高性能计算与云计算

第十讲 大规模数据处理及Map/Reduce编程

胡金龙，董守斌

华南理工大学计算机学院
广东省计算机网络重点实验室
Communication & Computer Network Laboratory (CCNL)

内容概要

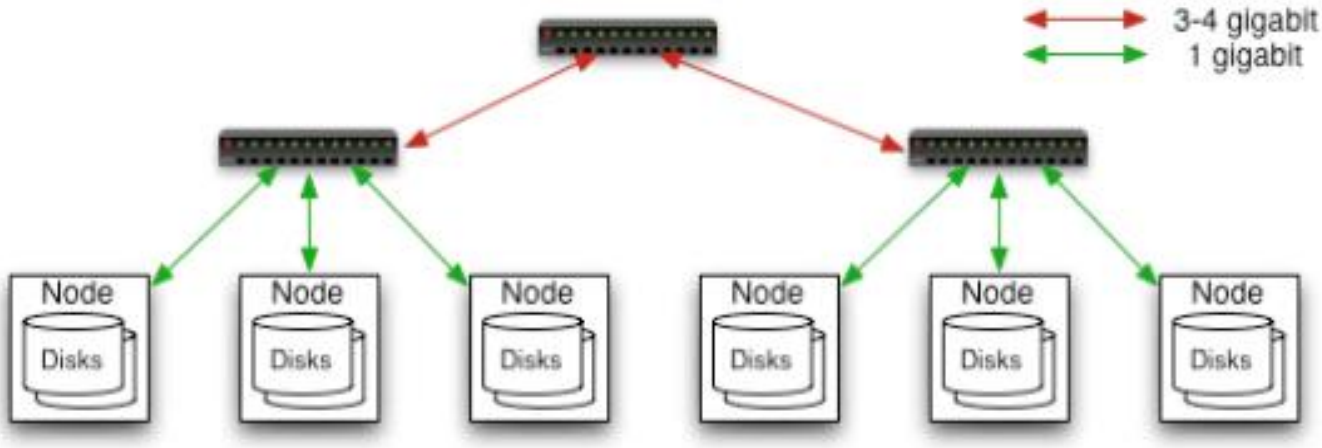
- 大规模数据处理的挑战
- **MapReduce编程模型**
- **MapReduce实现机制**

大规模数据处理

- 许多任务都需要处理大量的数据(> 1 TB)，需要能用到成百上千的CPUs
  - 例如：Google Earth：70.5 TB， 原始图像70 TB，索引数据 500 GB
  - ... 这并不容易

并行和分布式计算

- 高端的MPP及商用的PC集群



大规模数据处理面临的困难

- 大规模PC集群可靠性很差
  - 1节点的MTBF（Mean time between failures） = 3年
  - 1000个节点的MTBF = 1天
  - 商用网络 =低带宽

运行系统（Runtime System）：
- 良好可扩展性
- 良好的容错能力

- 并行/分布式程序开发、调试困难
  - 数据如何划分
  - 任务如何调度
  - 任务之间的通信
  - 错误处理，容错...

编程模型（Programming Model）：
- 一定的表达能力
- 很好的简单易用性

MapReduce：大规模数据处理

- **MapReduce是一个编程模型，提供：**
  - **自动的并行化和分布（Automatic parallelization & distribution）**
  - **容错 （Fault-tolerant）**
  - **I/O调度/状态和监控（Provides status and monitoring tools）**
  - **运行时系统处理输入数据的分割、执行调度、处理机器故障，以及管理所需的机器内部通信等细节（Clean abstraction for programmers）**
    - 这使得没有任何并行和分布式系统经验的程序员能够容易地利用一个大的分布式系统的资源。

Dean & Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004

MapReduce的推动力

- 多核时代：
  - 人们急于为即将来到的爆炸性提升的并行能力寻找易用的编程模型
- 大规模数据处理
  - Web检索和挖掘
  - 自然语言处理、机器学习等统计方法为主导的领域，数据规模不断增长，也特别关心
- Google公司最先提出了分布式并行编程模型MapReduce
- 开源的 Hadoop 计划使其流行
  - Yahoo!, Facebook, Amazon, …
- 云计算（Cloud Computing）的核心技术之一

MapReduce的应用例子

- **Google:**
  - **Google Search的索引构建（Index construction ）**
  - **Google News的文章聚类（clustering）**
  - **基于统计的机器翻译（machine translation）**
- **Facebook:**
  - **数据挖掘（Data mining）**
  - **广告优化（Ad optimization）**
  - **垃圾信息检测（Spam detection）**
- **Yahoo!:**
  - **Yahoo! Search的“Web map” 的构建**
  - **Yahoo! Mail的垃圾邮件检测**

内容概要

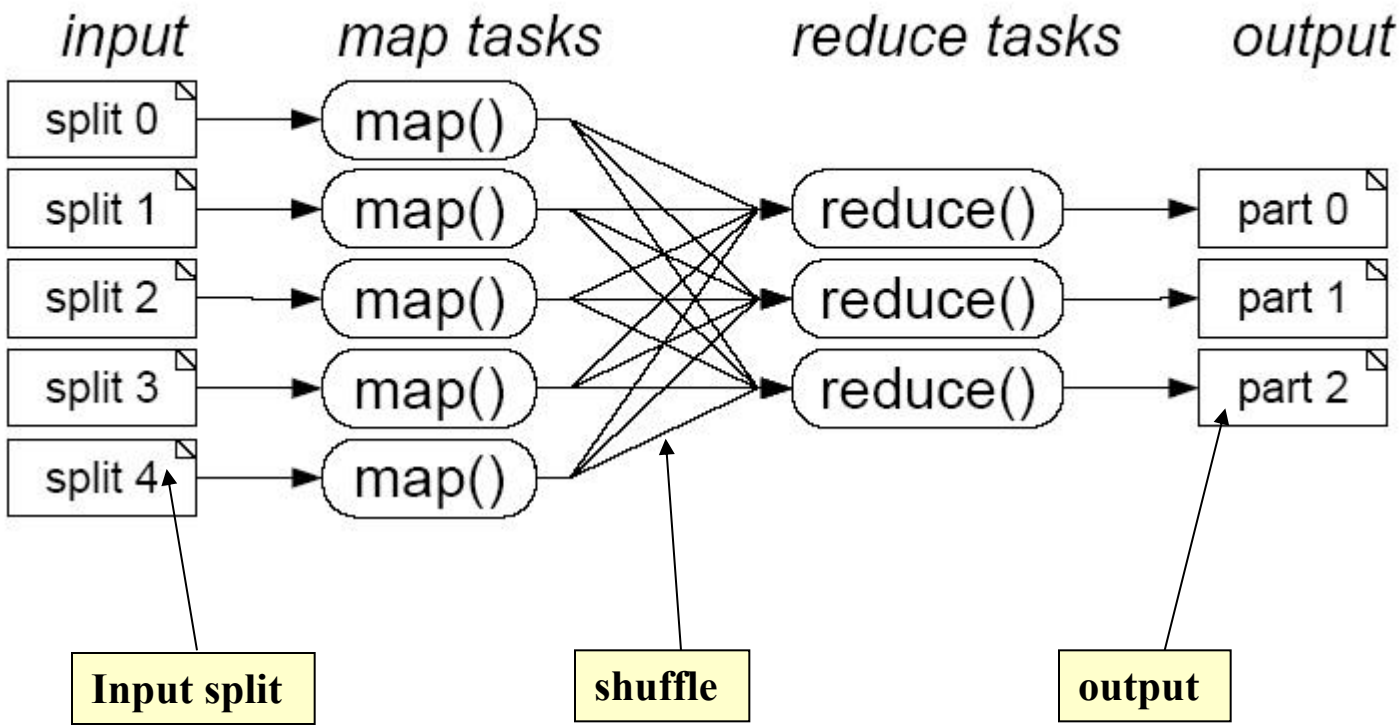
- 大规模数据处理的挑战
- **MapReduce编程模型**
- MapReduce实现机制
- Hadoop的MapReduce

MapReduce的设计理念

- 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
- 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设计理念是“计算向数据靠拢”，而不是“数据向计算靠拢”，因为数据的迁移需要大量的网络传输开销

MapReduce编程模型

- 并行/分布式计算编程模型



MapReduce的工作流程

- 读入数据：key/value 对的记录格式数据
- Map：从每个记录里提取关键信息
  - map (in_key, in_value) -> list(intermediate_key, intermediate_value)
    - 处理输入的 key/value对
    - 输出中间结果key/value对
- Shuffle：混排交换数据
  - 把相同key的中间结果汇集到相同节点上
- Reduce：聚合、摘要、过滤等
  - reduce (intermediate_key, list(intermediate_value)) -> list(out_value)
    - 归并某一个key的所有values，进行计算
    - 输出合并的计算结果
- 输出结果

编程模型

- **Map/Reduce借鉴函数式编程（functional programming）的思想，将复杂的运行于大规模集群上的并行计算过程高度的抽象到了两个函数，Map 和 Reduce, 这是一个令人惊讶的简单却又威力巨大的模型**

```
map    (Key_in, Value_in)-> list(Key_inter,Value_inter)

Reduce   (Key_inter,list(Value_inter))
                          ->list(Key_out, Value_out)
```

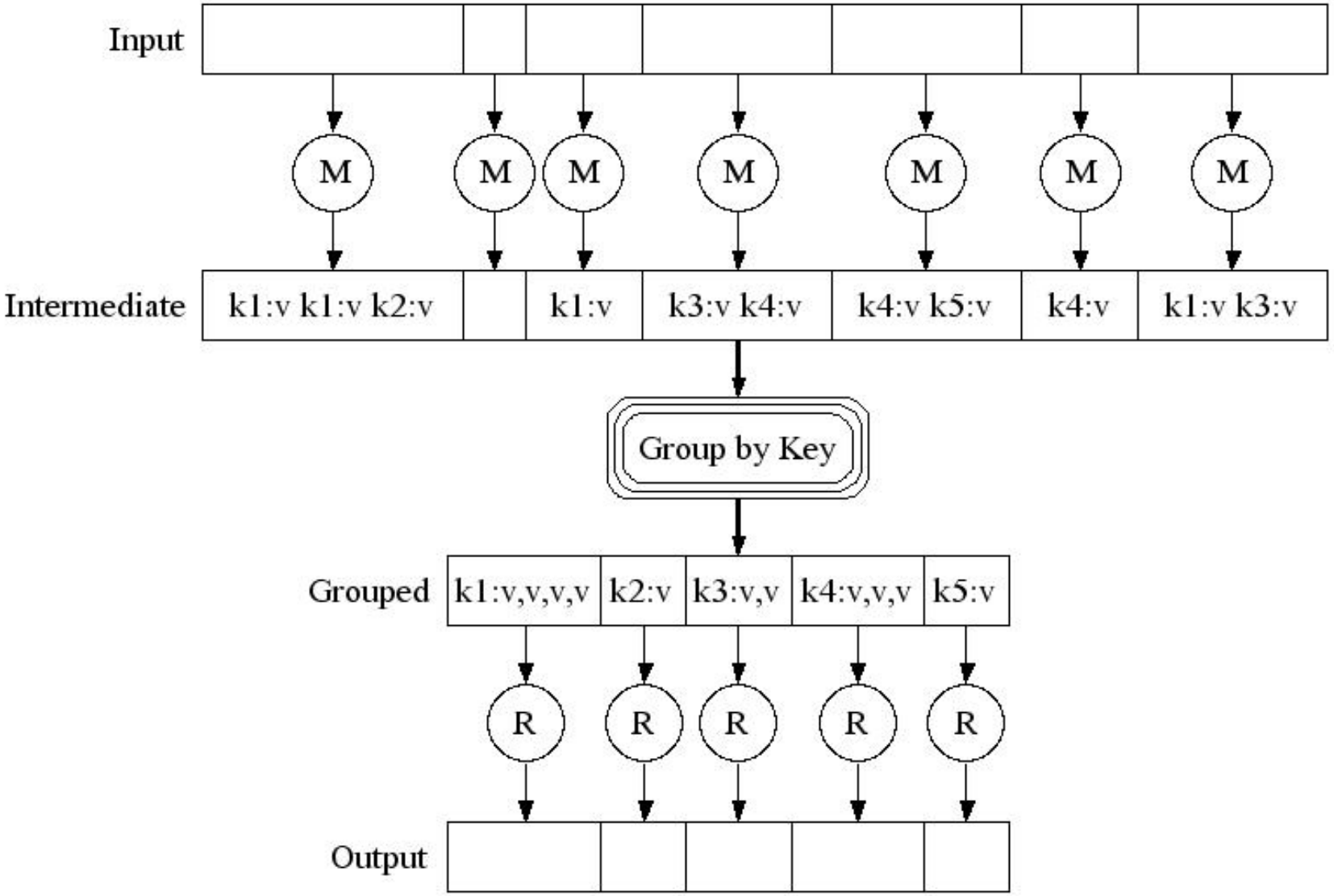
Map

- 数据源的记录（如：文件行、数据库的行等）以key*value 对（如：文件名，行）的形式送给map函数
- map()产生一个或多个中间结果值 （intermediate values ），每个中间结果值都带有一个输出key

Reduce

- 在map过程结束后，所有属于同一输出key的中间值并合并成一个列表
- reduce() 将具有相同输出key的中间值合并为一个或多个最终值（final values ）
  - 一般一个最终结果值对应一个key

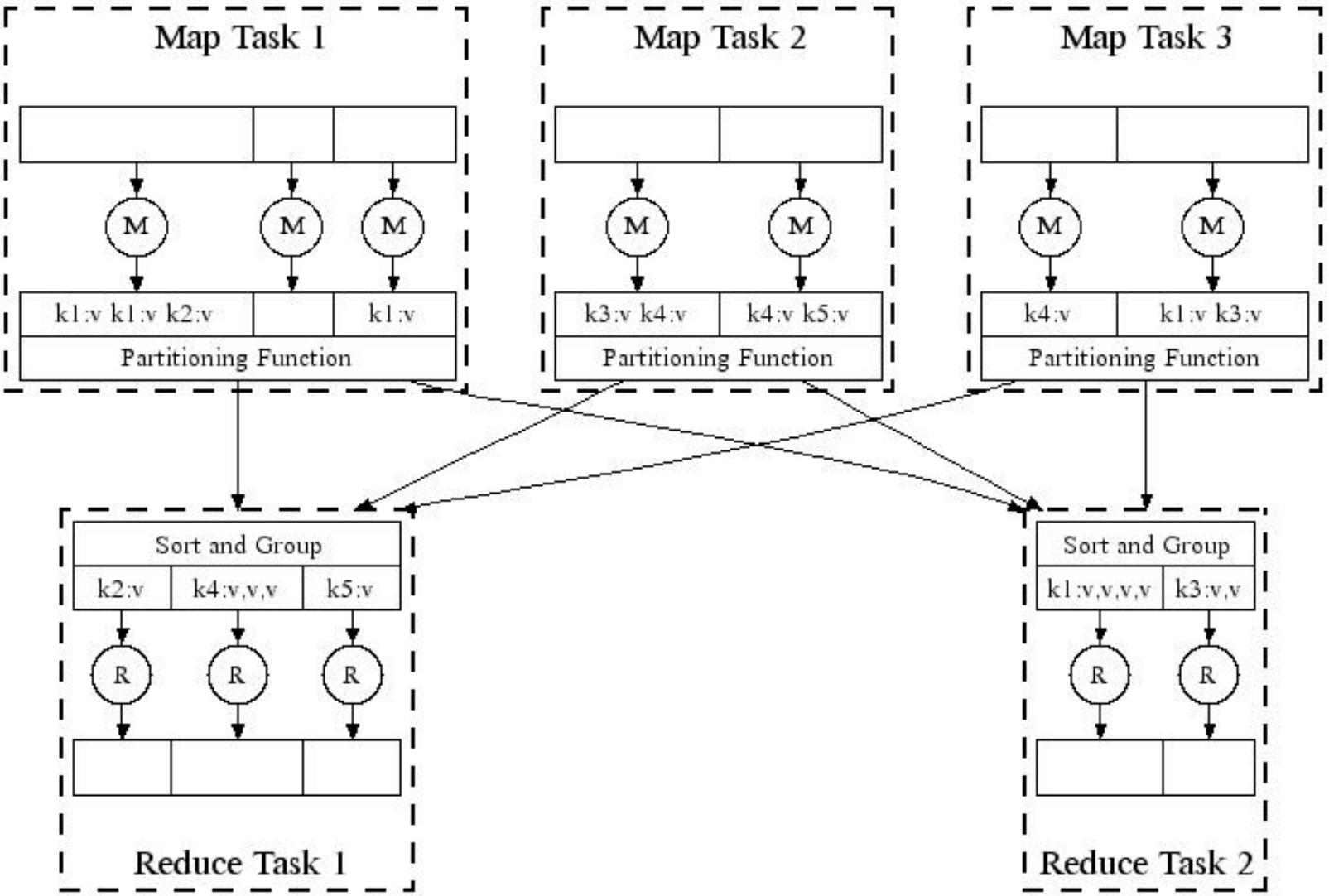
执行


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

并行化

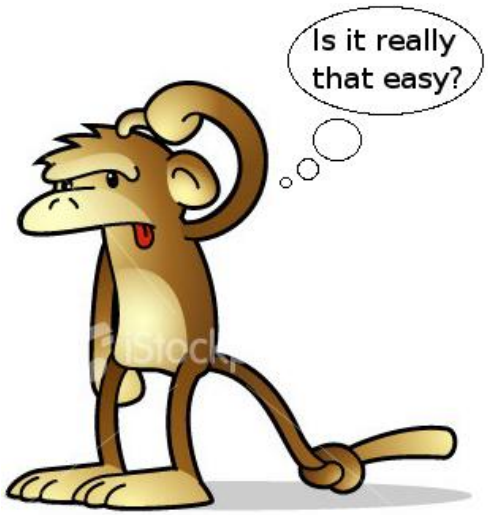
- **map()** 函数可以并行运行，从不同的数据数据集创建不同的中间值
- **reduce()** 函数也可以并行运行，在不同的输出key上工作
- 所有值都是被分别（independently）执行
- 瓶颈：只有在map阶段完全结束后才能启动reduce阶段

并行执行


Map Task 1
Map Task 2
Map Task 3
M
k1:v k1:v k2:v
k1:v
Partitioning Function
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Sort and Group
k2:v
k4:v,v,v
k5:v
k1:v,v,v,v
k3:v,v
R
Reduce Task 1
Reduce Task 2

如何用MapReduce？

- 用户需要做的：
  - 指定：
    - 输入/输出文件
    - M：做map的worker的个数
    - R：做reduce的worker的个数
    - W：机器数目
  - 写map 和 reduce 函数
  - 提交任务
- 无需任何并行/分布式系统的知识

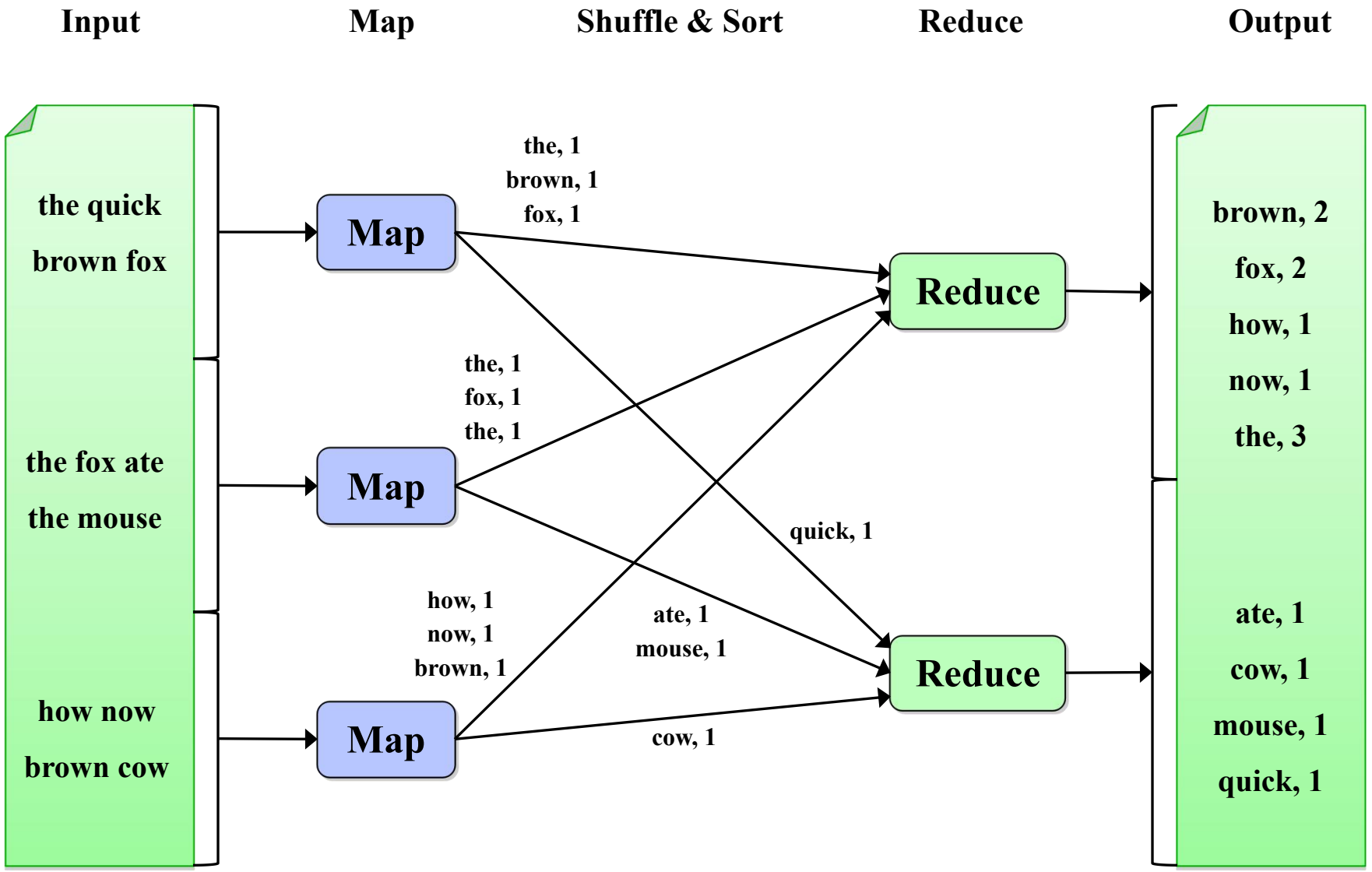


例子:统计一个文档集合中各个词出现的次数（Word Count）

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count的执行



Word Count的伪码表示

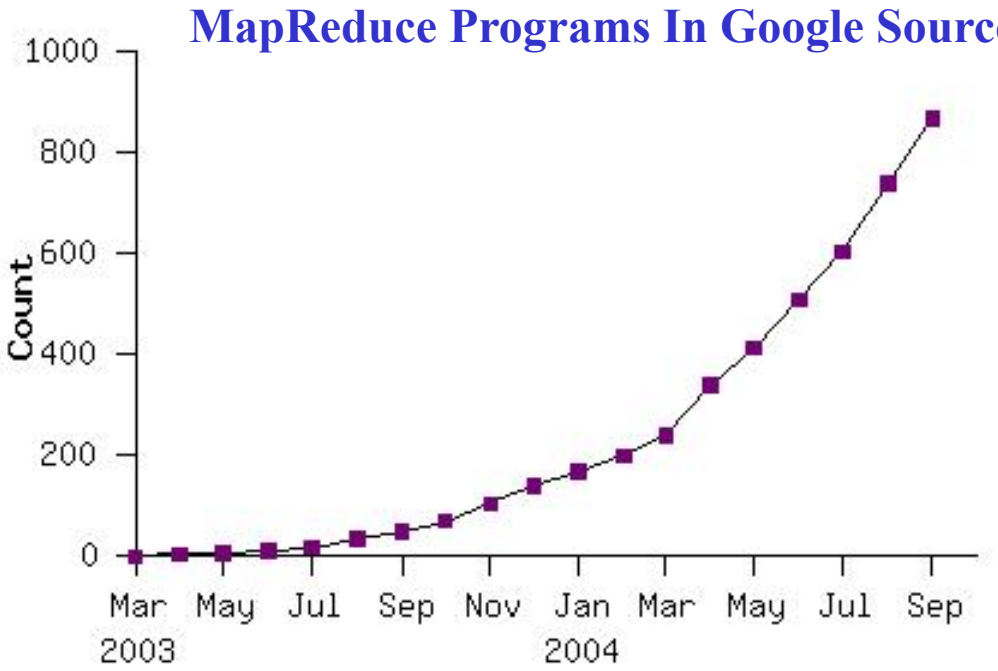
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");
```

```
reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

基于MapReduce已实现的算法

- 词的并现矩阵（Word Co-occurrence Matrices），成对文件相似度（Pairwise Document Similarity）
- PageRank算法
- K-Means， EM， SVM， PCA，线性回归（Linear Regression）, 朴素贝叶斯（Naïve Bayes），逻辑回归（Logistic Regression），神经网络（ Neural Network）
- 蒙特卡罗仿真（Monte Carlo simulation）
- ……

MapReduce计算模型被广泛应用



- 应用的领域

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

MapReduce的能力

MapReduce是否可能成为
解决大部分并行计算需求的主要手段？

- MapReduce难于有效实现的并行算法
  - 稠密/稀疏线性代数（Dense/Sparse Linear Algebra）
  - N体问题（N-Body Problems）
  - 动态规划（Dynamic Programming）
  - 图的遍历（Graph Traversal）
  - 组合逻辑（Combinational Logic）
  - …

"The landscape of parallel computing research: a view from Berkeley," 2006

MapReduce的能力

MapReduce是否可以用来进行在线实时数据处理？

- MapReduce作为一个编程模型，可以用来处理在线处理（online processing）问题
- 而目前多数实现是针对数据密集型（data-intensive）应用，实现中涉及到大量的Network I/O, File I/O，还有大量中间文件存取，这些因素使得它们不适合于响应时间要求高的应用

总结：Map/Reduce

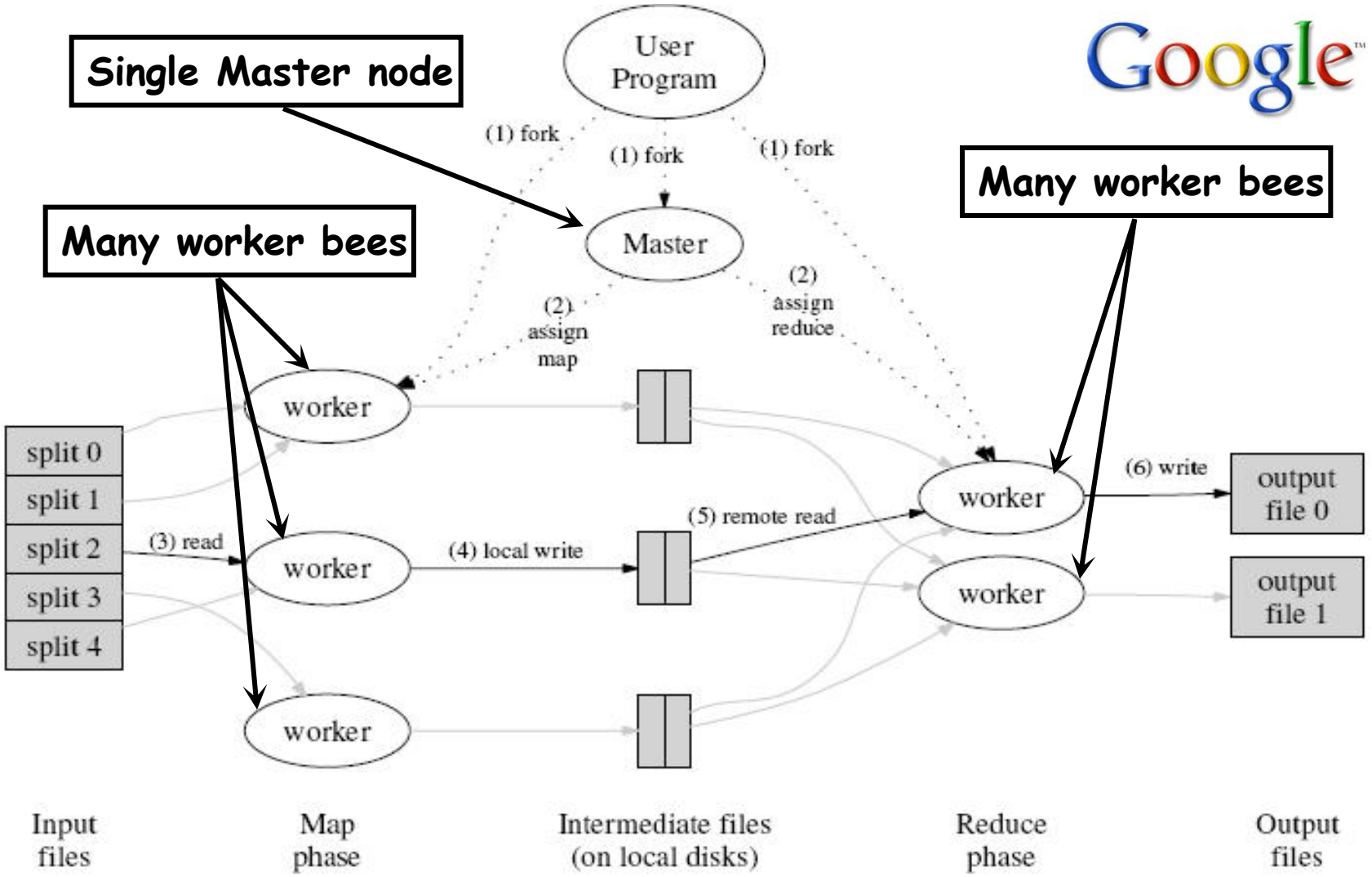
- 并行/分布式计算的编程模型
  - 面向领域（Domain specific）：基于函数编程方式进行大规模数据处理
  - 高可靠性（High reliability）：适用大规模分布式计算环境
  - 简单API：应用范围广泛

- MapReduce已被证明是有用的抽象（abstraction），极大地简化了大规模计算，使得程序员可将注意力放在问题本身，让底层去处理许多messy的细节
- MapReduce并不适用于所有问题，但如果适用，可节省很多时间

内容概要

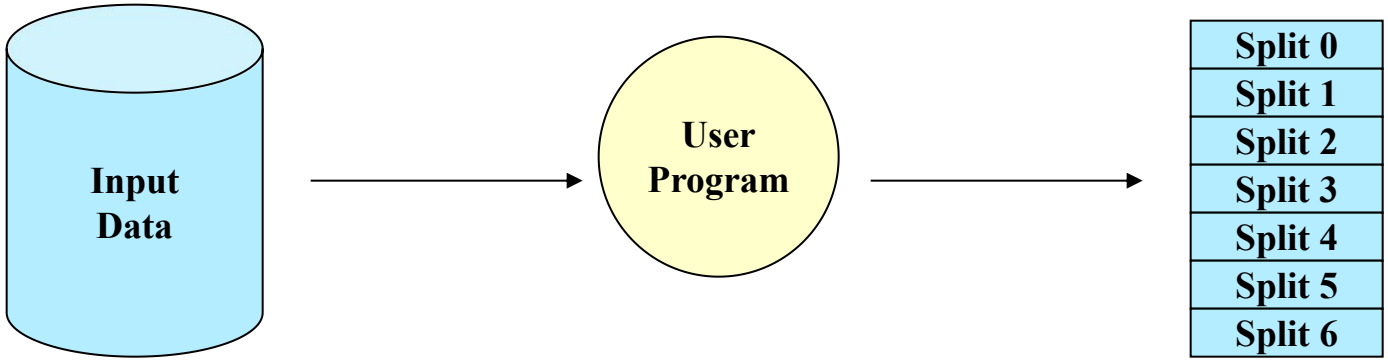
- 大规模数据处理的挑战
- MapReduce编程模型
- MapReduce实现机制

Google 的MapReduce实现架构



MapReduce的执行（1）

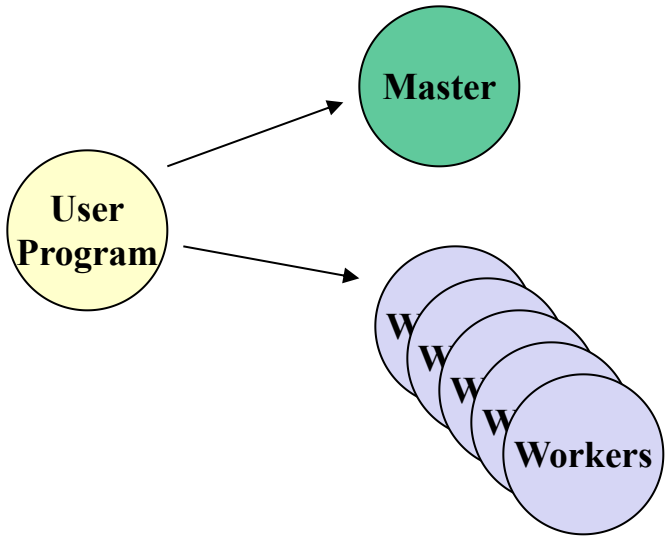
1. 用户程序，通过 MapReduce库，划分输入数据为分片（split）



* 分片大小通常为16-64MB

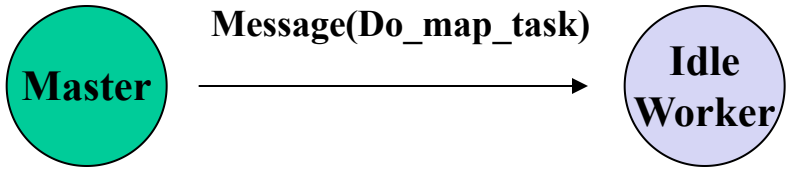
MapReduce的执行（2）

2. 用户在集群上创建进程拷贝。其中一个拷贝是 “Master”，其他是 worker线程



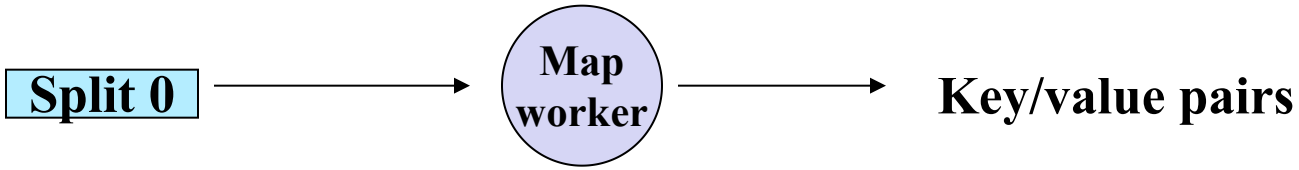
MapReduce资源

3. Master将 M个map任务（task） 和 R个reduce任务分配到空闲的 workers上
   - M == 分片数目
   - R == 中间结果的key空间被分为R个部分



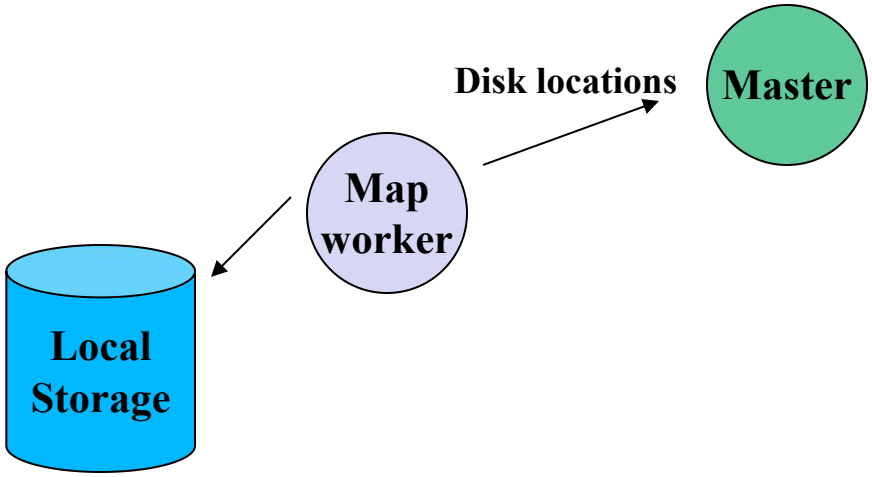
MapReduce资源

4. 每个map任务的worker读入所分配的输入分片，输出中间结果的key/value对
   - 输出放到RAM的缓冲区



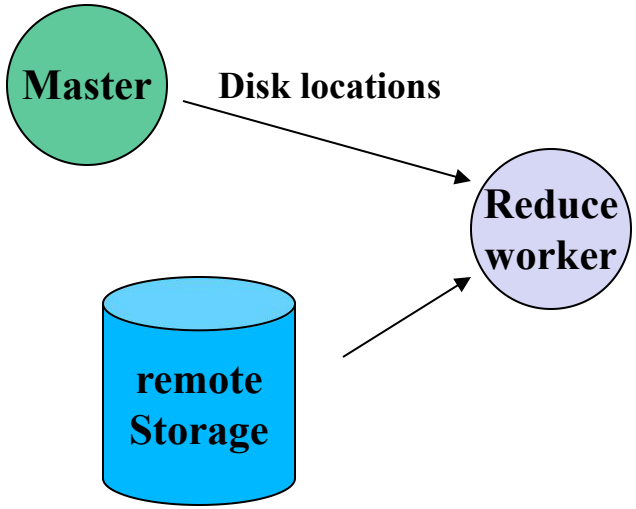
MapReduce 的执行

5. 每个worker 将中间值分为R个区域，存储到硬盘上，并通知 Master进程



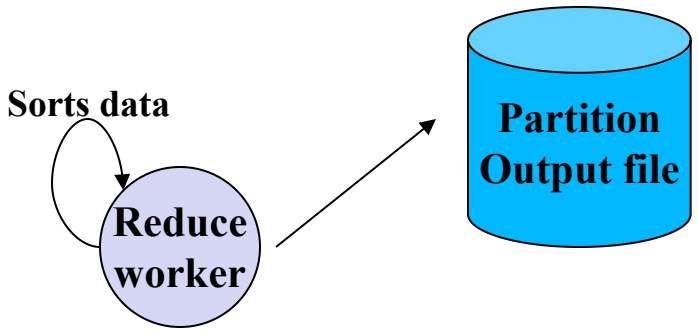
MapReduce的执行

6. Master进程告诉可用的reduce任务的worker数据所在的硬盘位置，使其可读入所有相关的中间结果数据



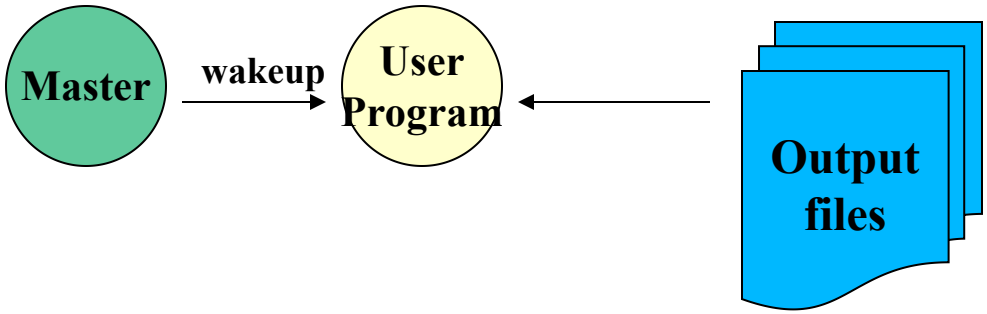
MapReduce的执行

7. 每个reduce任务的worker排序其获得的所有中间结果。调用 reduce函数，以中间结果的唯一性key以及相关键值为输入。Reduce函数的输出追加到reduce任务的分区输出文件（partition output file）

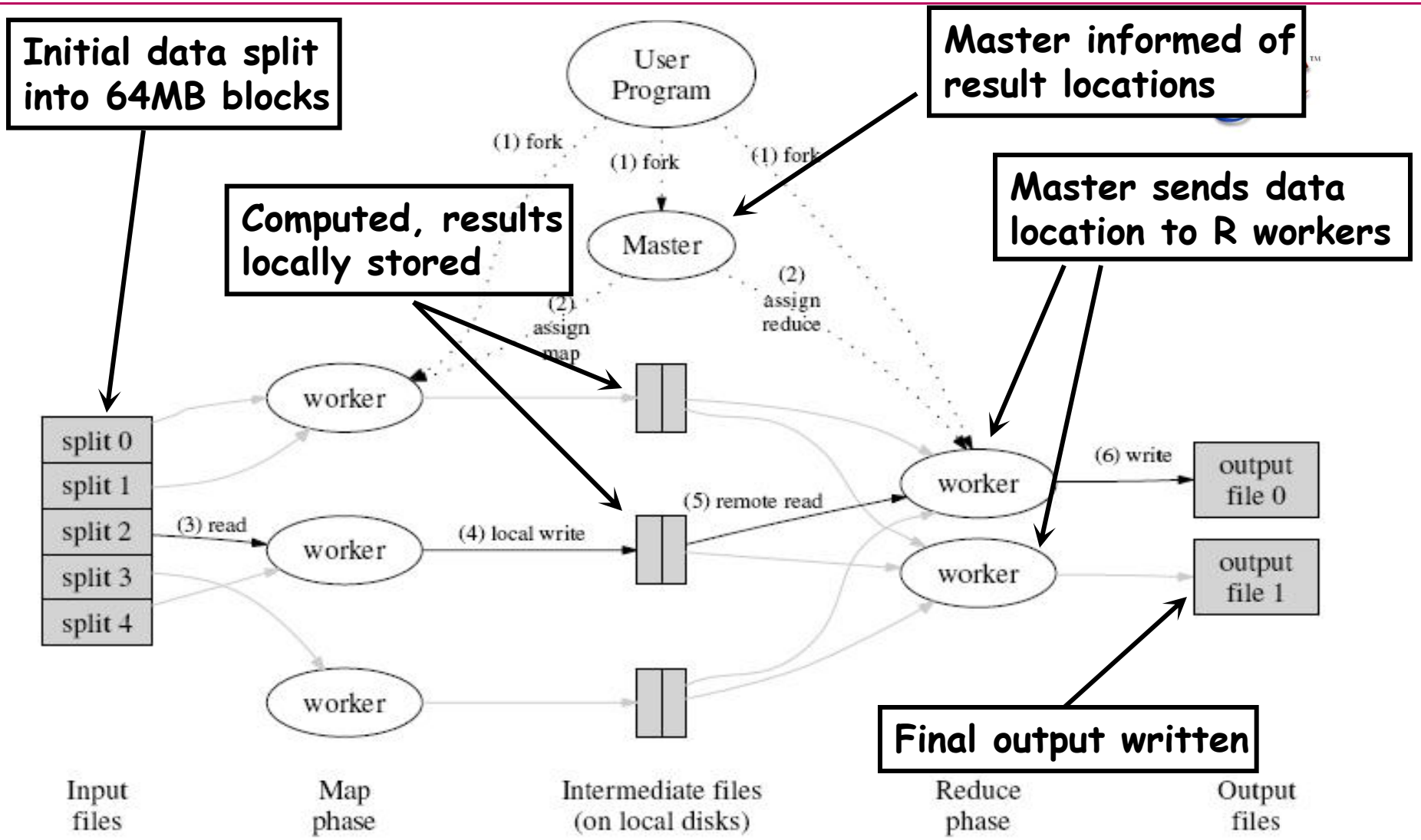


MapReduce的执行

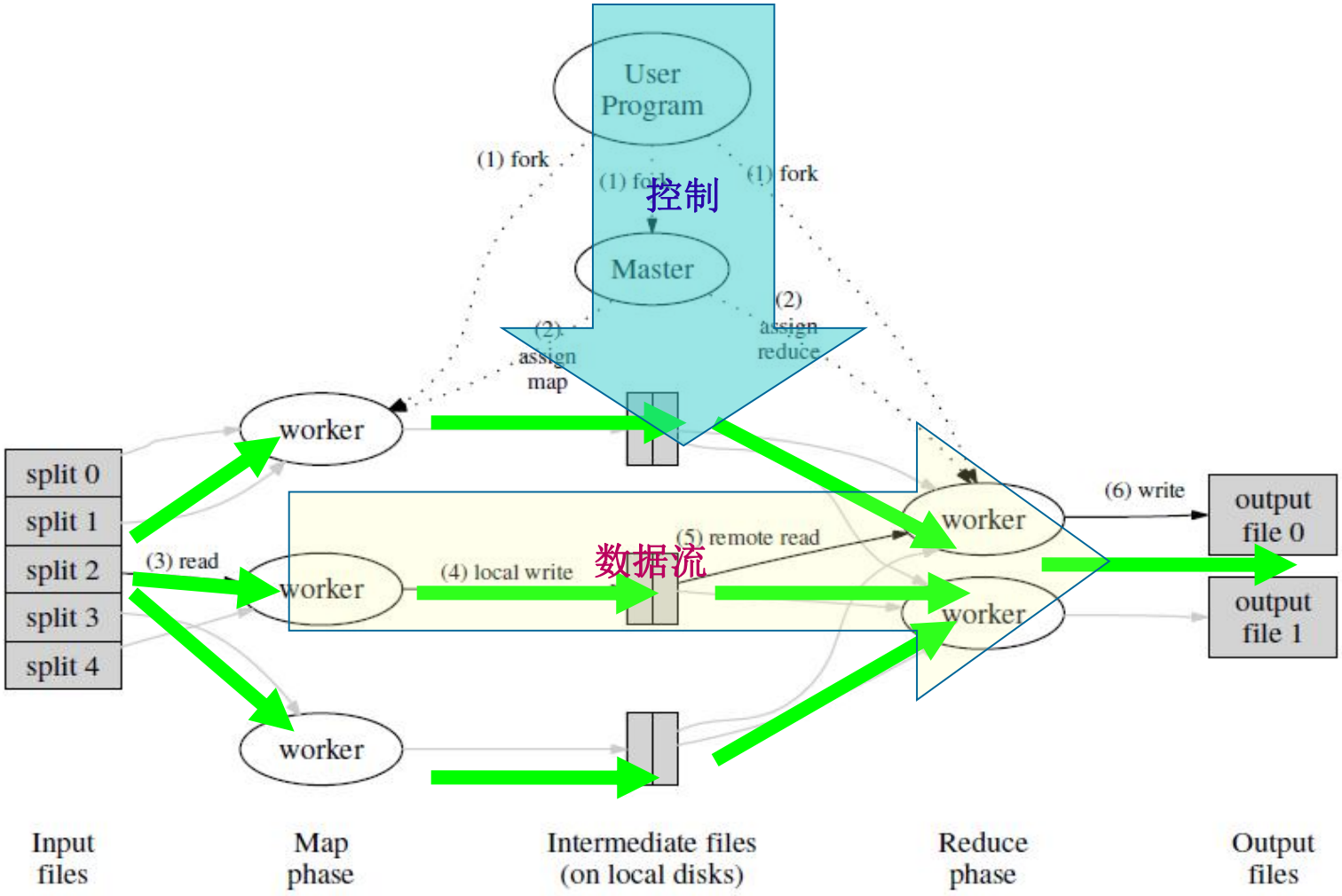
8. 当所有任务完成后，Master进程唤醒用户进程，结果存在于R个输出文件中



总结：MapReduce操作



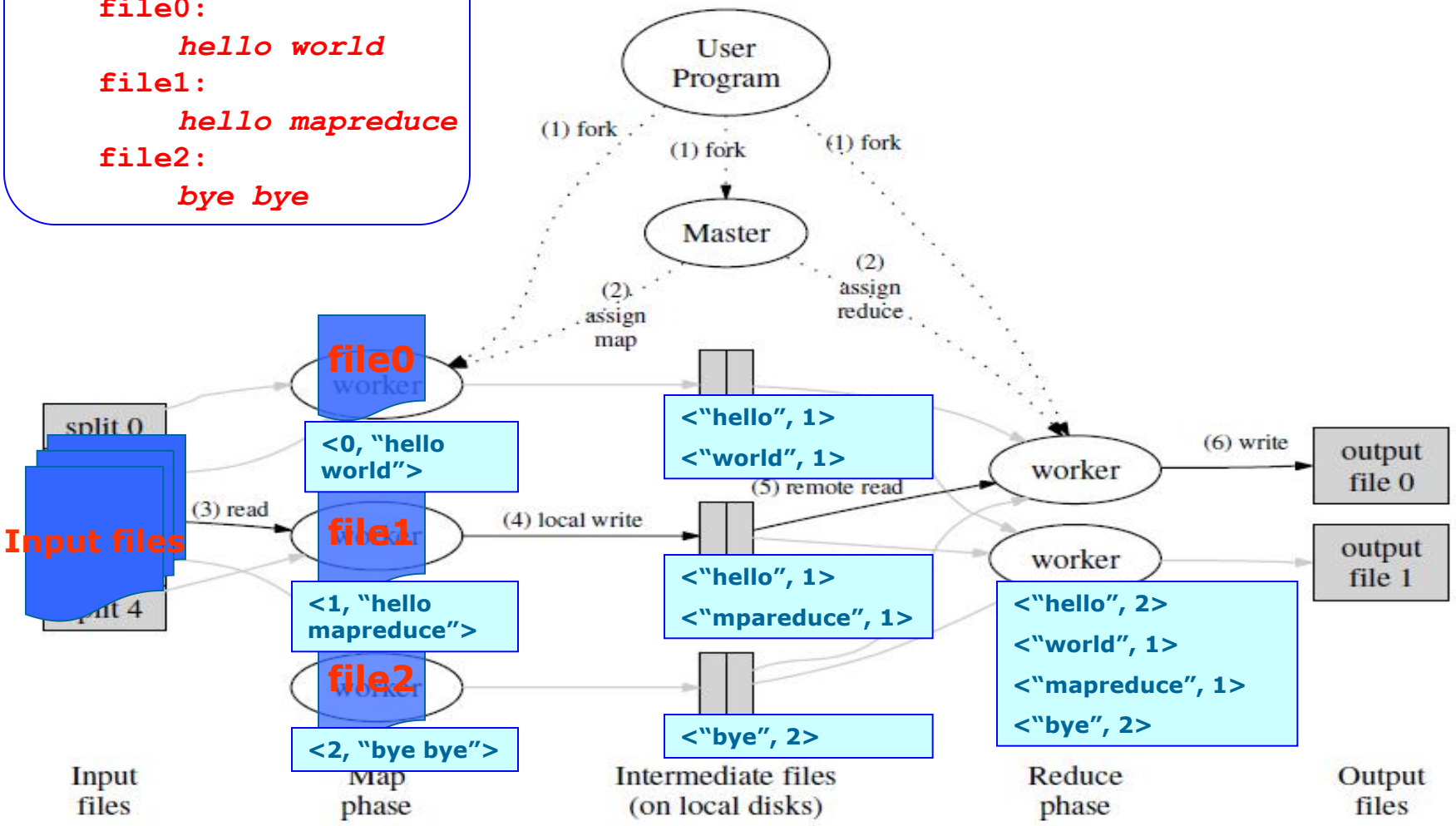
MapReduce的执行过程



MapReduce的执行过程

Word count:

```
file0:
    hello world
file1:
    hello mapreduce
file2:
    bye bye
```



关于MapReduce的观察

- 在map完成之前不能启动reduce
- 任务调度是基于数据存储位置而进行的
- 如果map的worker 在reduce完成之前失败，任务必须完全重新运行
- Master必须知道中间文件的存储位置
- 所有困难的工作由MapReduce库完成

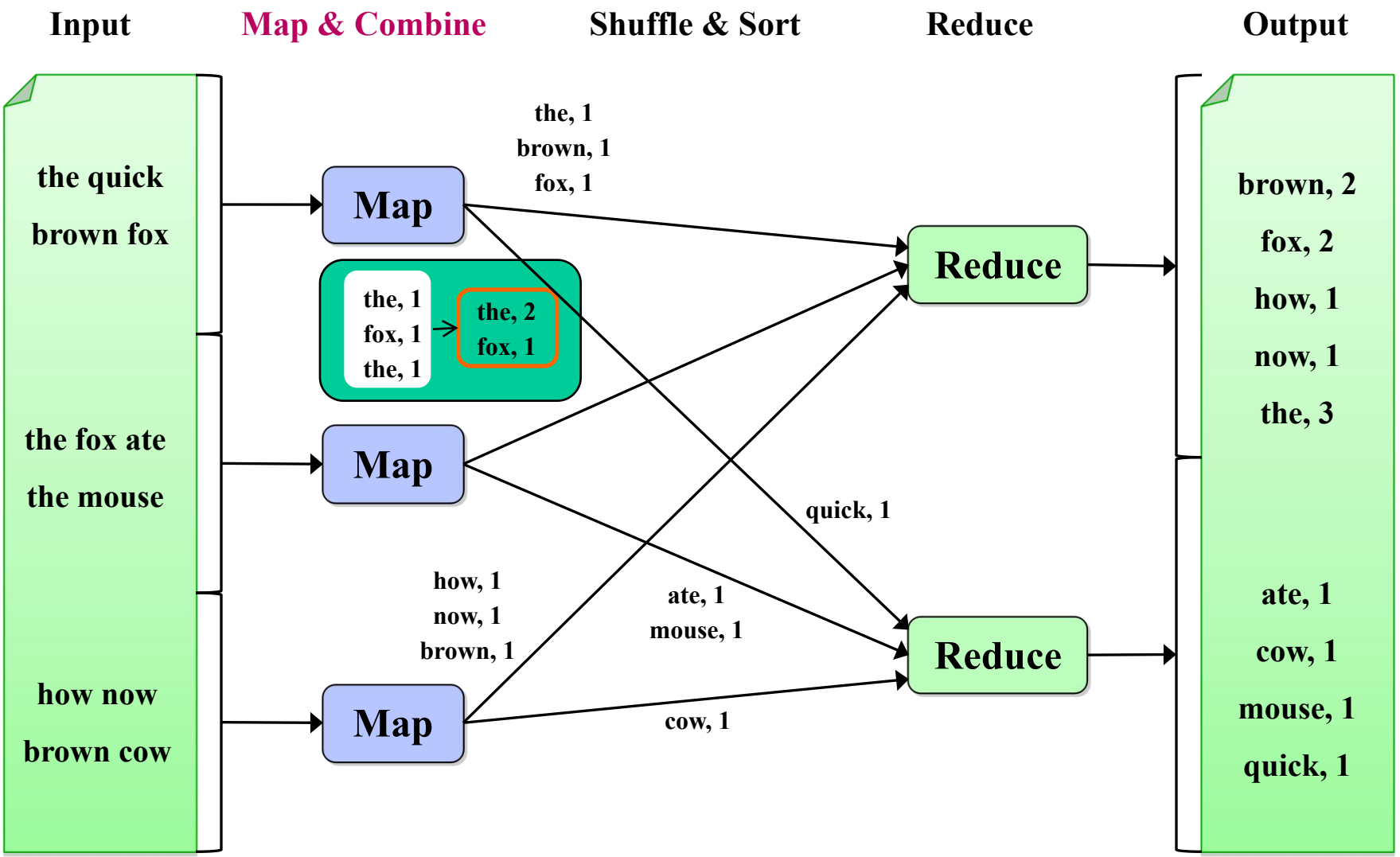
Map和Reduce任务数如何设置？

- M个 map任务（ tasks），R个reduce任务
- 实用规则：
  - M和 R一般远大于集群的节点数目
  - 一般分布式文系统的一个分片（16～64MB）对应一个 map的worker
  - R是机器节点数的小倍数 (例如，R=5000，如果机器节点数为2000）
  - 提高了动态负载均衡以及从失败的worker中恢复的速度
- 通常R小于 M，因为输出是分布到R个文件的

Combiners

- 一个map任务会产生许多值对形如 (k,v1)，(k,v2)， …，他们的k值都相同
  - 例如： Word Count中的常用词
- 如果在mapper中进行预聚合（pre-aggregating）可节省带宽
  - combine(k1, list(v1)) → v2
  - Combine函数一般与reduce 函数相同
  - “Combiner”函数和mapper运行在同一台机器上，在真正的reduce过程开始之前，先执行mini-reduce过程，以节约带宽
- reduce函数满足交换律和结合律时适用（ commutative and associative）

有Combiner的Word Count



Partition函数

- Map任务的输入是由输入文件的连续分片创建的
- 对reduce而言，我们需要确保具有相同中间键值的记录被送到同一个worker
- 系统实现了缺省的 partition函数，例如：hash(key) mod R
- 有时需要重写
  - 例如： hash(hostname(URL)) mod R 保证来自同一个站点的URLs被送到同一输出文件

负载均衡

- M和R的数值远大于机器节点数
  - 当一个worker失败，许多被分配给它的任务可被调度给其他worker
- Master 做出O(M+R)个调度决定，并在内存中维持 O(MR)个状态
- 拖尾（Stragglers）是 指那些在做最后的map或reduce时用时超长的任务
  - 负载不均衡导致
- 在MapReduce操作快要结束时，Master对余下的正在执行的任务调度备份机制
  - 不管初始任务或备份任务完成，该任务就被认为完成
  - 实验表明，这种机制极大提高了执行时间；在大规模排序时提高44%

容错机制

- Master进程周期性地 pings各个workers（No response = failed worker）
  - 如果Map-task失败，重新执行（Re-execute）
    - 所有输出都本地存储
  - 如果Reduce-task失败，只重新执行部分完成的任务
    - 所有输出存储在一个全局的文件系统中
- Master 写周期性的checkpoints
- 如果发生错误，workers发送 “last gasp”的 UDP包给master
  - 检测导致失败的记录，并跳过
- 输入文件存储在多个机器上
- 在计算快完成时，重新调度运行的任务（in-progress tasks），以避免拖尾（stragglers）

调试（Debugging）

- 提供可读的状态信息给http服务器
  - 用户可以看到已完成的作业（jobs），正在运行的作业，处理率等
- 串行执行
  - 在单个机器上串行执行
  - 允许使用gdb和其他的调试工具

Google实现概述

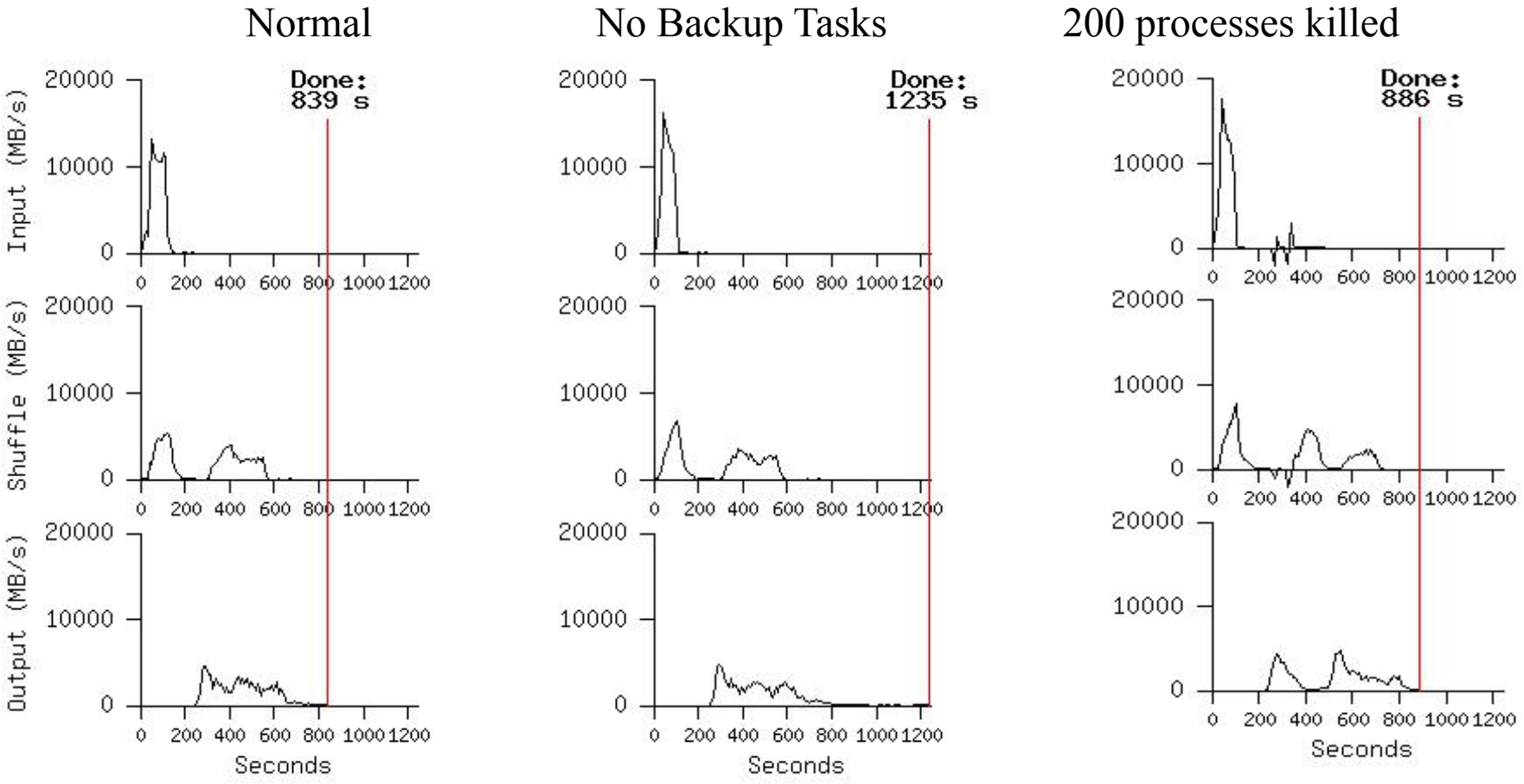
- 典型集群系统：
  - 100/1000台： 2个CPU的 x86机器， 2-4 GB内存
  - 有限的对剖带宽
  - 存储：本地IDE硬盘
  - 分布式文件系统（Google用GFS）用于管理数据
  - 任务调度系统：job由tasks组成，调度器分配task到机器
- Google用C++ 库实现，链接到用户程序

性能测试

- 在1800台机器上测试
  - 4GB内存
  - 双核2 GHz Xeons with Hyperthreading
  - 双160 GB IDE硬盘
  - 千兆以太网
- 在周末运行，机器基本上是空闲的
- 基准测试：Sort
  - 排序10^{10}个100-byte 的记录

Sort Benchmark： http://sortbenchmark.org

性能（MR_Sort）



- **Backup tasks reduce job completion time significantly**
- **System deals well with failures**

内容概要

- 大规模数据处理的挑战
- MapReduce编程模型
- MapReduce实现机制
  - Hadoop的MapReduce

Hadoop 简介

- Hadoop 是一个开源分布式计算平台，可运作于大规模cluster上的并行分布式计算框架。借助于Hadoop, 程序员可以轻松地编写分布式并行程序，将其运行于计算机集群上，完成海量数据的计算
- 实现了 Map/Reduce 计算模型
- 提供一个分布式文件系统HDFS，用来在各个节点上存储数据
- 高容错性，自动处理失败节点
- http://hadoop.apache.org

Hadoop与Google云计算系统

Hadoop云计算系统	Google云计算系统
Hadoop MapReduce	**Google MapReduce**
Hadoop HDFS	**Google GFS**
Hadoop HBase	**Google Bigtable**
Hadoop ZooKeeper	Google Chubby

Hadoop项目的模块在不断更新和增加：

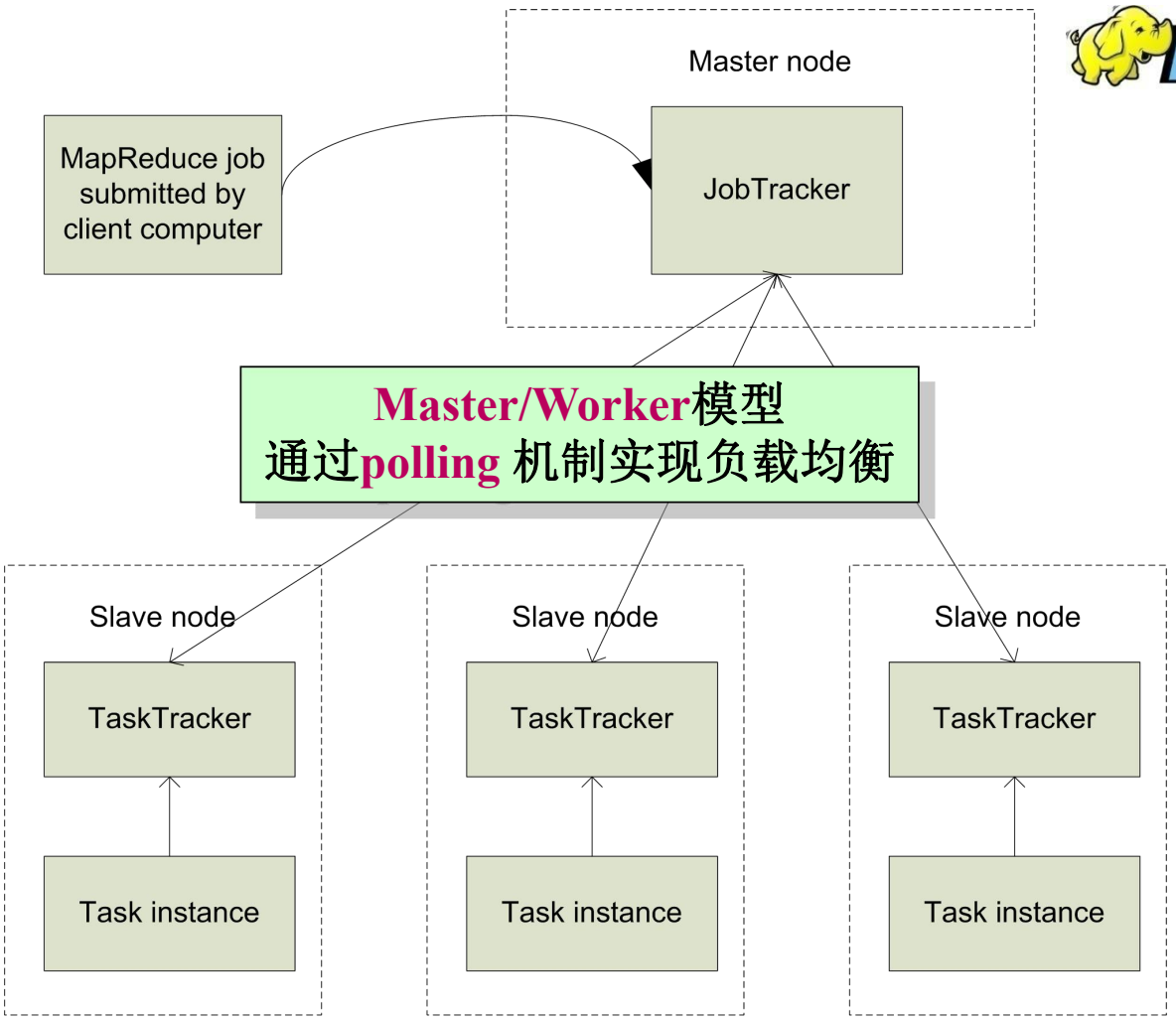
（https://hadoop.apache.org/ 2020-11-06）
目前包括以下模块：
Hadoop Common：支持其他Hadoop模块的通用实用程序。
Hadoop分布式文件系统（HDFS）：一种分布式文件系统，可提供对应用程序数据的高吞吐量访问。
Hadoop YARN：用于作业调度和集群资源管理的框架。
Hadoop MapReduce：基于YARN的系统，用于并行处理大数据集。
Hadoop Ozone： Hadoop的对象存储。

基于Hadoop实现的系统

- A9.com—Amazon
  - Amazon的产品检索索引。每天百万级的session，1 到100个节点
- Adknowlesge（adknowledge.com）—Ad network
  - 推荐系统，包括行为定位，和其他的点击分析。每天 500MM 点击，50到 200个节点，大部分位于 EC2
- Facebook （www.facebook.com/）
  - 内部日志和多维数据源，320个集群，2,560 cores和大约 1.3 PB的裸存储
- Powerset/Microsoft（www.powerset.com）—自然语言检索（Natural Language Search ）
  - 400个实例在Amazon.EC2
- Quantcast（www.quantcast.com）
  - 3000 cores, 3500TB。每天处理 1PB+。

更多应用可参考：
https://cwiki.apache.org/confluence/display/HADOOP2/PoweredBy

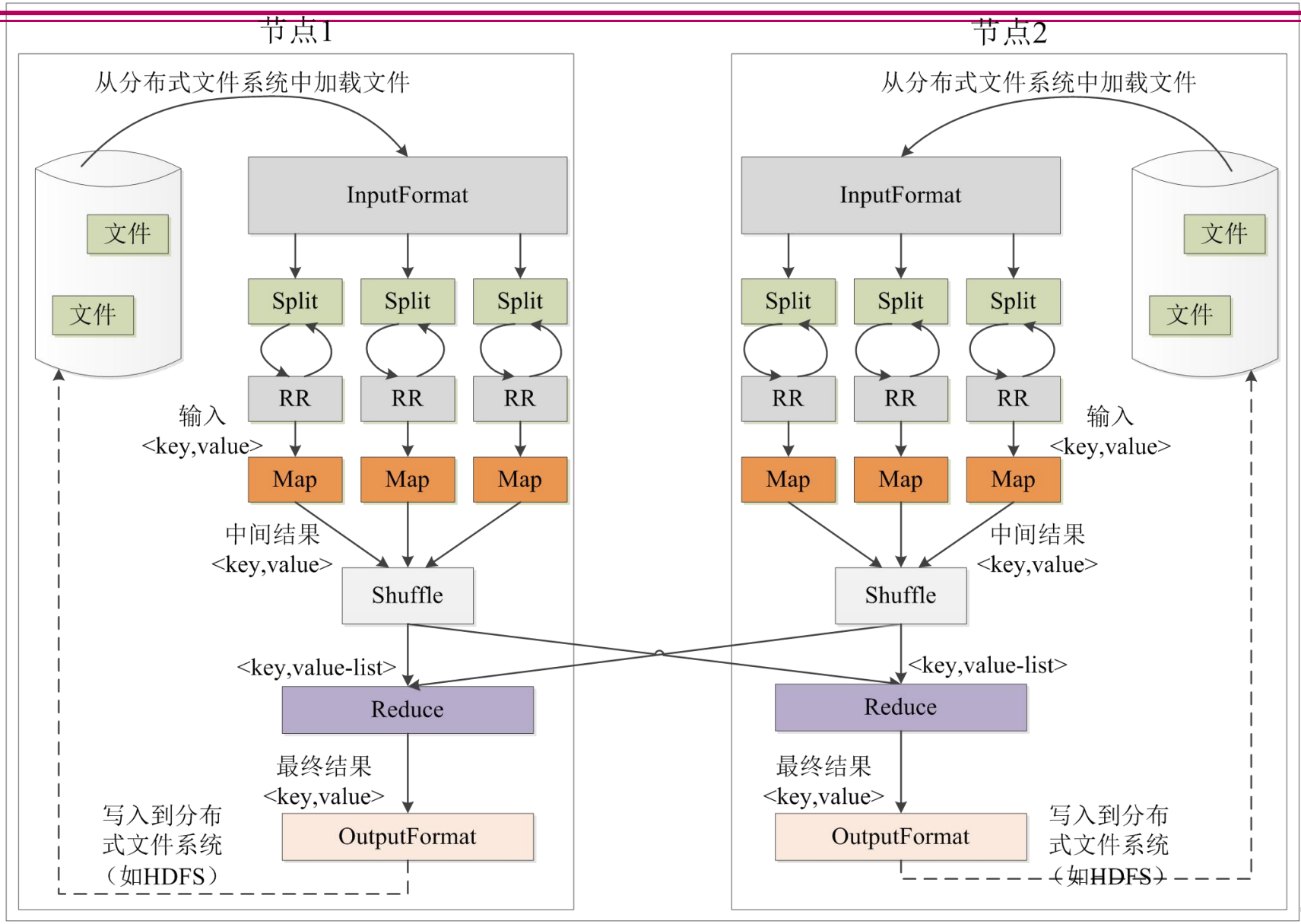
Hadoop的MapReduce架构



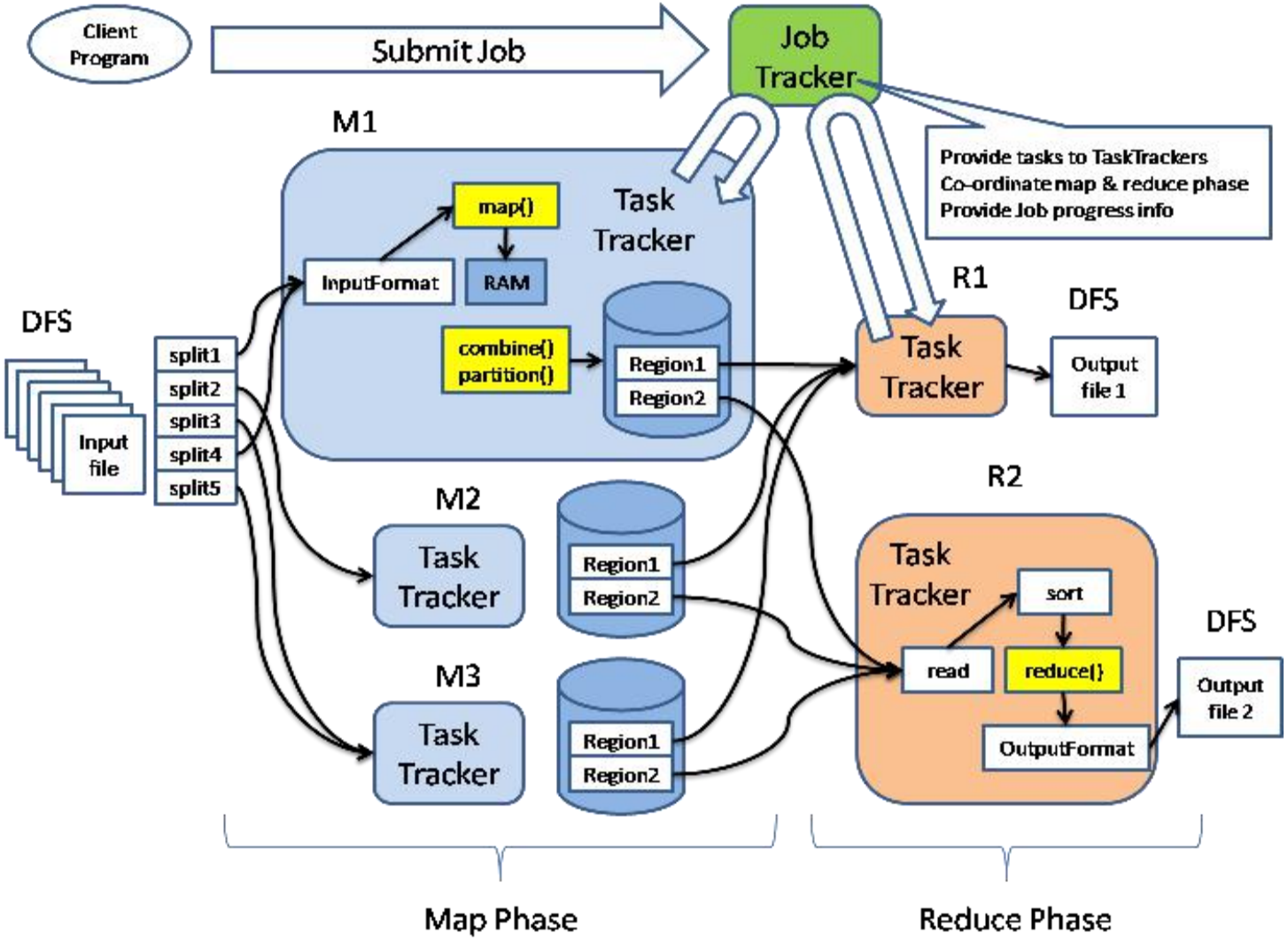
基本部件

- **Mapper**
  - **public static class TokenCounterMapper**
    extends Mapper<Object, Text, Text, IntWritable>
- **Reducer**
  - **public static class IntSumReducer**
    extends Reducer<Text,IntWritable,Text,IntWritable>
- **Driver**
  - **import org.apache.hadoop.mapreduce.Job;**

MapReduce的执行过程



MapReduce的执行过程



MapReduce的执行

- 一个JobTrackter，多个TaskTracker
- JobTrackter（Master）
  - 接收任务（job）的提交
  - 提供任务的监控（monitoring）和控制（control）
  - 把job划分成多个tasks，交给Tasktracker执行，并管理这些tasks的执行
- TaskTracker（Worker）
  - 管理单个task的map任务和reduce任务的执行

数据输入和输出

- <key, Value>对
  - 最基本的数据单元，其中key实现了WritableComparable接口，value实现了Writable接口
  - 可以对其序列化并对key执行排序
- 数据输入
  - InputFormat、InputSplit、RecordReader三种格式
  - InputFormat将输入数据切分为多个块，每个块都是InputSplit类型
  - RecordReader将每个InputSplit块分解为多个<key, value>对传送给Map任务
- 数据输出
  - FileOutputFormat、RecordWriter两个编程接口
  - FileOutputFormat将数据输出到文件
  - RecordWriter输出一个<key , value>对

容错

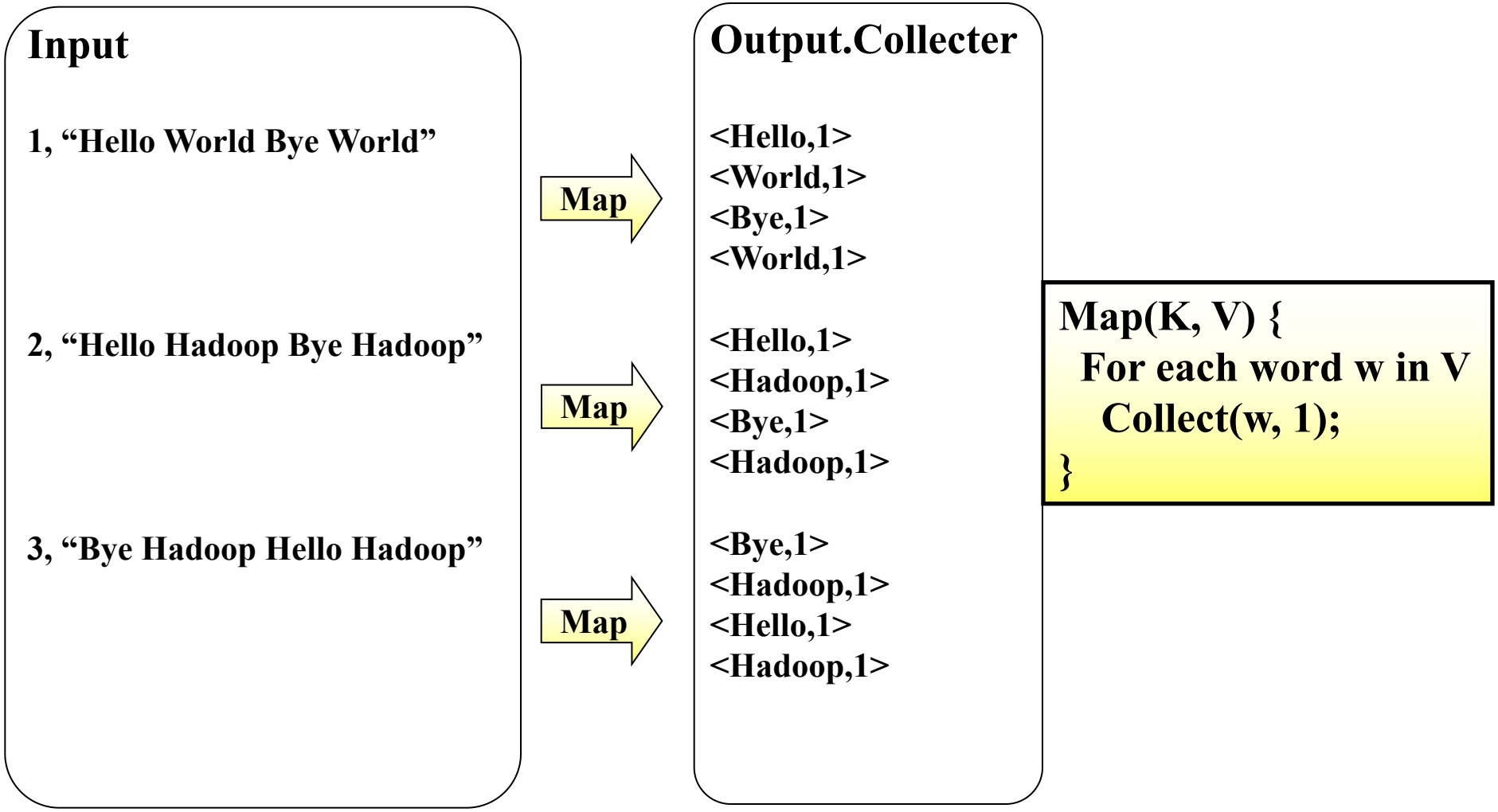
- 由Hadoop系统自己解决
- 主要方法是将失败的任务进行再次执行
- TaskTracker会把状态信息汇报给JobTracker，最终由JobTracker决定重新执行哪一个任务

- 为了加快执行的速度，Hadoop也会自动重复执行同一个任务，以最先执行成功的为准—投机执行（speculative execution）
  - mapred.map.tasks.speculative.execution
  - mapred.reduce.tasks.speculative.execution

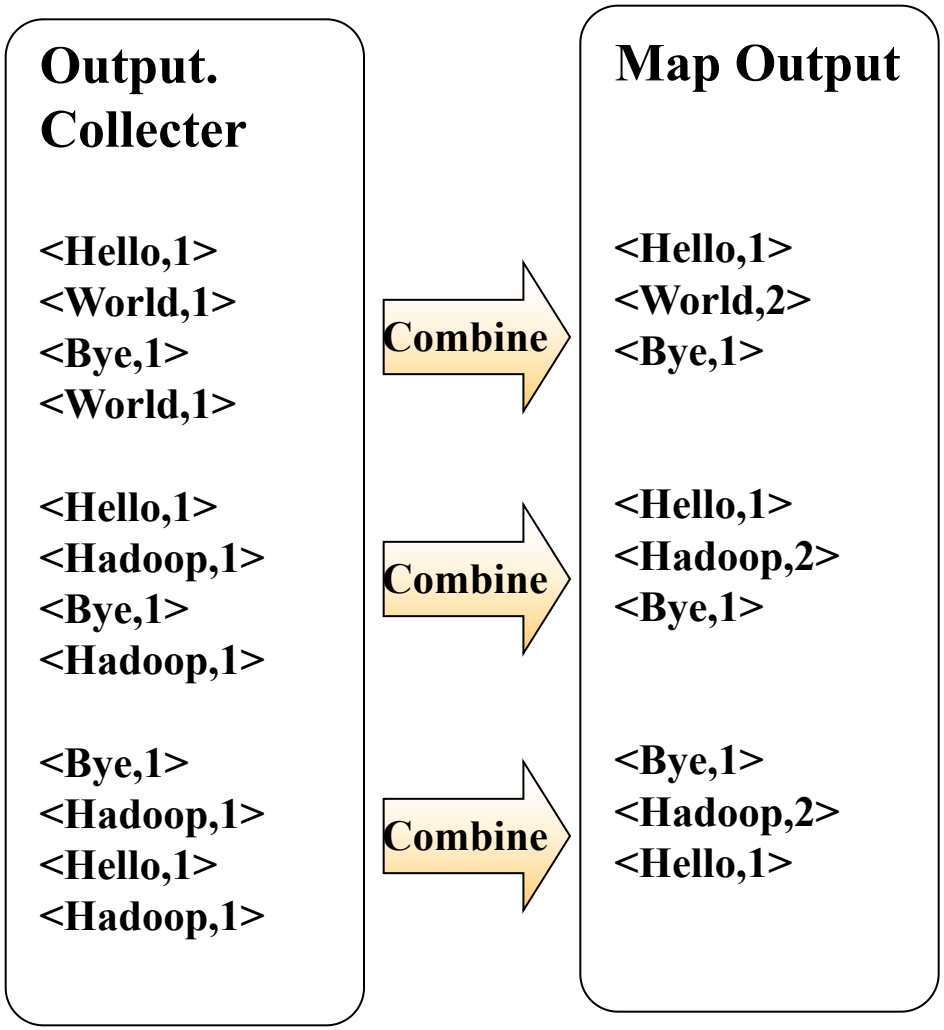
程序的调试及任务的控制

- 日志文件
  - 分门别类存放在hadoop-version/logs目录下面，hadoop-*username*-*service*-*hostname*.log
  - 对于用户程序来说，TaskTracker的log可能是最重要的
  - 每一个计算节点上有日志logs/userlogs日志，也有重要的日志信息
- 在单机上首先执行，看看是否能够正确执行，而后再在多机的集群系统上执行
- 任务的控制
  - bin/hadoop job –list
  - bin/hadoop job –kill jobid

实例分析WordCount：Map

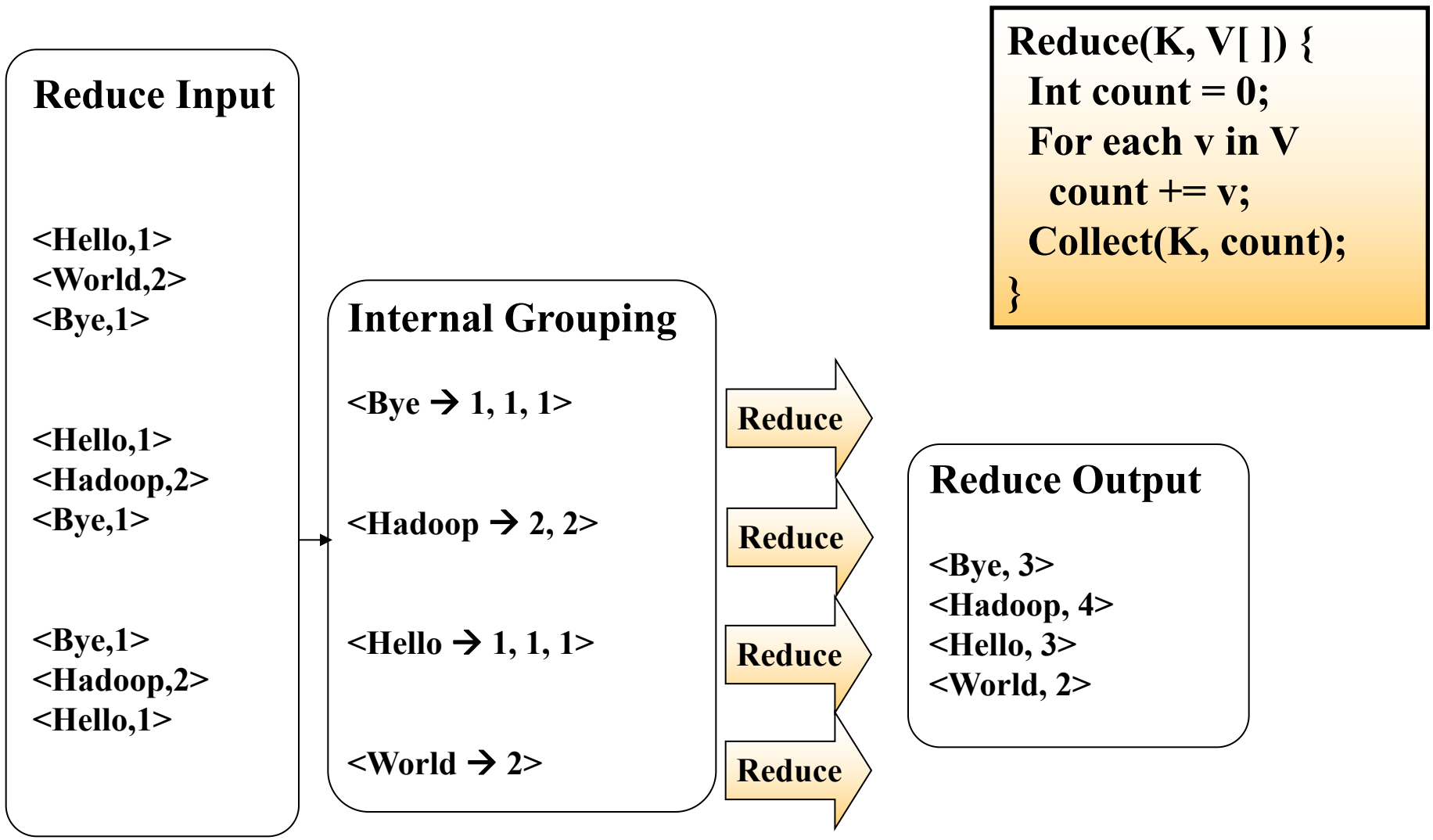


实例分析WordCount：Combine




```
Combine(K, V[ ]) {
  Int count = 0;
  For each v in V
    count += v;
  Collect(K, count);
}
```

实例分析WordCount：Reduce



MapReduce WordCount的Java代码

```
1.  package org.myorg;
2.
3.  import java.io.IOException;
4.  import java.util.*;
5.
6.  import org.apache.hadoop.fs.Path;
7.  import org.apache.hadoop.conf.*;
8.  import org.apache.hadoop.io.*;
9.  import org.apache.hadoop.mapreduce.*;
10. import org.apache.hadoop.mapreduce.lib.input.*;
11. import org.apache.hadoop.mapreduce.lib.output.*;
12. import org.apache.hadoop.util.*;
13.
14. public class WordCount extends Configured implements Tool {
15.
16.    public static class Map
17.         extends Mapper<LongWritable, Text, Text, IntWritable> {
18.       private final static IntWritable one = new IntWritable(1);
19.       private Text word = new Text();
20.
21.       public void map(LongWritable key, Text value, Context context)
22.           throws IOException, InterruptedException {
23.         String line = value.toString();
24.         StringTokenizer tokenizer = new StringTokenizer(line);
25.         while (tokenizer.hasMoreTokens()) {
26.           word.set(tokenizer.nextToken());
27.           context.write(word, one);
28.         }
29.       }
30.    }
31.
32.    public static class Reduce
33.         extends Reducer<Text, IntWritable, Text, IntWritable> {
34.       public void reduce(Text key, Iterable<IntWritable> values,
35.           Context context) throws IOException, InterruptedException {
36.
37.         int sum = 0;
38.         for (IntWritable val : values) {
39.           sum += val.get();
40.         }
41.         context.write(key, new IntWritable(sum));
42.       }
```

```
44.
45.    public int run(String [] args) throws Exception {
46.      Job job = new Job(getConf());
47.      job.setJarByClass(WordCount.class);
48.      job.setJobName("wordcount");
49.
50.      job.setOutputKeyClass(Text.class);
51.      job.setOutputValueClass(IntWritable.class);
52.
53.      job.setMapperClass(Map.class);
54.      job.setCombinerClass(Reduce.class);
55.      job.setReducerClass(Reduce.class);
56.
57.      job.setInputFormatClass(TextInputFormat.class);
58.      job.setOutputFormatClass(TextOutputFormat.class);
59.
60.      FileInputFormat.setInputPaths(job, new Path(args[0]));
61.      FileOutputFormat.setOutputPath(job, new Path(args[1]));
62.
63.      boolean success = job.waitForCompletion(true);
64.      return success ? 0 : 1;
65.    }
66.
67.    public static void main(String[] args) throws Exception {
68.      int ret = ToolRunner.run(new WordCount(), args);
69.      System.exit(ret);
70.    }
71. }
72.
```

第一步：实现Map类

- 这个类实现 Mapper 接口中的 map 方法，输入参数中的 value 是文本文件中的一行，利用 StringTokenizer 将这个字符串拆成单词，然后用Mapper.Context 输出结果 <单词,1>

WordCount的Map 类

```
16.   public static class Map
17.       extends Mapper<LongWritable, Text, Text, IntWritable> {
18.     private final static IntWritable one = new IntWritable(1);
19.     private Text word = new Text();
20.
21.     public void map(LongWritable key, Text value, Context context)
22.         throws IOException, InterruptedException {
23.       String line = value.toString();
24.       StringTokenizer tokenizer = new StringTokenizer(line);
25.       while (tokenizer.hasMoreTokens()) {
26.         word.set(tokenizer.nextToken());
27.         context.write(word, one);
28.       }
29.     }
30.   }
31.
```

第二步：实现 Reduce 类

- 这个类实现 Reducer 接口中的 reduce 方法, 输入参数中的 key, values 是由 Map 任务输出的中间结果，values 是一个 Iterator, 遍历这个 Iterator, 就可以得到属于同一个 key 的所有 value. 此处，key是一个单词，value 是词频。只需要将所有的 value 相加，就可以得到这个单词的总的出现次数。

WordCount的Reduce类

```
32.   public static class Reduce
33.       extends Reducer<Text, IntWritable, Text, IntWritable> {
34.     public void reduce(Text key, Iterable<IntWritable> values,
35.         Context context) throws IOException, InterruptedException {
36.
37.       int sum = 0;
38.       for (IntWritable val : values) {
39.         sum += val.get();
40.       }
41.       context.write(key, new IntWritable(sum));
42.     }
43.   }
44.
```

第三步：运行Job

- 在 Hadoop 中一次计算任务称之为一个 job, 可以通过一个 JobConf 对象设置如何运行这个 job。然后将 JobConf 对象作为参数，调用 JobClient 的 runJob, 开始执行这个计算任务

实例分析：WordCount

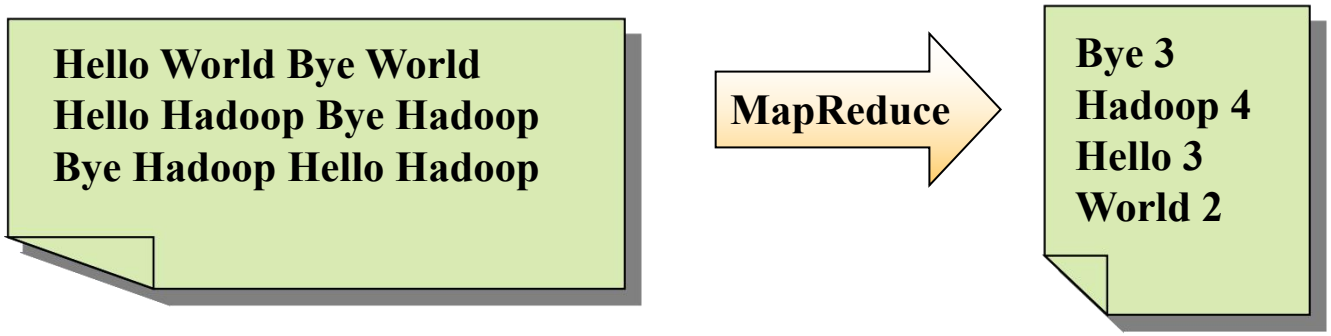
```
44.
45.     public int run(String [] args) throws Exception {
46.       Job job = new Job(getConf());
47.       job.setJarByClass(WordCount.class);
48.       job.setJobName("wordcount");
49.
50.       job.setOutputKeyClass(Text.class);
51.       job.setOutputValueClass(IntWritable.class);
52.
53.       job.setMapperClass(Map.class);
54.       job.setCombinerClass(Reduce.class);
55.       job.setReducerClass(Reduce.class);
56.
57.       job.setInputFormatClass(TextInputFormat.class);
58.       job.setOutputFormatClass(TextOutputFormat.class);
59.
60.       FileInputFormat.setInputPaths(job, new Path(args[0]));
61.       FileOutputFormat.setOutputPath(job, new Path(args[1]));
62.
63.       boolean success = job.waitForCompletion(true);
64.       return success ? 0 : 1;
65.     }
66.
67.     public static void main(String[] args) throws Exception {
68.       int ret = ToolRunner.run(new WordCount(), args);
69.       System.exit(ret);
70.     }
71. }
72.
```

WordCount实例分析：实验结果

- 实验结果

输入：包含词的文件

输出：每个词的出现次数



课程小结

- MapReduce是一个简单易用的并行编程模型，它极大简化了大规模数据处理问题的实现
- MapReduce编程模型
  - 工作流程及实现机制，Map函数，Reduce函数
- 基于MapReduce的算法
  - Word Count
- MapReduce的实现机制
  - 实现架构，负载均衡，容错，Combiner，Partitioner
- MapReduce的开源实现—Hadoop MapReduce
  - Word Count的实现例子

推荐网站和读物

- 《云计算》（第三版）
  - 第2章 Google云计算原理及应用
    - 2.2 分布式数据处理MapReduce
  - 第5章 Hadoop2.0：主流开源云架构
    - 5.6 Hadoop2.0编程接口
- J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Osdi 2004, pp. 137-150.
- Hadoop MapReduce Tutorial
  - http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

下一讲

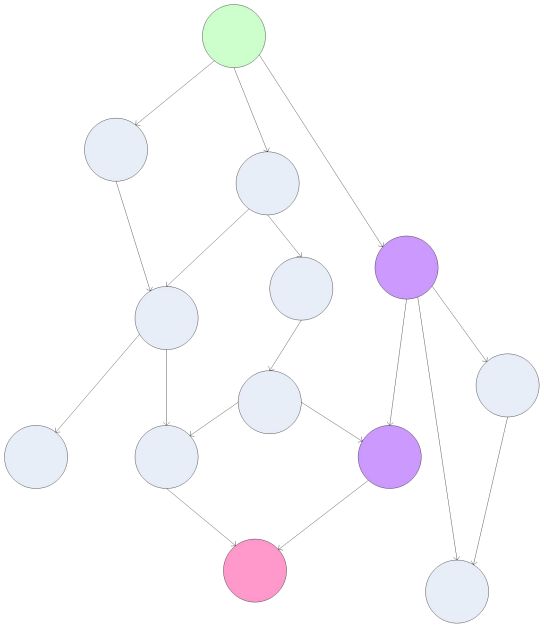
- 云计算系统

附录-1

- 基于MapReduce的算法的实现思路

最短路径算法

- 最短路经算法是经典的图搜索应用
- 单源最短路径问题（single-source shortest path problem，SSSP）的经典算法—— Dijkstra算法
- 是否可用MapReduce实现?



最短路径算法：基本思路

- 可以如下递归地解决这个问题：
  - **DistanceTo(startNode) = 0**
  - 对所有从 **startNode**直达的节点 ***n***，**DistanceTo(n) = 1**
  - 对所有从其他节点***S***直达的节点***n***,

    DistanceTo(n) = 1 + min(DistanceTo(m), m ∈ S)

最短路径的Map/Reduce算法

- **Map函数以节点 *n* 为 key，(*D*, *points-to*) 为输入值**
  - ***D* 是节点*n*到 startNode的距离**
  - ***points-to* 是可从 *n*直达的节点列表**

  对所有节点∀p ∈ points-to， 输出 (p, D+1)

- **Reduce函数对所有的 *p* 收集所有可能的距离，并选择最小值**

讨论

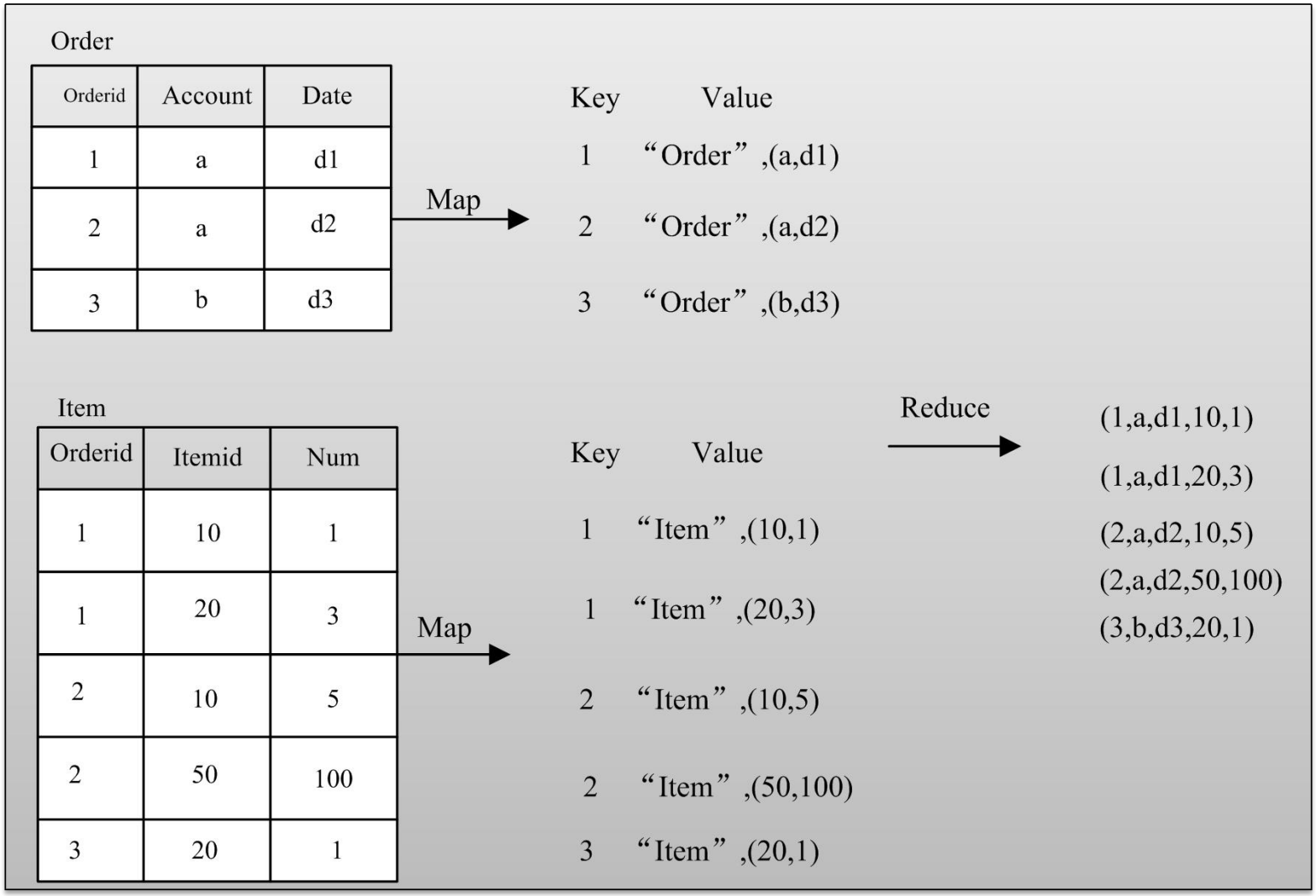
- 收敛的条件?
  - 当不再存在更佳路径时，算法会收敛
- 对有权重的最短路径
  - 只需做个简单的变换：map函数中的points-to 列表中包含到每个pointed-to节点的‘w’值
  - 对每个节点输出 (p, D+w_p)，而不是 (p, D+1)
  - 适用于正权重的图
- 与Dijkstra算法相比
  - Dijkstra算法更为高效，因为在每一步只搜索前沿中最小开销的边
  - MapReduce算法并行地搜索所有可能的路径，所以效率不太高，但扩展性很好

其他算法

- 表关系连接
- 排序
- 模式匹配

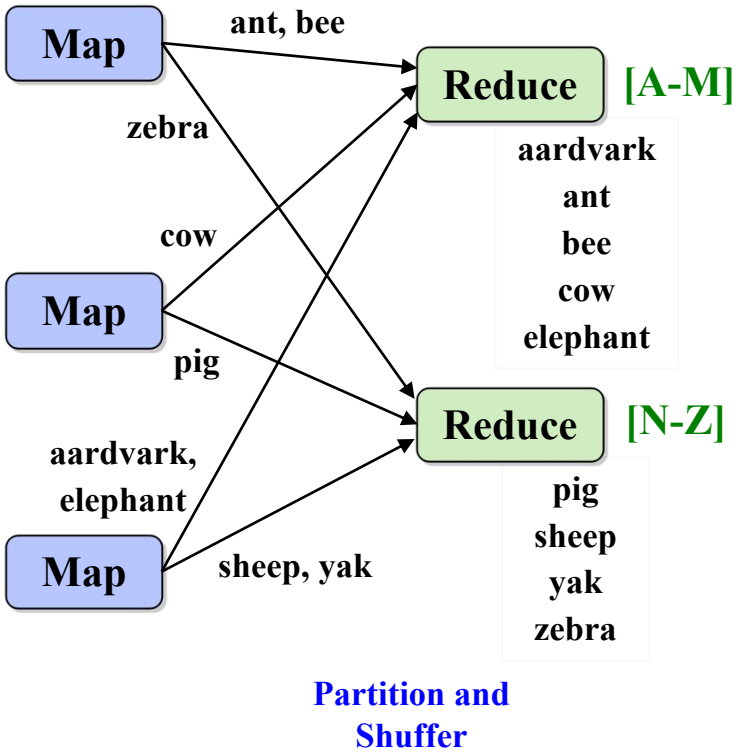
- 整个Web的词频统计（Term frequencies through the whole Web repository）
- URL访问频率的统计（Count of URL access frequency）
- 反向web链接图（Reverse web-link graph）

自然关系连接


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)

排序：基本思路

- 输入：
  - 一系列文件，每行一个值
  - Mapper的key是文件名，行号
  - Mapper的value是行的内容
- 利用reducer的特性： (key, value) 对按key的顺序被处理，reducers自动被排序



排序：技巧

- **MapReduce函数**
  - **Mapper： 恒等函数（Identity function），输出(key, value)**
  - **Reducer：恒等函数，输出(key,value)**

- **从mapper输出的(key, value)对须基于 hash(key) 被送到特定的 reducer**
- **必须为所要处理的数据选择哈希函数，使得$k_1 < k_2$ => $hash(k_1) < hash(k_2)$**

模式匹配：基本思路

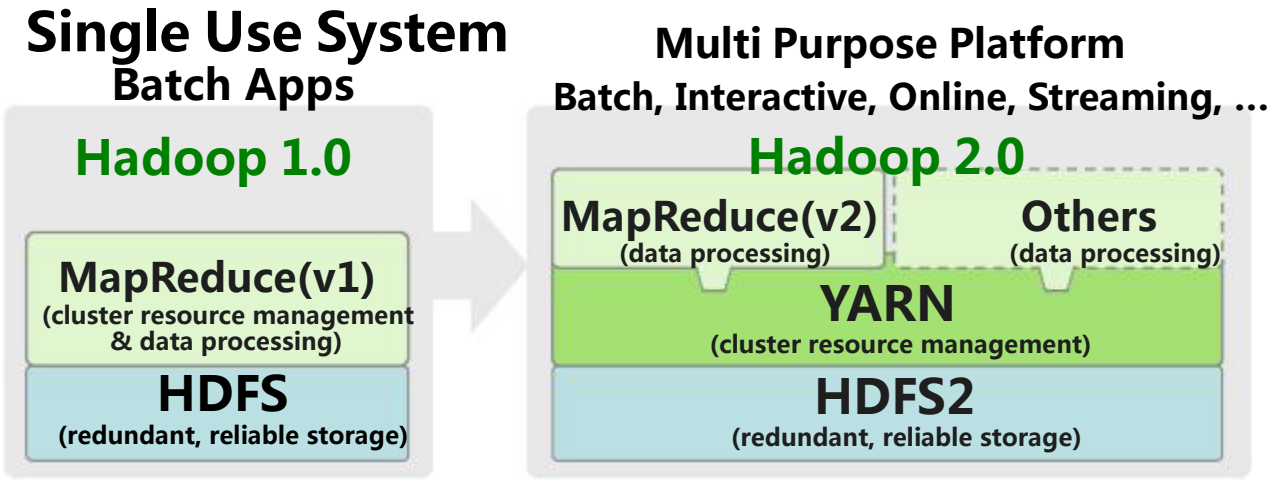
- 输入：
  - 包含文本行的一系列文件
  - 要匹配的模式（pattern）
- Mapper的key是文件名，行号
- Mapper的value是行的内容
- 模式作为特殊的参数

- MapReduce函数
  - Mapper：给定 (filename, some text) 和“pattern”，如果 “text” 匹配 “pattern”，则输出 (filename, 1)
  - Reducer：输出(filename, sum(value))

附录-2

YARN

Hadoop 2.0



- ✓ **Hadoop 2.0**：由HDFS、MapReduce和YARN三个分支构成
- ✓ **HDFS**：支持NN Federation、HA
- ✓ **MapReduce**：运行在YARN上的MR，编程模型不变
- ✓ **YARN**：资源管理系统

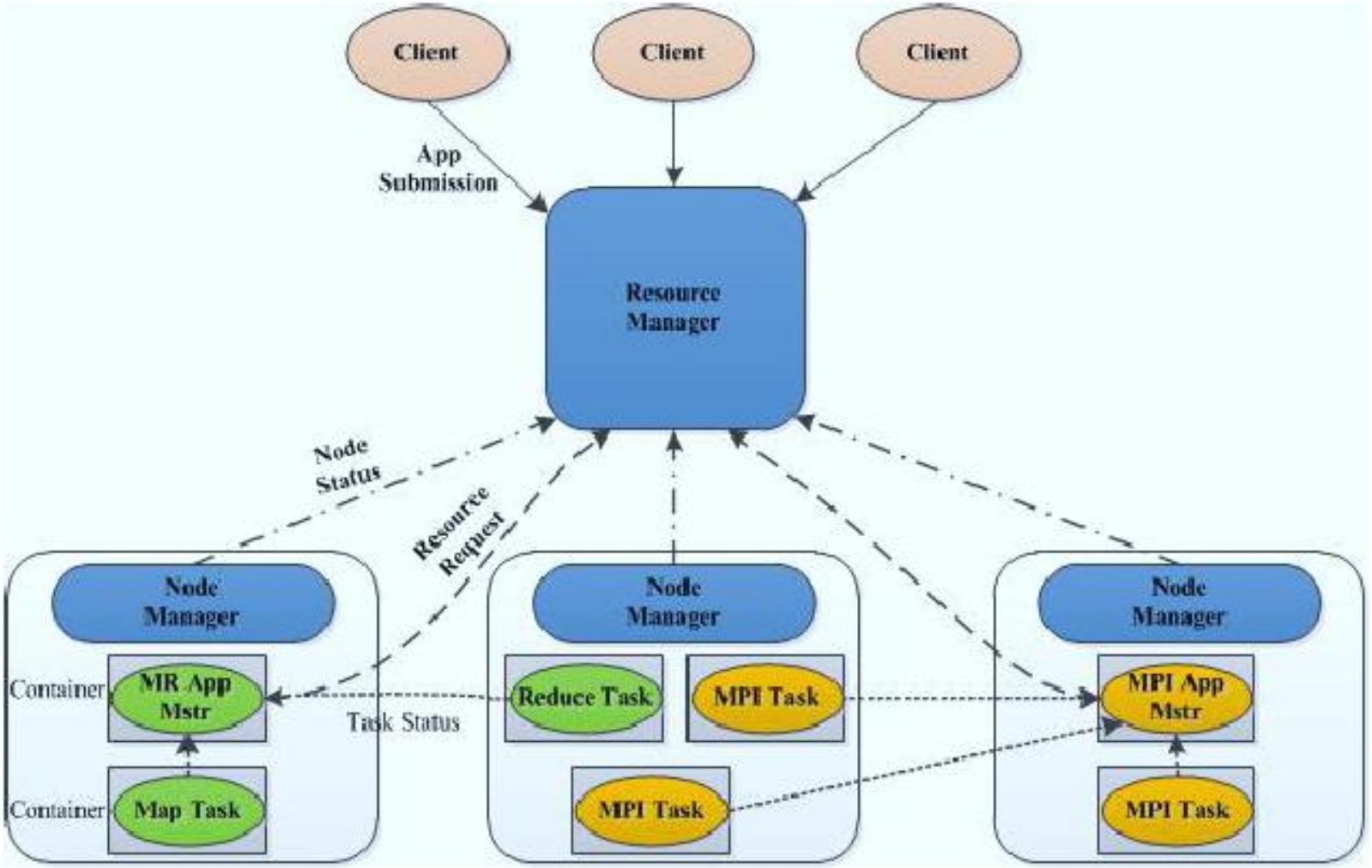
…

YARN-产生背景

MapReduce v1的不足：

- 扩展性差，JobTracker成为瓶颈
- 可靠性差，NameNode单点故障
- 扩展性差，难以支持MR之外的计算
- 资源利用率低
- 多计算框架各自为战，数据共享困难
  - MapReduce v1：离线计算框架
  - Storm：实时计算框架
  - Spark：内存计算框架

Yarn 基本框架与组件



YARN-架构及组件

ResourceManager

处理客户端请求

启动/监控**ApplicationMaster**

监控**NodeManager**

资源分配与调度

NodeManager

单个节点上的资源管理

处理来自**ResourceManager**的命令

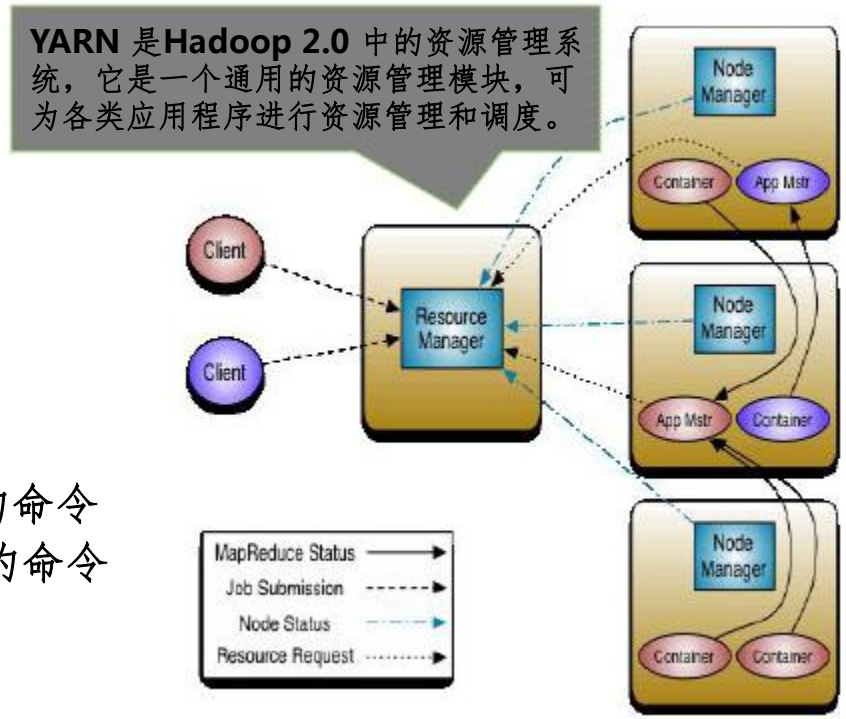
处理来自**ApplicationMaster**的命令

ApplicationMaster

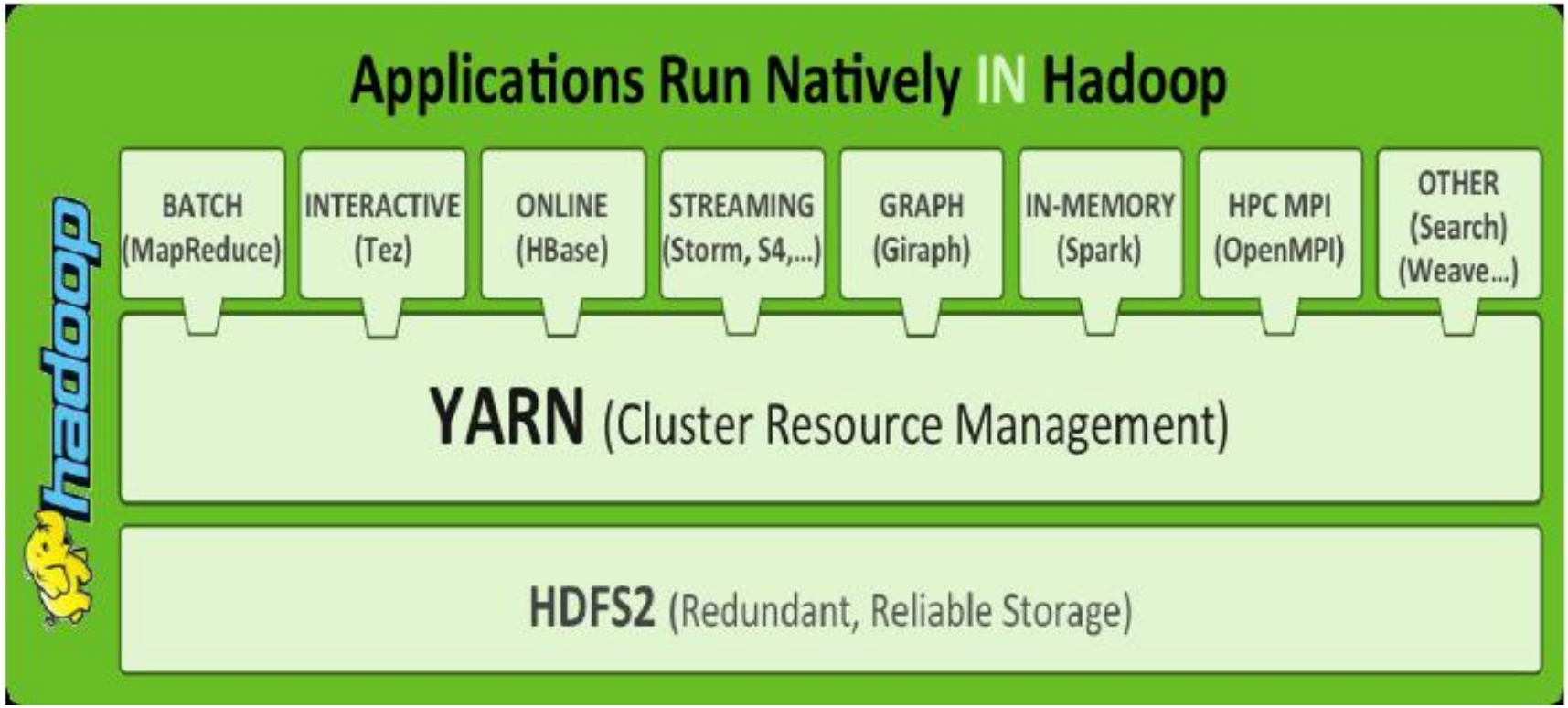
数据切分

为应用程序申请资源，并分配给内部任务

任务监控与容错



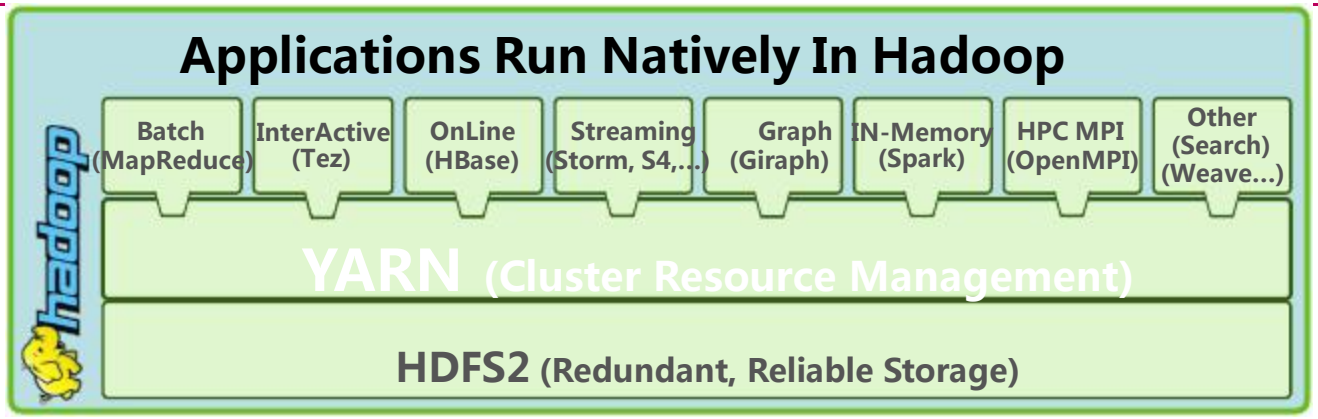
以YARN为核心的生态系统



运行在YARN上的计算框架

- 离线计算框架：**MapReduce**
- **DAG**计算框架：**Tez**
- 流式计算框架：**Storm**
- 内存计算框架：**Spark**
- 图计算框架：**Giraph**、**GraphLib**
- **……**

YARN带来的好处



运行在YARN上带来的好处：

- 一个集群部署多个版本
- 计算资源按需伸缩
- 不同负载应用混搭，集群利用率高
- 共享底层存储，避免数据跨集群迁移