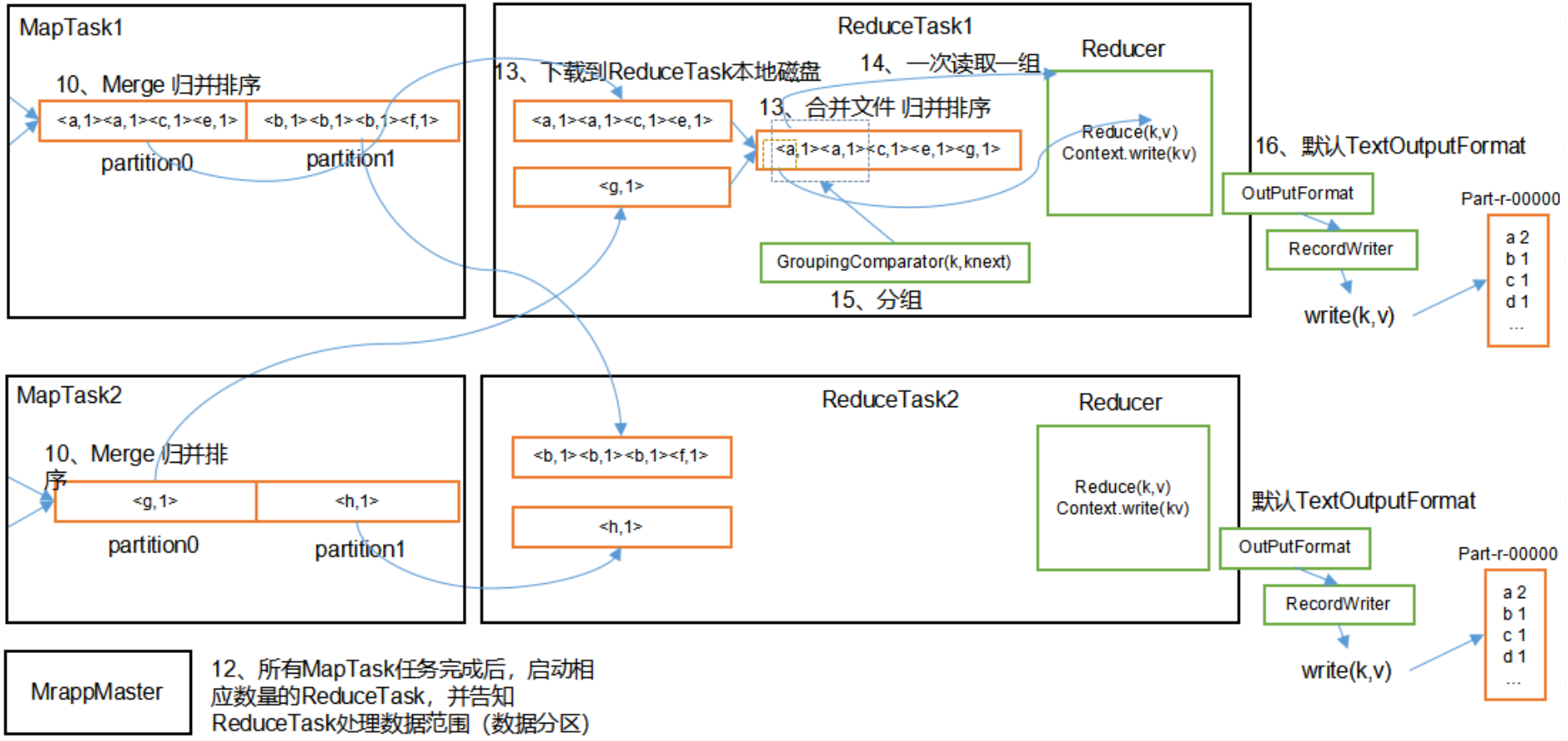


MapReduce详细工作流程（一）



MapReduce详细工作流程（二）

归并排序



4.4 MapReduce 具体应用

- MapReduce应用于各种计算问题
  - 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

4.4.1 MapReduce 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

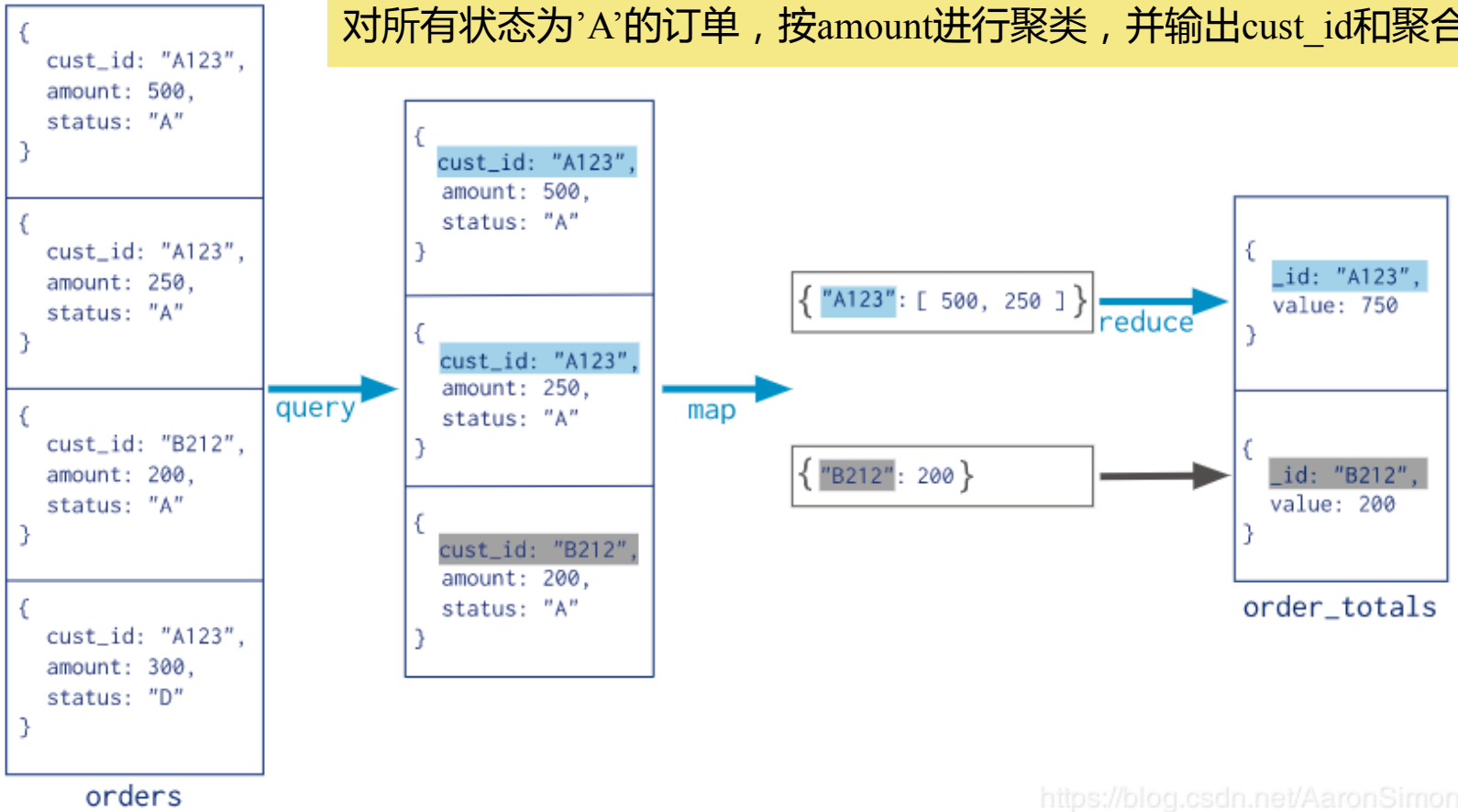
Key	Value
1	“Item”,(10,1)
1	“Item”,(20,3)
2	“Item”,(10,5)
2	“Item”,(50,100)
3	“Item”,(20,1)

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)

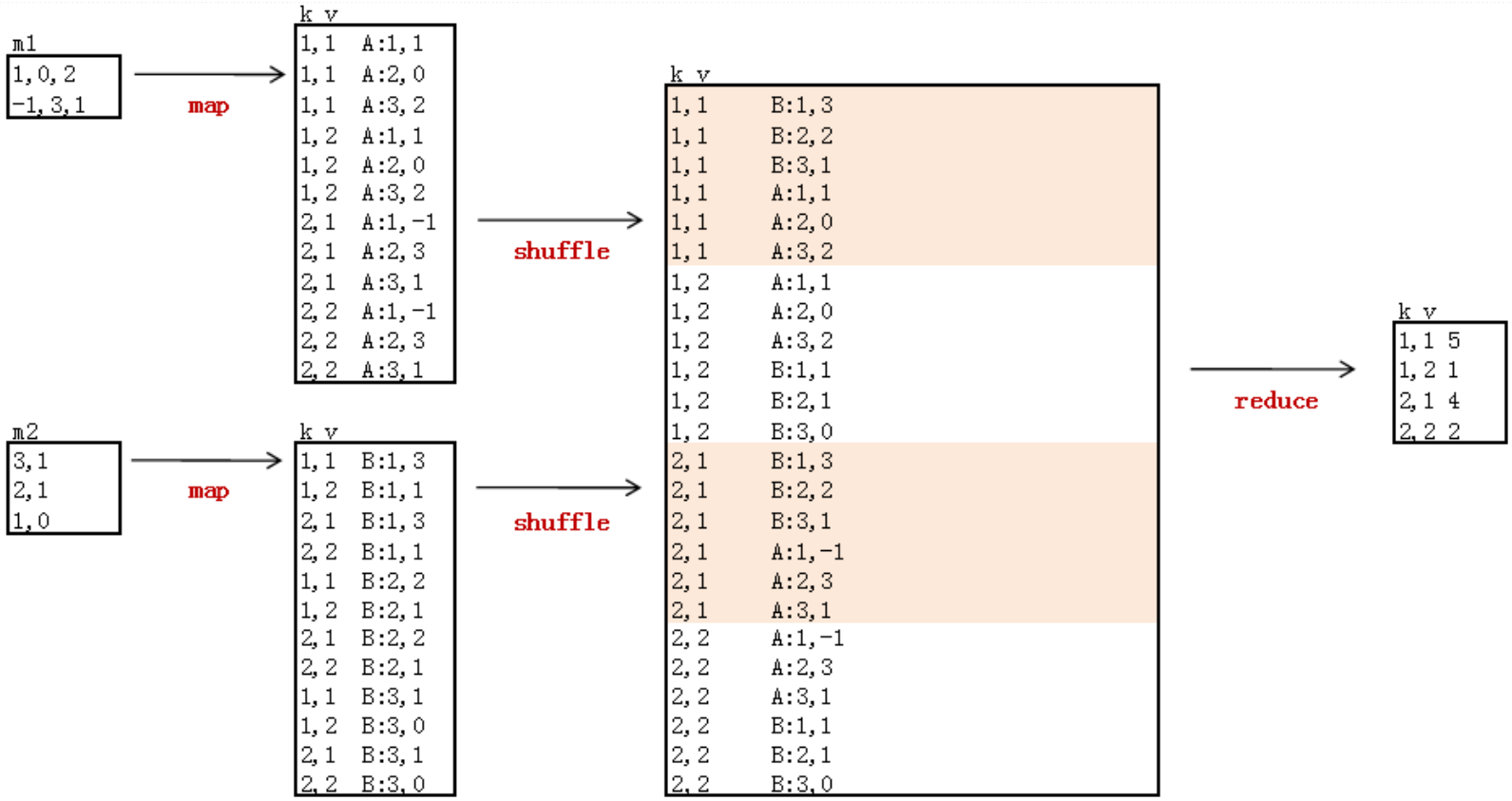
4.4.2 MapReduce 聚合操作

对所有状态为'A'的订单，按amount进行聚类，并输出cust_id和聚合之后的总量



4.4.3 MapReduce 矩阵乘法

思考：它的基本原理是什么？



4.4 MapReduce应用

思考与讨论：除了上述任务之外，

MapReduce还可用于哪些计算任务？

Thank You!