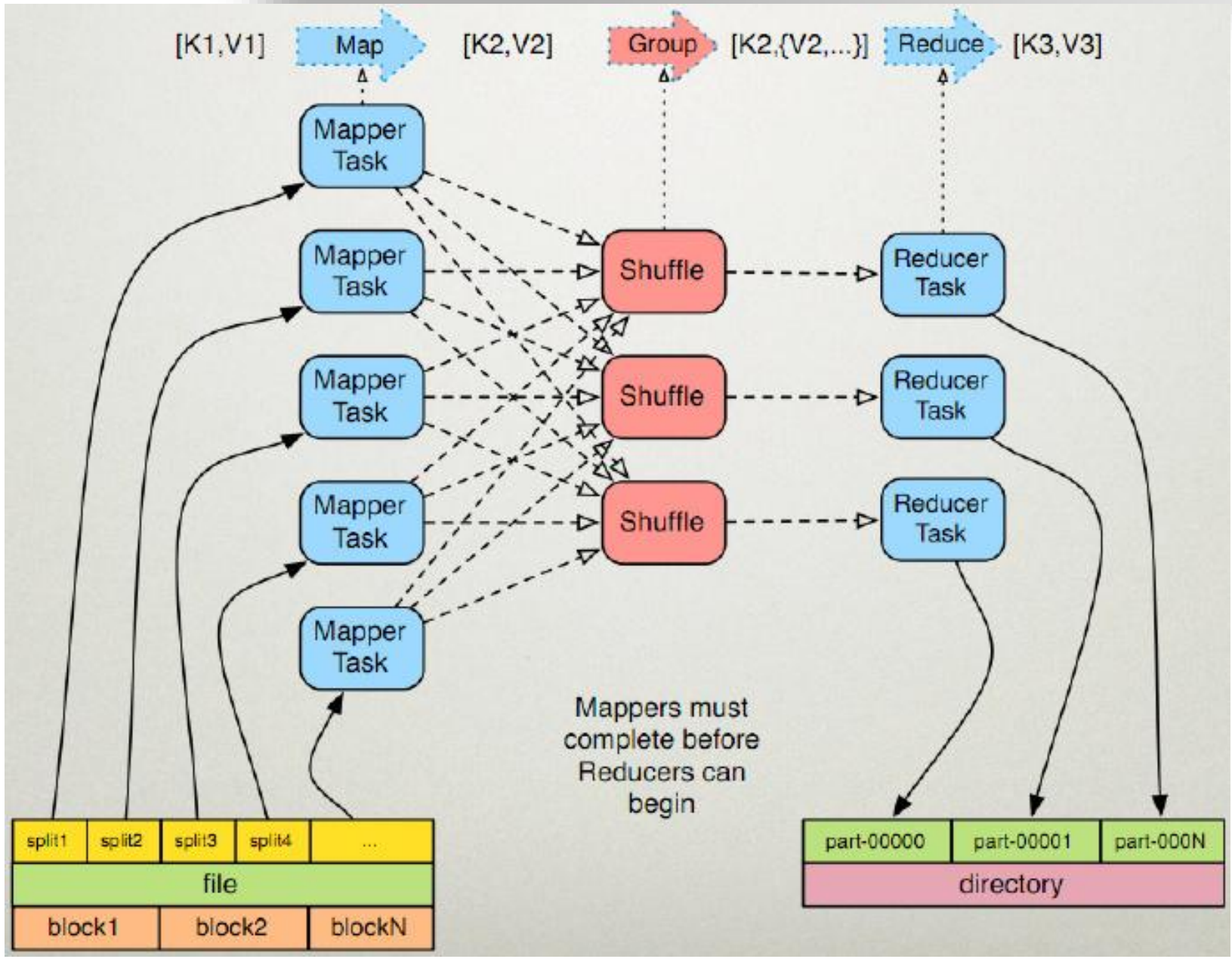
2017.09


# Hadoop离线大数据分析

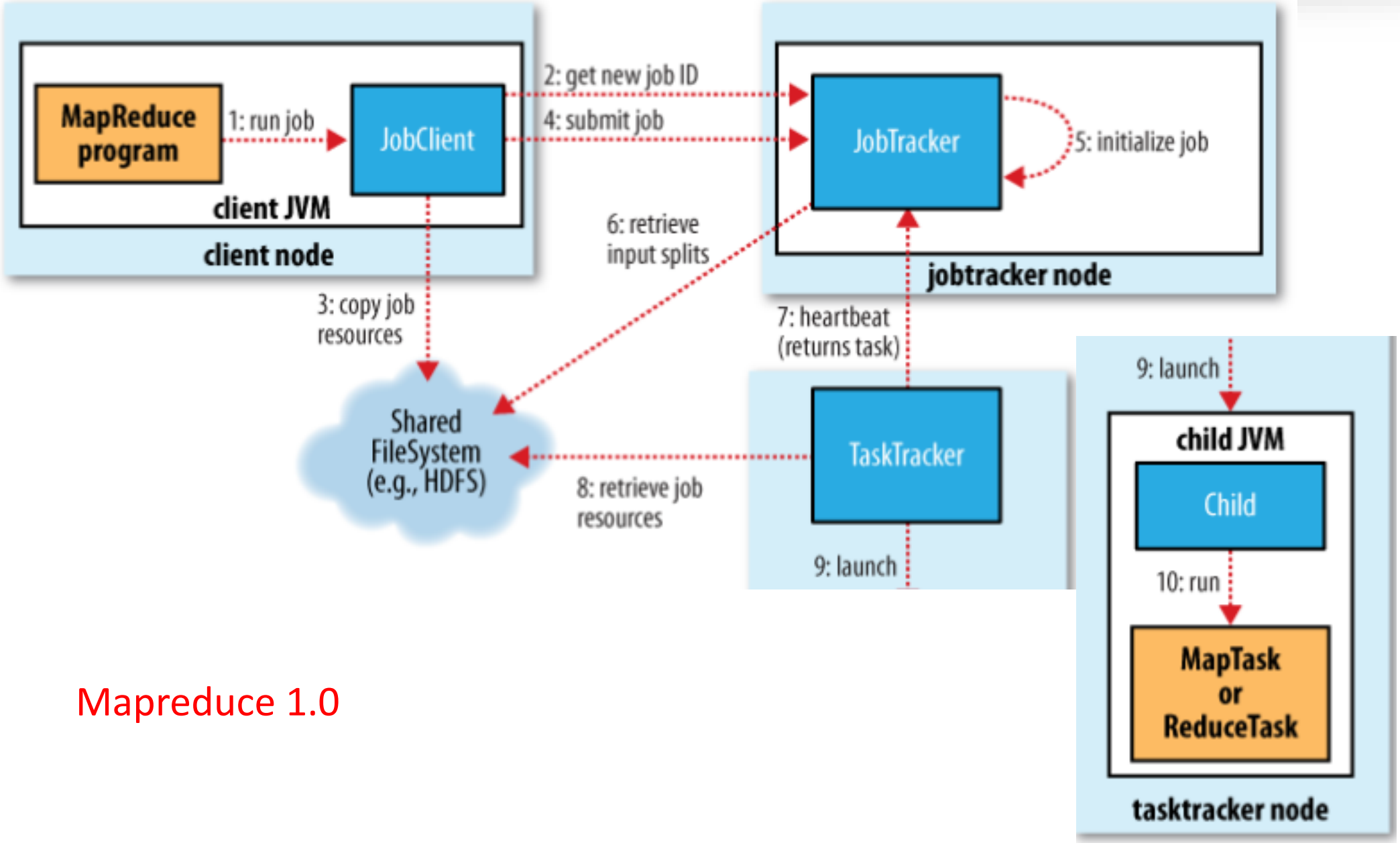
MapReduce工作机制

贝毅君 beiyj@zju.edu.cn

# 3 MR工作机制

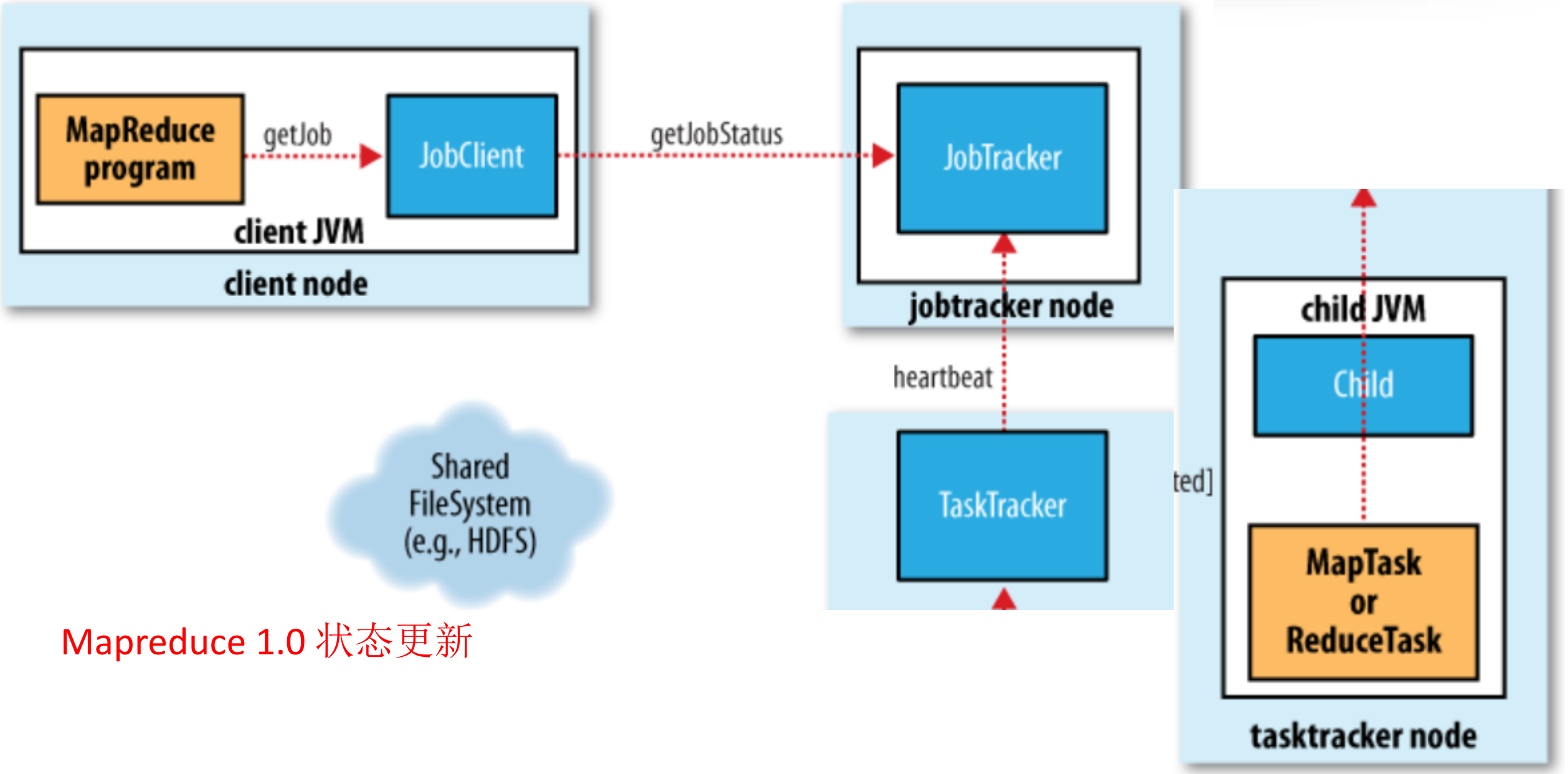

[K1,V1]
Map
[K2,V2]
Group
[K2,{V2,...}]
Reduce
[K3,V3]
Mapper Task
Mapper Task
Mapper Task
Mapper Task
Mapper Task
Shuffle
Shuffle
Shuffle
Reducer Task
Reducer Task
Reducer Task
Mappers must complete before Reducers can begin
split1
split2
split3
split4
...
file
block1
block2
blockN
part-00000
part-00001
part-000N
directory

# 3 MR工作机制



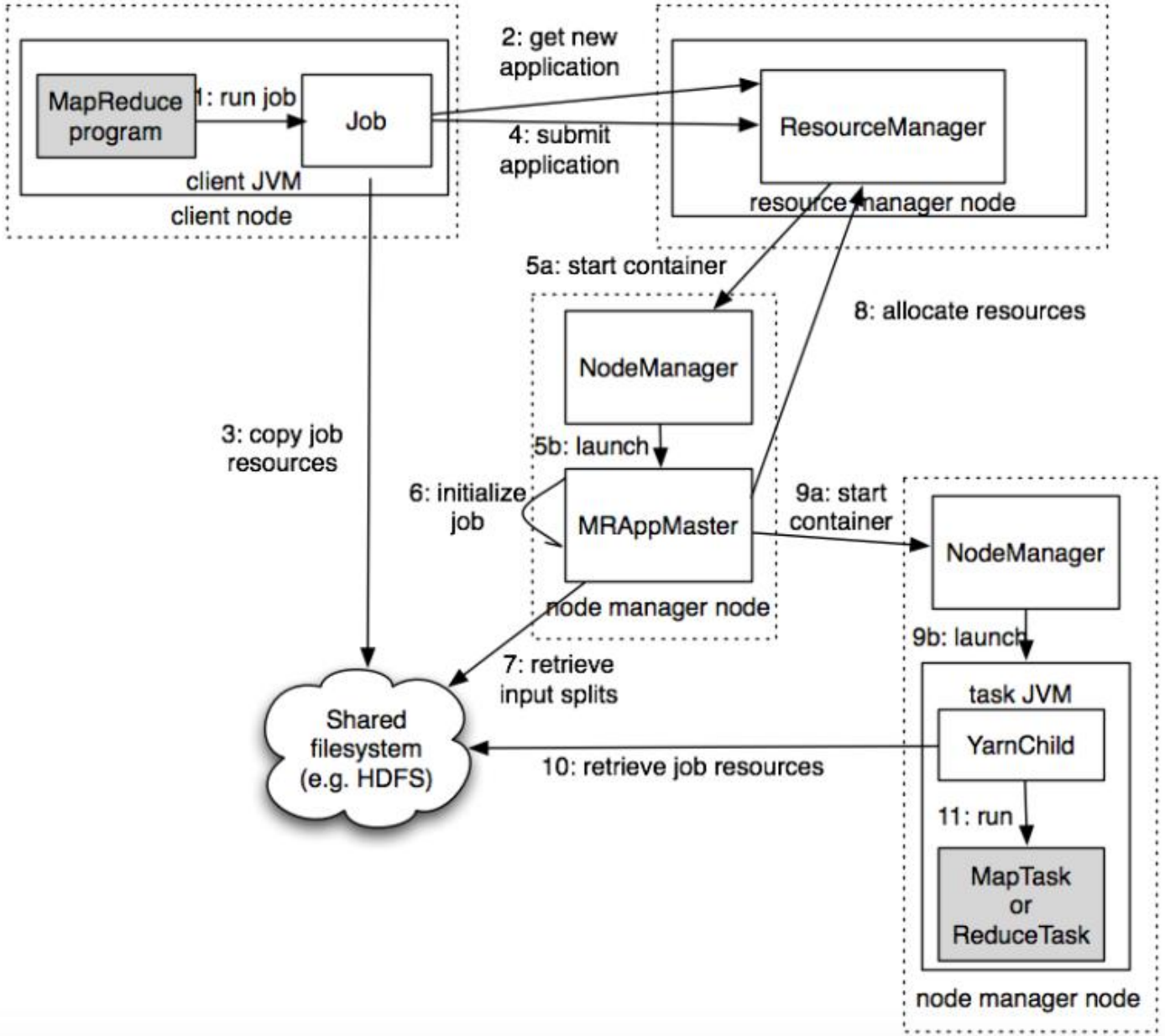

Mapreduce 1.0

# 3 MR工作机制




Mapreduce 1.0 状态更新

# 3 MR on YARN




第一代MapReduce存在的问题：

- JobTracker是集群事务的集中处理点，存在单点故障
- JobTracker需要完成的任务太多，既要维护job的状态又要维护job的task的状态，造成过多的资源消耗
- 在taskTracker端，用map/reduce task作为资源的表示过于简单，没有考虑到CPU、内存等资源情况，当把两个需要消耗大内存的task调度到一起，很容易出现OOM
- 把资源强制划分为map/reduce slot,当只有map task时，reduce slot不能用；当只有reduce task时，map slot不能用，容易造成资源利用不足。

# Yarn架构

Yarn/MRv2最基本的想法是将原JobTracker主要的资源管理和job调度/监视功能分开作为两个单独的守护进程。有一个全局的ResourceManager(RM)和每个Application有一个ApplicationMaster(AM)

Application相当于map-reduce job或者DAG jobs。ResourceManager和NodeManager(NM)组成了基本的数据计算框架。ResourceManager协调集群的资源利用，任何client或者运行着的applicatitonMaster想要运行job或者task都得向RM申请一定的资源

ApplicatonMaster是一个框架特殊的库，对于MapReduce框架而言有它自己的AM实现，用户也可以实现自己的AM，在运行的时候，AM会与NM一起来启动和监视tasks

# MRv2运行流程

1. MR JobClient向resourceManager(AsM)提交一个job
2. AsM向Scheduler请求一个供MR AM运行的container，然后启动它
3. MR AM启动起来后向AsM注册
4. MR JobClient向AsM获取到MR AM相关的信息，然后直接与MR AM进行通信
5. MR AM计算splits并为所有的map构造资源请求
6. MR AM做一些必要的MR OutputCommitter的准备工作
7. MR AM向RM(Scheduler)发起资源请求，得到一组供map/reduce task运行的container，然后与NM一起对每一个container执行一些必要的任务，包括资源本地化等
8. MR AM 监视运行着的task 直到完成，当task失败时，申请新的container运行失败的task
9. 当每个map/reduce task完成后，MR AM运行MR OutputCommitter的cleanup 代码，也就是进行一些收尾工作
10. 当所有的map/reduce完成后，MR AM运行OutputCommitter的必要的job commit或者abort APIs
11. MR AM退出

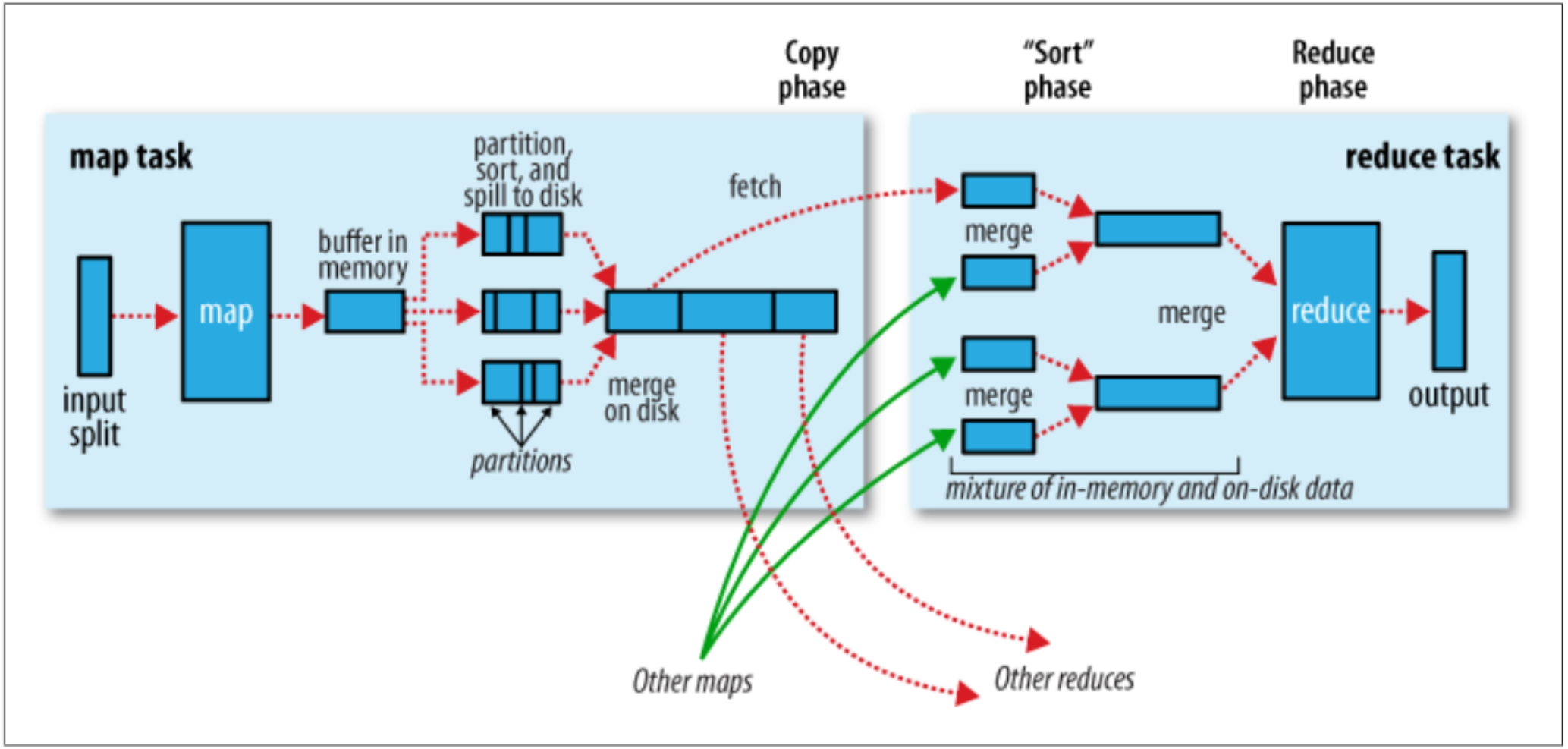
# YARN主要优势

YARN大大减少了Job Tracker的资源消耗，并且让监测每个Job子任务状态的程序分布式化了

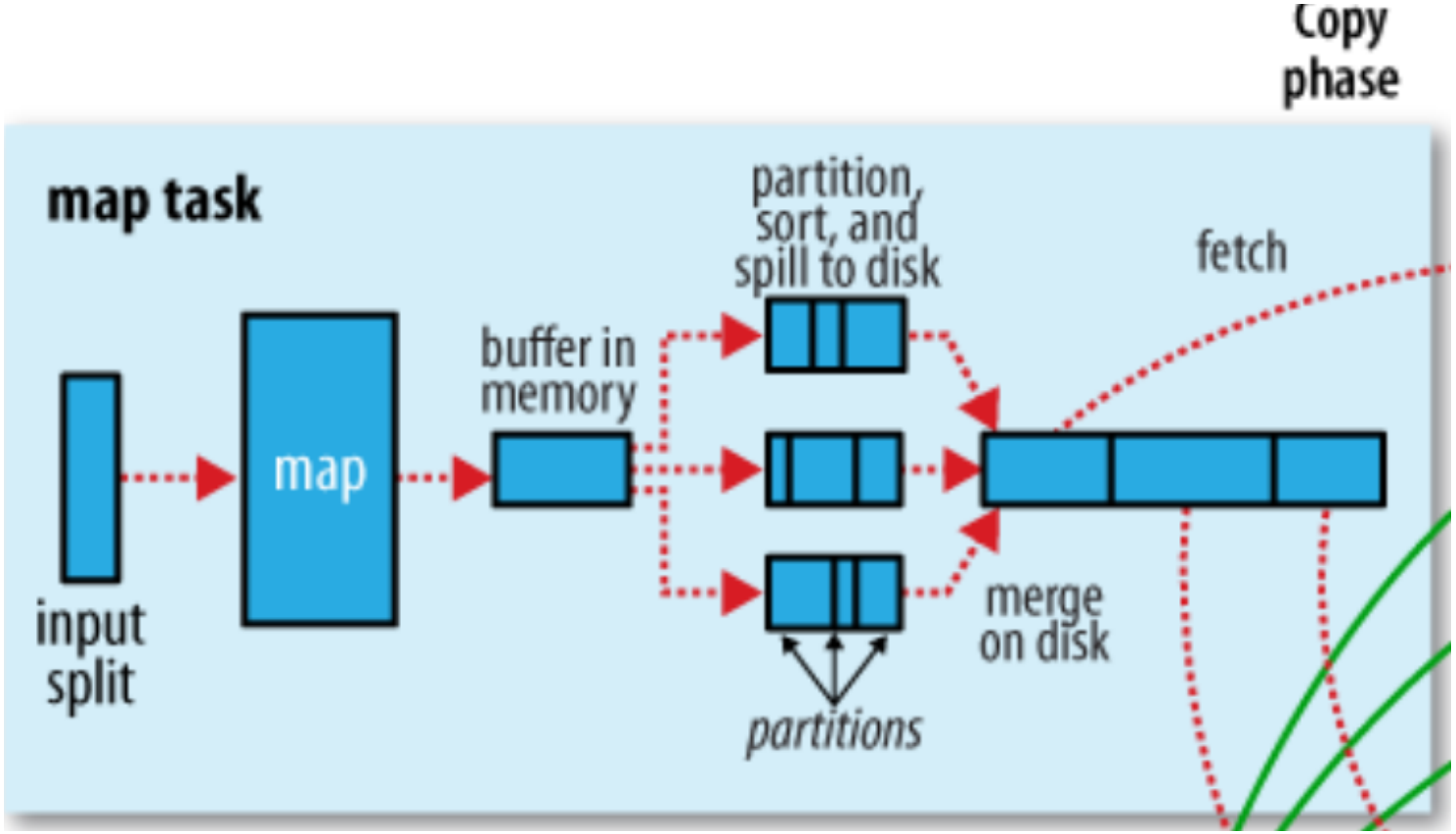
YARN中Application Master是一个可变更部分，用户可以对不同编程模型编写自己的AppMst，让更多类型的编程模型能跑在Hadoop集群中（spark on YARN）

老的框架中，Job Tracker一个很大的负担就是监控Job下任务的运行状况，现在由Application Master去做，而Resource Manager是监测Application Master的运行状况，如果出问题，会将其在其他机器上重启

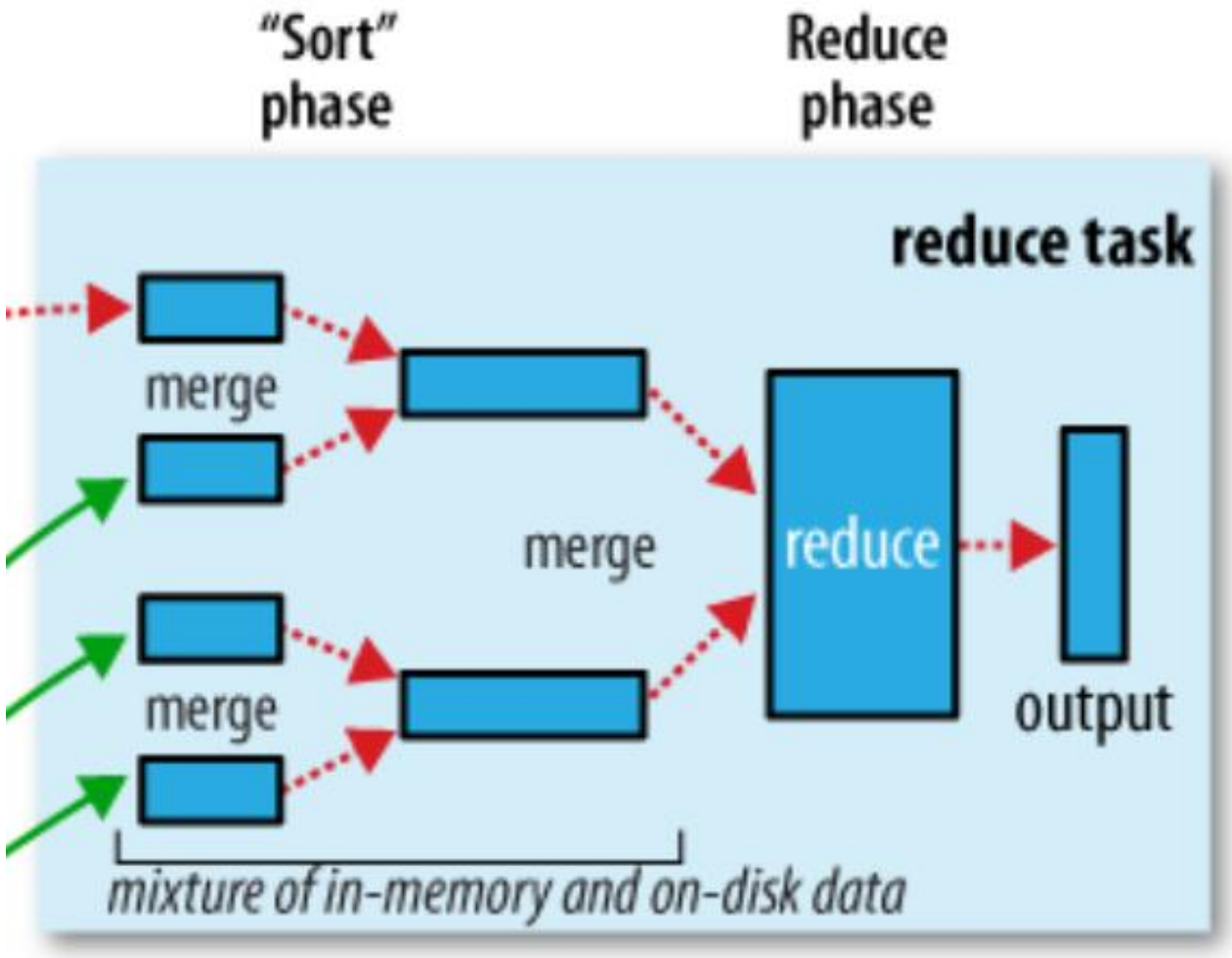
# 3 shuffle


Copy phase
"Sort" phase
Reduce phase
map task
reduce task
input split
map
buffer in memory
partition, sort, and spill to disk
partitions
merge on disk
fetch
merge
merge
merge
reduce
output
mixture of in-memory and on-disk data
Other maps
Other reduces

# 3 shuffle


Copy phase
map task
partition, sort, and spill to disk
fetch
buffer in memory
map
input split
merge on disk
partitions

# 3 shuffle


"Sort" phase
Reduce phase
reduce task
merge
merge
merge
reduce
output
mixture of in-memory and on-disk data

# 3 shuffle

shuffle是MR处理流程中的一个过程，它的每一个处理步骤是分散在各个map task和reduce task节点上完成的，整体来看，分为3个操作：

1. 分区partition
2. Sort根据key排序
3. Combiner进行局部value的合并

1.每个map有一个环形内存缓冲区，用于存储任务的输出。默认大小100MB（io.sort.mb属性），一旦达到阀值0.8（io.sort.spill.percent）,一个后台线程把内容写到(spill)磁盘的指定目录（mapred.local.dir）下的新建的一个溢出写文件。
2.写磁盘前，要partition,sort。如果有combiner，combine排序后数据。
3.等最后记录写完，合并全部溢出写文件为一个分区且排序的文件。

1.Reducer通过Http方式得到输出文件的分区。
2.TaskTracker为分区文件运行Reduce任务。复制阶段把Map输出复制到Reducer的内存或磁盘。一个Map任务完成，Reduce就开始复制输出。
3.排序阶段合并map输出。然后走Reduce阶段。

# map端调优

*Table 6-1. Map-side tuning properties*

| Property name | Type | Default value | Description |
|---|---|---|---|
| io.sort.mb | int | 100 | The size, in megabytes, of the memory buffer to use while sorting map output. |
| io.sort.record.percent | float | 0.05 | The proportion of io.sort.mb reserved for storing record boundaries of the map outputs. The remaining space is used for the map output records themselves. This property was removed in release 0.21.0 as the shuffle code was improved to do a better job of using all the available memory for map output and accounting information. |

# map端调优

| | | | |
|---|---|---|---|
| io.sort.spill.percent | float | 0.80 | The threshold usage proportion for both the map output memory buffer and the record boundaries index to start the process of spilling to disk. |
| io.sort.factor | int | 10 | The maximum number of streams to merge at once when sorting files. This property is also used in the reduce. It's fairly common to increase this to 100. |

# map端调优

| Property name | Type | Default value | Description |
| --- | --- | --- | --- |
| min.num.spills.for.combine | int | 3 | The minimum number of spill files needed for the combiner to run (if a combiner is specified). |
| mapred.compress.map.output | boolean | false | Compress map outputs. |
| mapred.map.output.compression.codec | Class name | org.apache.hadoop.io.compress.DefaultCodec | The compression codec to use for map outputs. |
| task tracker.http.threads | int | 40 | The number of worker threads per tasktracker for serving the map outputs to reducers. This is a cluster-wide setting and cannot be set by individual jobs. Not applicable in MapReduce 2. |

# reduce端调优

*Table 6-2. Reduce-side tuning properties*

| Property name | Type | Default value | Description |
|---|---|---|---|
| mapred.reduce.parallel.copies | int | 5 | The number of threads used to copy map outputs to the reducer. |
| mapred.reduce.copy.backoff | int | 300 | The maximum amount of time, in seconds, to spend retrieving one map output for a reducer before declaring it as failed. The reducer may repeatedly reattempt a transfer within this time if it fails (using exponential backoff). |

# reduce端调优

| | | | |
|---|---|---|---|
| mapred.reduce.copy.backoff | int | 300 | The maximum amount of time, in seconds, to spend retrieving one map output for a reducer before declaring it as failed. The reducer may repeatedly reattempt a transfer within this time if it fails (using exponential backoff). |
| io.sort.factor | int | 10 | The maximum number of streams to merge at once when sorting files. This property is also used in the map. |

# reduce端调优

| | | | |
|---|---|---|---|
| mapred.job.shuffle.input.buffer.percent | float | 0.70 | The proportion of total heap size to be allocated to the map outputs buffer during the copy phase of the shuffle. |
| mapred.job.shuffle.merge.percent | float | 0.66 | The threshold usage proportion for the map outputs buffer (defined by mapred.job.shuffle.input.buffer.percent) for starting the process of merging the outputs and spilling to disk. |

# reduce端调优

| | | | |
|---|---|---|---|
| mapred.inmem.merge.threshold | int | 1000 | The threshold number of map outputs for starting the process of merging the outputs and spilling to disk. A value of 0 or less means there is no threshold, and the spill behavior is governed solely by mapred.job.shuffle.merge.percent. |
| mapred.job.reduce.input.buffer.percent | float | 0.0 | The proportion of total heap size to be used for retaining map outputs in memory during the reduce. For the reduce phase to begin, the size of map outputs in memory must be no more than this size. By default, all map outputs are merged to disk before the reduce begins, to give the reducers as much memory as possible. However, if your reducers require less memory, this value may be increased to minimize the number of trips to disk. |

# 3 Partitioner编程

在某些时候，需要让某个mapper的计算结果进入指定的reducer逻辑，在mapreduce中，使用Partitioner完成

Partitioner是partitioner机制的基类，如果需要定制partitioner机制也需要继承该类。

HashPartitioner是mapreduce的默认partitioner。计算方法是:
which reducer =
(key.hashCode() &
Integer.MAX_VALUE) %
numReduceTasks
得到当前的目的reducer。

# 3 combiner编程

- 每一个map可能会产生大量的输出，combiner的作用就是在map端对输出先做一次合并，以减少传输到reducer的数据量。
- combiner最基本是实现本地key的归并，combiner具有类似本地的reduce功能
- 如果不用combiner，那么，所有的结果都是reduce完成，效率会相对低下。使用combiner，先完成的map会在本地聚合，提升速度。

具体实现步骤：
自定义一个combiner继承Reducer，重写reduce方法
在job中设置：
job.setCombinerClass(CustomCombiner.class)

**注意**：combiner使用的原则是：有或没有都不能影响业务逻辑
Combiner的输出是Reducer的输入，如果Combiner是可插拔的，添加Combiner绝不能改变最终的计算结果。

Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，且不影响最终结果的场景。

比如累加，最大值等

# 需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(1)定义一个 mapper 类

```
//首先要定义四个泛型的类型
//keyin:  LongWritable      valuein: Text
//keyout: Text              valueout:IntWritable

public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{
    //map 方法的生命周期：  框架每传一行数据就被调用一次
    //key :  这一行的起始点在文件中的偏移量
    //value: 这一行的内容
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        //拿到一行数据转换为 string
        String line = value.toString();
        //将这一行切分出各个单词
        String[] words = line.split(" ");
        //遍历数组，输出<单词，1>
        for(String word:words){
            context.write(new Text(word), new IntWritable(1));
```

# 需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(2)定义一个 reducer 类

```
    //生命周期：框架每传递进来一个 kv 组，reduce 方法被调用一次
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException {
        //定义一个计数器
        int count = 0;
        //遍历这一组 kv 的所有 v，累加到 count 中
        for(IntWritable value:values){
            count += value.get();
        }
        context.write(key, new IntWritable(count));
    }
}
```

# 需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

(3)定义一个主类，用来描述 job 并提交 job

```
public class WordCountRunner {
    //把业务逻辑相关的信息（哪个是 mapper，
哪个是 reducer，要处理的数据在哪里，
输出的结果放哪里……）描述成一个 job 对象
    //把这个描述好的 job 提交给集群去运行
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job wcjob = Job.getInstance(conf);
        //指定我这个 job 所在的 jar 包
//      wcjob.setJar("/home/hadoop/wordcount.jar");
        wcjob.setJarByClass(WordCountRunner.class);

        wcjob.setMapperClass(WordCountMapper.class);
        wcjob.setReducerClass(WordCountReducer.class);
```

```
//设置我们的业务逻辑 Mapper 类的输出 key 和 value 的数据类型
wcjob.setMapOutputKeyClass(Text.class);
wcjob.setMapOutputValueClass(IntWritable.class);
//设置我们的业务逻辑 Reducer 类的输出 key 和 value 的数据类型
wcjob.setOutputKeyClass(Text.class);
wcjob.setOutputValueClass(IntWritable.class);

//指定要处理的数据所在的位置
FileInputFormat.setInputPaths(wcjob, "hdfs://hdp-server01:9000/wordcount/data/big.txt");
//指定处理完成之后的结果所保存的位置
```

2017.08


# THE END