海量数据处理中的云计算

C5. MapReduce（一）

北京邮电大学信息与通信工程学院

2013年春季学期

课后问题记录及解答

- 问题：？
  - 解答：？

上节作业

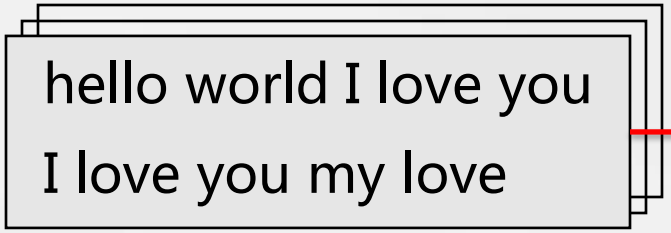
- HDFS还存在哪些问题？针对这些问题有什么解决方法？（任选一个问题）
  - 对HDFS存在问题的详细说明
  - 这些问题目前已有的解决方法，以及这些方法的优点和缺点

- 与RAID、NFS、SAN、NAS等存储技术的比较
  - 比较的详细说明

本节目录

- WordCount
- MapReduce原理及流程

从WordCount开始



Input

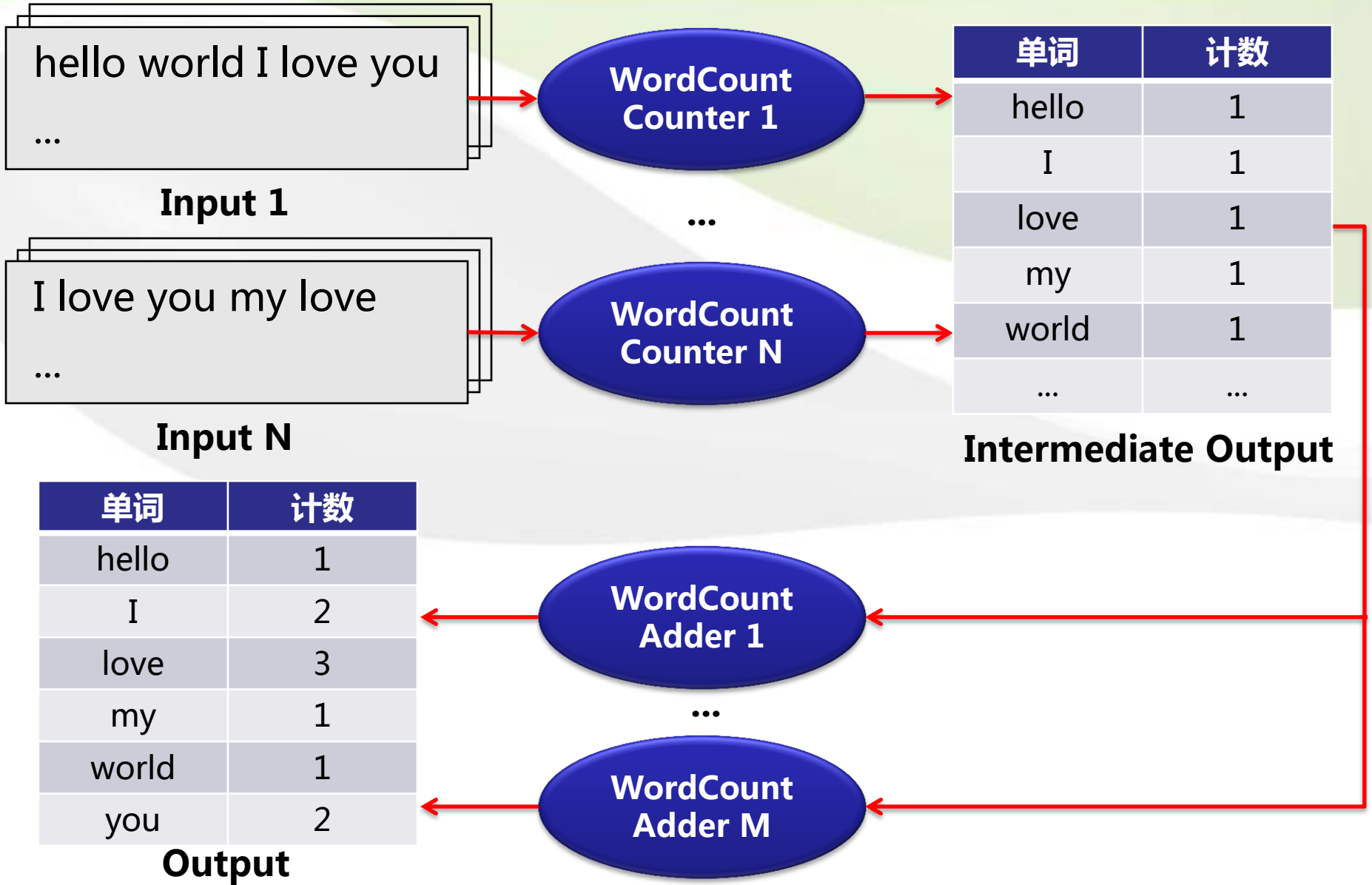
单词	计数
hello	1
I	2
love	3
my	1
world	1
you	2

Output

```
1:  define wordCount as Map
2:  for each doc in docSet
3:      words = tokenize(doc)
4:      for each word in words
5:        wordCount[word] ++
6:      end for
7:  end for
8:  output(wordCount)
```

- 串行代码的问题：
  - 海量数据
  - 运行效率

将WordCount拆分为两阶段



将WordCount拆分为两阶段

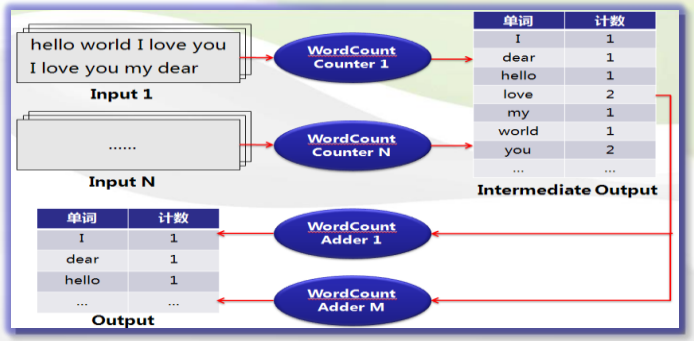
Counter

```
1:  define wordCount as Map
2:  for each doc in docSet
3:    words = tokenize(doc)
4:    for each word in words
5:      wordCount[word] ++
6:    end for
7:  end for
8:  send (wordCount, Adder)
```

Adder

```
1:  define totalCount as Map
2:  for each wordCount from Counter
3:    for each word in wordCount
4:      totalCount[word] += wordCount[word]
5:    end for
6:  end for
7:  output(totalCount)
```

并行WordCount

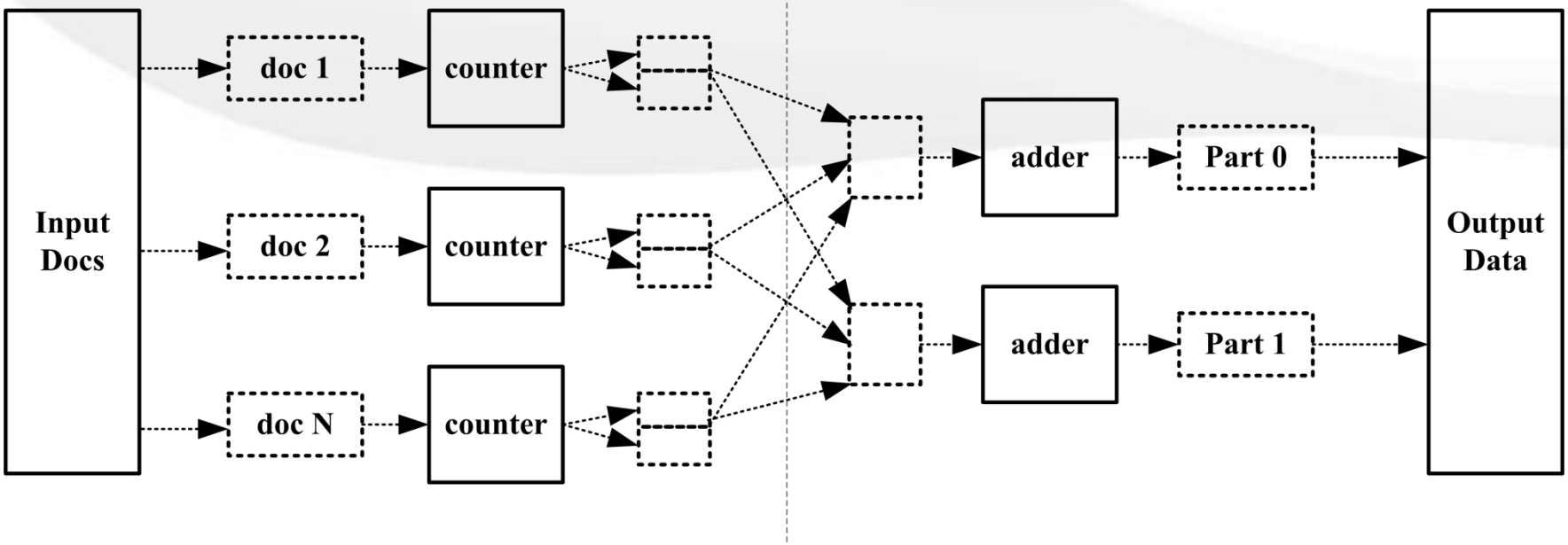


Counter

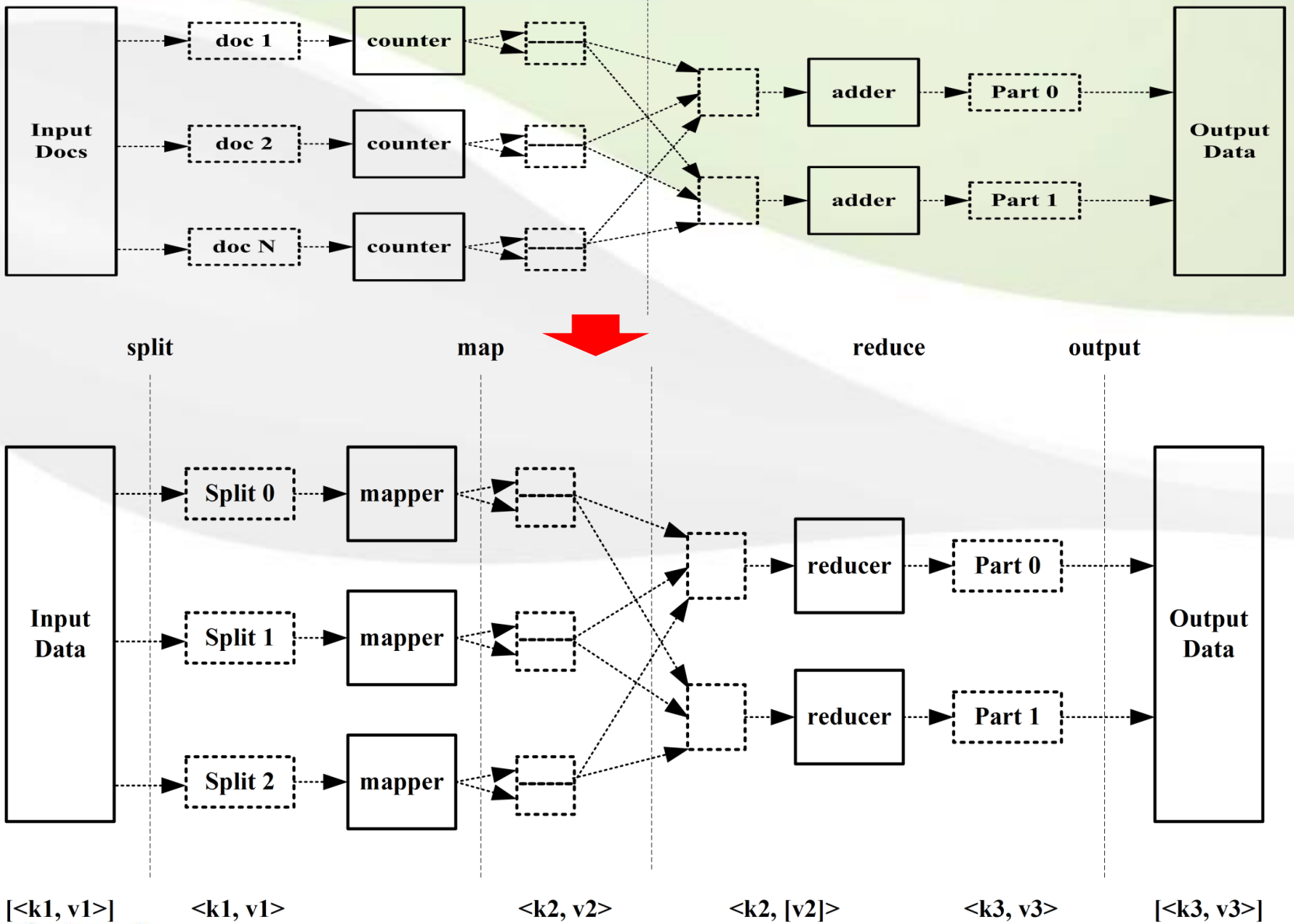
```
1: define wordCount as Map
2: for each doc in docSet
3:   words = tokenize(doc)
4:   for each word in words
5:     wordCount[word] ++
6:   end for
7: end for
8: sendToAdder(wordCount)
```

Adder

```
1: define totalCount as Map
2: for each wordCount from Counter
3:   for each word in wordCount
4:     totalCount[word] += wordCount[word]
5:   end for
6: end for
7: output(totalCount)
```



从并行WordCount到MapReduce



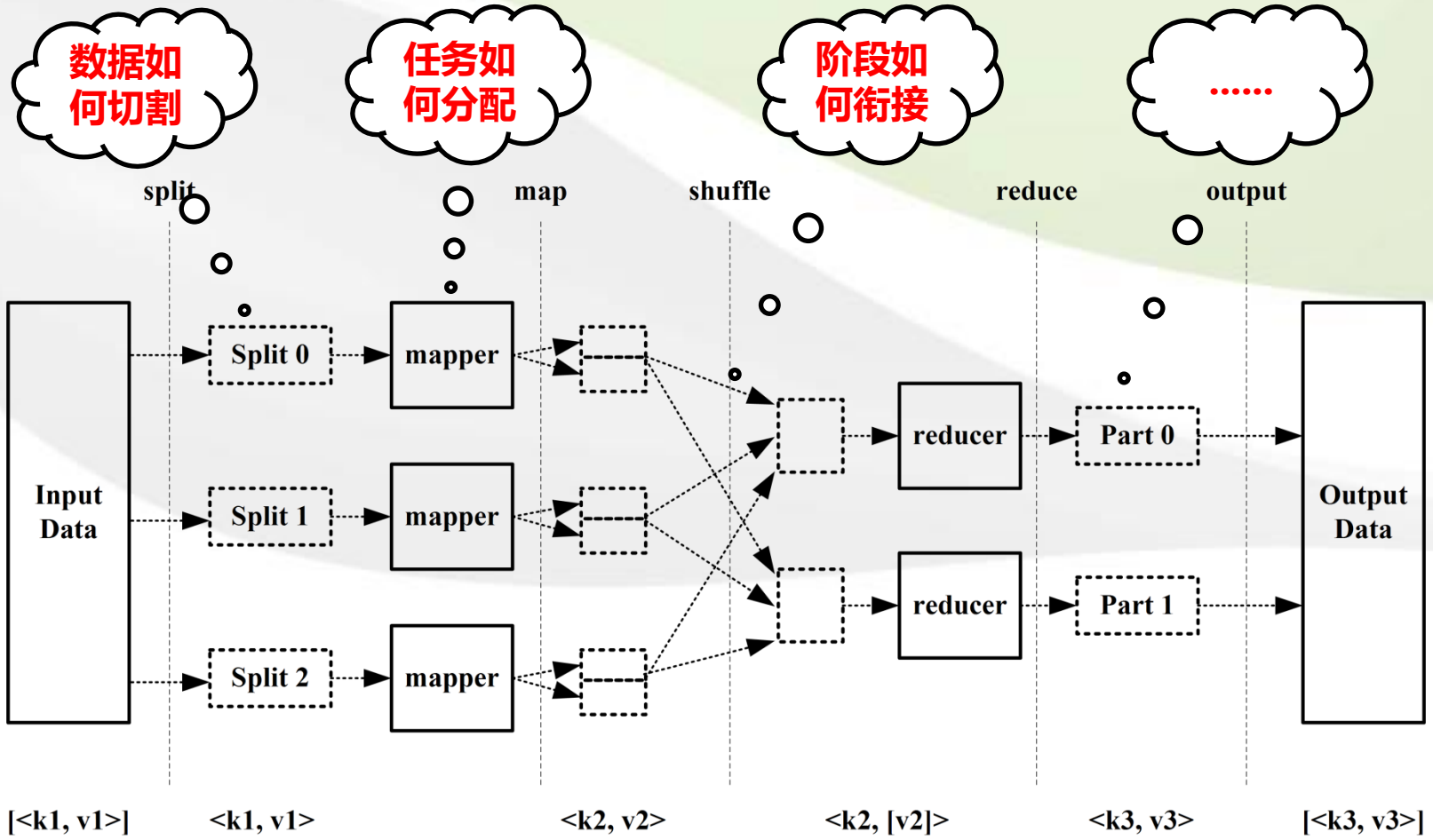
MapReduce版的WordCount

```
public static class TokenizerMapper extends Mapper
  <Object, Text, Text, IntWritable>{
  private final static IntWritable one =  new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context
    context) throws IOException, InterruptedException{
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
} } }
```

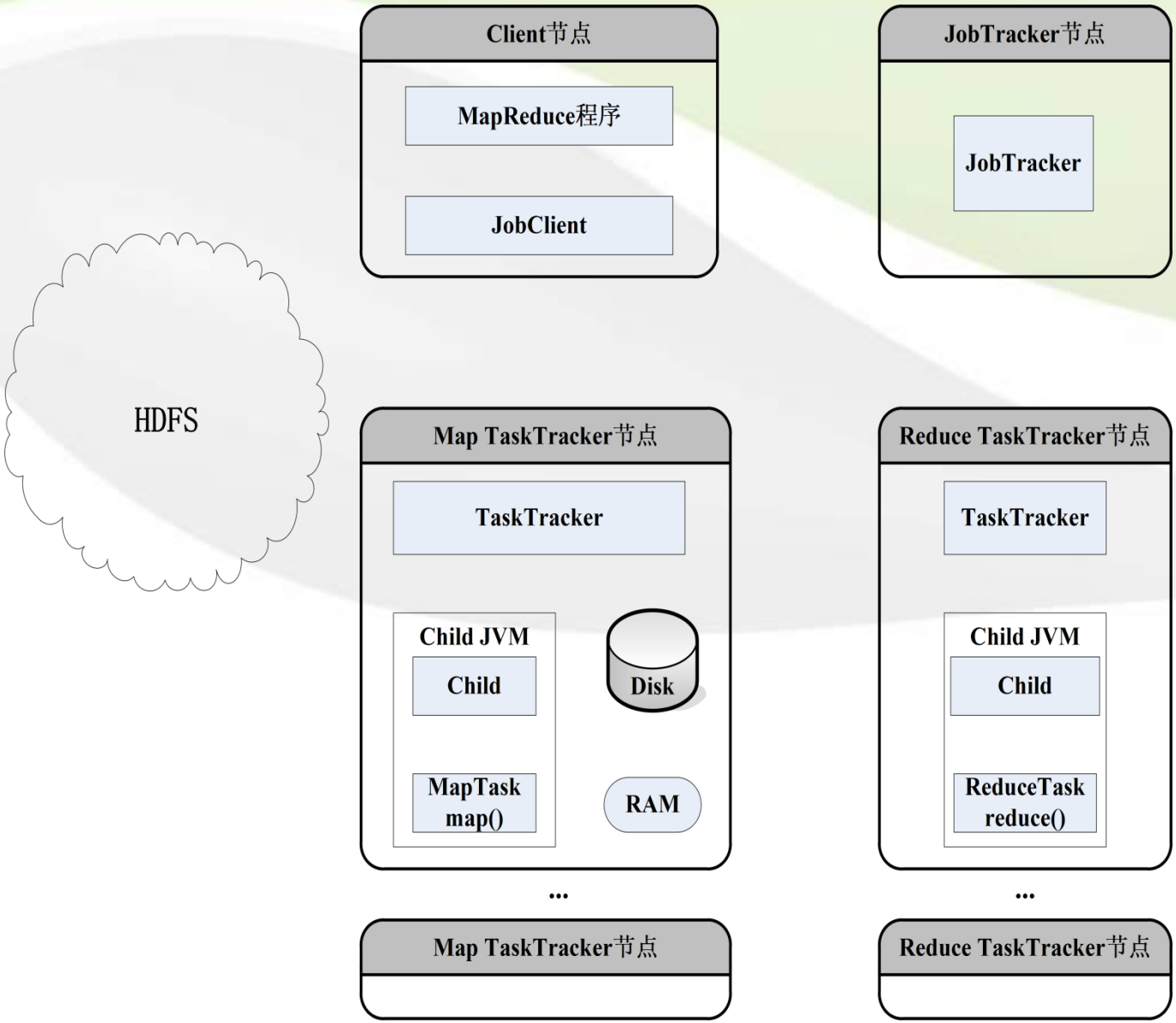
```
public static class IntSumReducer extends Reducer
  <Text,IntWritable,Text,IntWritable>{
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable>
    values,  Context context) throws IOException{
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();}
      result.set(sum);
      context.write(key, result);
} } }
```

```
public static void main(String[] args) throws Exception{
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(
    conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount <in> <out>");
    System.exit(2);
  }
  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job,
    new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job,
    new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

MapReuduce版WordCount中要解决的问题



解决之道 - MapReduce计算框架



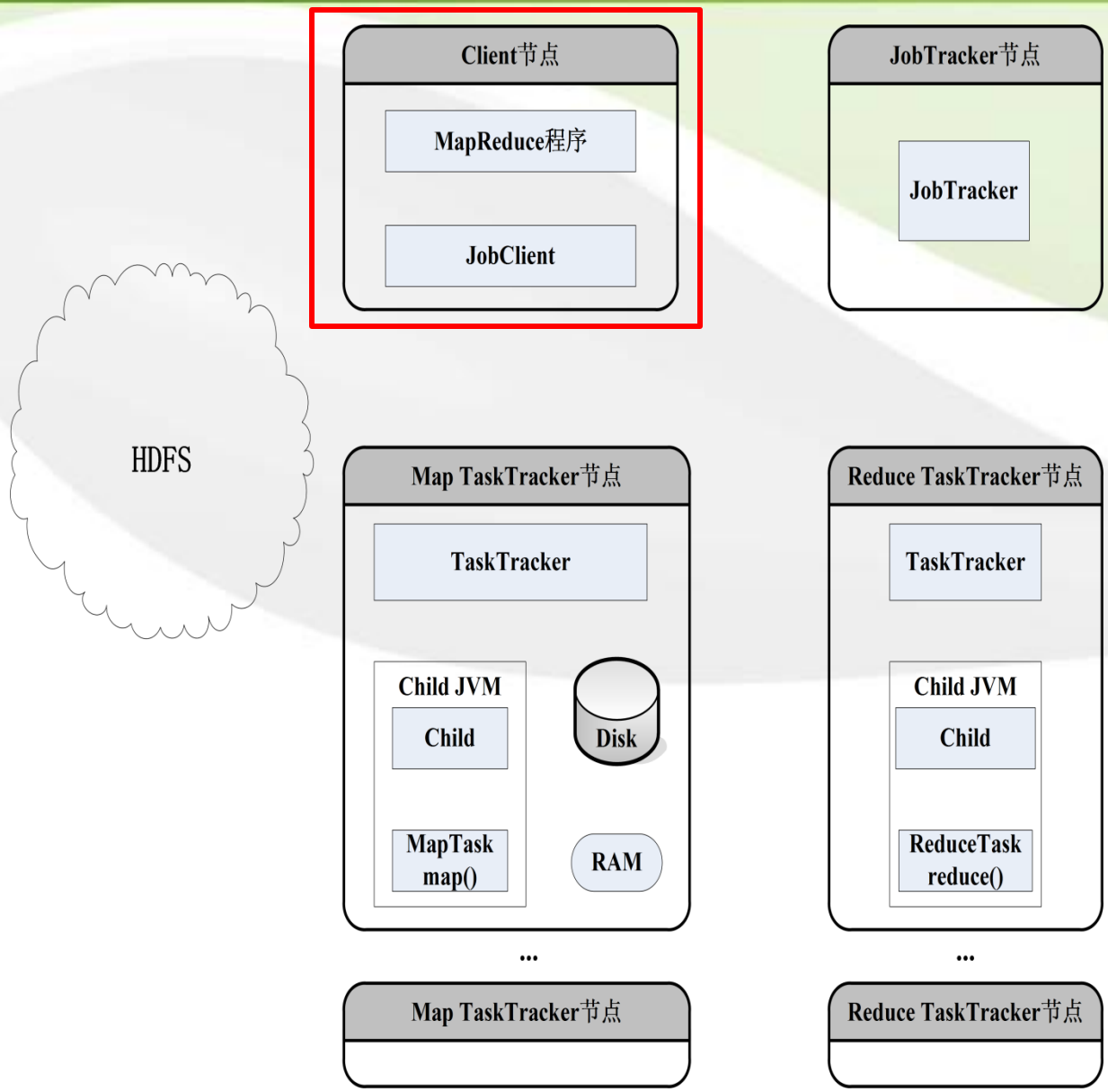
作业（Job）

- MapReduce程序指定的一个完整计算过程
- 一个作业在执行过程中可以被拆分为若干Map和Reduce任务完成

任务（Task）

- MapReduce框架中进行并行计算的基本事务单元
- 分为Map 和Reduce任务，一个作业通常包含多个任务

MapReduce框架中的角色



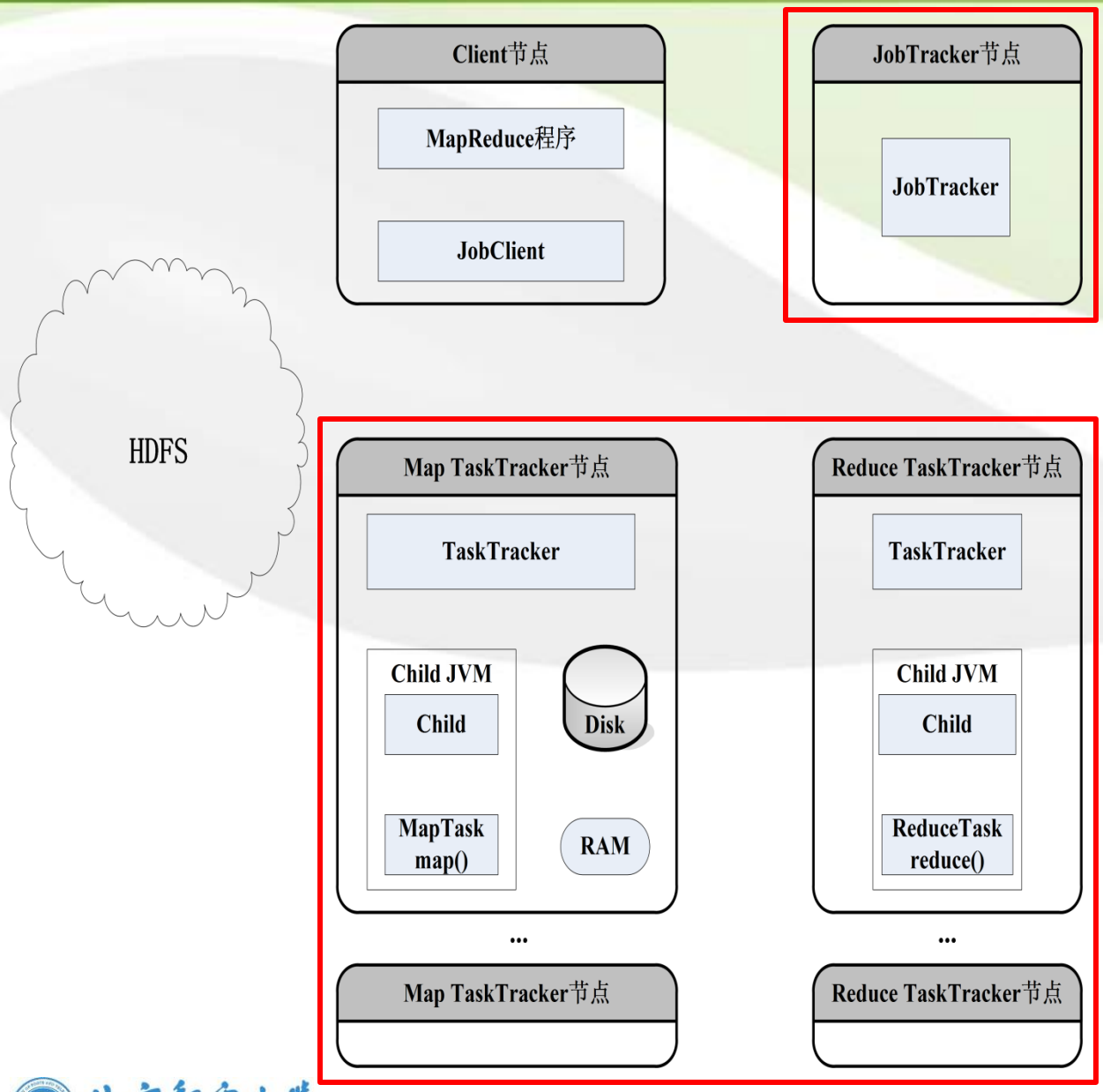
MapReduce程序

- 我们编写的程序

JobClient

- 替程序与MapReduce运行框架交互的对象

MapReduce框架中的角色



JobTracker

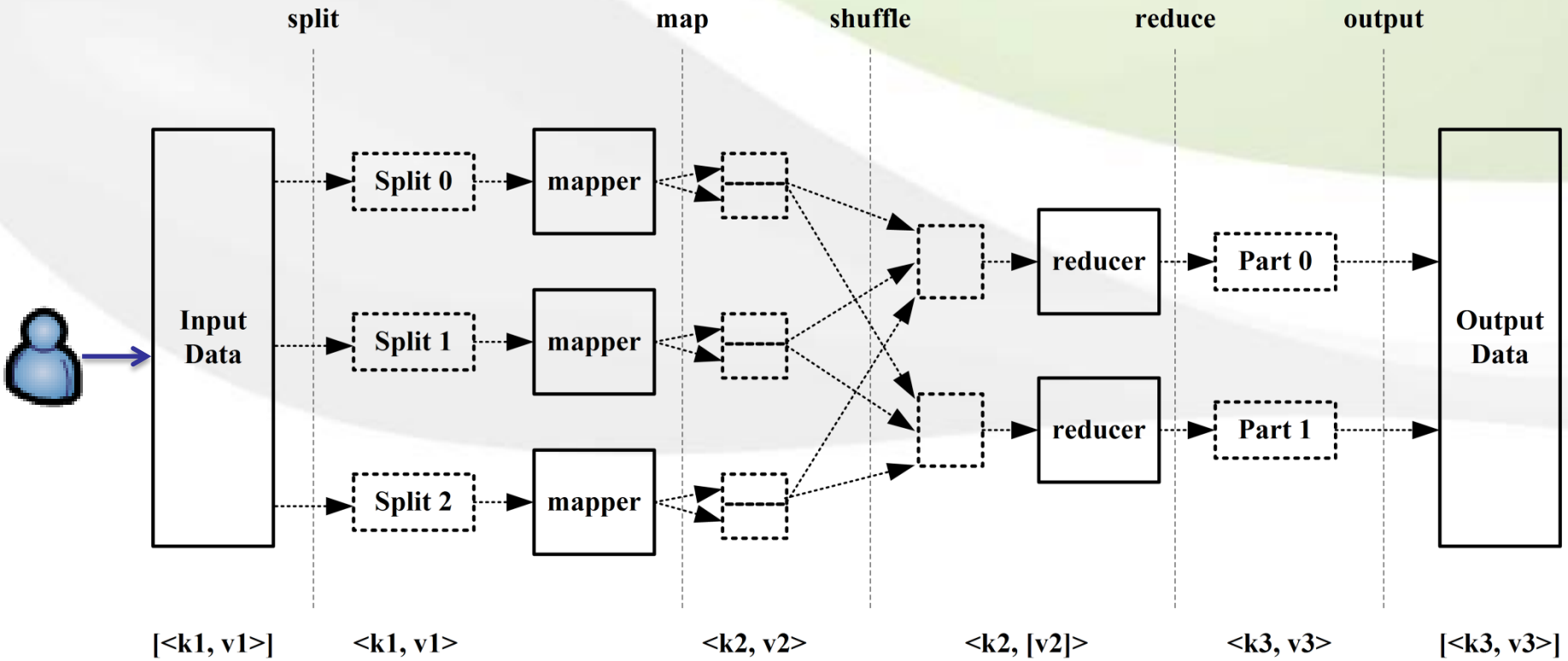
- MapReduce框架的管理者
- 协调MapReduce作业
- 分配任务
- 监控任务

TaskTracker

- 执行JobTracker分配的任务
- 分为Map和Reduce两类

MapReduce运行流程

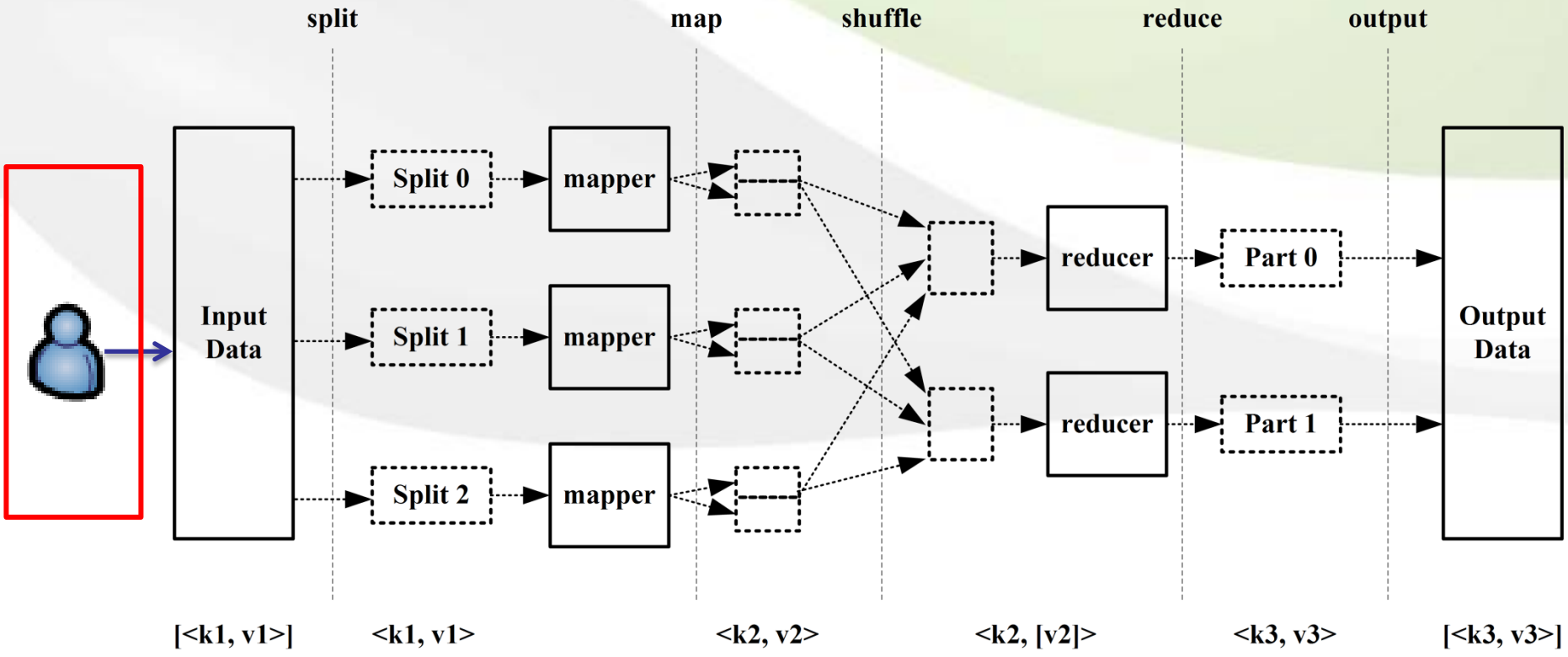
① 作业启动　③ 作业/任务调度　⑤ shuffle　⑦ 作业完成



② 作业初始化　④ Map任务执行　⑥ Reduce任务执行

MapReduce运行流程（1） - 作业启动

① 作业启动



作业启动

./hadoop jar ./../hadoop-examples-1.0.3.jar wordcount data_in data_out

```
JAVA=$JAVA_HOME/bin/java
JAVA_HEAP_MAX=-Xmx1000m
# check envvars which might override default args
if [ "$HADOOP_HEAPSIZE" != "" ]; then
  #echo "run with heapsize $HADOOP_HEAPSIZE"
  JAVA_HEAP_MAX="-Xmx""$HADOOP_HEAPSIZE""m"
  #echo $JAVA_HEAP_MAX
fi
```

.....

```
elif [ "$COMMAND" = "jar" ] ; then
  CLASS=org.apache.hadoop.util.RunJar
  HADOOP_OPTS="$HADOOP_OPTS $HADOOP_CLIENT_OPTS"
```

.....

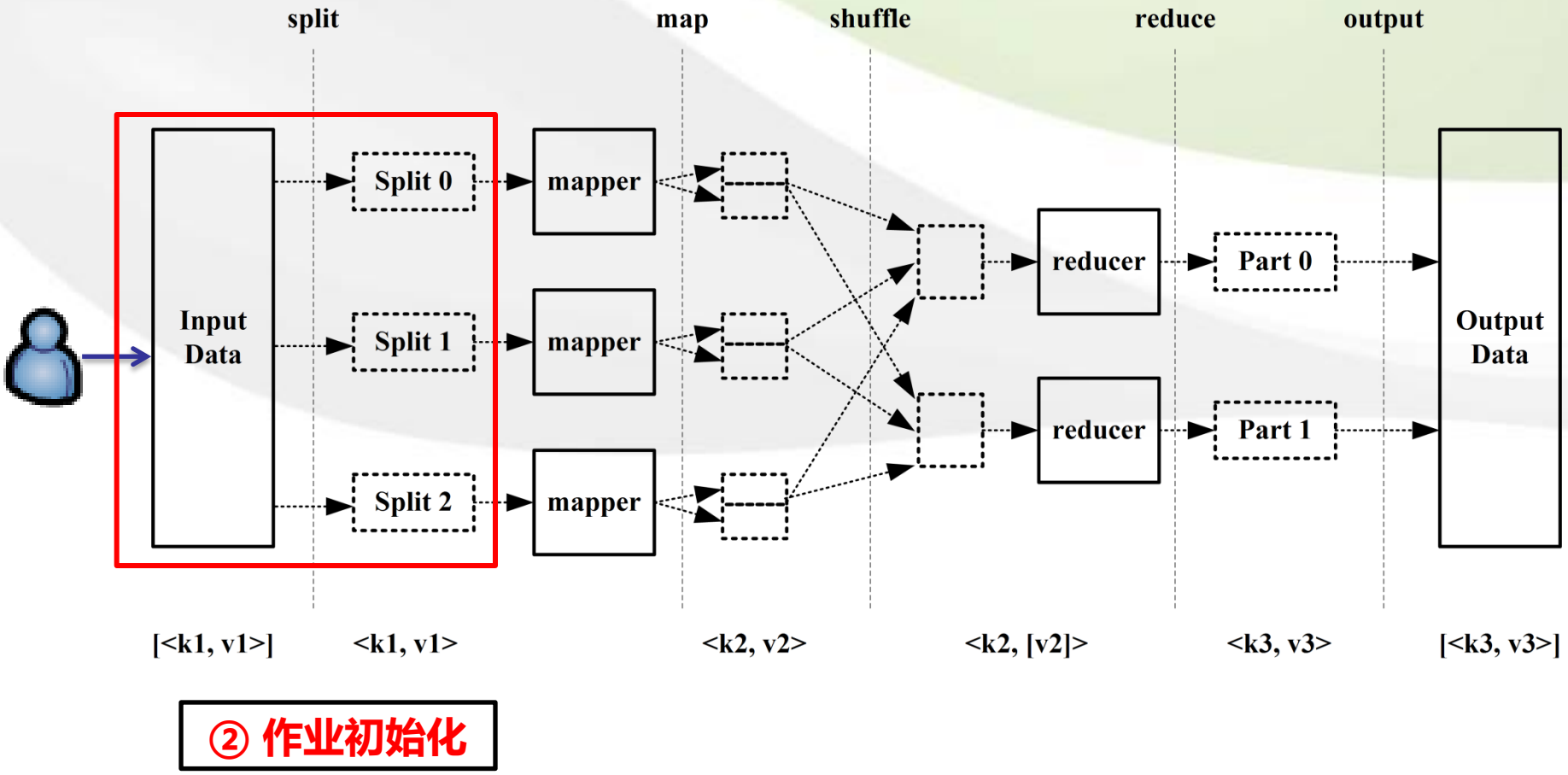
```
# run it
exec "$JAVA" -Dproc_$COMMAND $JAVA_HEAP_MAX $HADOOP_OPTS
    -classpath "$CLASSPATH" $CLASS "$@"
```

作业提交

```
public static void main(String[] args) throws Exception{
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(
    conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount <in> <out>");
    System.exit(2);
  }
  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job,
    new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job,
    new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

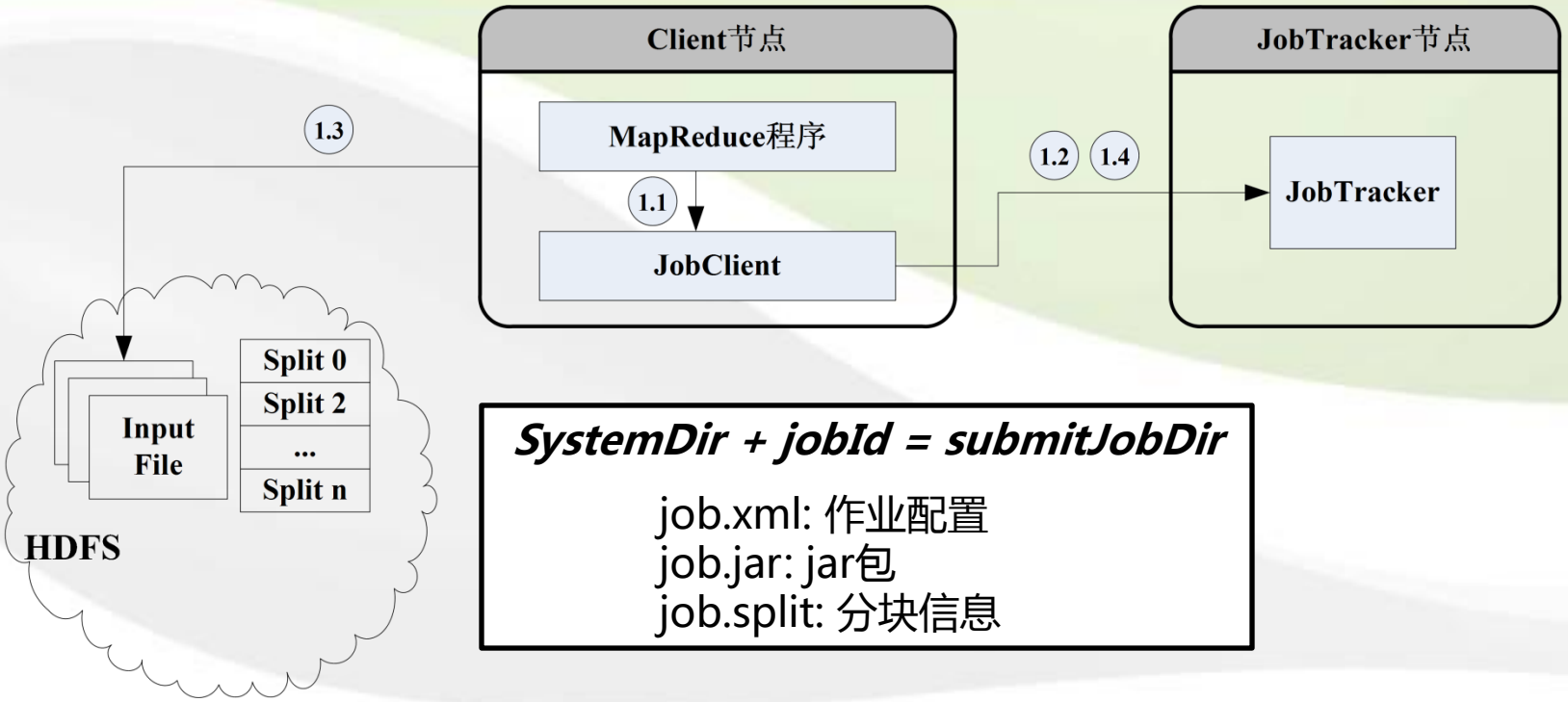
- job.waitForCompletion(true)
  - 调用JobClient的submit()
  - true代表打印作业信息
- submit时发生了什么？

MapReduce运行流程（2） - 作业初始化



② 作业初始化

作业初始化



1.1 MapReduce程序创建新的JobClient实例

1.2 JobClient向JobTracker请求获得一个新的JobId标识本次作业

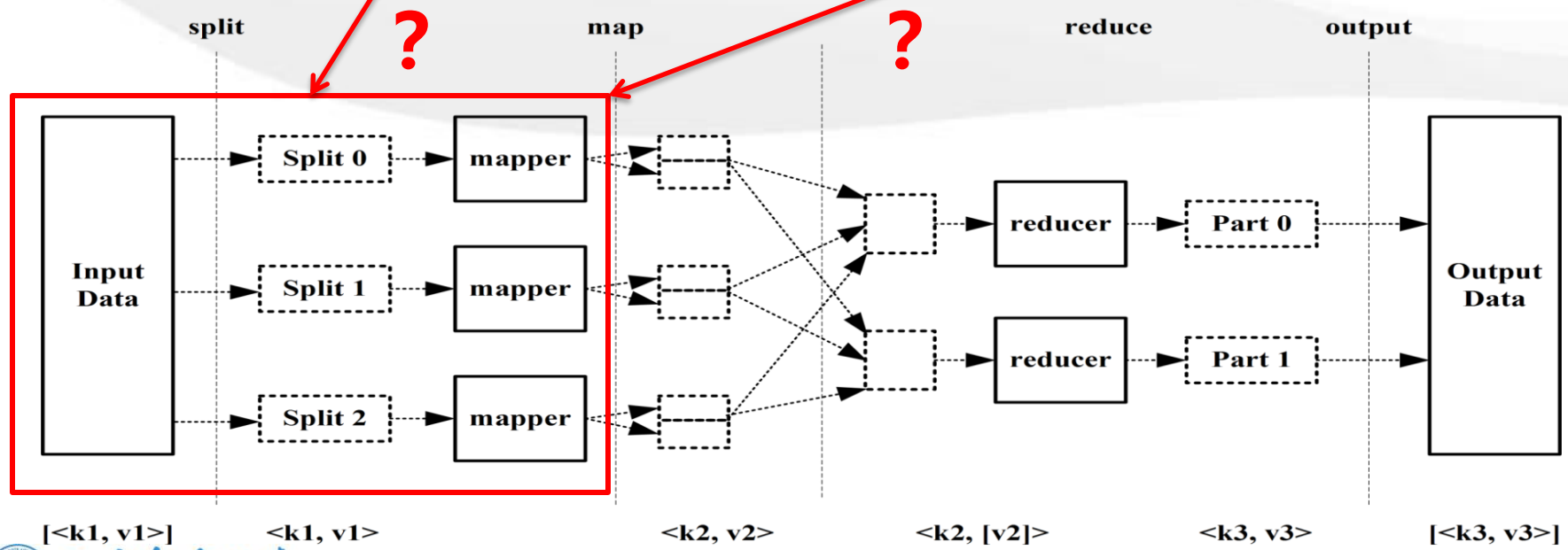
1.3 JobClient将运行作业需要的相关资源放入作业对应的HDFS目录、计算分片数量和map任务数量

1.4 向JobTracker提交作业，并获得作业的状态对象句柄

作业初始化 - 关于split和Map

```
public static class TokenizerMapper extends Mapper
  <Object, Text, Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context
    context) throws IOException, InterruptedException{
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
}}}
```

```
public static void main(String[] args) throws
Exception{
  ...
  job.setMapperClass(TokenizerMapper.class);
  ...
  FileInputFormat.addInputPath(job,
    new Path(otherArgs[0]));
  ...
}
```

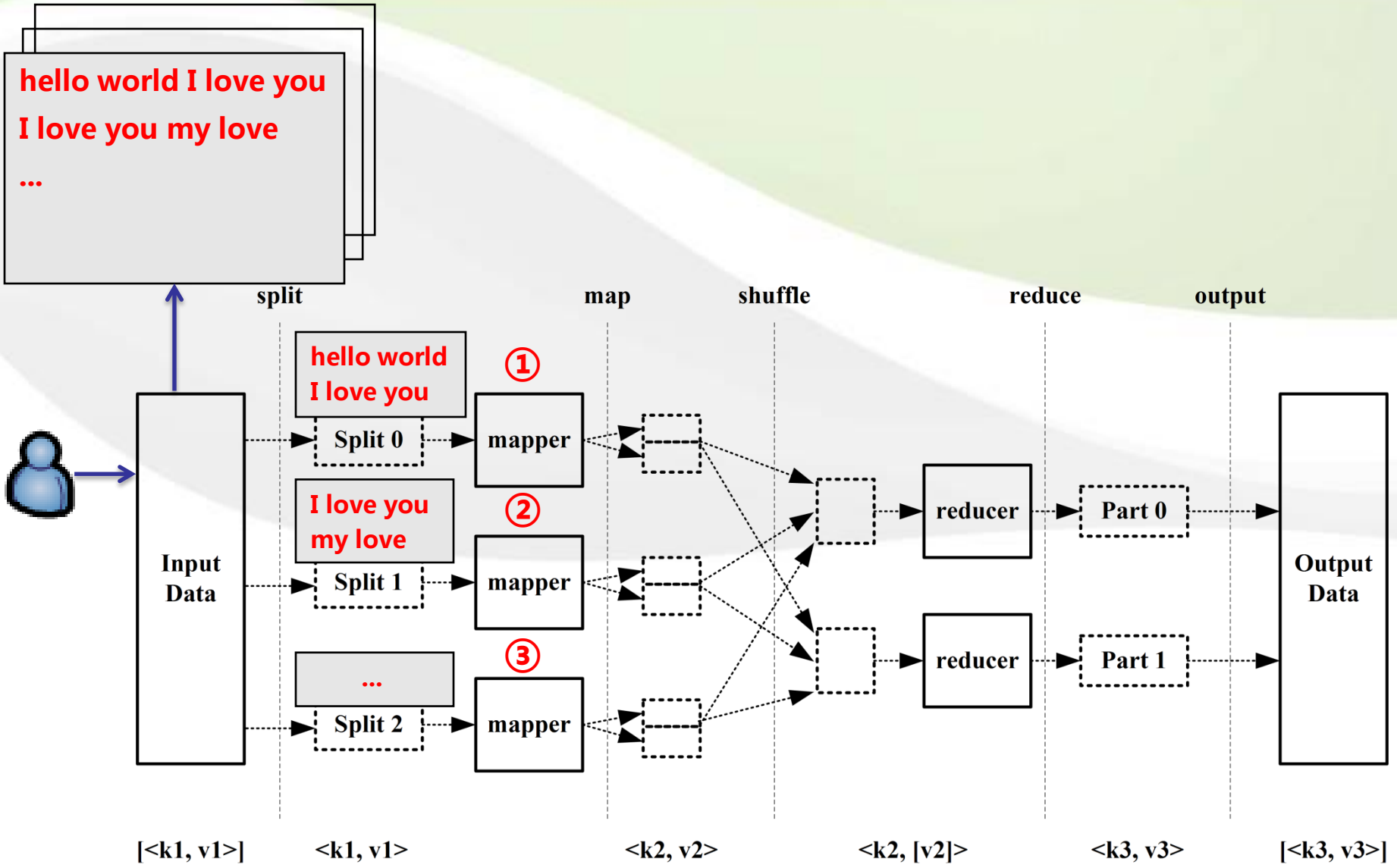


作业初始化 - 计算Split和Map数

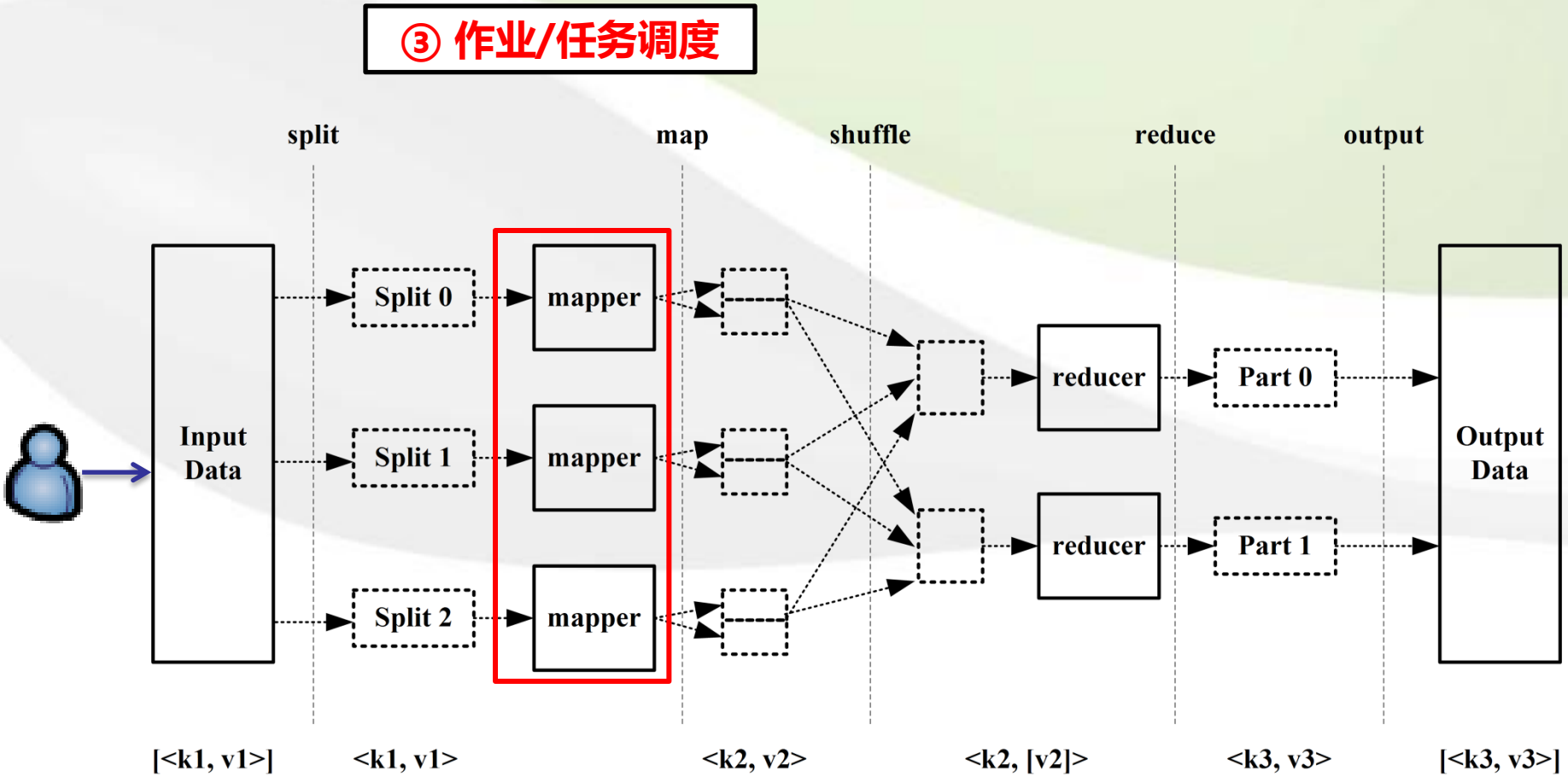
- FileInputFormat.getSplits　　Map数=Split数
  ① 期望的Map数（参数mapred.map.tasks，默认为0）
  ② 每个split的最小值minSize（参数mapred.min.split.size，默认为0）
  ③ goalsize = input文件的总字节数/(mapred.map.tasks==0？1: mapred.map.tasks)
  ④ splitsize= Math.max(minSize, Math.min(goalSize, blockSize))，通常=blockSize，如输入的文件较小，文件字节数之和小于blocksize时，splitsize=输入文件字节数之和
  ⑤ 对于input的每个文件，按以下过程计算split的个数
    a. 文件剩余字节数/splitsize>1.1，则创建一个split（字节数=splitsize），文件剩余字节数=文件大小-splitsize
    b. 文件剩余字节数/splitsize<1.1，剩余的部分作为一个split，否则重复a、b

- 示例：（Hadoop 默认设置）
  - 一个文件，100M，则split数为2，第一个split为64M，第二个为36M
  - 一个文件，129M， 则split数为2，第一个split为64M，第二个为65M
  - 两个文件，100M和20M， 则split数为3，第一个文件分为两个split，第一个split为64M，第二个为36M，第二个文件为一个split，大小为20M
  - **1个100M文件和10个10M的文件时，Map数为？如何增大前者Map数？**

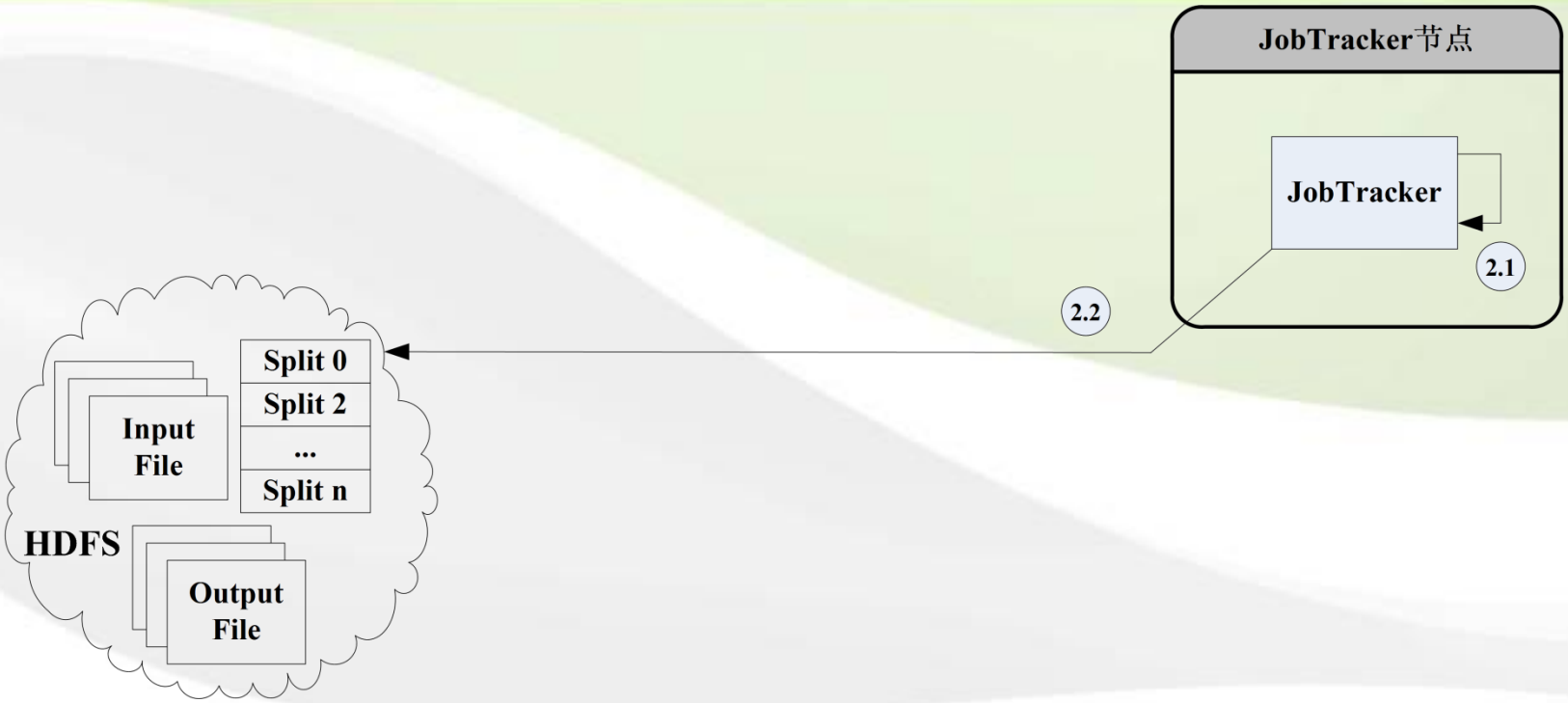
作业初始化完成后


hello world I love you
I love you my love
...
split
map
shuffle
reduce
output
hello world
I love you
Split 0
mapper
①
I love you
my love
Split 1
mapper
②
...
Split 2
mapper
③
Input
Data
reducer
reducer
Part 0
Part 1
Output
Data
[<k1, v1>]
<k1, v1>
<k2, v2>
<k2, [v2]>
<k3, v3>
[<k3, v3>]

MapReduce运行流程（3） - 作业/任务调度



作业调度



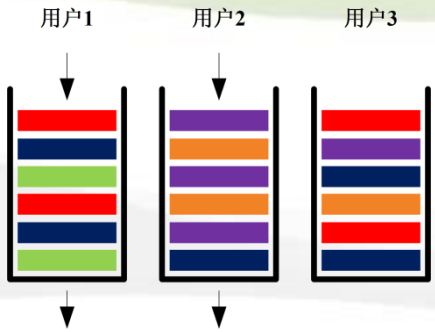
2.1 作业提交请求放入队列等待调度

2.2 从HDFS中取出作业分片信息，创建对应数量的TaskInProgress调度和监控Map任务

作业调度

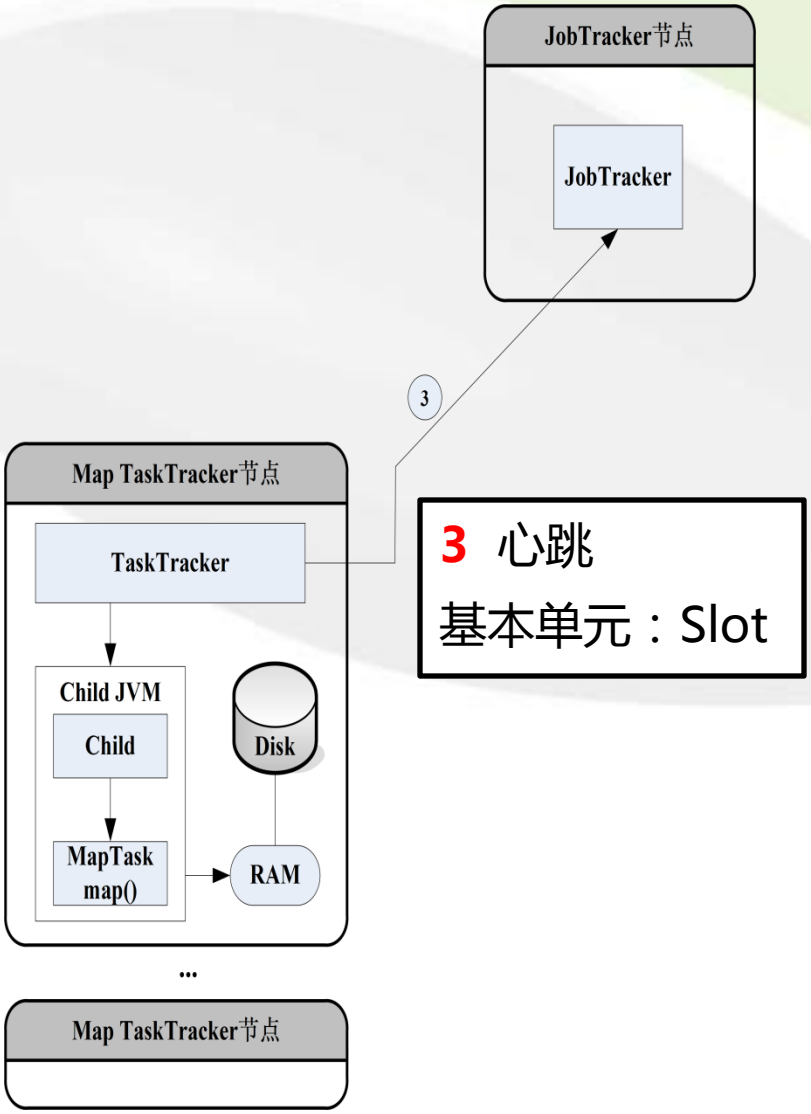
- 先进先出（FIFO）调度器（默认）
  - VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW五个等级
  - 先按优先级高低、再按作业到达时间，选择下一个被执行的作业
  - 不支持抢占

- 公平（Fair）调度器
  - 所有用户公平地共享计算能力，不被独占
  - 每个用户一个资源池，计算资源按照用户设定被公平地分配到这些资源池中
  - 每个用户可以提交多个作业，会按照公平共享的方式分享该用户占有的资源池
  - 无作业的资源也可以被其他用户共享
  - 支持资源抢占



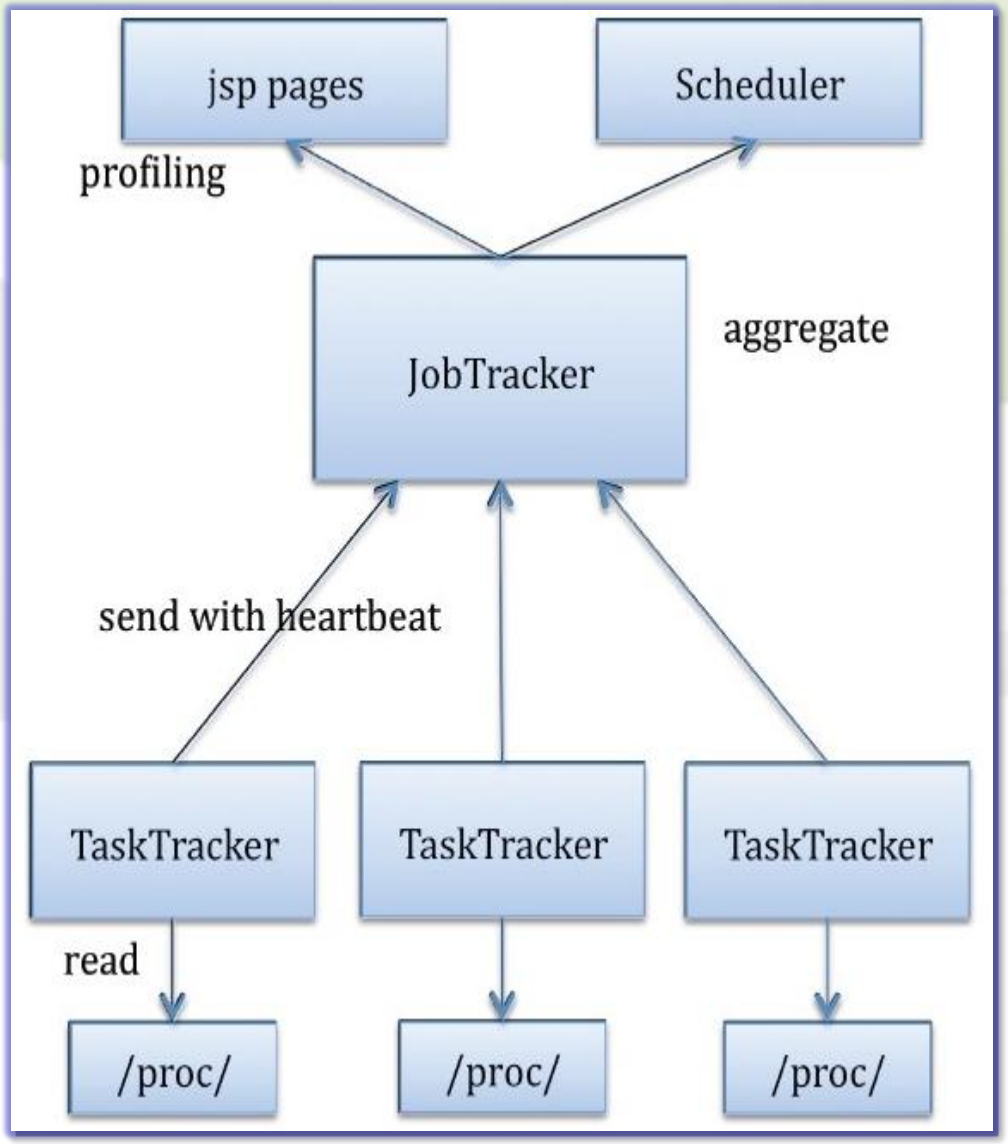
- 能力（Capacity）调度器
  - 采用分层次多队列的形式组织计算资源
  - 队列内采用支持优先级的先进先出调度方式
  - 计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个比值最小的队列，然后按FIFO从队列中选择一个作业执行
  - 对同一用户提交的作业所占资源总量进行限制

任务分配



3 心跳

基本单元：Slot

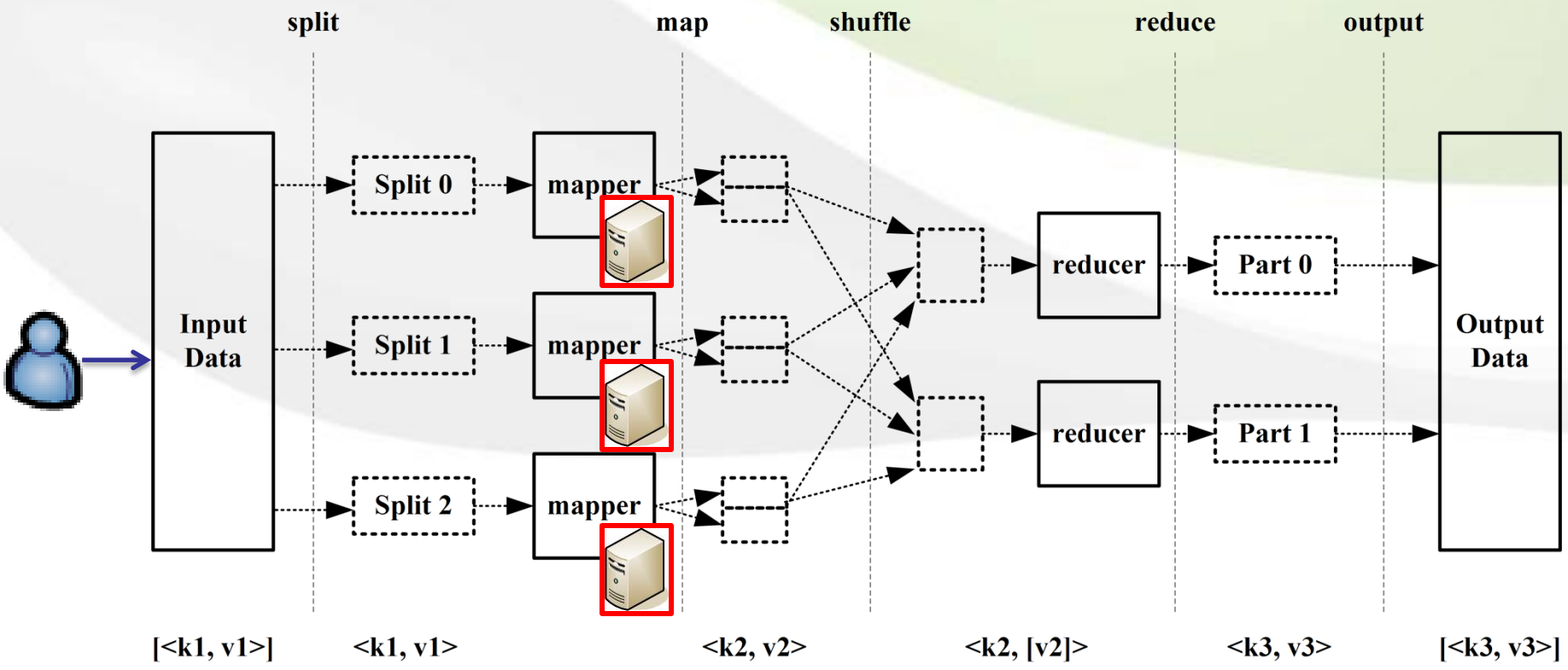


Map数量 - 影响MapReduce性能的重要因素

- split数量决定Map任务数量
- 节点CPU/内存能力决定Map Slot数量
- split大小决定Map执行时间和传输效率

- Map任务数 + Map Slot + 节点数量 = Map阶段并行程度

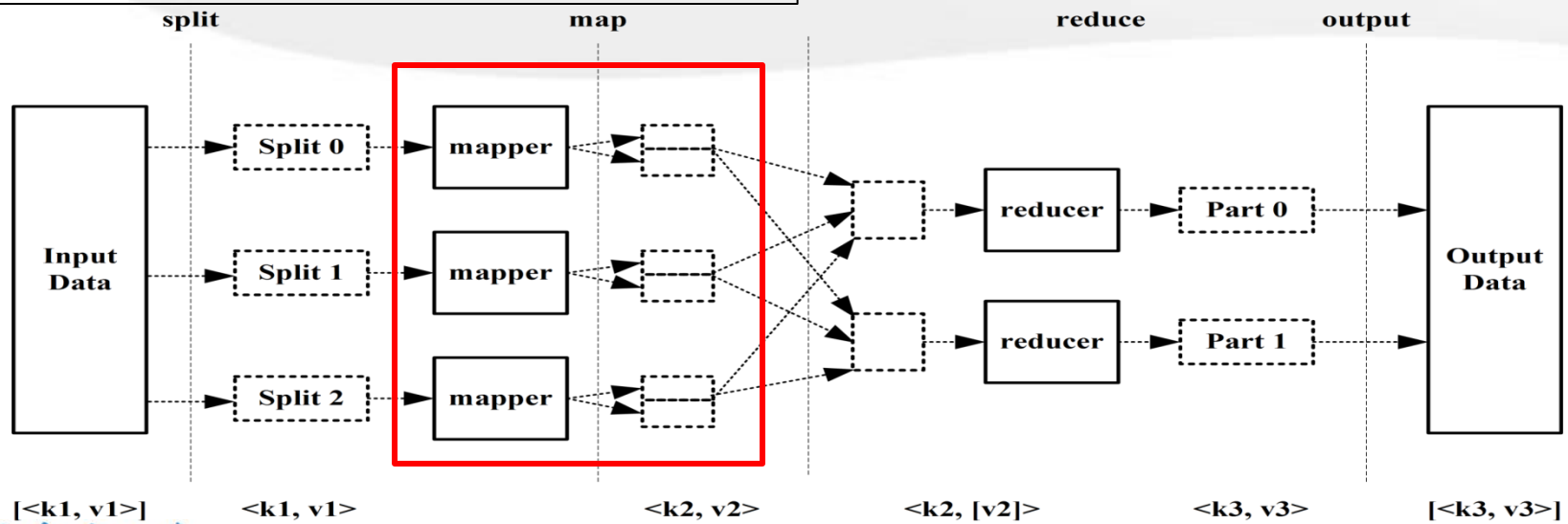
任务分配完成后


split
map
shuffle
reduce
output
Input Data
Split 0
Split 1
Split 2
mapper
mapper
mapper
reducer
reducer
Part 0
Part 1
Output Data
[<k1, v1>]
<k1, v1>
<k2, v2>
<k2, [v2]>
<k3, v3>
[<k3, v3>]

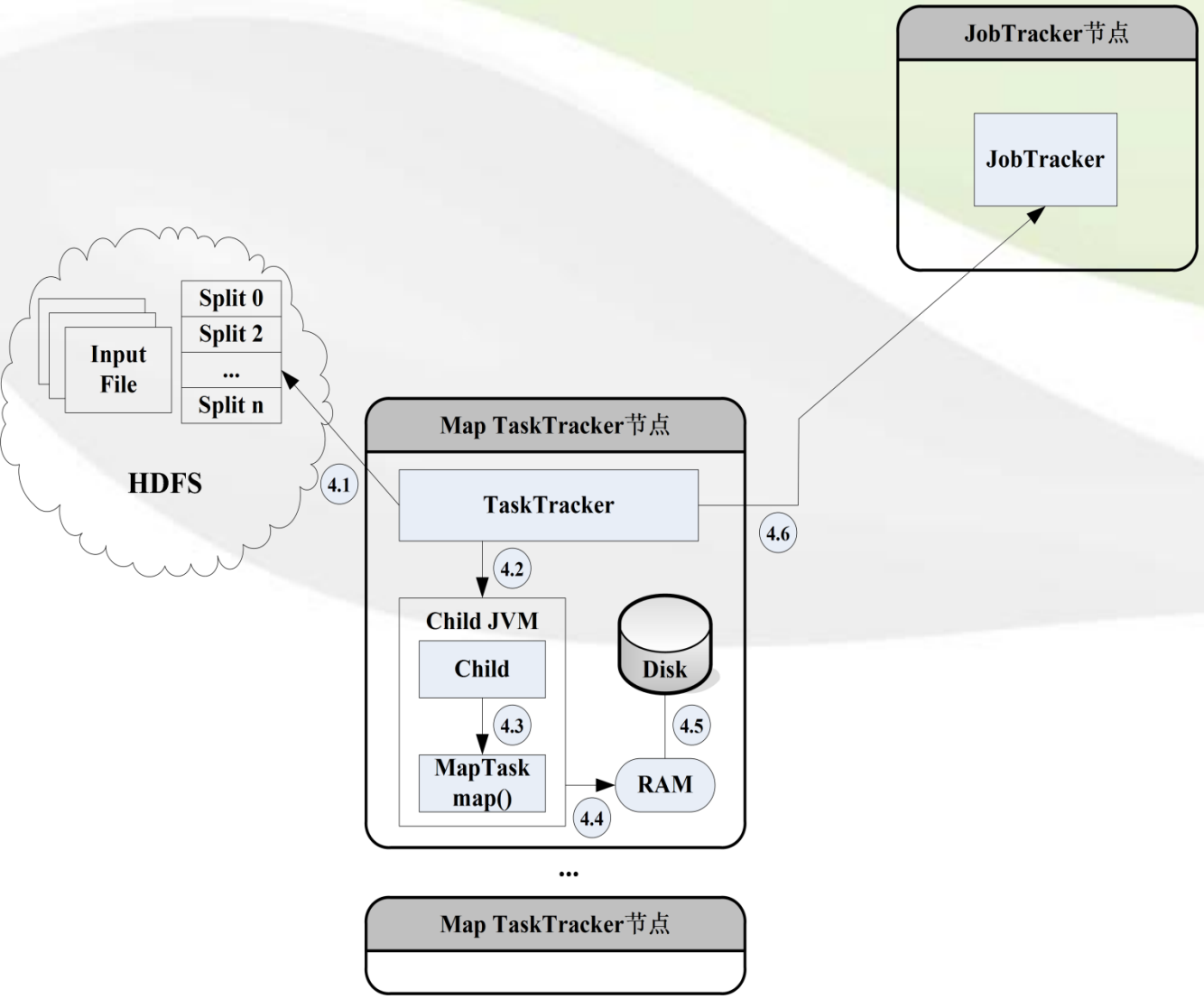
MapReduce运行流程（4） - Map任务执行

```
public static class TokenizerMapper extends Mapper
  <Object, Text, Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(Object key, Text value, Context
    context) throws IOException, InterruptedException{
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
} } }
```

④ Map任务执行



Map任务执行



4.1 从HDFS提取相关资源（Jar包、数据）

4.2 创建TaskRunner运行Map任务

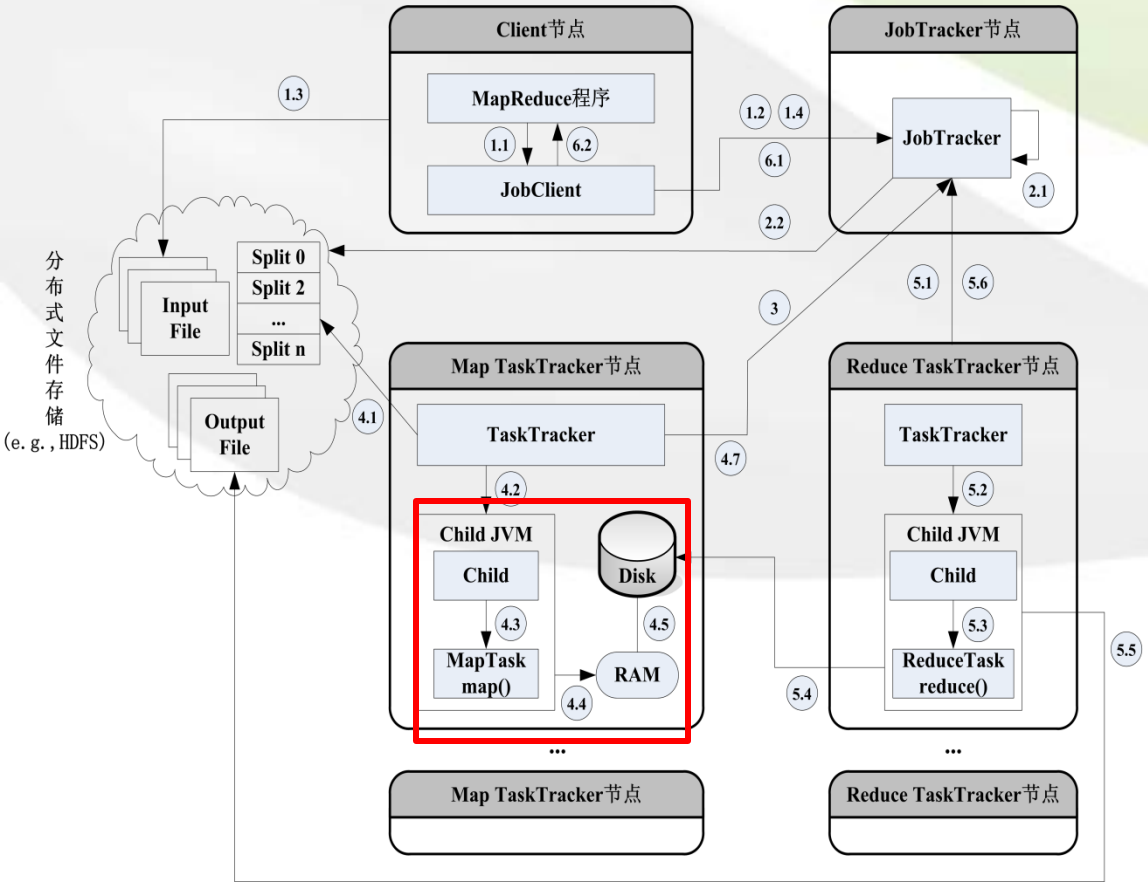
4.3 在单独的JVM中启动MapTask执行map函数

4.4 中间结果数据定期存入缓存

4.5 缓存写入磁盘

4.6 定期报告进度

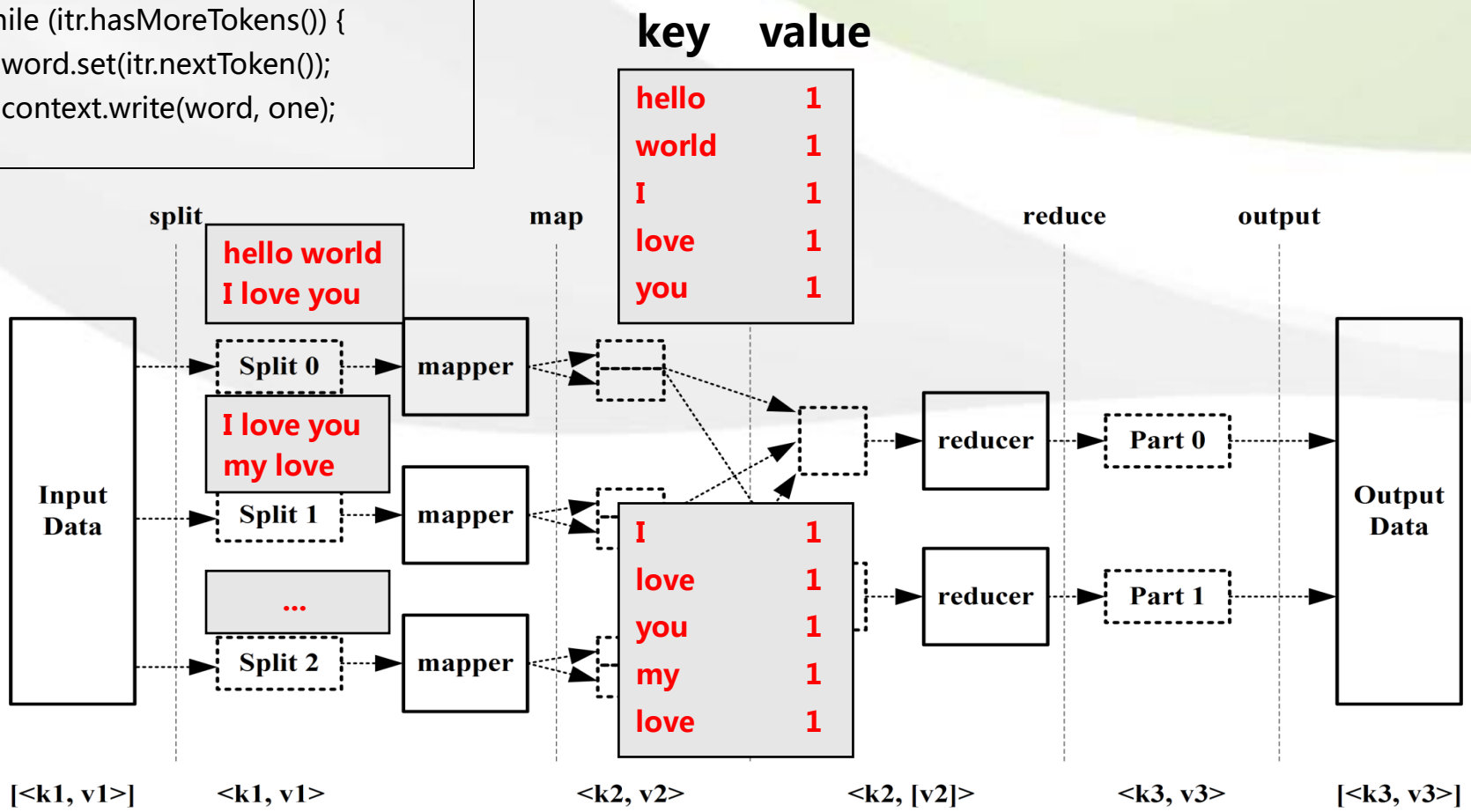
计算与数据靠近



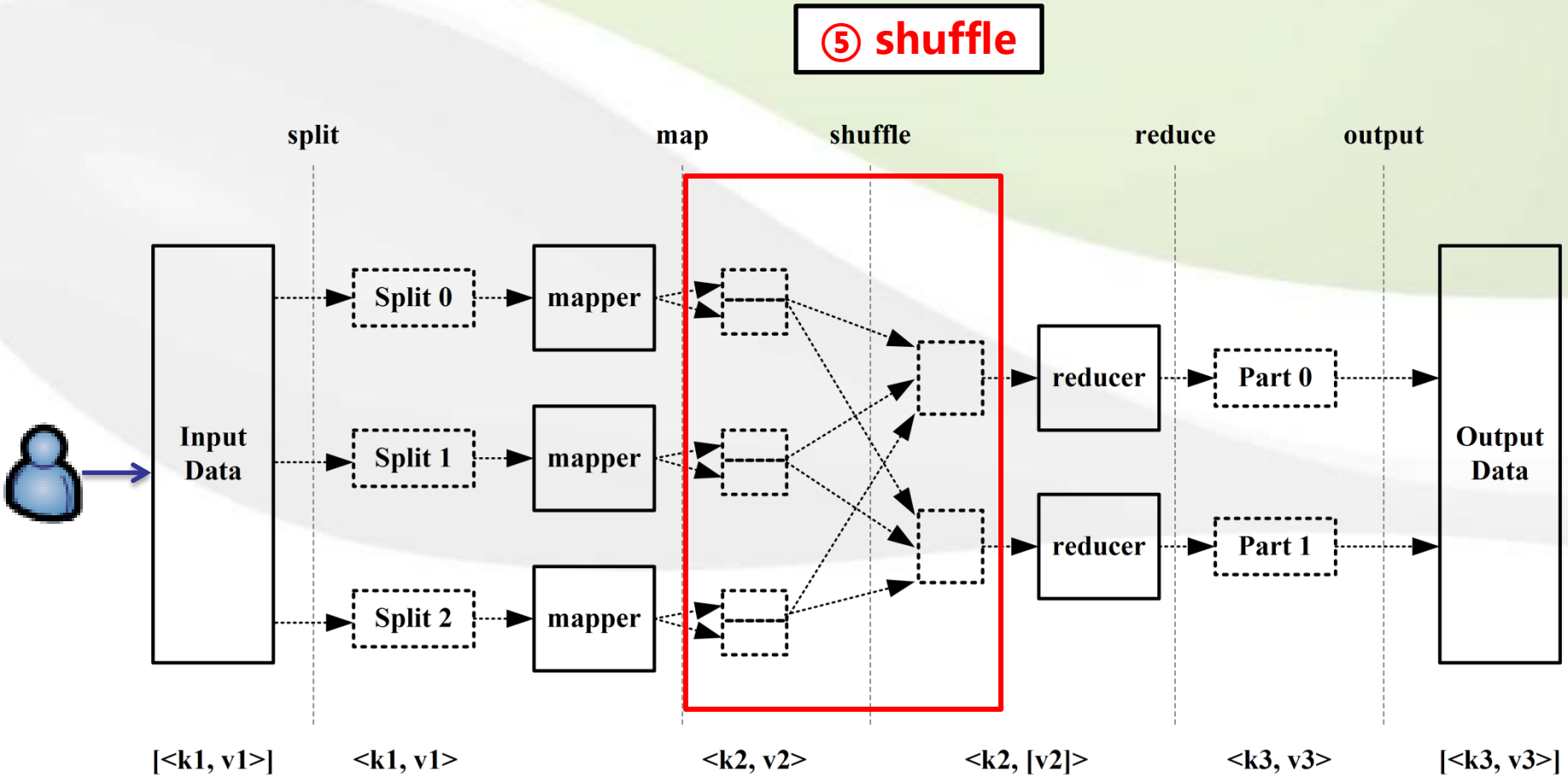
- 大规模数据处理时外存文件数据I/O访问会成为一个制约系统性能的瓶颈
- **代码靠近数据**
  - 原则：本地化数据处理（locality），即一个计算节点尽可能处理其本地磁盘上所存储的数据
  - 尽量选择数据所在DN启动Map任务
  - 减少数据通信，提高计算效率
- **数据靠近代码**
  - 当本地没有数据处理时，尽可能从同一机架的其他节点传输数据进行处理

Map任务执行完成后

```
public void map(Object key, Text value,
Context context) throws IOException,
    InterruptedException{
    StringTokenizer itr = new
        StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
} }
```

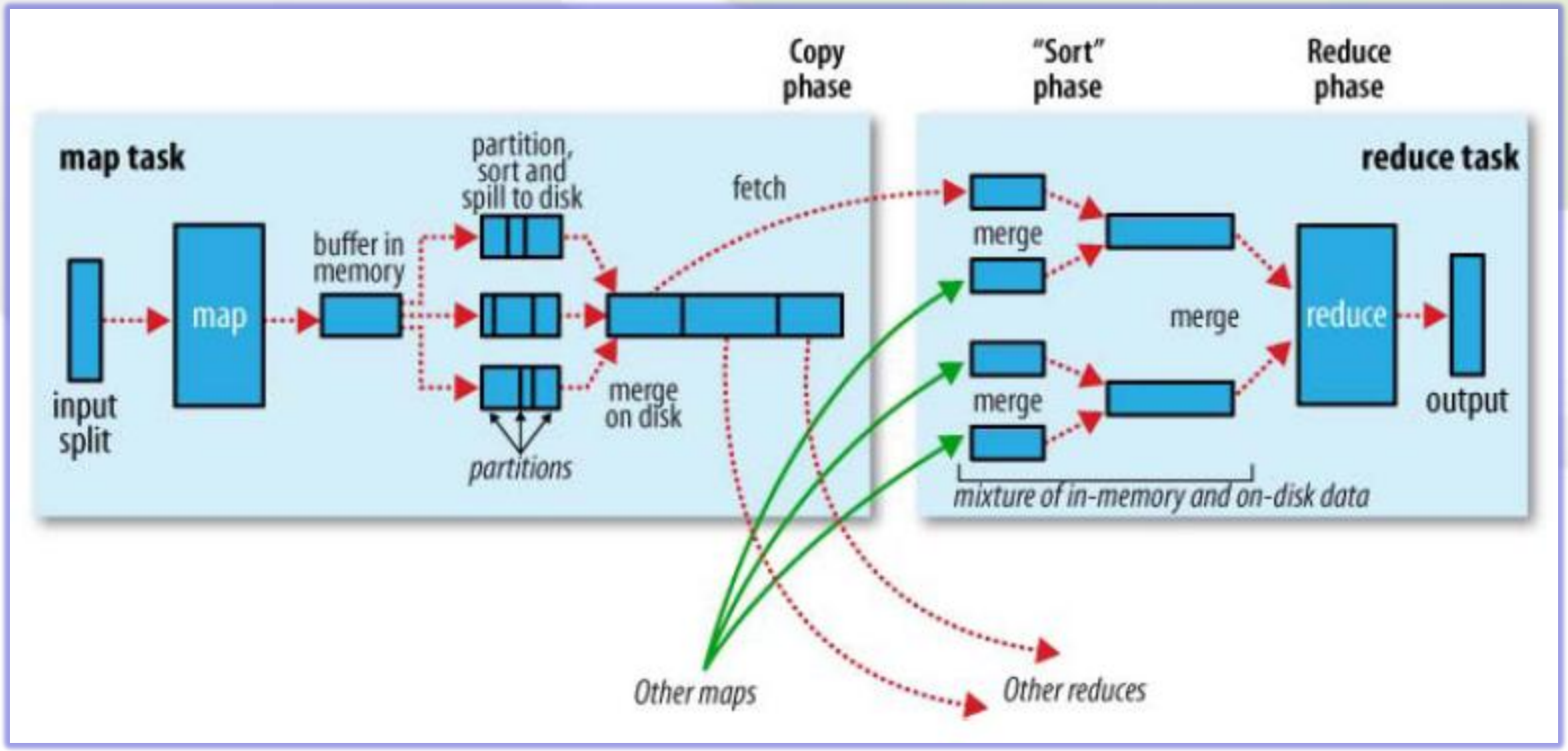


MapReduce运行流程（5） - shuffle

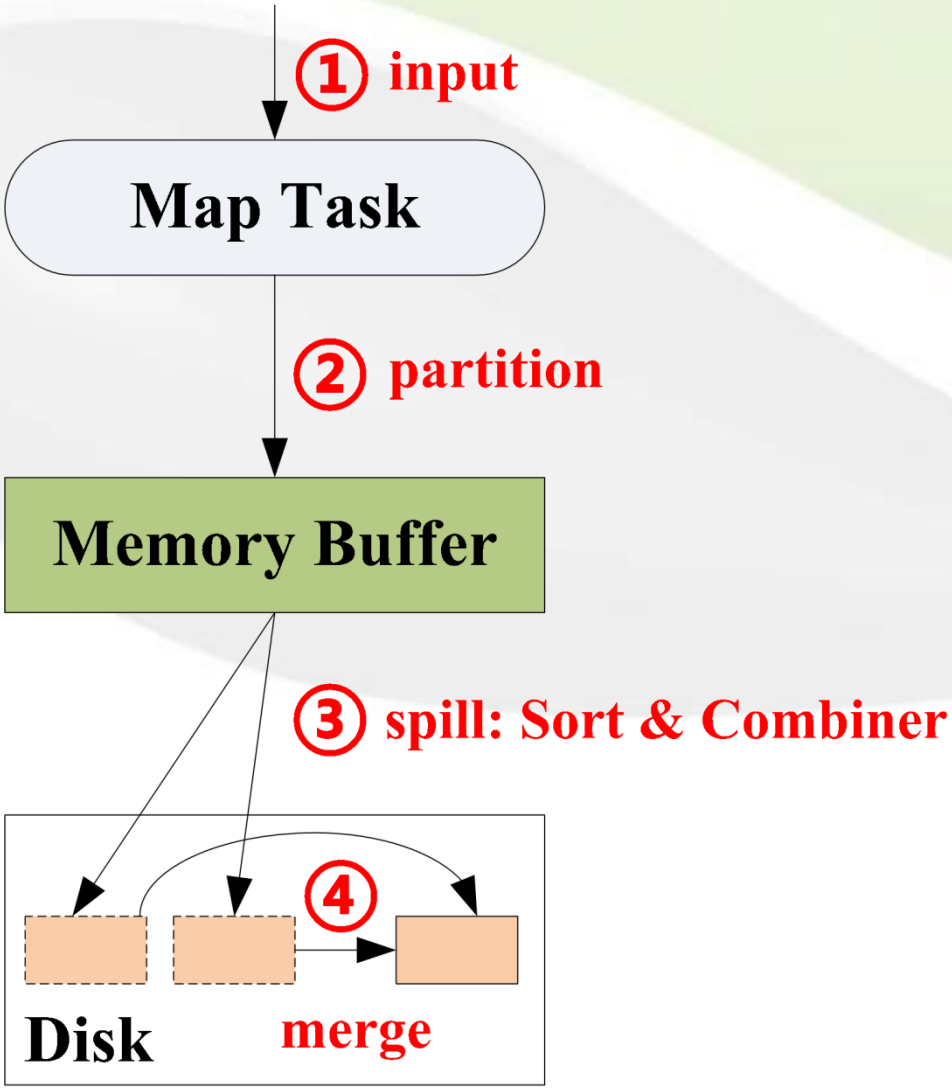


shuffle - 奇迹发生的地方

- Map与Reduce之间的神秘Shuffle



Map端结果保存



① 处理输入数据

② 寻找对应Reduce（Partition）

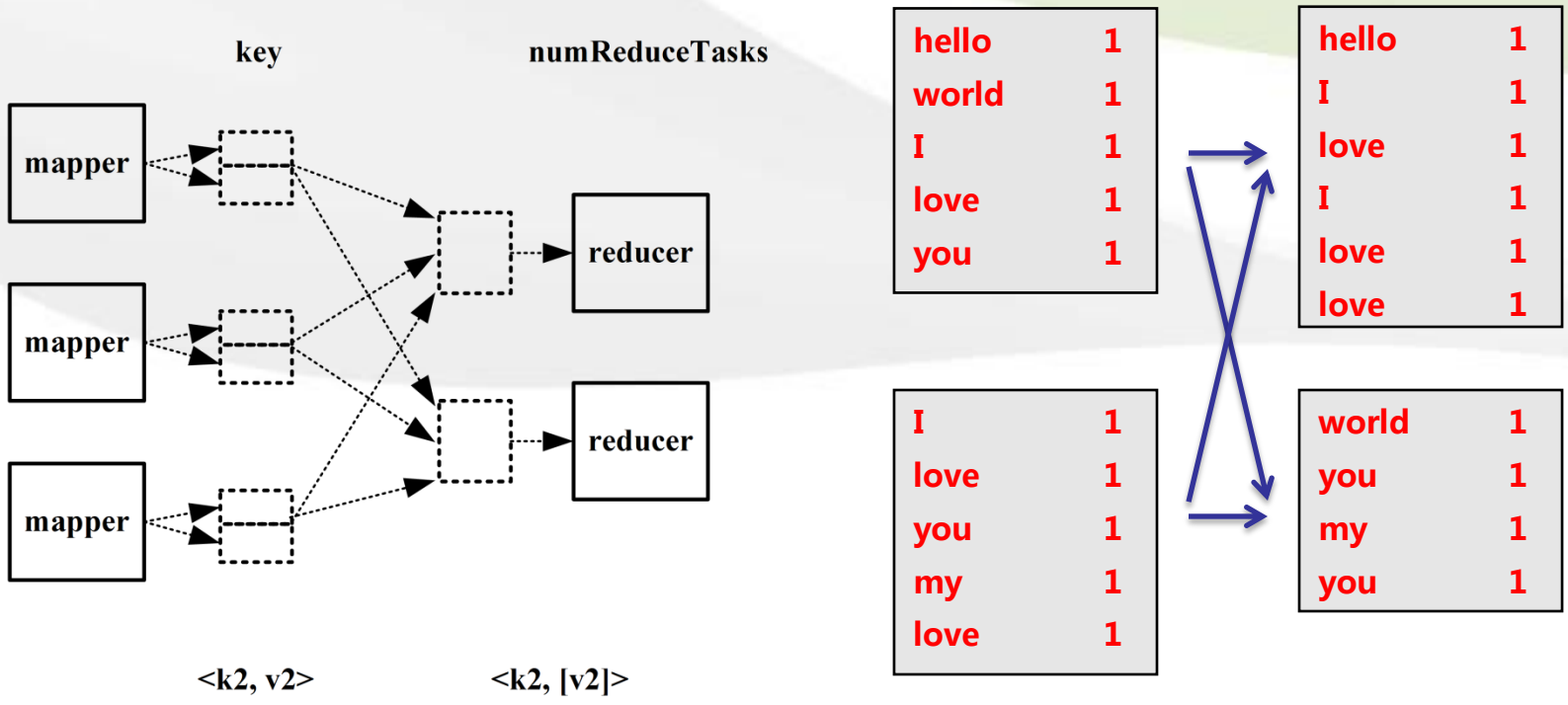
③ 内存数据溢出到磁盘（Spill）
- Sort
- Combiner

④ 合并中间结果文件（Merge）

Partition

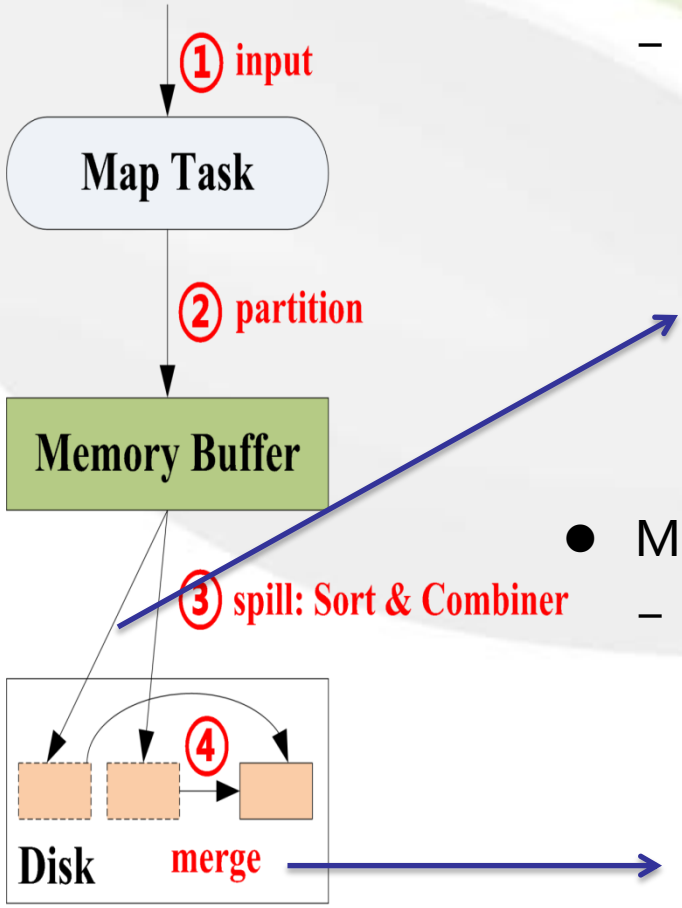
- 作用：将map的结果发送到相应的reduce
- 要求：负载均衡、效率
- 默认HashPartitioner

(key.hashCode() & Integer.MAX_VALUE) % numReduceTasks



- 可自定义：job.setPartitionerClass(MyPartitioner.class);

Sort



- Map后的第1次排序：文件内部快速排序（Sort）
  - map函数处理完输入数据之后，会将中间数据存在本机的一个或者几个文件当中，并且针对这些文件内部的记录进行一次快速排序

I	1
love	1
you	1
my	1
love	1

→

I	1
my	1
love	1
you	1
love	1

- Map后的第2次排序：多个文件归并排序（Merge）
  - Map任务执行完成后会对这些排好序的文件做一次归并排序，并将排好序的结果输出到一个大的文件中

I	1
love	1
you	1
my	1
love	1

→

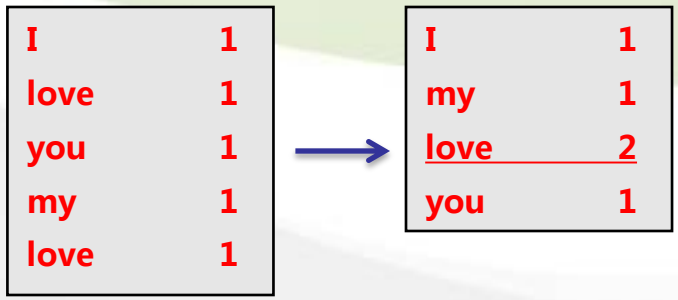
...	
I	1
my	1
love	1
you	1
love	1
...	

Combiner

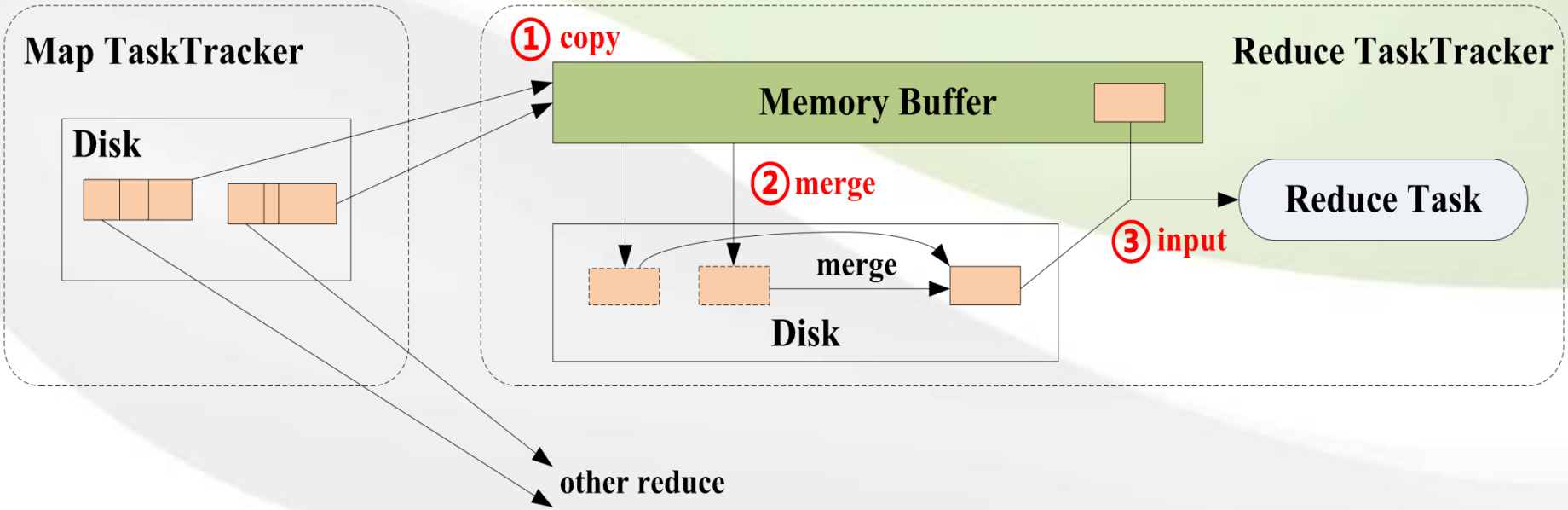
- 作用：合并Map输出的中间数据，减少数据传输、提高处理效率

```
public static void main(String[] args) throws
Exception{
  …
  job.setCombinerClass(IntSumReducer.class);
  …
}
```

```
public static class IntSumReducer extends Reducer
  <Text,IntWritable,Text,IntWritable>{
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable>
    values,  Context context) throws IOException{
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();}
      result.set(sum);
      context.write(key, result);
} }
```

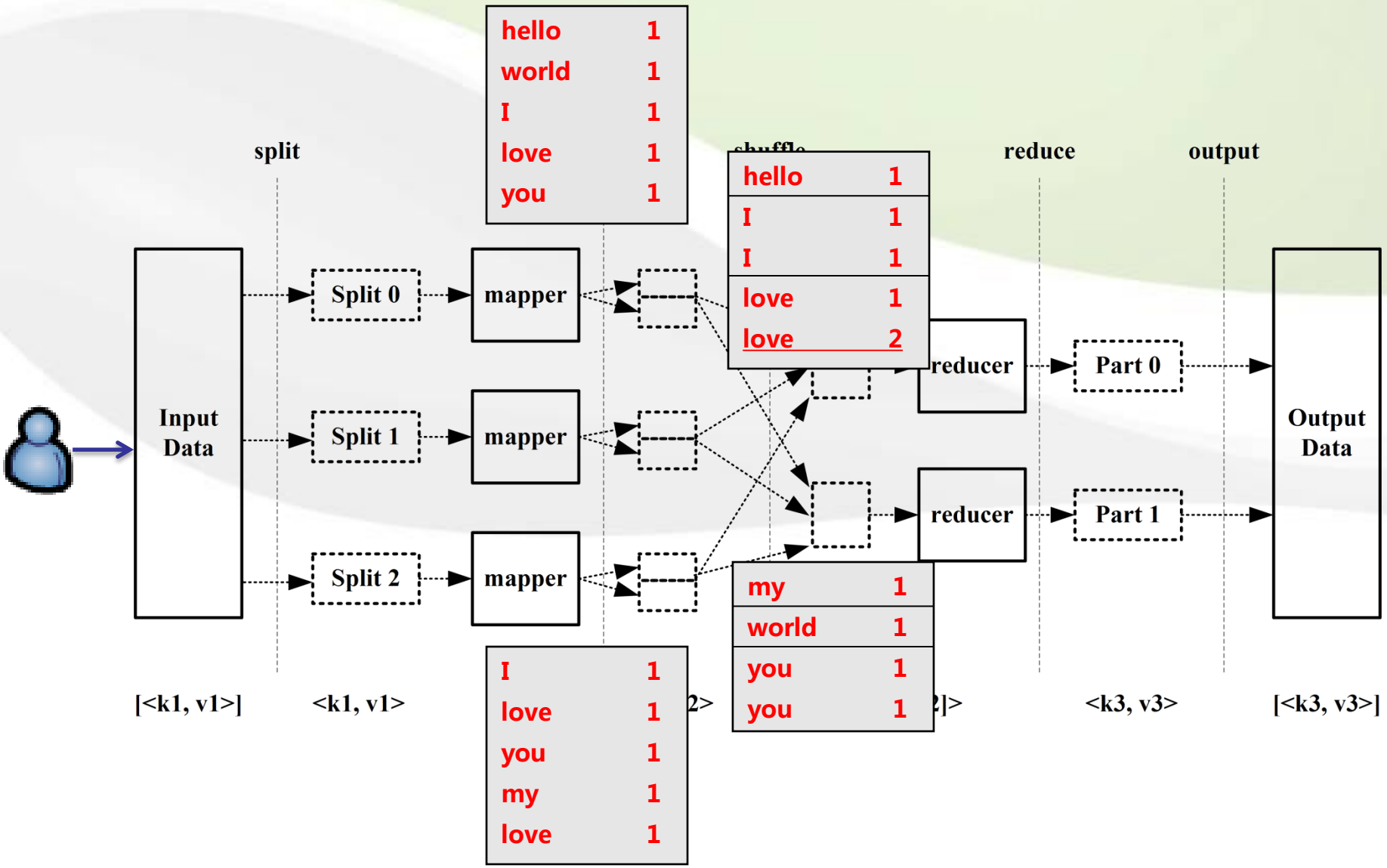


Reduce端拉取数据



① 拉取数据（Copy）
② 合并中间结果文件（Merge）
③ 处理数据

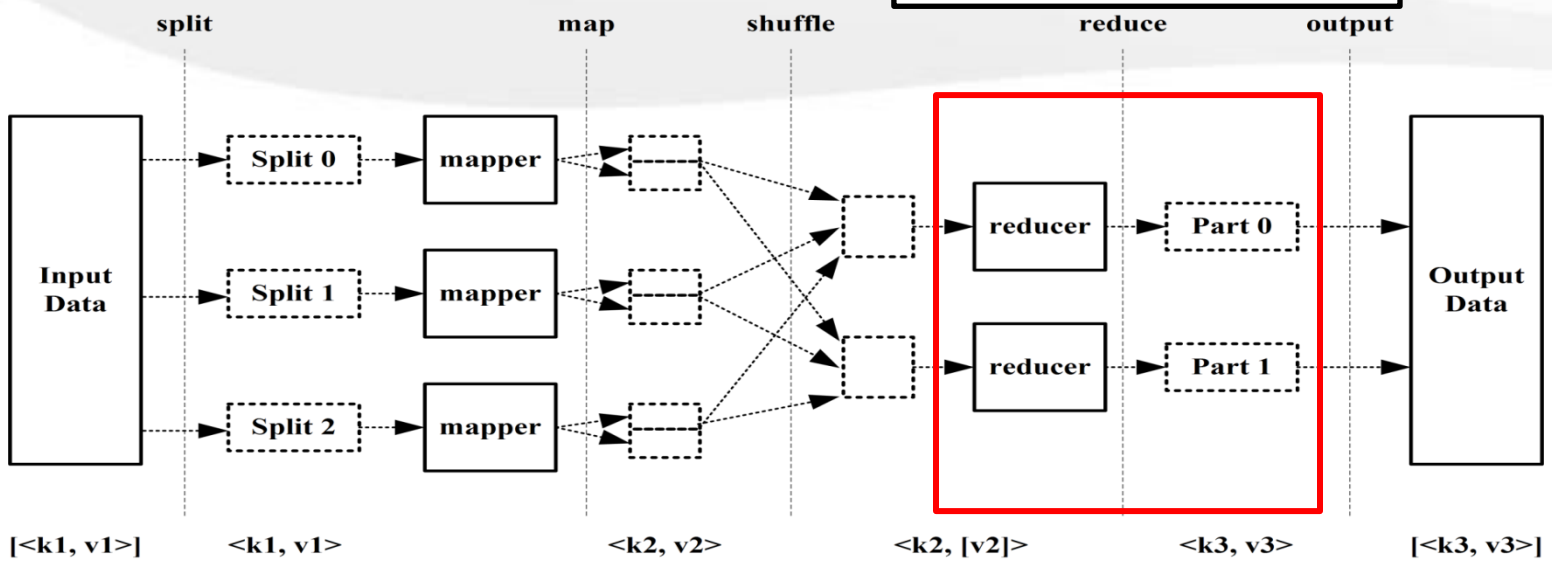
shuffle完成后


hello 1
world 1
I 1
love 1
you 1
split
shuffle
reduce
output
hello 1
I 1
I 1
love 1
love 2
Input Data
Split 0
Split 1
Split 2
mapper
mapper
mapper
reducer
reducer
Part 0
Part 1
Output Data
my 1
world 1
you 1
you 1
I 1
love 1
you 1
my 1
love 1
[<k1, v1>]
<k1, v1>
<k3, v3>
[<k3, v3>]

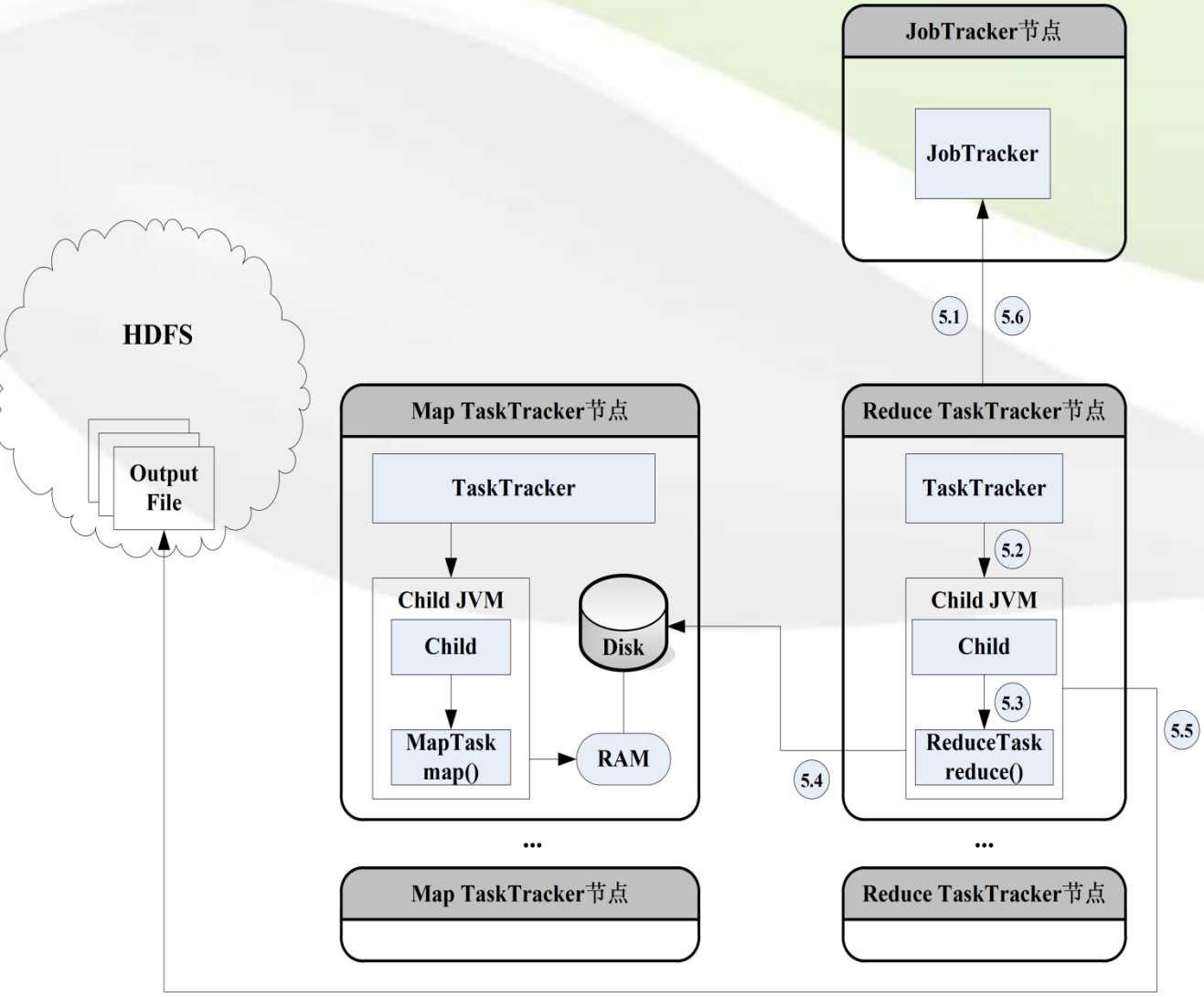
MapReduce运行流程（6） - Reduce任务执行

```
public static class IntSumReducer extends Reducer
  <Text,IntWritable,Text,IntWritable>{
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable>
    values,  Context context) throws IOException{
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();}
      result.set(sum);
      context.write(key, result);
} } }
```

⑥ Reduce任务执行



Reduce任务执行



5.1 分配Reduce任务

5.2 创建TaskRunner运行Reduce任务

5.3 在单独的JVM中启动ReduceTask执行reduce函数

5.4 从Map节点下载中间结果数据

5.5 输出结果临时文件

5.6 定期报告进度

计算容错：TaskTracker

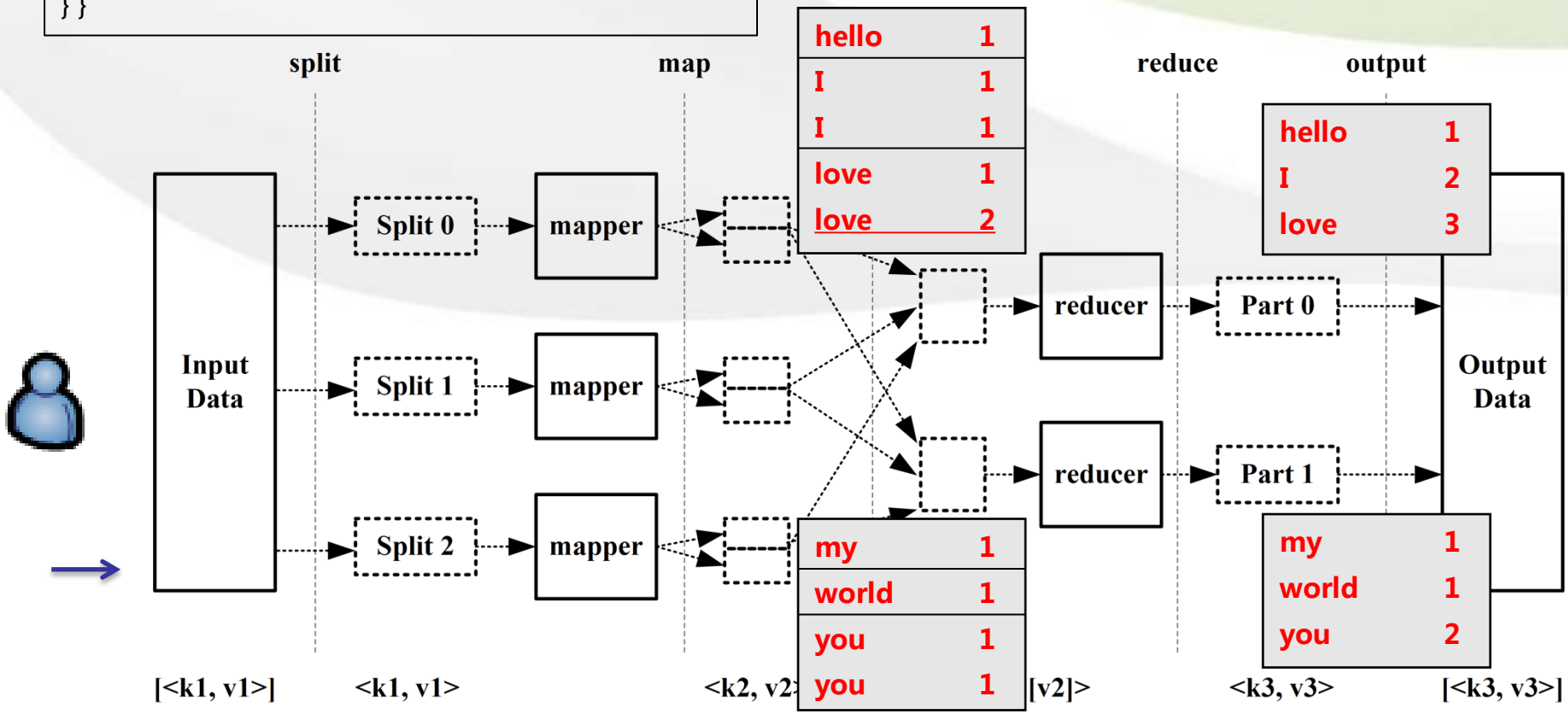
- 心跳监测
  - mapred.tasktracker.expiry.interval，默认10分钟
  - 已完成的任务会正常返回，未完成的任务则重新分配TaskTracker节点执行
- 黑名单机制
  - Job黑名单：每个Job会维护一个TaskTracker黑名单
  - 集群黑名单：整个集群的TaskTracker黑名单
  - 当一个job成功结束时，对该Job黑名单中的tasktracker做如下三个判断：
    - 该TaskTracker被4个Job加入了黑名单（mapred.max.tracker.blacklists）
    - 该TaskTracker被加入Job黑名单的次数，超过了集群中所有tasktracker被加入Job黑名单平均次数的50%（mapred.cluster.average.blacklist.threshold）
    - 已经加入黑名单的TaskTracker个数不超过集群总TaskTracker的50%
  - 以上三条如果均满足，则将Job黑名单中的该TaskTracker加入集群黑名单，以后将不在该TaskTracker上调度任何task
  - 恢复：重启TaskTracker

计算容错：Task失败

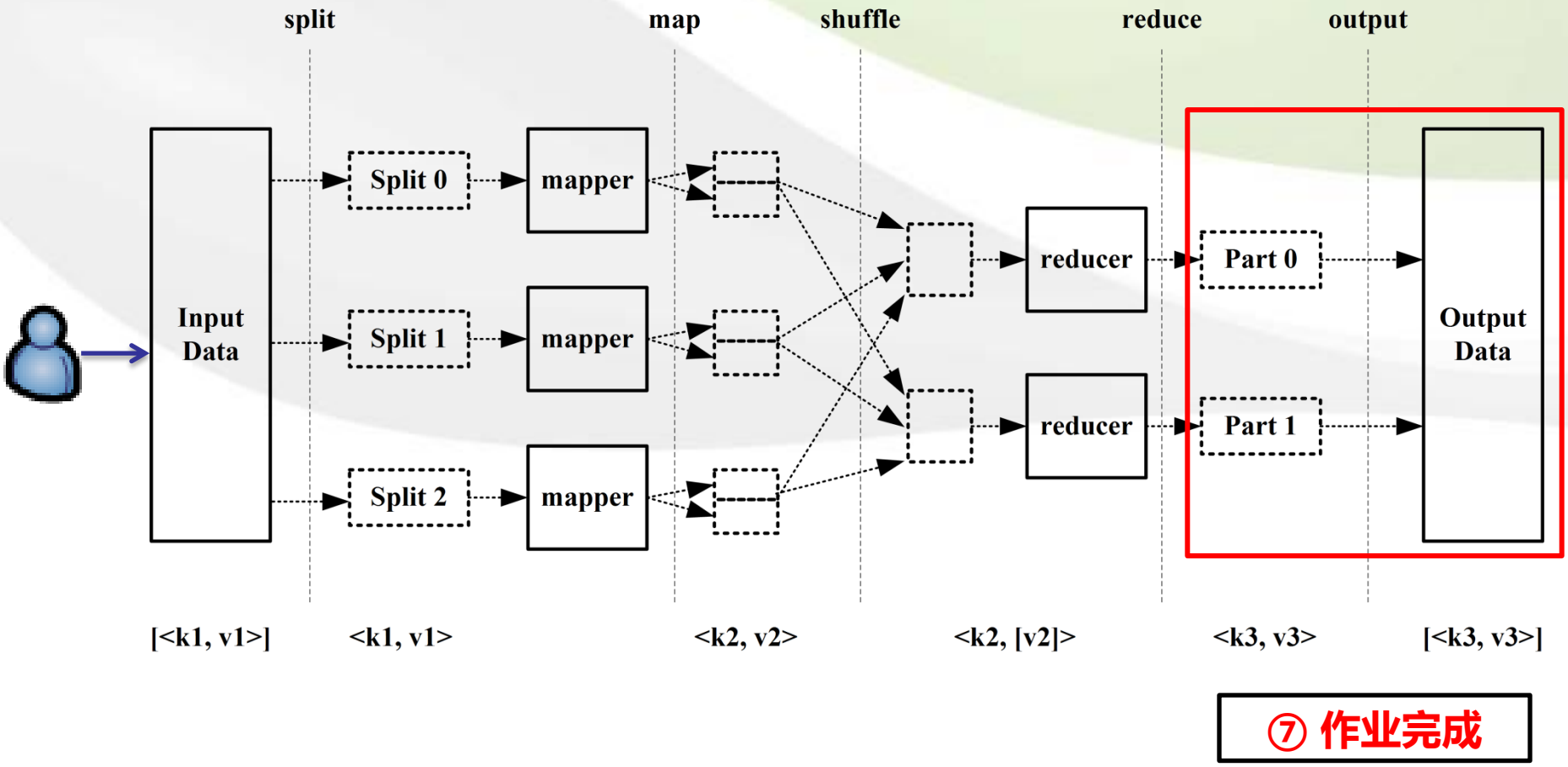
- Task异常：失败（failed）和终止（killed）
- 失败原因：
  - map或reduce函数代码不正确
  - 任务所在的JVM出现运行异常
  - 任务进度更新超时
- 失败处理：
  - TaskTracker将此任务的失败信息报告给JobTracker
  - JobTracker分配新的节点执行此任务
  - 如果同一个任务出现多次失败，且失败次数超过由参数指定的最大次数时，作业会在未完成的情况下被终止
    - mapred.max.map.attempts
    - mapred.reduce.max.attempts

Reduce任务执行完成后

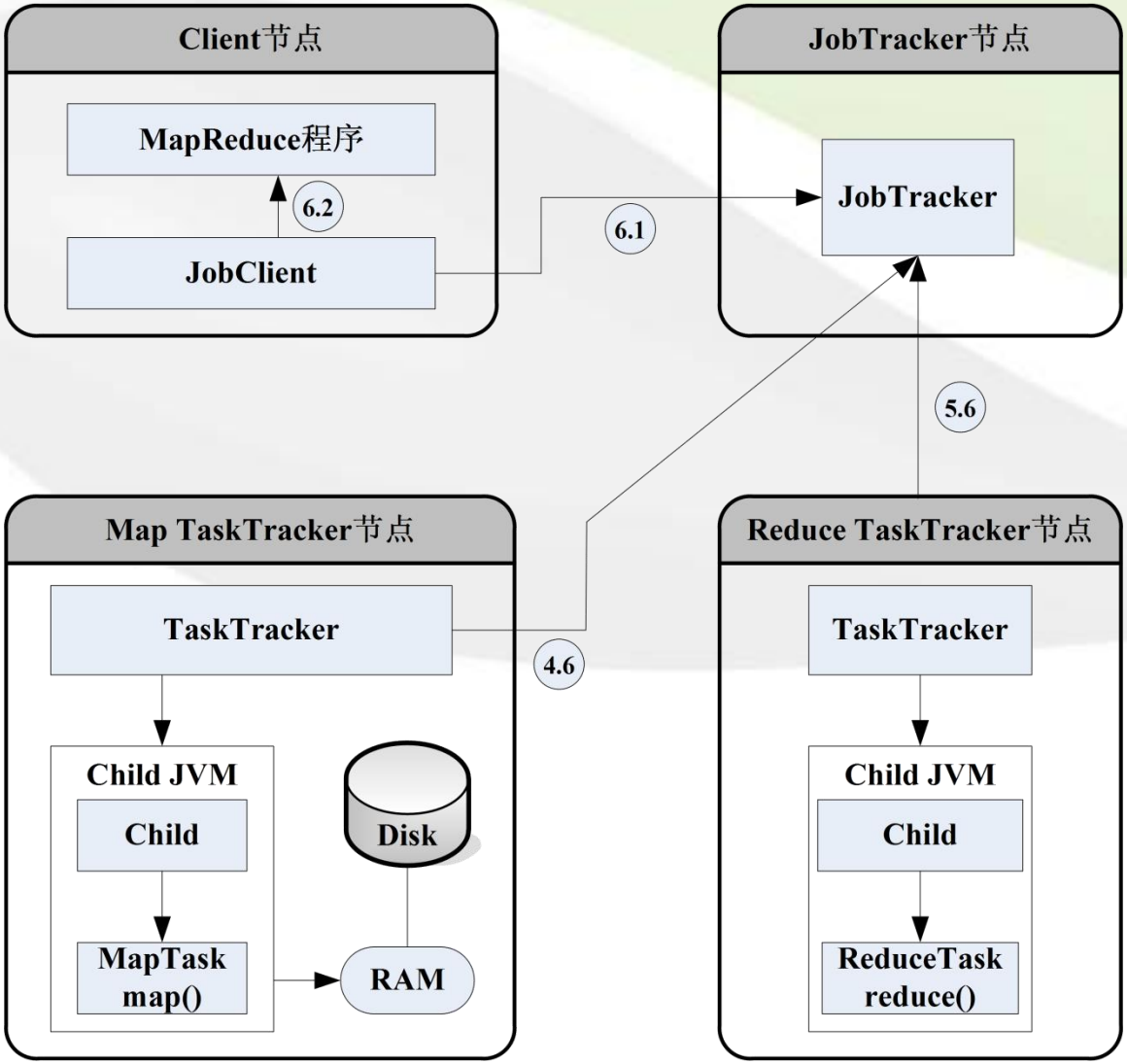
```
public void reduce(Text key, Iterable<IntWritable>
    values,  Context context) throws IOException{
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();}
      result.set(sum);
      context.write(key, result);
} }
```



MapReduce运行流程（7） - 作业完成



作业完成

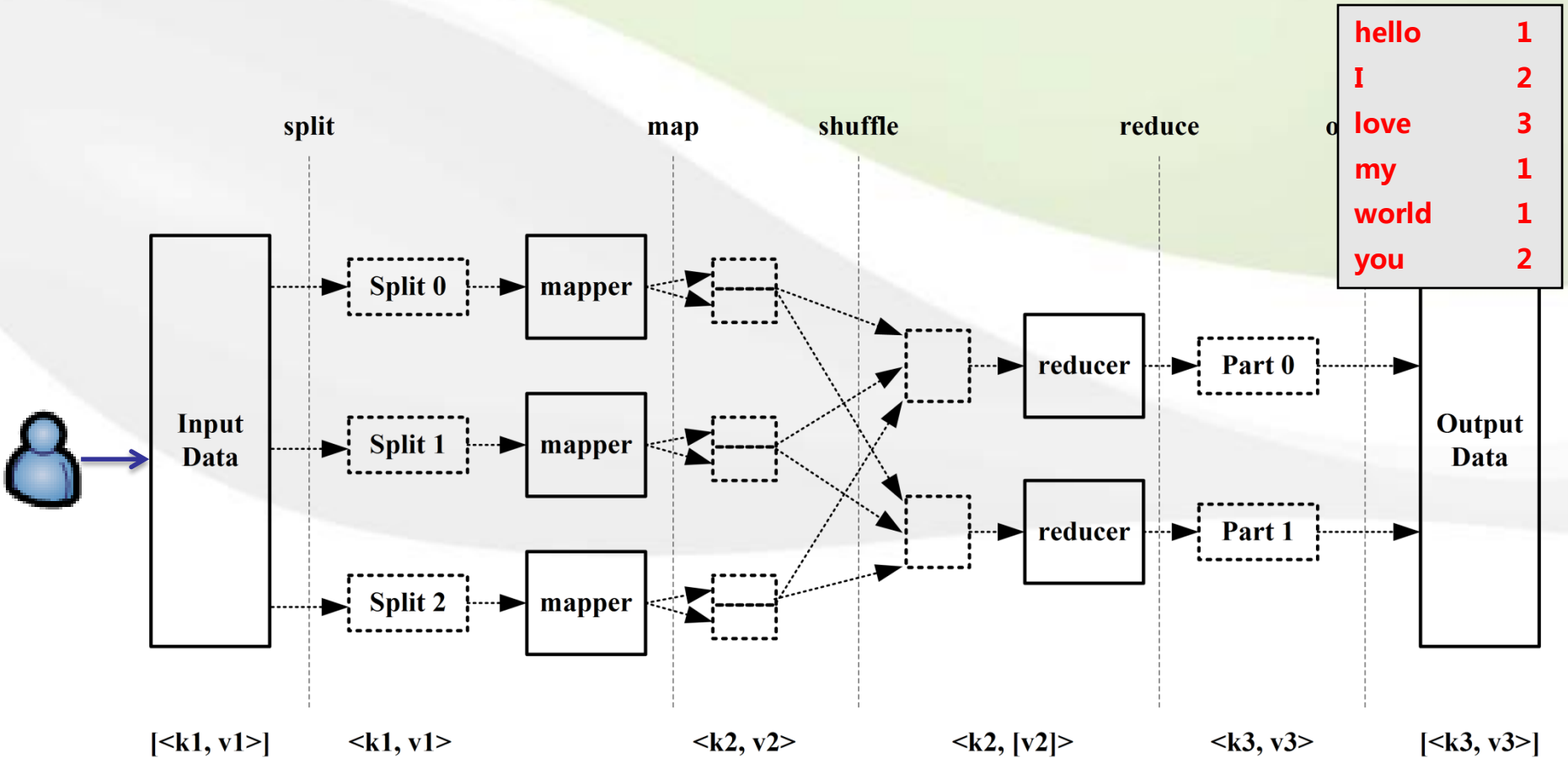


4.6/5.6 进度更新

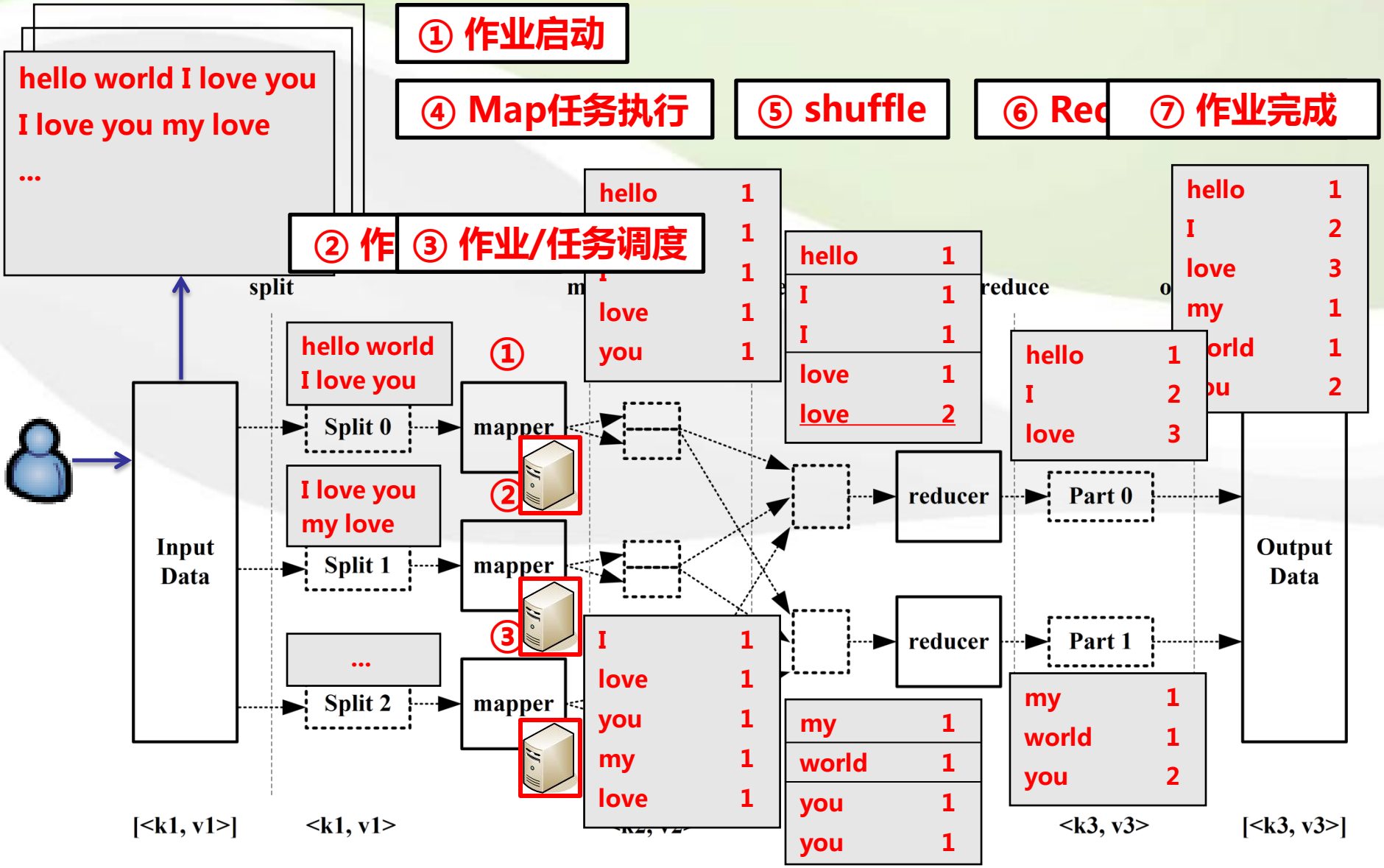
6.1 JobClient轮询获知任务完成

6.2 通知用户

作业完成后


split
map
shuffle
reduce
o
hello 1
I 2
love 3
my 1
world 1
you 2
Input Data
Split 0
Split 1
Split 2
mapper
mapper
mapper
reducer
reducer
Part 0
Part 1
Output Data
[<k1, v1>]
<k1, v1>
<k2, v2>
<k2, [v2]>
<k3, v3>
[<k3, v3>]

完整的WordCount过程



本节问题及下节课程预告

- 本节问题：
  - 编写MapReduce程序，处理access.log（QQ群共享）文件，统计每个用户访问每个SP的次数、总流量

- 下节课程预告（欢迎大家踊跃提问）
  - MapReduce（2）
  - 2013.4.7