大数据技术概论

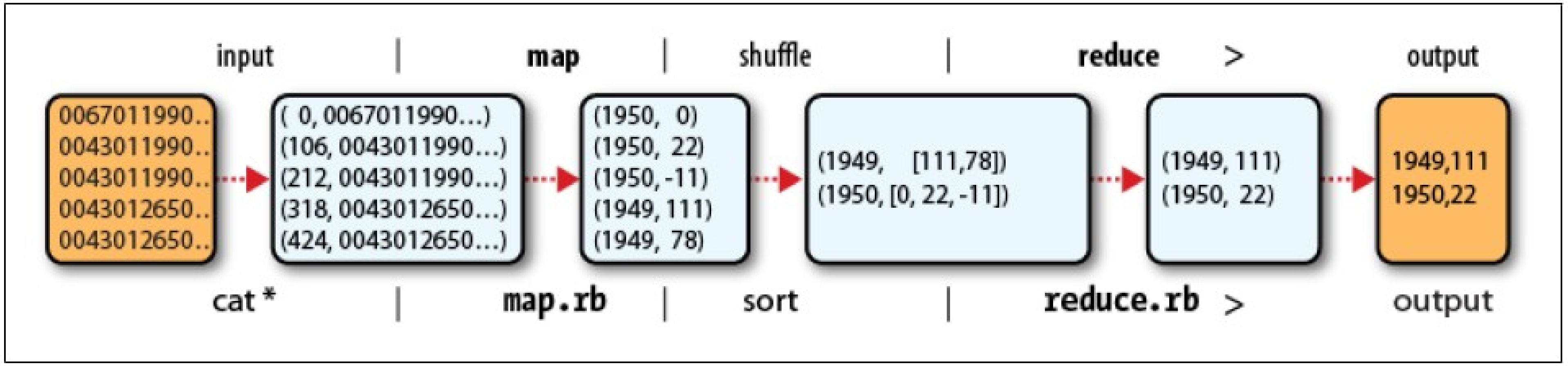
中国科学院大学人工智能学院

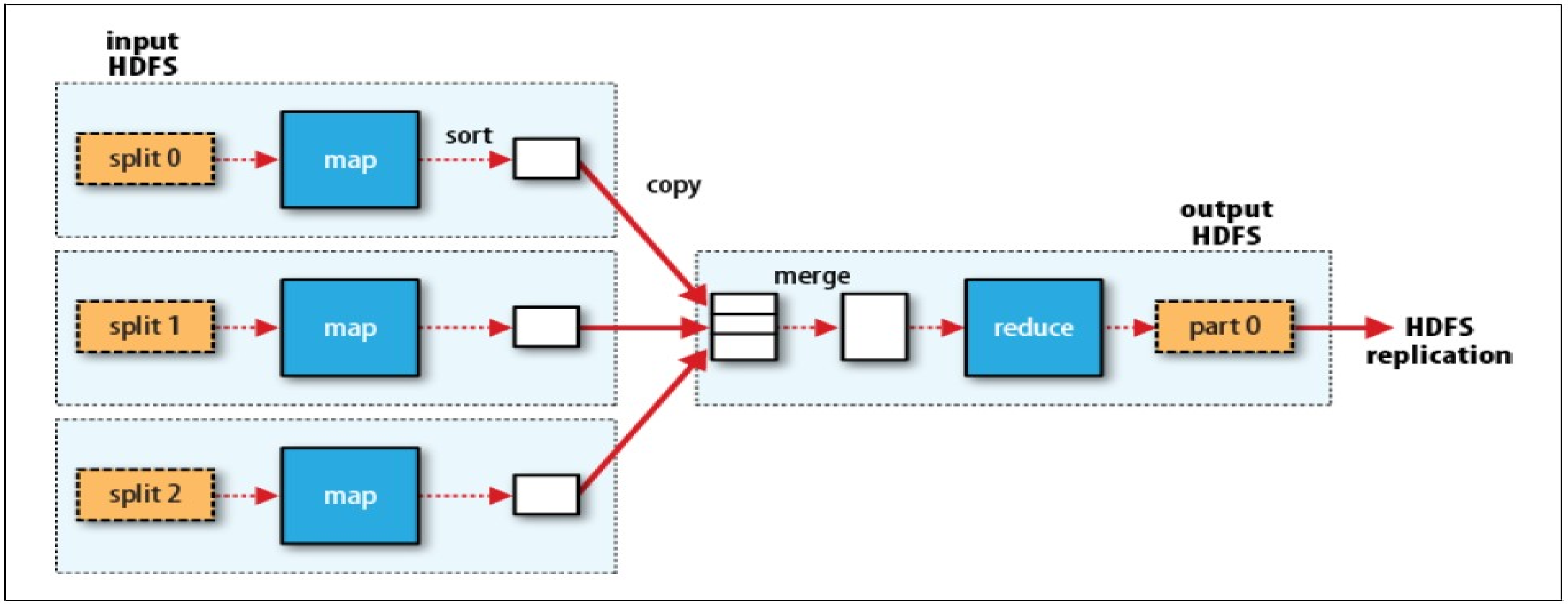
2017年秋季学期

提纲

- MapReduce Programming Model
- WordCount等几个例子
- Hadoop的 I/O操作

使用Hadoop来分析程序



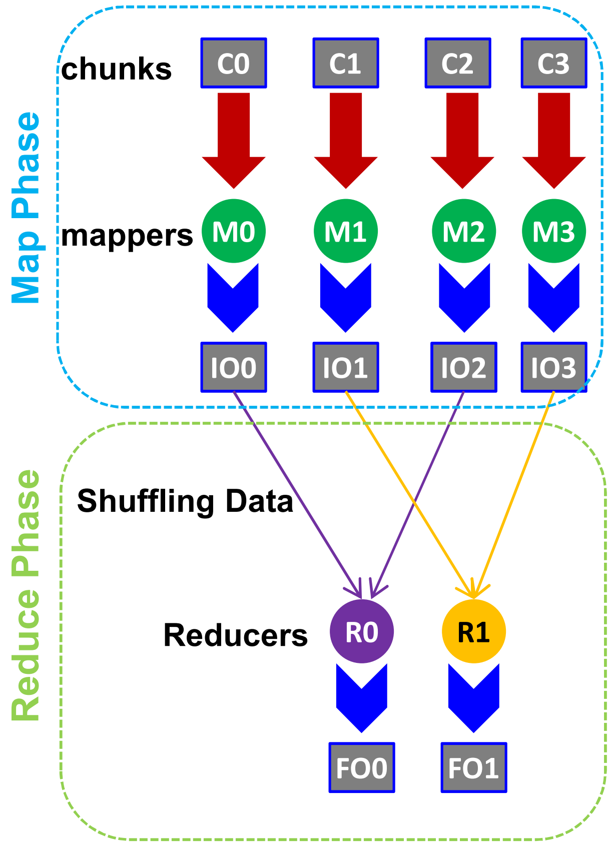


MapReduce Programming Model

- 输入一系列的键、值对，生成一系列的键值对
  - $\{ < k_i, v_i > \} \rightarrow \{ < k_o, v_o > \}$
- 用两个函数来表示:
  - Map task (映射) :对独立的分块列表中的每一条记录进行指定的操作
    - $<k_i, v_i> \rightarrow \{ < k_t, v_t > \}$
  - Reduce task (规约) :通过定义的化简函数对一个列表中的元素进行适当的合并，得到最终的结果
    - $< k_t, \{v_t\} > \rightarrow < k_o, v_o >$

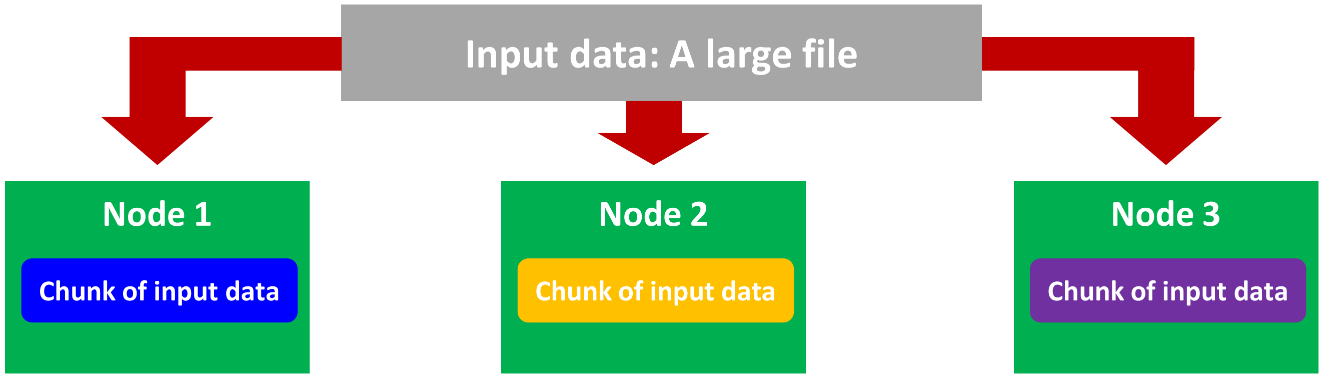
MapReduce: 概要框架

- Map（映射）：对独立的分块列表中的每一条记录进行指定的操作
- Shuffle(混洗）:实现从Map输出到Reduce输入的中间操作;对中间输出中同一关键词对应的所有值进行聚合;把中间结果传输给Reduce
- Reduce(规约）：通过定义的化简函数对一个列表中的元素进行适当的合并，得到最终的结果



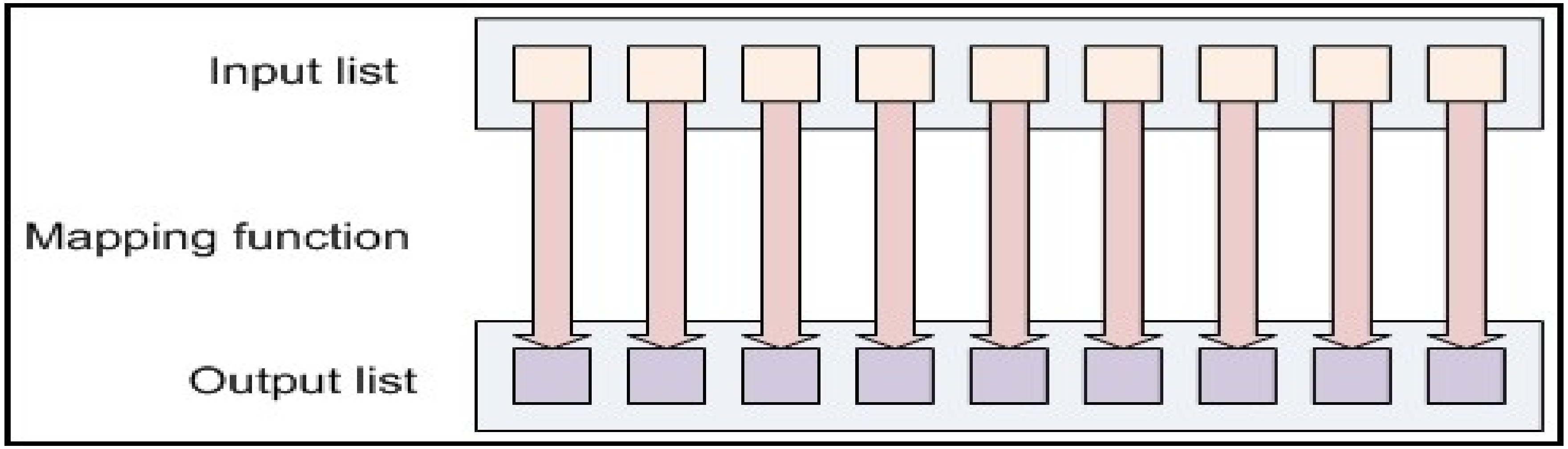
数据分发

- Hadoop将MapReduce的输入数据划分为等长的小数据块(input split)，简称“分片”。
- Hadoop为每个分片构建一个Map任务，并由该任务来运行用户自定义的map函数从而处理分片中的每条记录



map

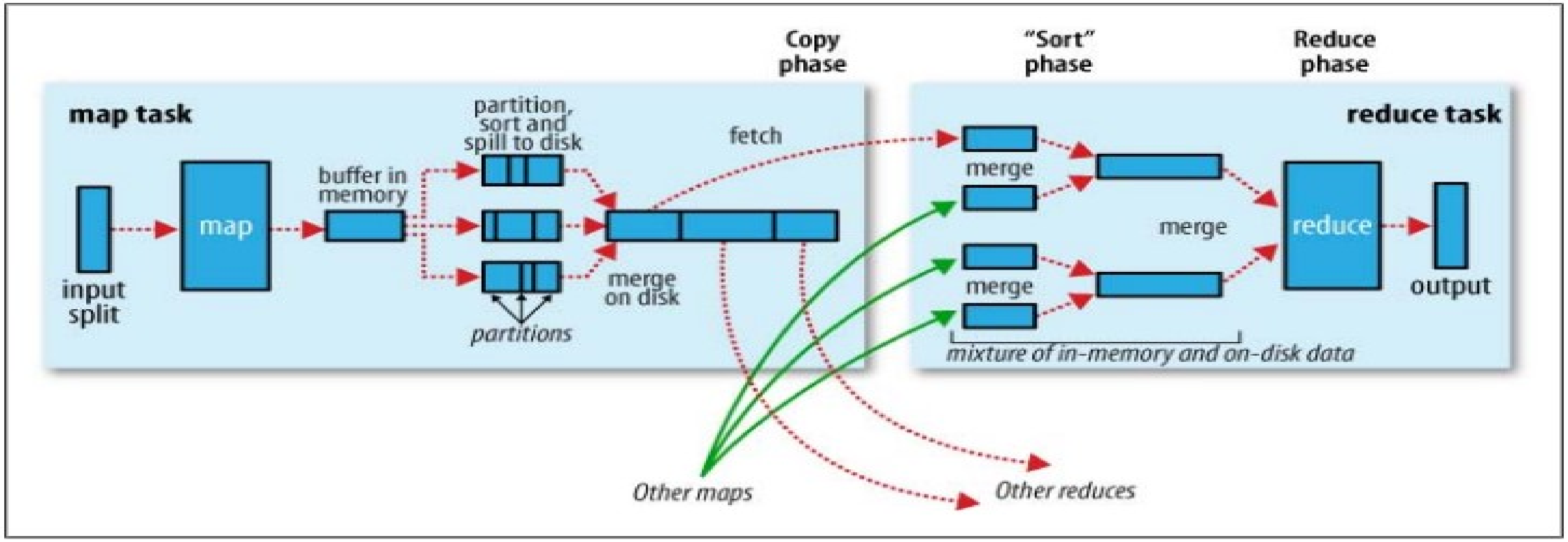
map: (k1,v1) → list(k2,v2)



- 数 据 本 地 化 优 化： Hadoop 在 存 储 有 输 入 数 据 (HDFS中的数据)的节点上运行map任务，可以获得最佳性能
- Map任务将其输出写入本地硬盘，而非HDFS

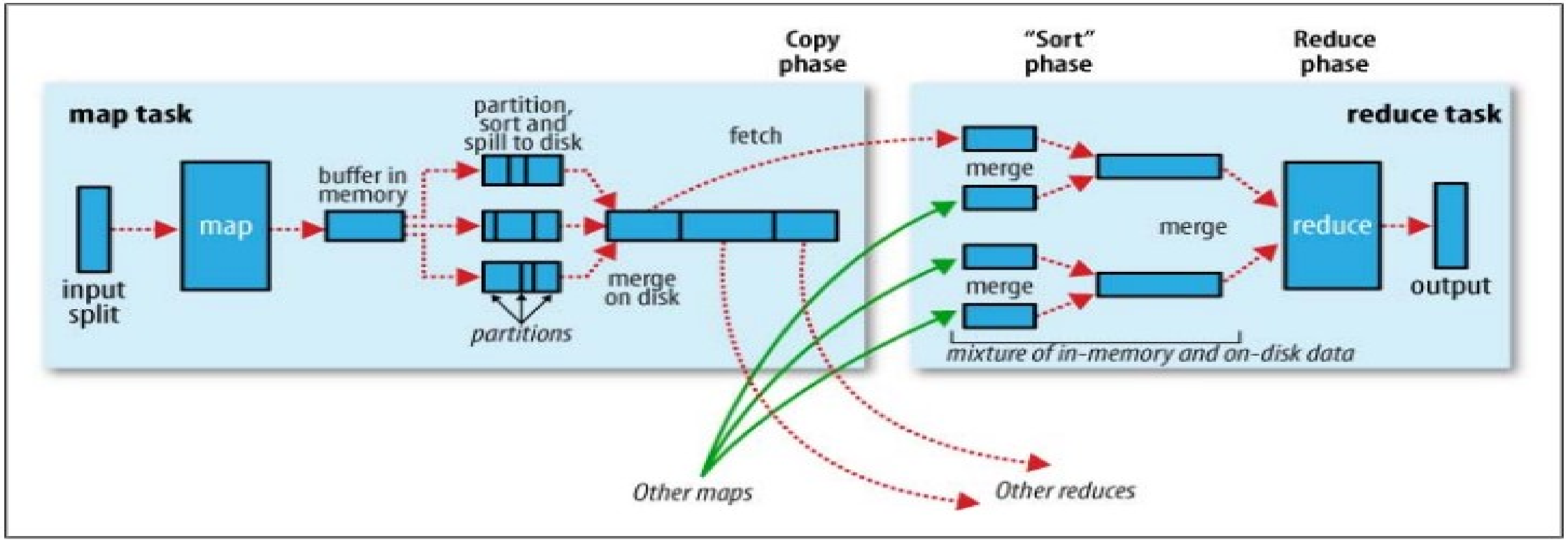
Shuffle:奇迹发生的地方

- Shuffle 把 Map输出结果处理后分发给 reducers
- 对中间输出中同一关键词对应的所有值进行聚合
- 对reducer的输出严格按照关键词排序
- 同一关键词对应的所有记录输出给同一reducer



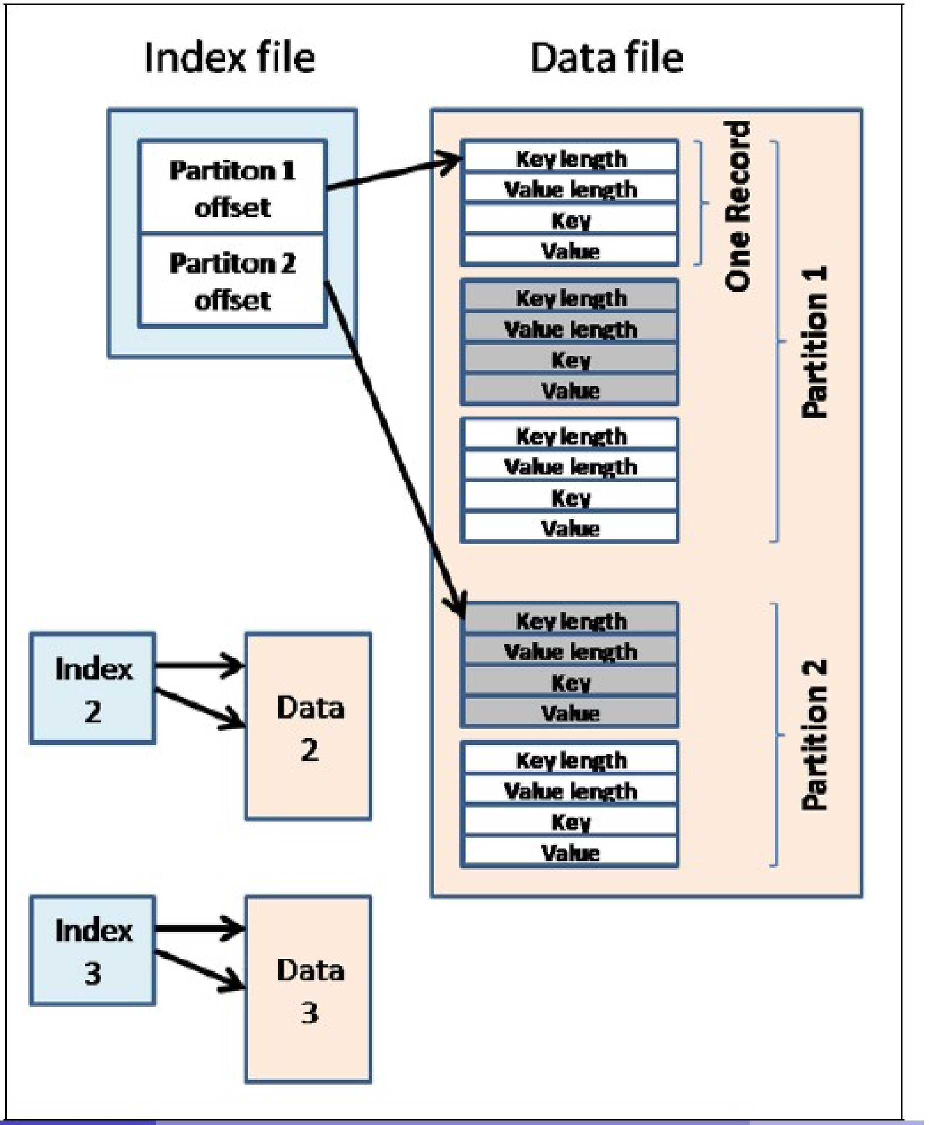
Shuffle: Map Side

- Map函数输出写到内存并进行预排序
- 采用环形内存缓冲区存储Map任务输出(100MB)
  - 缓冲内容达到阈值，后台线程便把内容溢出(Spill)到磁盘
  - 写磁盘(mapred.local.dir)之前，把数据划分为分区(partition)
  - 在每个分区内，按键进行内排序
- Independent（独立性）;Idempotent(幂等性)



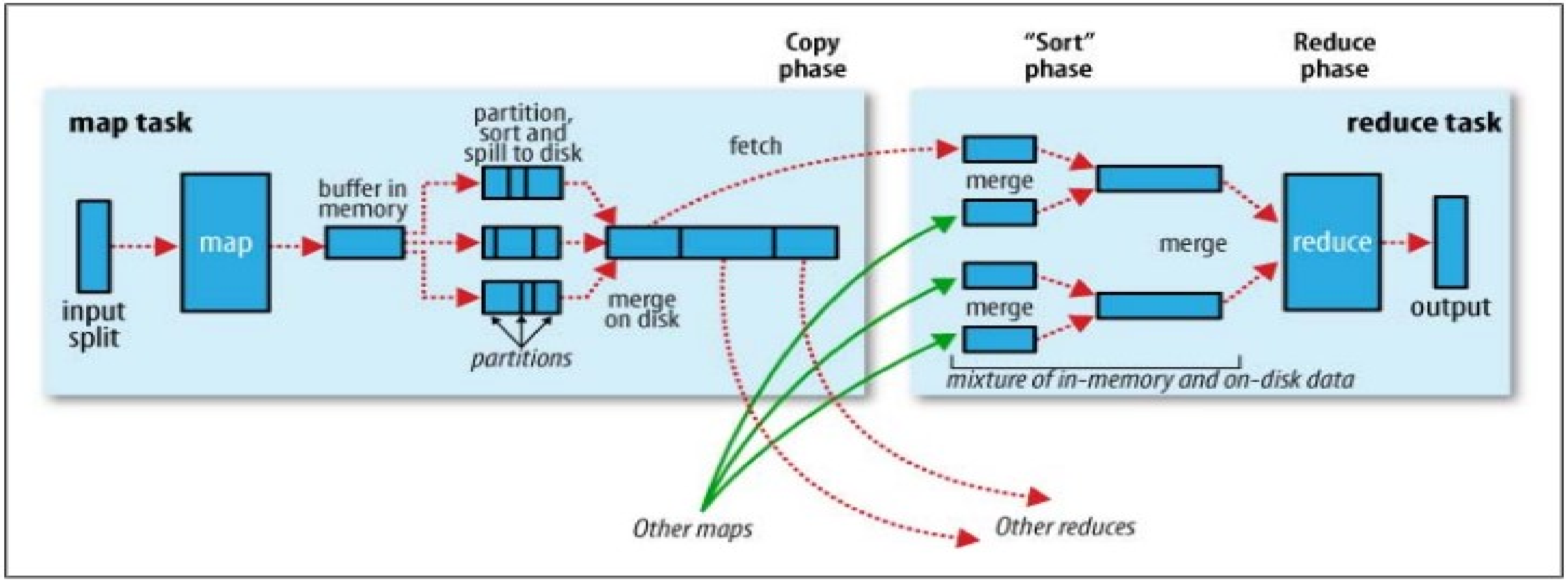
Shuffle(Map Side)

- Map任务完成时，多个溢出(Spill)文件被合并成一个已分区且已排序的输出文件
- Reduce通过HTTP方式得到输出文件的分区



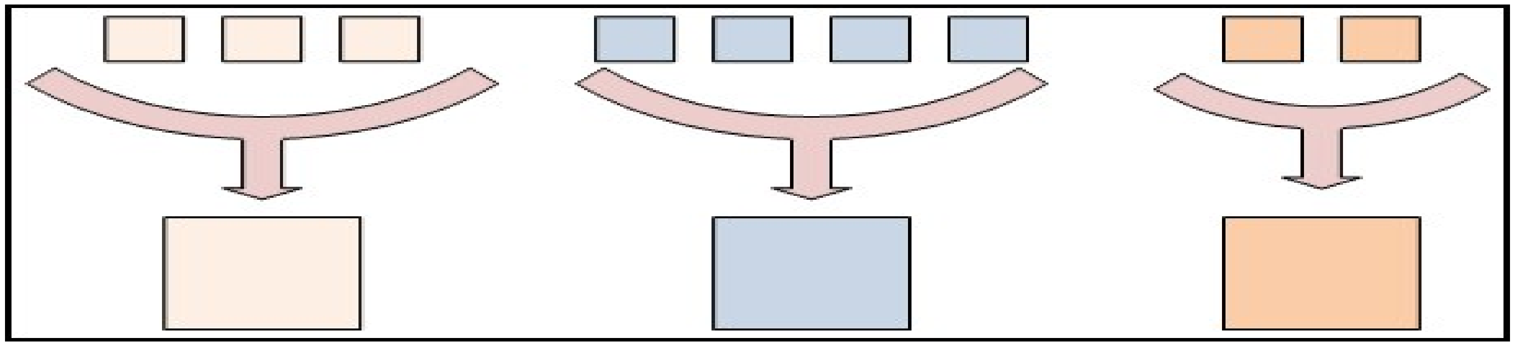
Shuffle: Reduce Side

- 只有有一个Map任务完成，Reduce任务就开始复制其输出
- 执行多轮合并，在合并过程中维持顺序排序



Reduce

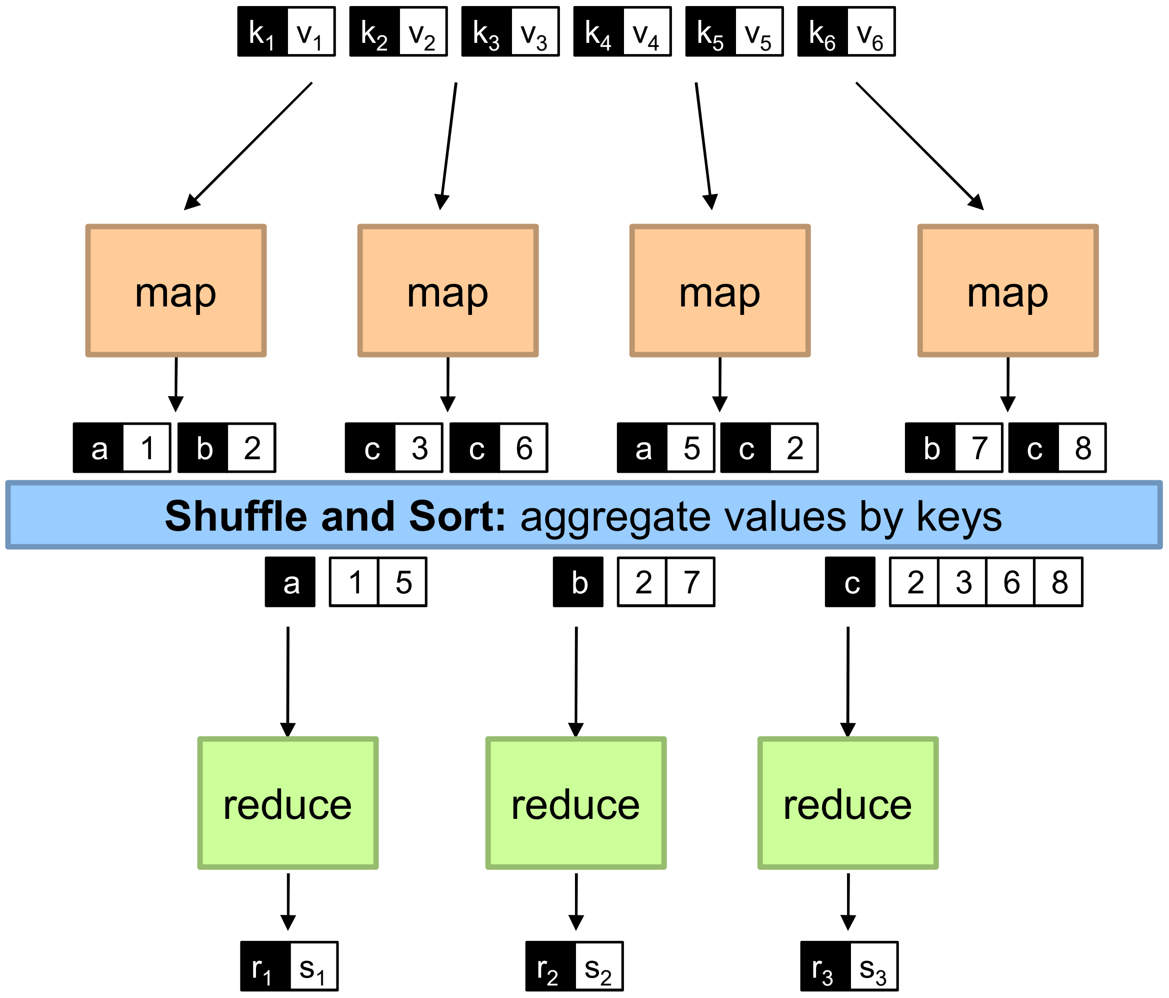
reduce: (k2,list(v2)) → list(k3,v3)



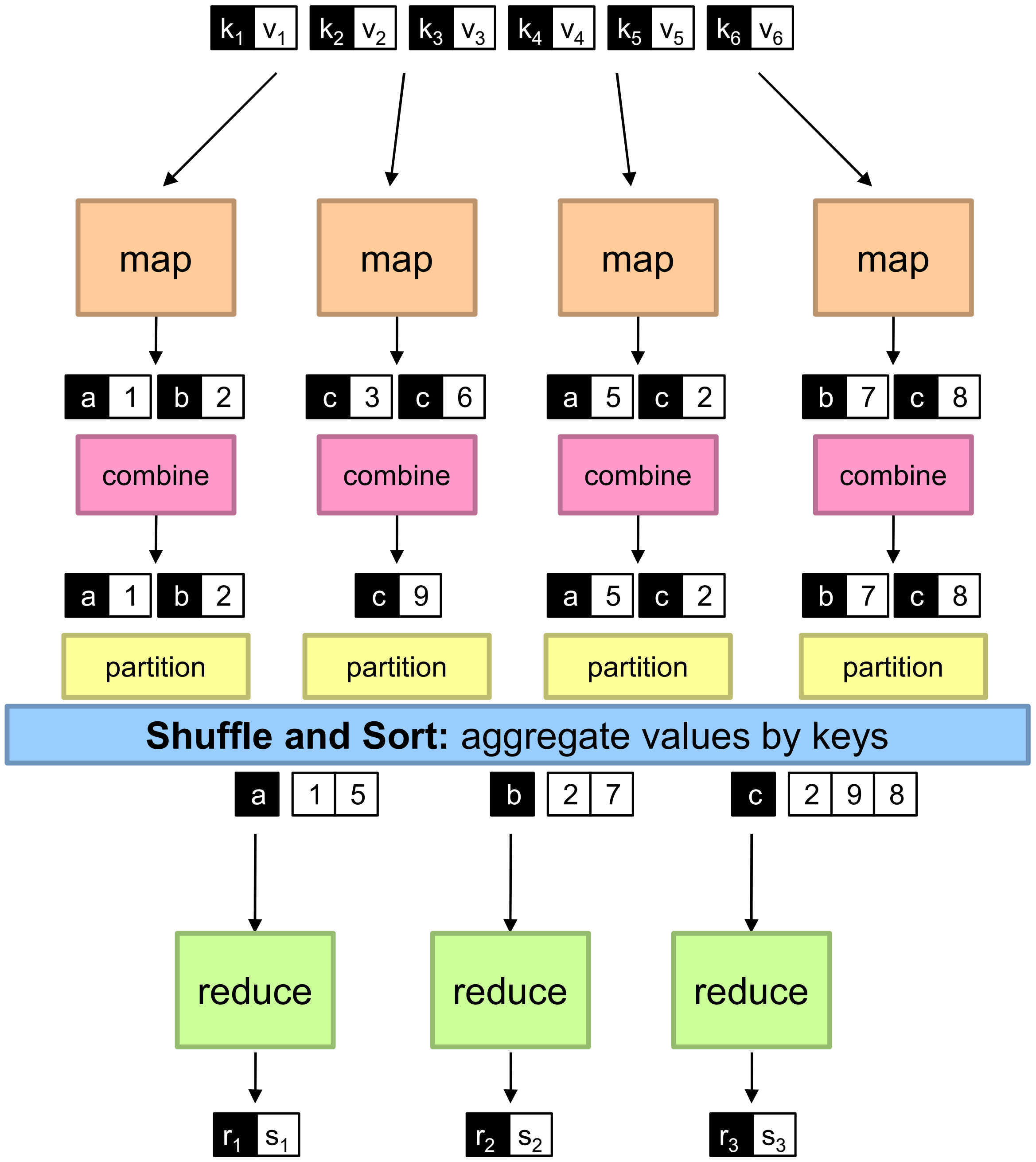
- 如果有若干Reduce任务，每个map任务就会针对输出进行分区(partition)，即为每个Reduce任务建一个分区
- Reduce任务的数量并非由输入数据的大小决定，而事实上是独立指定的

MapReduce 处理流程

- 读取输入文件
  - 把输入文件划分为分片, 把每个分片分配给一个Map任务
- Map任务
  - 使用Map任务处理分片中的每一条记录
  - 每个Map任务返回一系列(key, value) 对
- Shuffle/Partition and Sort
  - Shuffle 把 sorting & aggregation 分发给 reducers
  - 关键词 k 对应的所有记录转发给同一个reduce processor
  - Sort 按照关键词k来分组, 为aggregation做准备
- Reduce 任务
  - 使用Reduce处理每个key对应的记录
  - Reduce 函数的最后结果是一系列(key, value) pairs



k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3



k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce “Runtime”

- 管理调度
  - Assigns workers to map and reduce tasks
- 管理数据
  - Moves processes to data
- 管理同步
  - Gathers, sorts, and shuffles intermediate data
- 管理错误
  - Detects worker failures and restarts

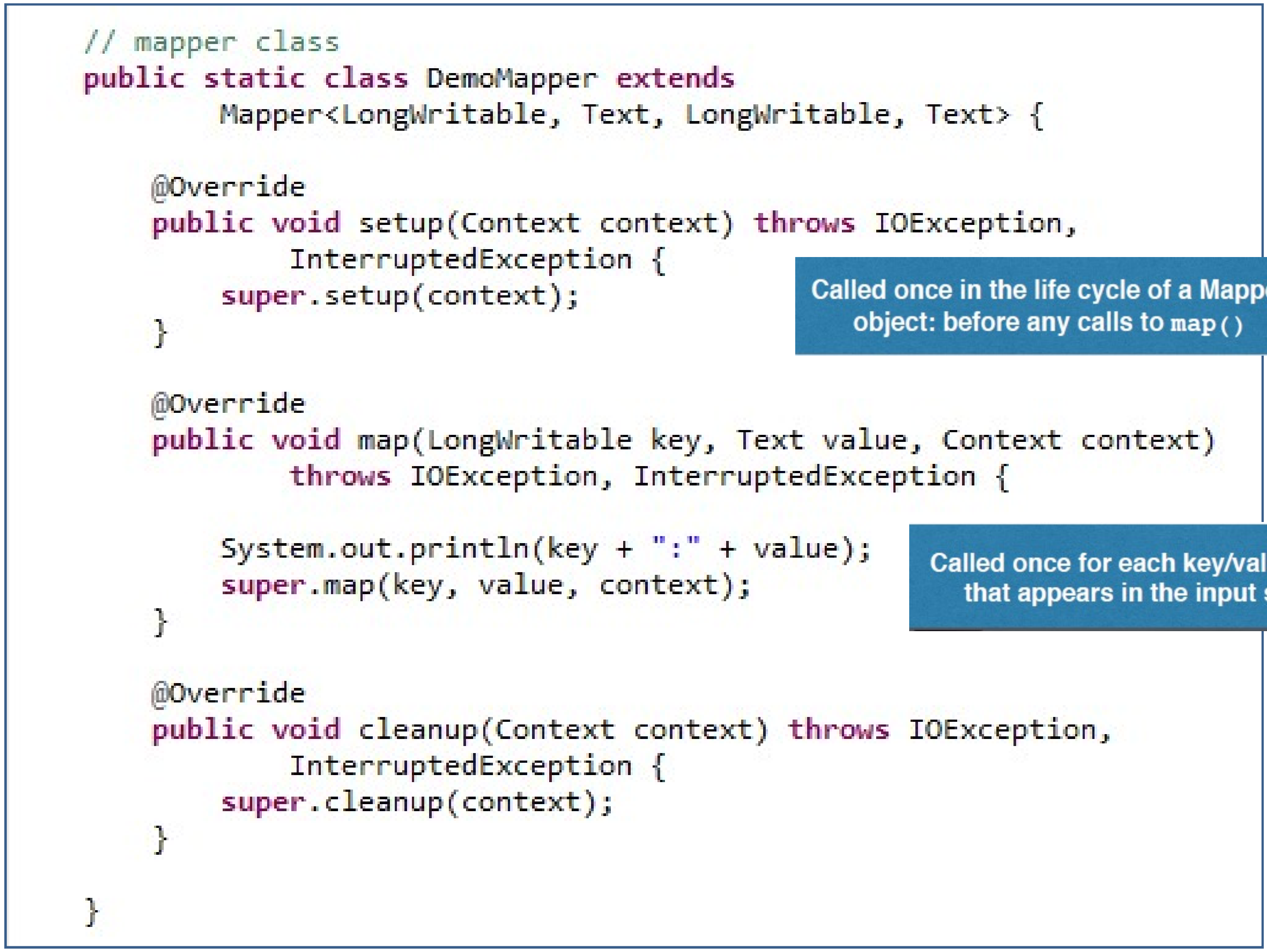
提纲

- MapReduce Programming Model
- WordCount等几个例子
- Hadoop的 I/O操作

默认的MapRedue作业

```
  Job job = new Job(getConf());
  job.setJarByClass(getClass());

  // 1) set input
 FileInputFormat.addInputPath(conf, new Path(args[0]));

// 2) set output
 FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  return job.waitForCompletion(true)?0:1;
```

Mapper class

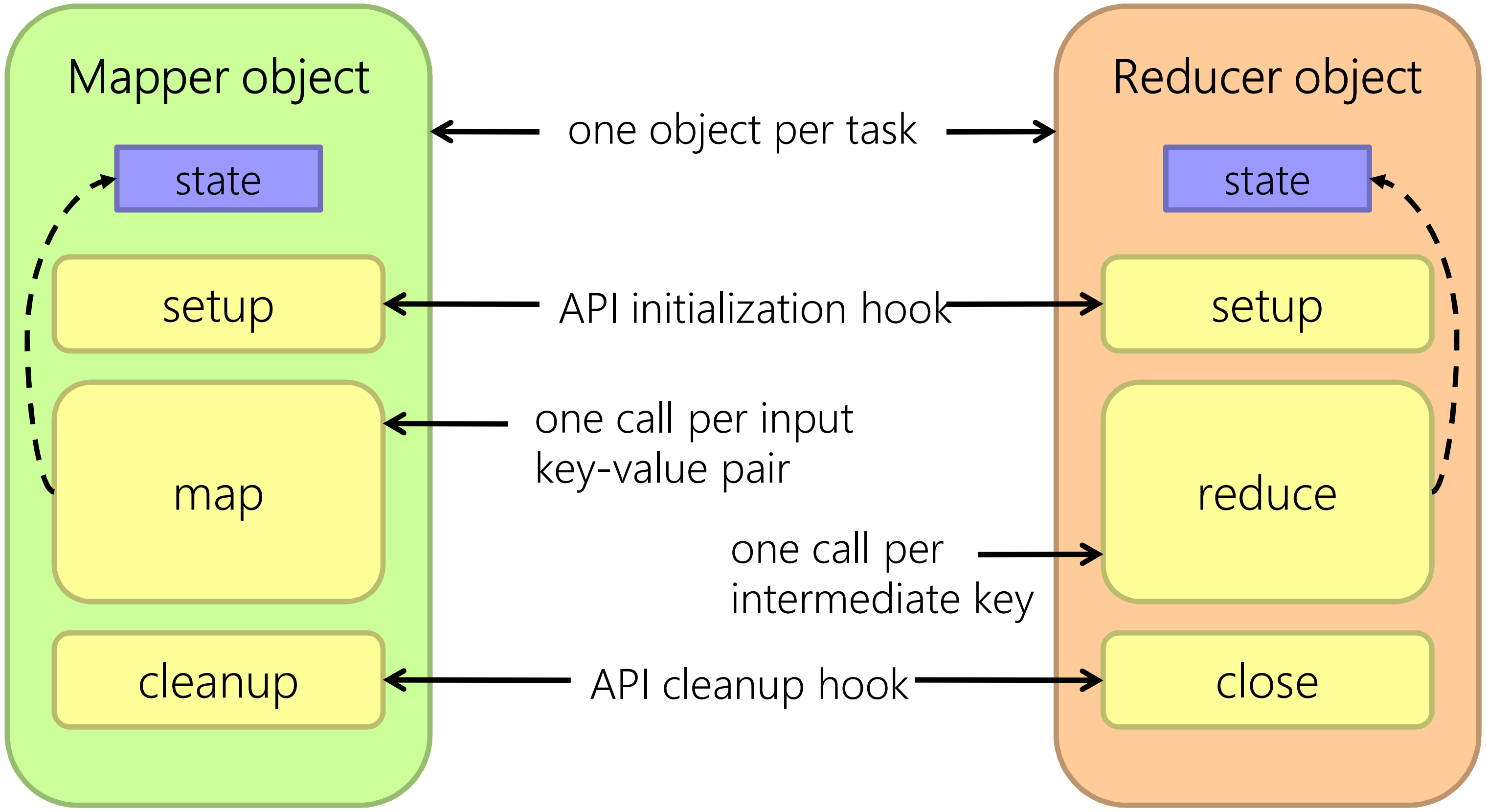


```
// mapper class
public static class DemoMapper extends
        Mapper<LongWritable, Text, LongWritable, Text> {

    @Override
    public void setup(Context context) throws IOException,
            InterruptedException {
        super.setup(context);
    }

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        System.out.println(key + ":" + value);
        super.map(key, value, context);
    }

    @Override
    public void cleanup(Context context) throws IOException,
            InterruptedException {
        super.cleanup(context);
    }

}
```

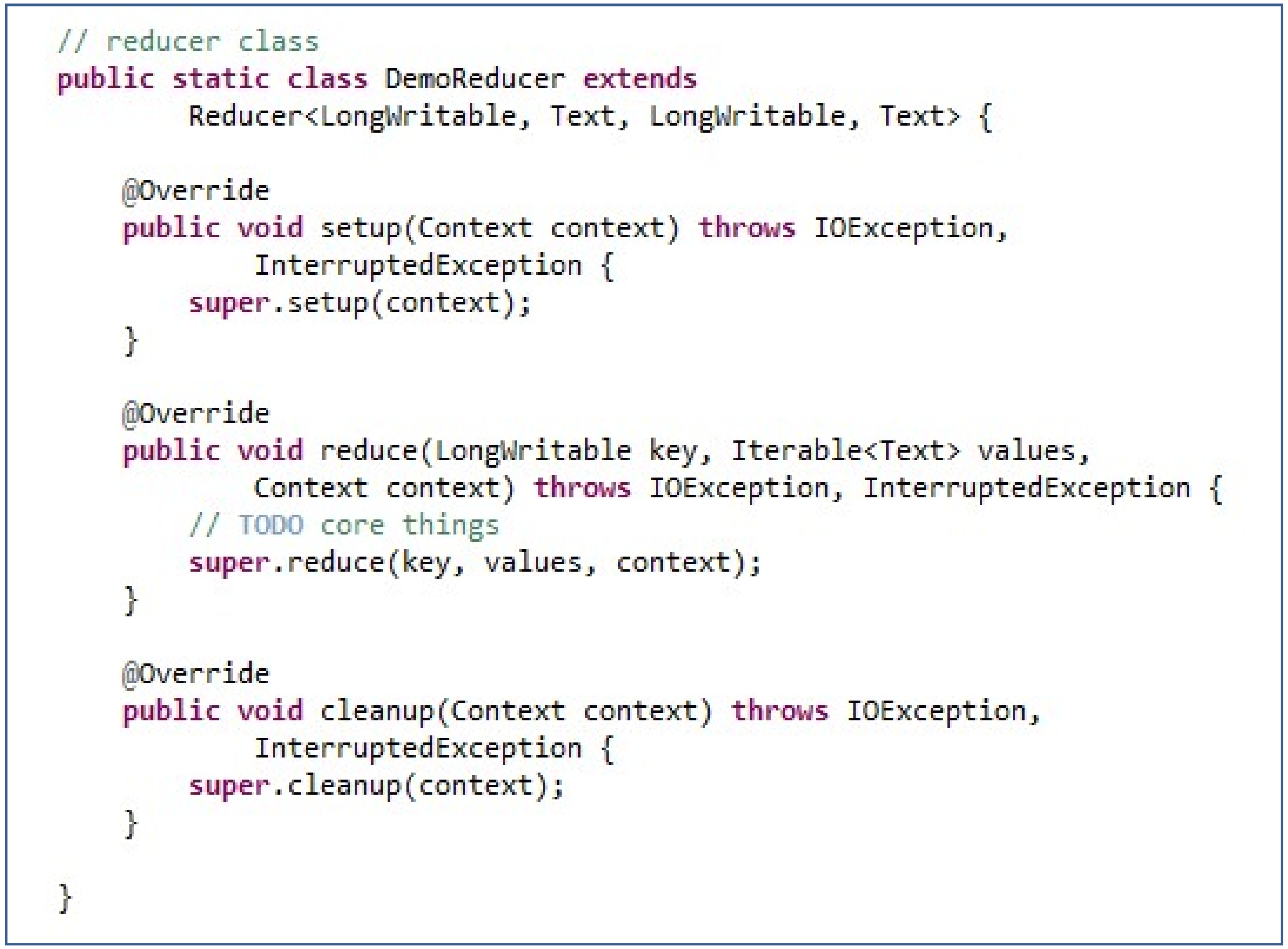
Called once in the life cycle of a Mapper object: before any calls to `map()`

Called once for each key/value pair that appears in the input split

Preserving State



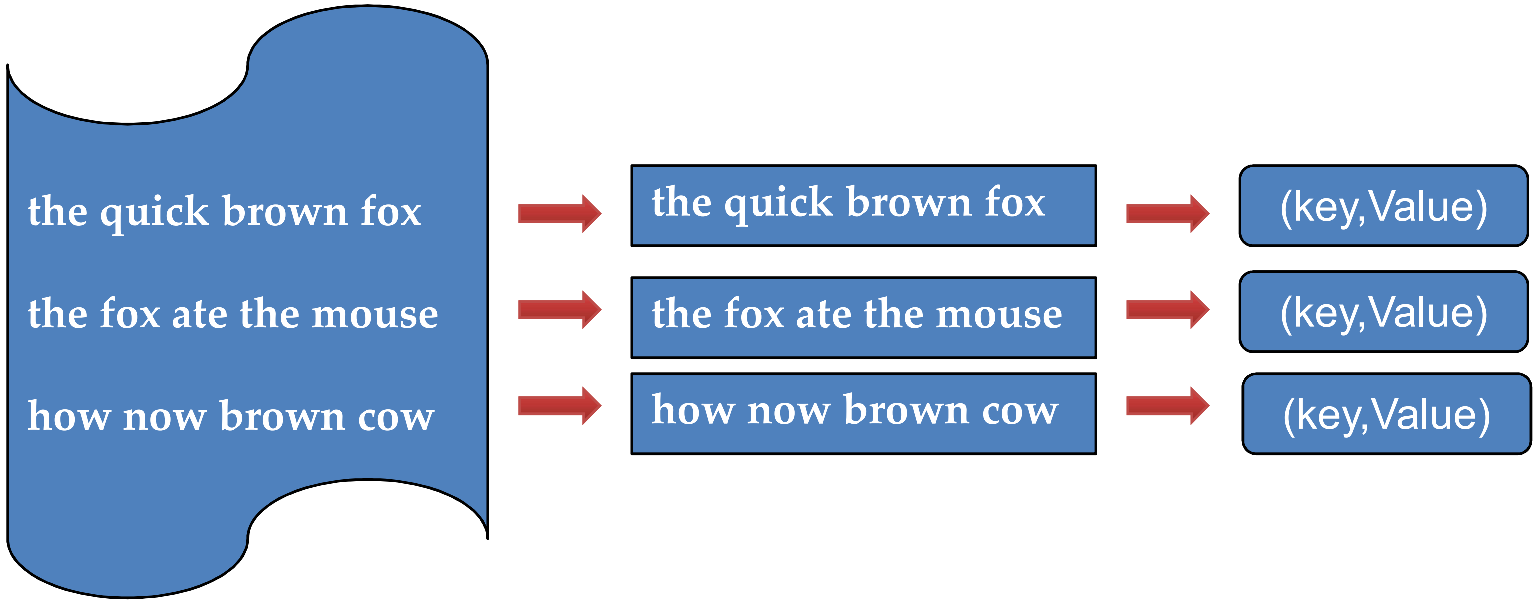
Reducer Class



```
// reducer class
public static class DemoReducer extends
        Reducer<LongWritable, Text, LongWritable, Text> {

    @Override
    public void setup(Context context) throws IOException,
            InterruptedException {
        super.setup(context);
    }

    @Override
    public void reduce(LongWritable key, Iterable<Text> values,
            Context context) throws IOException, InterruptedException {
        // TODO core things
        super.reduce(key, values, context);
    }

    @Override
    public void cleanup(Context context) throws IOException,
            InterruptedException {
        super.cleanup(context);
    }

}
```

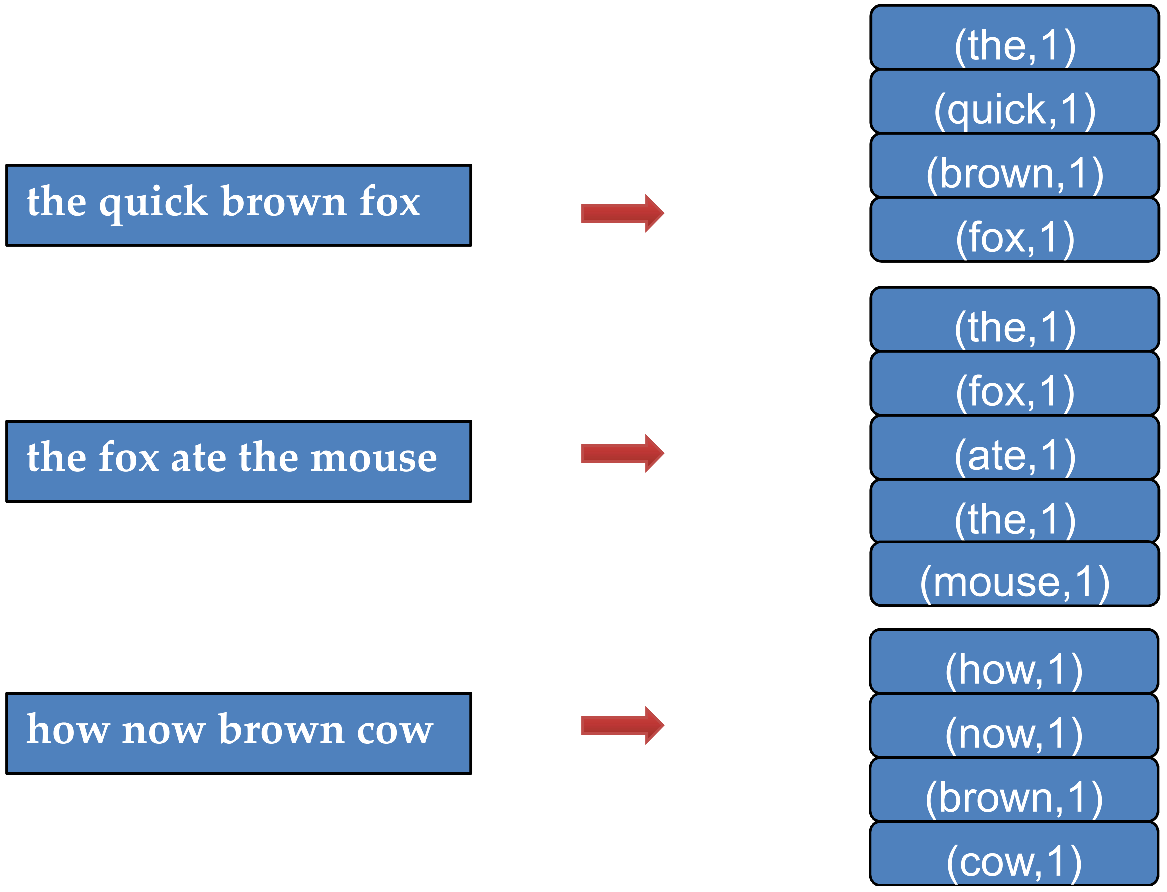
MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 1: 自动对文本进行分割



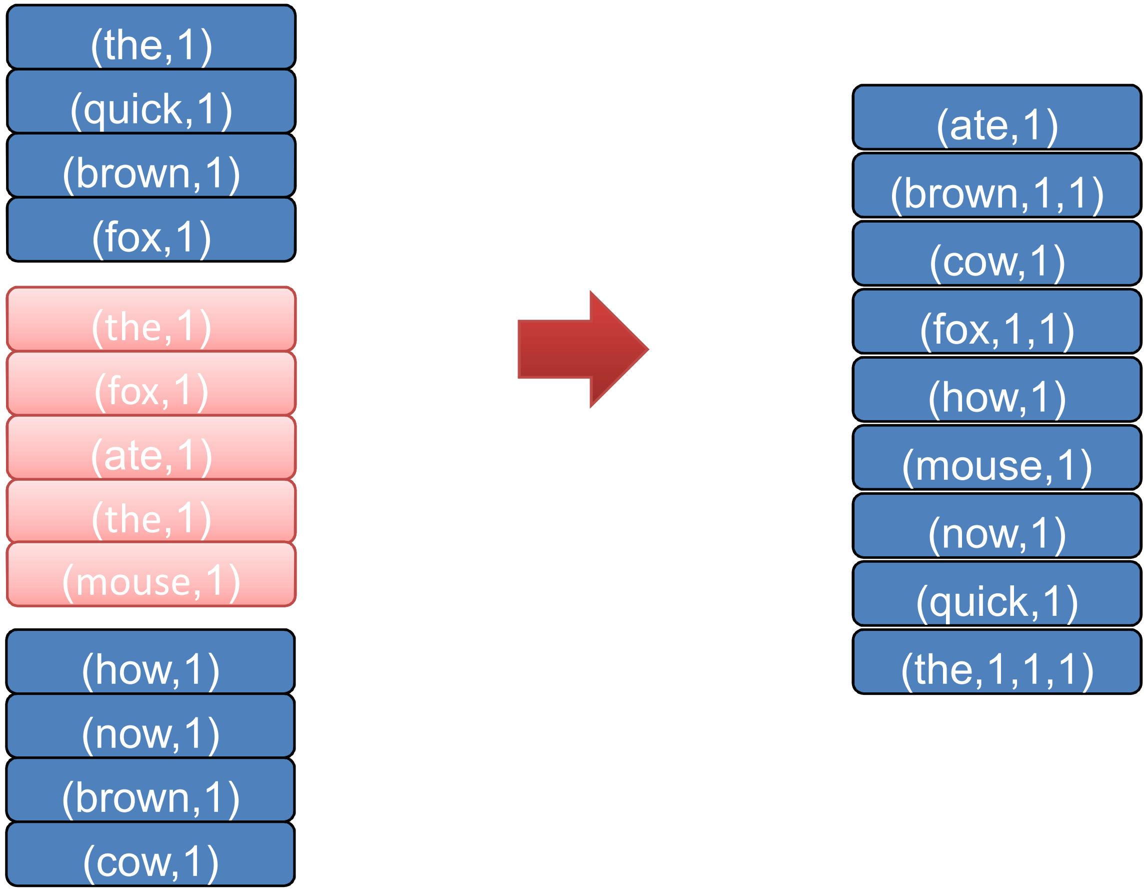
MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 2:在分割之后的每一对<key,value>进行用户定义的Map进行处理，再生成新的<key,value>对



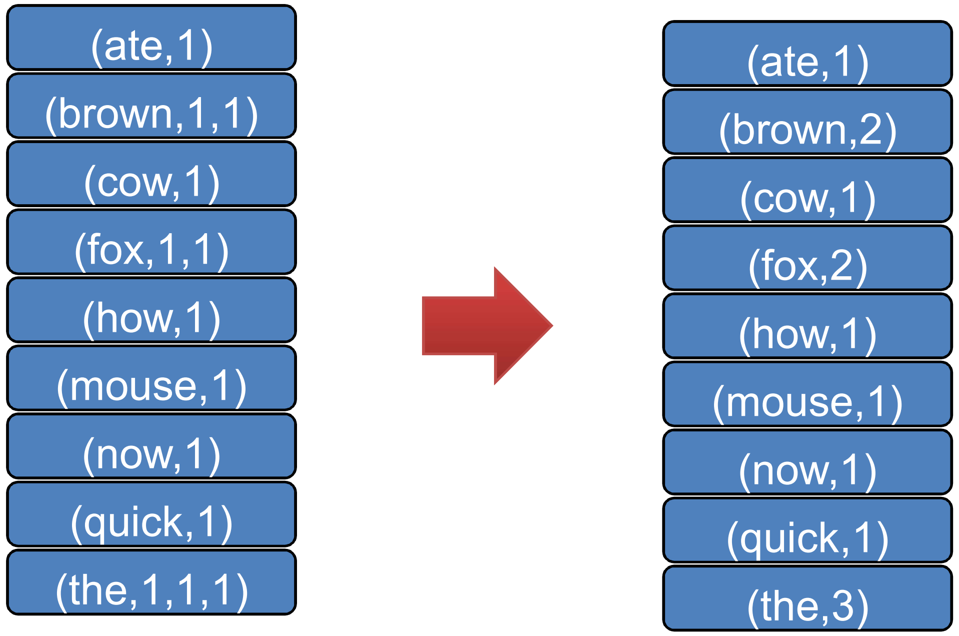
MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 3:对Map输出的结果进行Shuffle

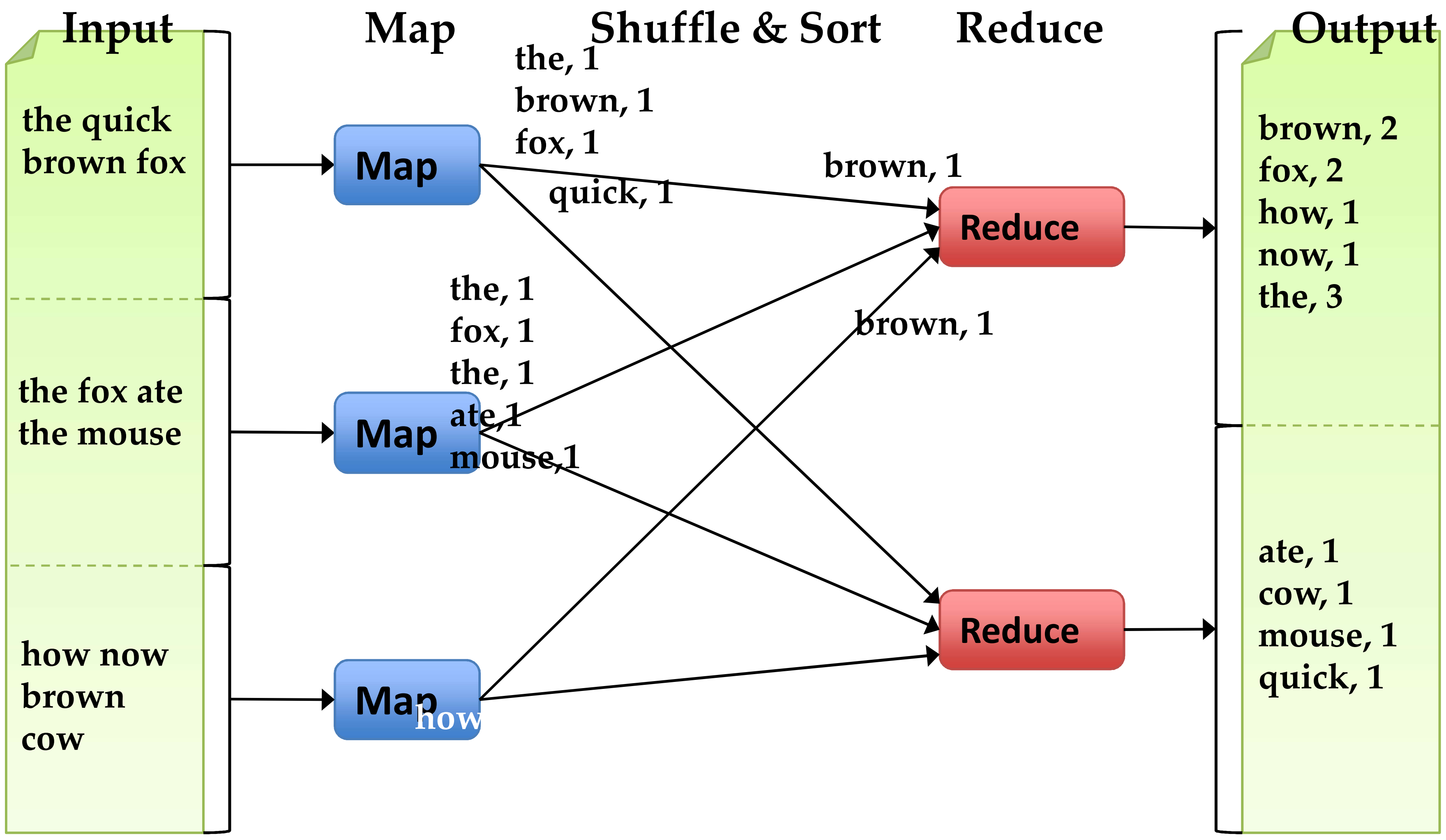


MapReduce示例：单词计数

- 使用MapReduce求解该问题
  - Step 4:通过Reduce操作生成最后结果



Word Count Execution



MapReduce WordCount.java

```
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);
     }
   }
 }
```

Reduce code in WordCount.java

```
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, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

The driver to set things up and start

```
//   Usage: wordcount <in> <out>
 public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
   String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
   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);
  }
}
```

Word Count: Version 1

```
class Mapper
    method Map(docid $a$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$                ▷ Tally counts for entire document
        for all term $t \in H$ do
            Emit(term $t$, count $H\{t\}$)
```

Are combiners still needed?

Word Count: Version 2

```
class MAPPER
    method INITIALIZE
        $H \leftarrow$ new ASSOCIATIVEARRAY
    method MAP(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$                ▷ Tally counts *across* documents
    method CLOSE
        for all term $t \in H$ do
            EMIT(term $t$, count $H\{t\}$)
```

Key idea: preserve state across input key-value pairs!

Are combiners still needed?

Mean

- 实例描述
  - 对输入文件中数据进行计算学生平均成绩。输入文件中的每行内容均为一个学生的姓名和他相应的成绩，如果有多门学科，则每门学科为一个文件。要求在输出中每行有两个间隔的数据，其中，第一个代表学生的姓名，第二个代表其平均成绩。
- 实际作用
  - 计算平均值

平均成绩

样例输入

1）math：

张三	88
李四	99
王五	66
赵六	77

2）china：

张三	78
李四	89
王五	96
赵六	67

3）english：

张三	80
李四	82
王五	84
赵六	86

样例输出

张三	82
李四	90
王五	82
赵六	76

IP地址，上网时长 $\Rightarrow$ IP地址，平均上网时长

Computing the Mean: Version 1

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, integer $r$)

class Reducer
    method Reduce(string $t$, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Why can't we use reducer as combiner?

Computing the Mean: Version 2

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, integer $r$)

class Combiner
    method Combine(string $t$, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        Emit(string $t$, pair $(sum, cnt)$)                ▷ Separate sum and count

class Reducer
    method Reduce(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Why doesn't this work?

Computing the Mean: Version 3

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, pair $(r, 1)$)

class Combiner
    method Combine(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        Emit(string $t$, pair $(sum, cnt)$)

class Reducer
    method Reduce(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, pair $(r_{avg}, cnt)$)
```

- Reducer input key/value type 要和mapper key/value type相匹配
- Combiner input & output type 要和 mapper output key/value type相匹配
- Combiners 是优化器, 必须谨慎使用,不能影响程序的正确性

Fixed**?**

Computing the Mean: Version 4

```
class Mapper
    method Initialize
        $S \leftarrow$ new AssociativeArray
        $C \leftarrow$ new AssociativeArray
    method Map(string $t$, integer $r$)
        $S\{t\} \leftarrow S\{t\} + r$
        $C\{t\} \leftarrow C\{t\} + 1$
    method Close
        for all term $t \in S$ do
            Emit(term $t$, pair $(S\{t\}, C\{t\})$)
```

Are combiners still needed?

沃尔玛：啤酒加尿布

- 沃尔玛基于4PB销售终端的交易数据进行关联规则挖掘，利用大数据扩大销售
- 为了能够准确了解顾客在其门店的购买习惯，利对其顾客的购物行为进行购物篮分析，想知道顾客经常一起购买的商品有哪些
- 在美国，一些年轻父亲下班后经常要到超市去买婴儿尿布，而他们中有30%到40%的人同时也为自己买一些啤酒

购物篮模型（The Market-Basket Model)

- 项（items）
  - e.g., things sold in a supermarket
- 购物篮（baskets）
  - 每个购物篮由多个项组成的集合
- 提取关联规则（association rules）
  - People who bought {x,y,z} tend to buy {v,w}

Input:

TID	*Items*
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output:

Rules Discovered:

{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

Applications

- Baskets = 购物篮; Items = 商品;
- Baskets = 文档; Items = 词;
- Baskets = 病人; Items = 药物&副作用;

频繁项集（Frequent Itemsets）

- 目标: 寻找经常在一个购物篮中出现的项集
- 假设***I*** 是一个项集， ***I*** 的支持度（ *Support* for itemset ***I*** ） 是指包含***I 的购物篮数目。***
- 给定一个支持度阈值 **s**,那么支持度大于等于 ***s*** 的项集称为频繁项集（frequent itemsets ） 。

TID	***Items***
1	**Bread, Coke, Milk**
2	**Beer, Bread**
3	**Beer, Coke, Diaper, Milk**
4	**Beer, Bread, Diaper, Milk**
5	**Coke, Diaper, Milk**

Support of {Beer, Bread} = 2

For Example

POS Transactions

Customer	Items Purchased
1	**Bread, Coke, Milk**
2	**Beer, Bread**
3	**Beer, Coke, Diaper, Milk**
4	**Beer, Bread, Diaper, Milk**
5	**Coke, Diaper, Milk**

Co-occurrence of Products

	Bread	Coke	Milk	Beer	Diaper
Bread	3	1	2	2	1
Coke	1	3	3	1	2
Milk	2	3	4	2	3
Beer	2	1	2	3	2
Diaper	1	2	3	2	3

构建单词共现矩阵算法

- word co-occurrence matrix
  - 语料库的单词共现矩阵是一个二维 N×N矩阵
  - N是语料库的词汇量（即，不同单词的数目）
  - 矩阵元素M[i, j] 代表单词W[i] 与单词W [j]在一定范围内同时出现的次数（一个语句中，一个段落中，一篇文档中，或文本串中一个宽度为M个单词的窗口中，这些都依具体问题而定)

The Problem of Lots of Data

- 快餐店…假设菜单上有100 样菜品
  - 三元素项集的个数： 161,700 !
- Supermarket…假设10,000 样商品
  - 50 million 双元素项集
- Building the words co-occurrence matrix
  - 如果内存足够大，把整个矩阵放在内存中，矩阵元素的计算会非常简单
  - 实际上，web-scale的文档的词汇量可能有数十万，甚至数亿
  - 共现矩阵的空间开销为$O(n^2)$
  - 简单地在单机上的实现，内存与磁盘之间的换页会使任务的执行十分缓慢

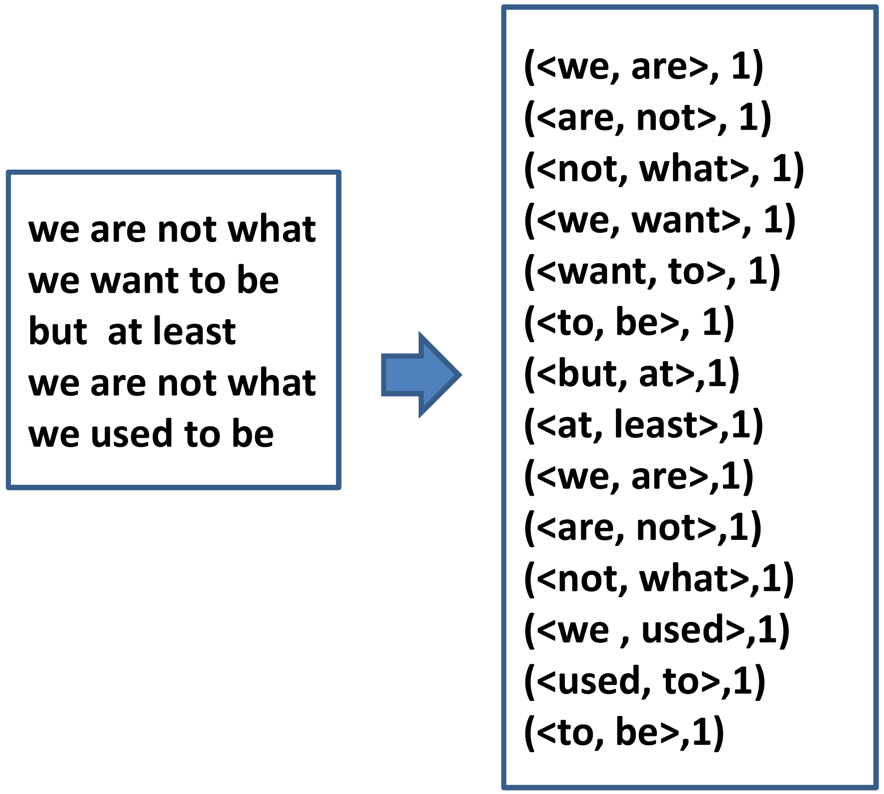
构建单词共现矩阵算法

- A simple “pairs” approach example
  - 语料

we are not what
we want to be
but at least
we are not what
we used to be

构建单词共现矩阵算法

- A simple “Pairs” approach example (cont.)
  - after map

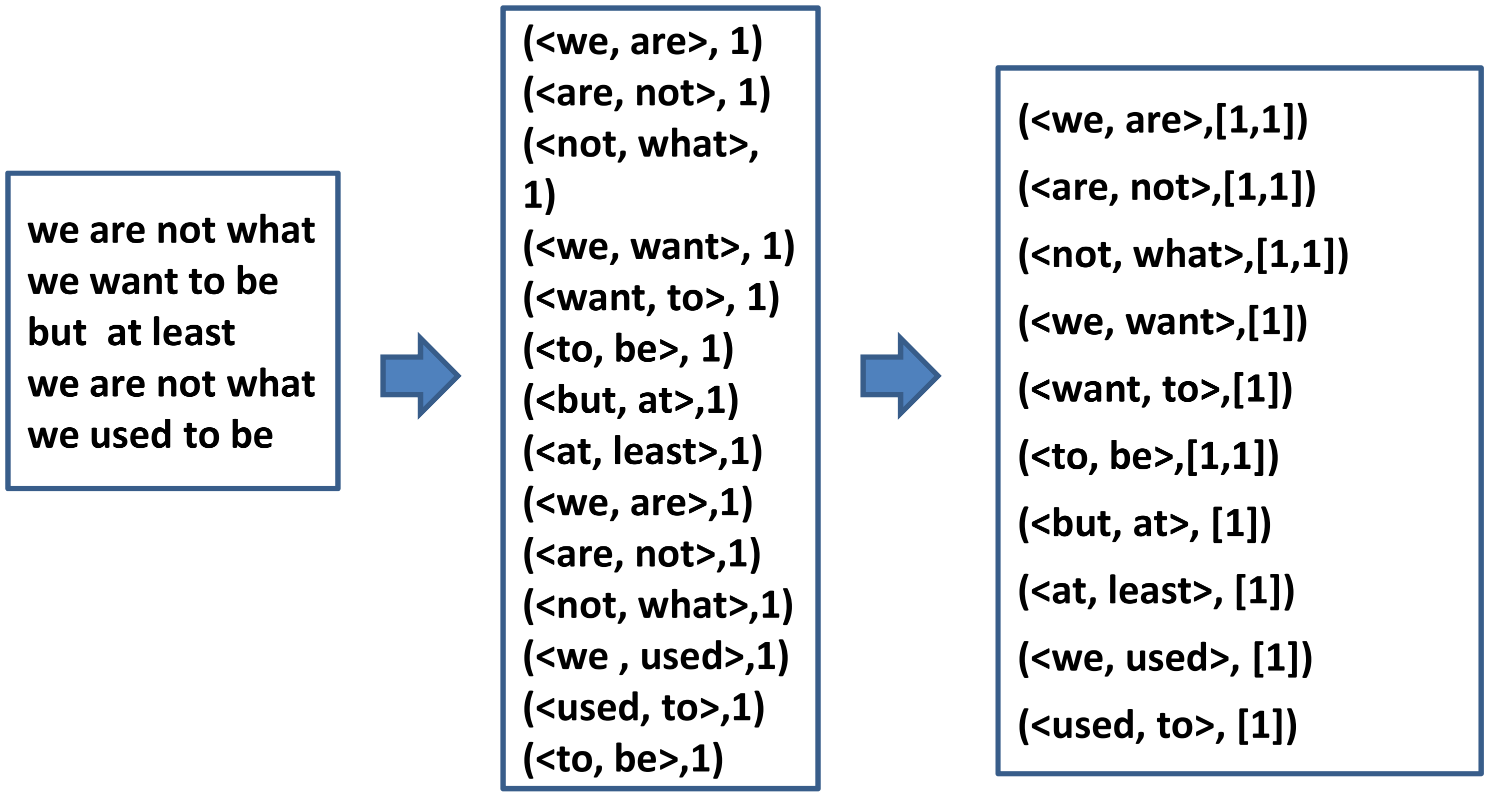


```
we are not what
we want to be
but  at least
we are not what
we used to be
```

```
(<we, are>, 1)
(<are, not>, 1)
(<not, what>, 1)
(<we, want>, 1)
(<want, to>, 1)
(<to, be>, 1)
(<but, at>,1)
(<at, least>,1)
(<we, are>,1)
(<are, not>,1)
(<not, what>,1)
(<we , used>,1)
(<used, to>,1)
(<to, be>,1)
```

构建单词共现矩阵算法

- A simple “Pairs” approach example (cont.)
  - after shuffle and sort



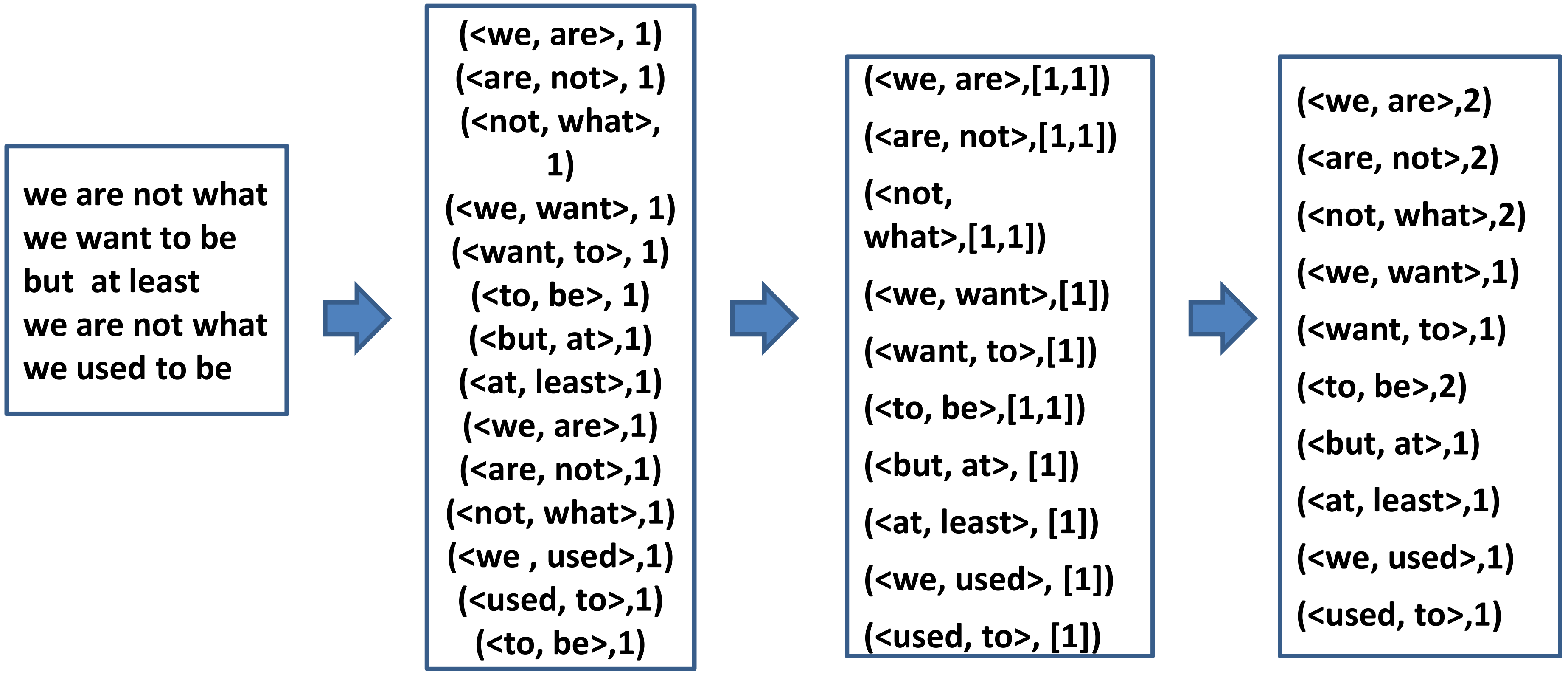
we are not what
we want to be
but at least
we are not what
we used to be

(<we, are>, 1)
(<are, not>, 1)
(<not, what>, 1)
(<we, want>, 1)
(<want, to>, 1)
(<to, be>, 1)
(<but, at>,1)
(<at, least>,1)
(<we, are>,1)
(<are, not>,1)
(<not, what>,1)
(<we , used>,1)
(<used, to>,1)
(<to, be>,1)

(<we, are>,[1,1])
(<are, not>,[1,1])
(<not, what>,[1,1])
(<we, want>,[1])
(<want, to>,[1])
(<to, be>,[1,1])
(<but, at>, [1])
(<at, least>, [1])
(<we, used>, [1])
(<used, to>, [1])

构建单词共现矩阵算法

- A simple "Pairs" approach example (cont.)
  - after reduce



we are not what
we want to be
but at least
we are not what
we used to be

(<we, are>, 1)
(<are, not>, 1)
(<not, what>, 1)
(<we, want>, 1)
(<want, to>, 1)
(<to, be>, 1)
(<but, at>,1)
(<at, least>,1)
(<we, are>,1)
(<are, not>,1)
(<not, what>,1)
(<we , used>,1)
(<used, to>,1)
(<to, be>,1)

(<we, are>,[1,1])
(<are, not>,[1,1])
(<not, what>,[1,1])
(<we, want>,[1])
(<want, to>,[1])
(<to, be>,[1,1])
(<but, at>, [1])
(<at, least>, [1])
(<we, used>, [1])
(<used, to>, [1])

(<we, are>,2)
(<are, not>,2)
(<not, what>,2)
(<we, want>,1)
(<want, to>,1)
(<to, be>,2)
(<but, at>,1)
(<at, least>,1)
(<we, used>,1)
(<used, to>,1)

构建单词共现矩阵算法

	we	are	not	what	want	to	be	but	at	least	used
we		2			1						1
are	2		2								
not		2		2							
what			2								
want	1					1					
to					1		1				1
be						1					
but									1		
at								1			
least											
used	1					1					

figure: the co-occurrence matrix

方法一: “Pairs”

```
map(docid a, doc d):
    foreach term w in d:
        foreach term u in d:
            EmitIntermediate(pair(w,u),1)
```

共现矩阵中的元素

```
reduce(pair p, counts [c1, c2, ..]
    s = 0;
    foreach c in counts:
        s += c;
    Emit(pair p, count s);
```

- 定义包括自然键和自然值的复合键
- 定义WordPair类,实现WritableComparable接口
  - Hashcode()
  - CompareTo()

“Pairs”分析

- 优点
  - 易于实现，便于理解
- 缺点
  - 需要进行sort和shuffle的pairs数量多
  - combiners难以起作用

方法二:“Stripes”

- 思想：使用关联数组(associative arrays)
- 每个 mapper 读取一个句子:
  - 生成所有的 co-occurring term pairs
  - 对于每一个 term, emit a → { b: countb, c: countc, d: countd … }
- Reducers 对关联数组列表进行求和

  a →[{ b:1, c:2, d:5, e:3, f:2 },{ b:1, d:5, e:3 }, { b:1, c:2, d:2, f:2 },{ b:2, c:2, d:7, e:3, f:2 }]

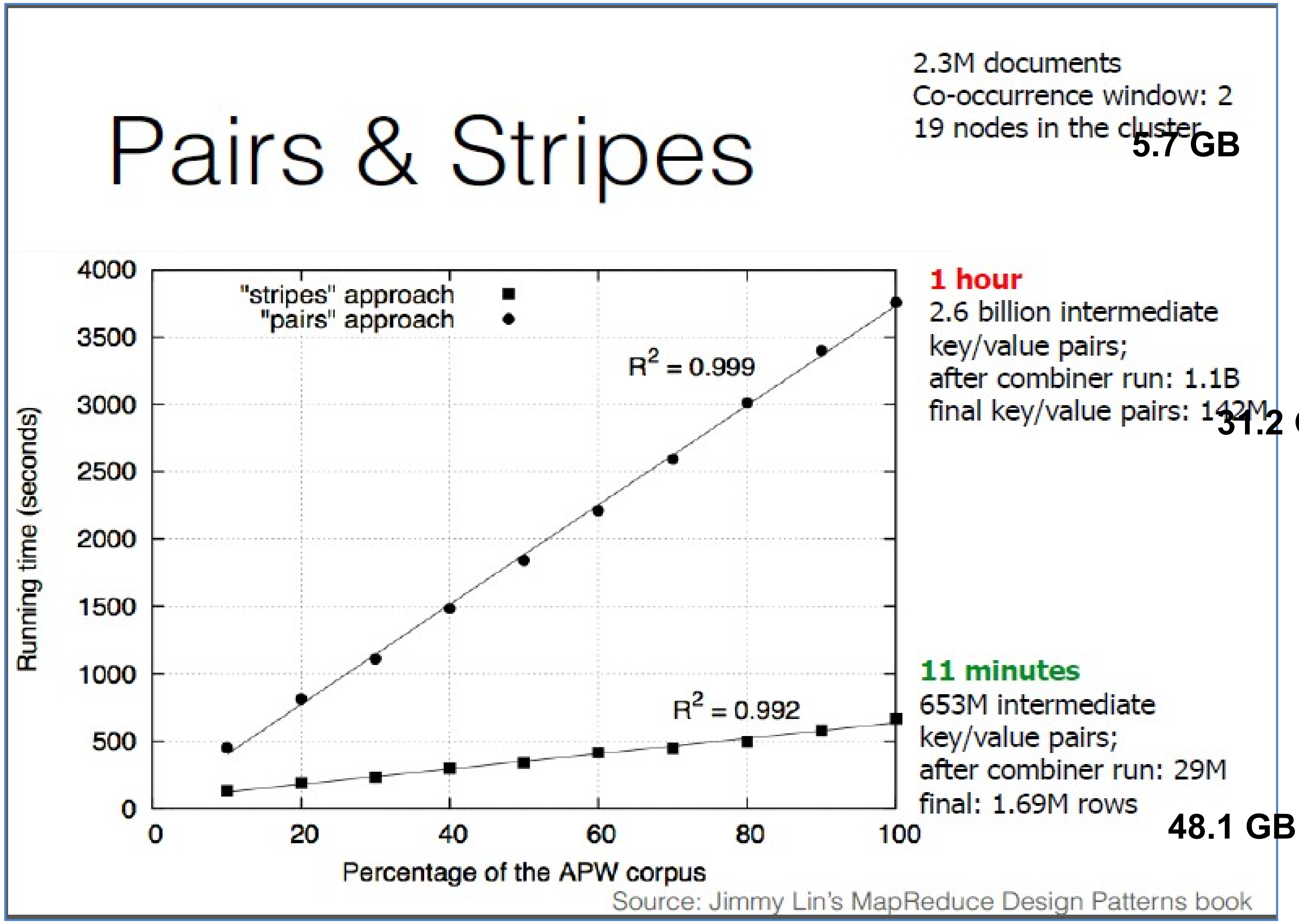
  a → [{b:5,c:6,d:19,e:9,f:6}]

"Stripes"

```
map(docid a, doc d):
    foreach term w in d:
        H = new associative_array;
        foreach term u in d:
            H{u}=H{u}+1;
        Emit(term w, Stripe H);

reduce(term w, stripes [H1,H2,H3,..])
    F = new associative_array;
    foreach H in stripes [H1,H2,H3,..]
        sum(F,H);
    Emit(term w, stripe F)
```

“Stripes”分析

- 优点
  - 减少了key—value pairs 进行sort shuffle的次数
  - 便于使用combiners
- 缺点
  - 代码实现难度大
  - 重量级内部对象
  - Fundamental limitation in terms of size of event space



Pairs & Stripes

2.3M documents
Co-occurrence window: 2
19 nodes in the cluster

5.7 GB

1 hour
2.6 billion intermediate key/value pairs;
after combiner run: 1.1B
final key/value pairs: 142M

31.2 GB

11 minutes
653M intermediate key/value pairs;
after combiner run: 29M
final: 1.69M rows

48.1 GB

Source: Jimmy Lin's MapReduce Design Patterns book

Figure 3: Running time of the pairs and stripes algorithms for computing word cooccurrence matrices on dierent fractions of the APW corpus. These experiments were performed on a Hadoop cluster with 19 slaves, each with two single-core processors and two disks.

Stripes

2.3M documents
Co-occurrence window: 2
Scaling up (on EC2)

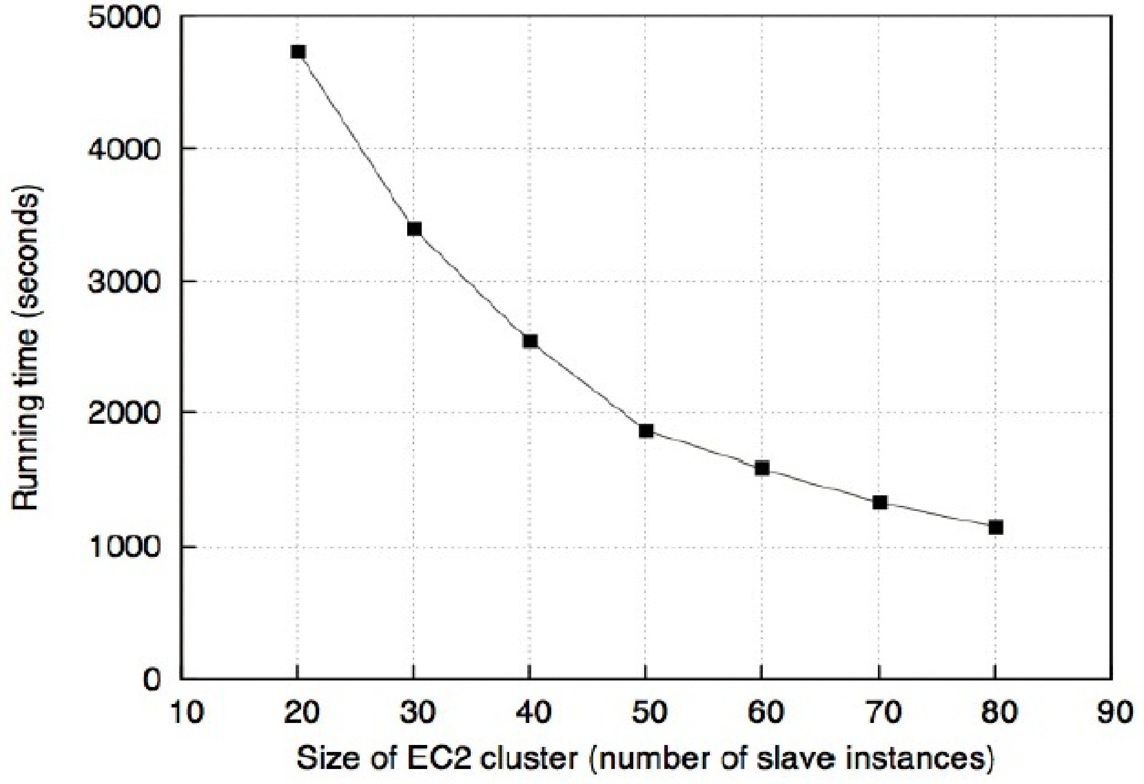


Figure 3.11: Running time of the stripes algorithm on the APW corpus with Hadoop clusters of different sizes from EC2

Order Inversion(反转排序)

更进一步：计算单词共现率

we are not what
we want to be
but at least
we are not what
we used to be

$$f(w_j \mid w_i) = \frac{\overbrace{N(w_i, w_j)}}{\sum_{w'} N(w_i, w')}$$

$f\,$**(used | we)= 1/(2+1+1) =1/4**

	we	are	not	what	want	to	be	but	at	least	used
we		2			1						1
are	2		2								
not		2		2							
what			2								
want	1					1					
to					1		1				1
be						1					
but									1		
at								1			
least											
used	1					1					

“Stripes”

```
map(docid a, doc d):
    foreach term w in d:
        H = new associative_array;
        foreach term u in d:
            H{u}=H{u}+1;
        Emit(term w, Stripe H);

reduce(term w, stripes [H1,H2,H3,..])
    F = new associative_array;
    foreach H in stripes [H1,H2,H3,..]
        sum(F,H);
    Emit(term w, stripe F)
```

增加代码计算单词同现率

“Pairs”

```
map(docid a, doc d):
    foreach term w in d:
        foreach term u in d:
            EmitIntermediate(pair(w,u),1)

reduce(pair p, counts [c1, c2, ..]
    s = 0;
    foreach c in counts:
        s += c;
    Emit(pair p, count s);
```

增加使用**associative_array**，缓存所有与**p.left**同现的单词个数

f(B|A): “Pairs”

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

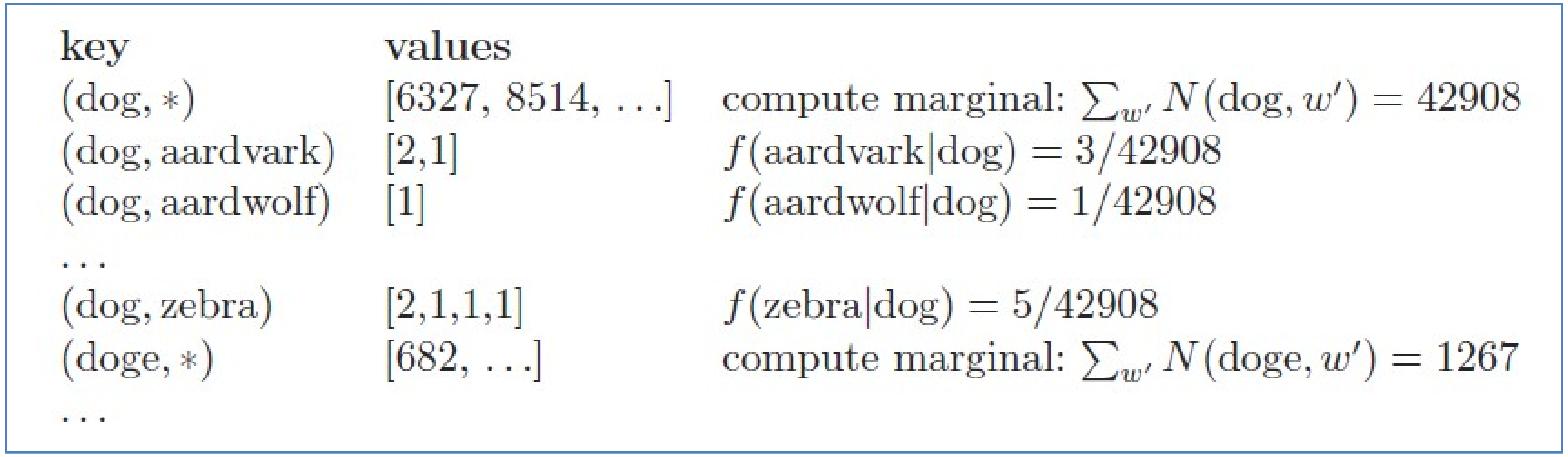
$(a, b_1) \rightarrow 3$		$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$\Rightarrow$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$		$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$		$(a, b_4) \rightarrow 1 / 32$
…		…

- For this to work:
  - Must emit extra (a, *) for every b_n in mapper
  - Must make sure all a’ s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

计算单词共现率(Pairs)

```
map(docid a, doc d):
        foreach term w in d:
                foreach term u in d:
                        EmitIntermediate(pair(w,u),1)
                        EmitIntermediate(pair(w,*),1)

reduce(pair p, counts [c1, c2, ..])
        s = 0;
        foreach c in counts:
                s += c;
        if(p.right==*)
                marginal=s;//keep marginal across reduce calls
        else
                Emit(pair p, s/marginal);
```

示例



key	values	
(dog, $*$)	[6327, 8514, $\ldots$]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
$\ldots$		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra}\|\text{dog}) = 5/42908$
(doge, $*$)	[682, $\ldots$]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$
$\ldots$		

Key Points: convert sequencing of computation into a sorting problem

Order Inversion小结

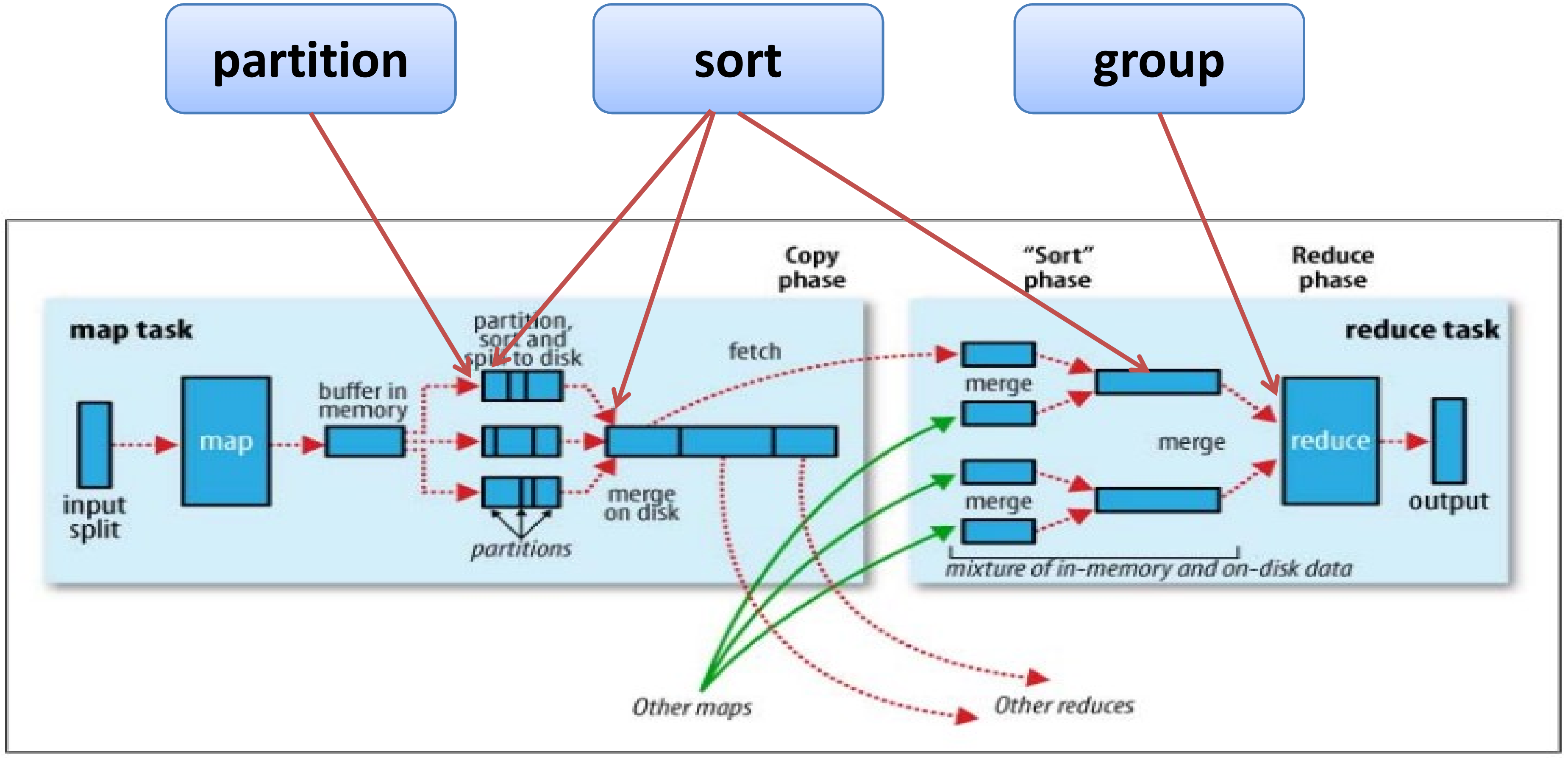
- 在Mapper阶段对于每一个共生词对输出一个特别的key-value对，其中的value将用于形成该共生词对的边界属性
- 控制mapreduce 的Sort过程，使得在reduce阶段上一步输出的key-value对将优于它对应的各个共生词对之前进行处理和计算
- 自定义一个 partitioner 以保障具有相同左边单词的每一个共生词对对应的key-value输出到同一个Reducer
- 在Reduce阶段对于具有相同左边单词的每一个共生词对，首先计算“特别”key-value的边界属性，然后与随后计算得到的各个value值相除后得最终结果

```
 Job job = Job.getInstance(getConf(),"DemoMapReduce");
 job.setJarByClass(DemoMapReduce.class);

 // 1) set input
job.setInputFormat(TextInputFormat.class);
FileInputFormat.addInputPath(conf, new Path(args[0]));

// 2) set mapper
 job.setMapperClass(Mapper.class);
 job.setMapOutputKeyClass(LongWritable.class);
 job.setMapOutputValueClass(Text.class);
 //3
  job.setPartitionerClass(HashPartitioner.class);
// 4) set reducer
 job.setReducerClass(Reducer.class);
 job.setNumReduceTasks(1);
 job.setOutputKeyClass(LongWritable.class);
 job.setOutputValueClass(Text.class);

// 5) set output
 job.setOutputFormat(TextOutputFormat.class);
 FileOutputFormat.setOutputPath(conf, new Path(args[1]));
```

```
public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf(),"DemoMapReduce");
        // run class
        job.setJarByClass(DemoMapReduce.class);
       // 1) set input
        job.setInputFormatClass(TextInputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
       // 2) set mapper
        job.setMapperClass(ModuleMapper.class);
        job.setMapOutputKeyClass(LongWritable.class);
        job.setMapOutputValueClass(Text.class);
       // 3) set shuffle
       // job.setPartitionerClass(HashPartitioner.class);
      // job.setSortComparatorClass(LongWritable.Comparator.class);
      // job.setCombinerClass(ModuleReducer.class);
     // job.setGroupingComparatorClass(LongWritable.Comparator.class);
     // 4) set reducer
       job.setReducerClass(ModuleReducer.class);
       job.setOutputKeyClass(LongWritable.class);
       job.setOutputValueClass(Text.class);
      // 5) set output
      job.setOutputFormatClass(TextOutputFormat.class);
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
     // submit job
      boolean isSuccess = job.waitForCompletion(true);
  }
```

自定义shuffle

Map&Shuffle&Reduce



提纲

- MapReduce Programming Model
- WordCount等几个例子
- Hadoop的 I/O操作

数据完整性

- 使用校验和机制来保障数据的完整性
  - Use CRC32
- 文件创建
  - 客户端计算校验和 per 512 byte
  - DataNode 存储校验和
- 文件存取
  - 客户端从DataNode获取数据和校验和
  - 如果校验失败，从其他副本节点重新获取

压缩

- 为什么使用压缩?
  - 减少存储文件所需要的磁盘空间
  - 加速数据在网络和磁盘上的传输
- 压缩 & 切分

Compression	Size (GB)	Compression Time (s)	Decompression Time (s)
None	8.0	-	-
Gzip	1.3	241	72
LZO	2.0	55	35

压缩格式	算法	文件扩展名	可切分
DEFLATE	DEFLATE	.deflate	否
Gzip	DEFLATE	.gz	否
bzip2	bzip2	.bz2	是
LZO	LZO	.lzo	否
LZ4	LZ4	.lz4	否
Snappy	Snappy	.snappy	否

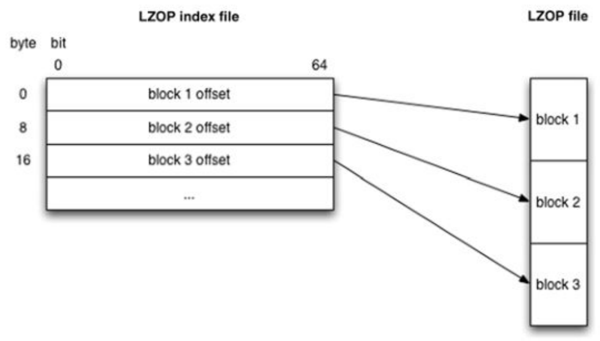


Figure 5.5 LZOP index file is a binary containing a consecutive series of 64-bit numbers.

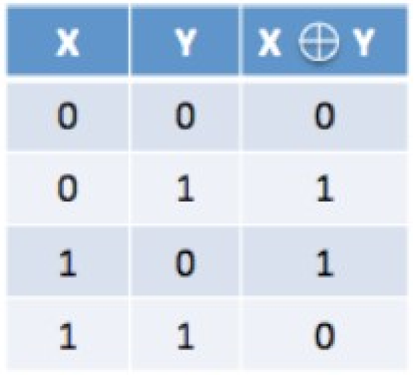
启用数据压缩

- 设置启动压缩功能
  - mapred.compress.map.output
  - mapred.output.compress
- 设置压缩方式
  - mapred.map.output.compression.codec
  - mapred.output.compression.codec

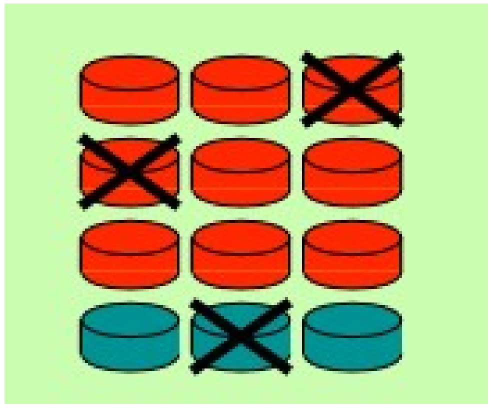
```
Configuration conf = new Configuration();
Conf.setBoolean("mapred.output.compress",true);
Conf.setClass("mapred.output.compress.codec",GzipCodec.cl
ass,CompressionCodec.class);
```

HDFS with Erasure Codes

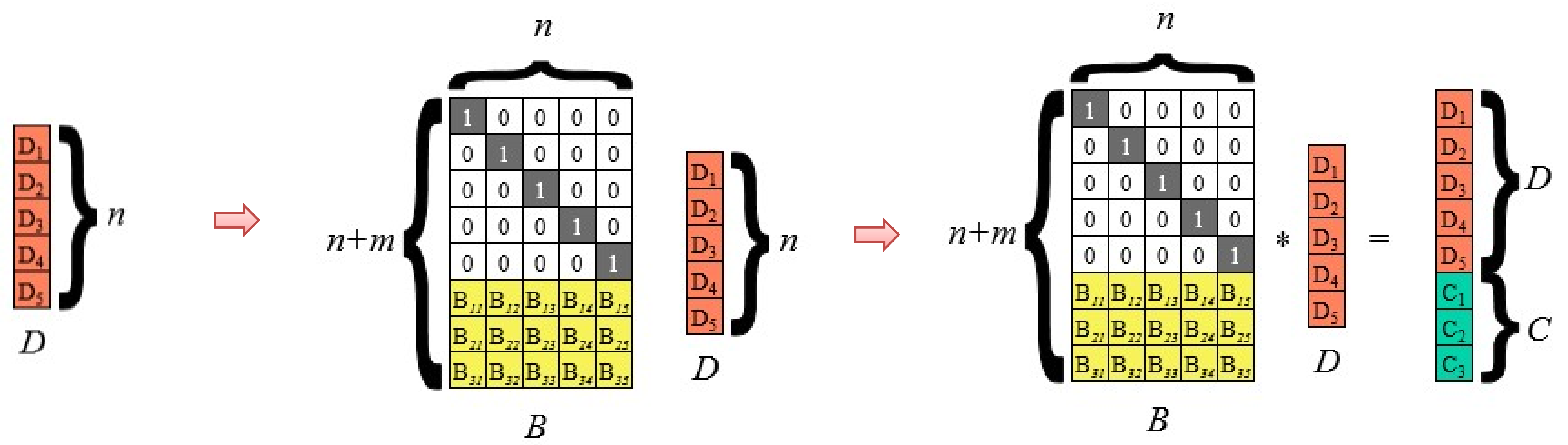
- Replication: 1 1 0 0 — 2 extra bits
- XOR Coding: $1 \oplus 0 = 1$ — 1 extra bit

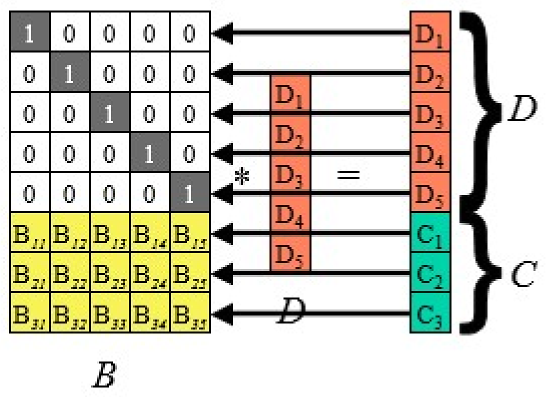


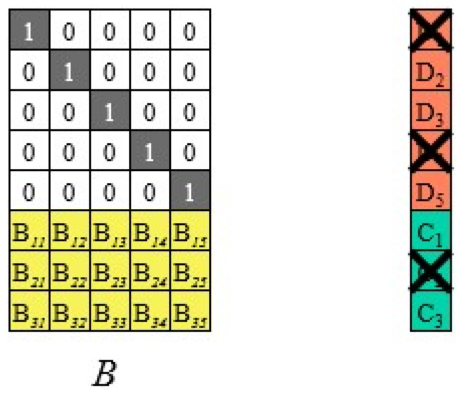
X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



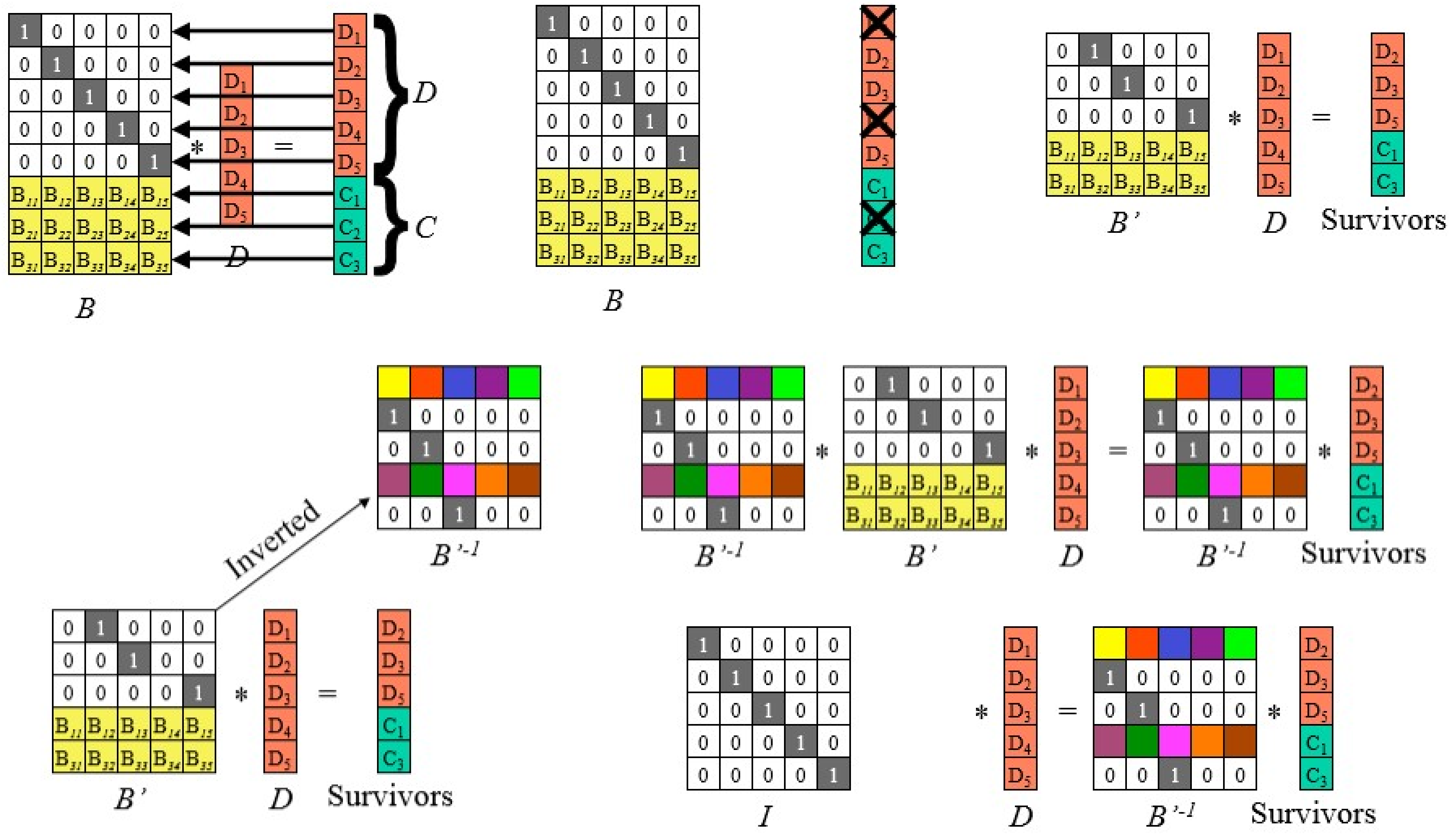
Reed-Solomon Codes







Reed-Solomon Codes

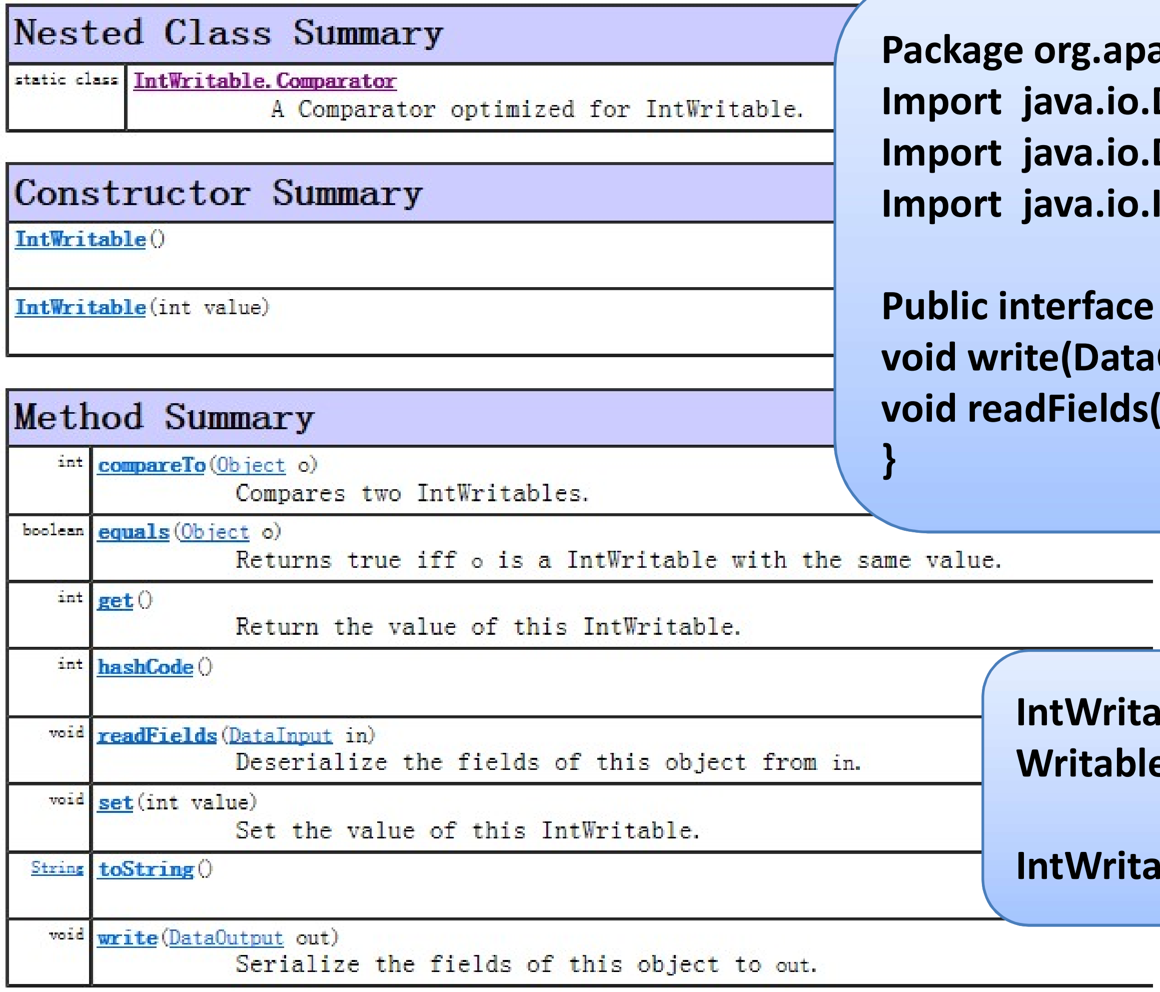


序列化

- 序列化（Serialization)是指将结构化对象转化为字节流以便在网络上传输或写到磁盘进行永久存储的过程
  - 进程间通信;永久存储
- Hadoop 使用自己定义的序列化格式:Writable
  - 紧凑：紧凑格式能充分利用网络带宽
  - 快速：尽量减少序列化和反序列化的性能开销
  - 可扩展：可以透明地读取老格式的数据
  - 支持互操作：可以使用不同给的语言读/写永久存储的数据

序列化

- 对hadoop而言，类型比较比较重要 (Shuffle and Sort phase)
  - 实现WritableComparable接口支持基于键的比较和排序
  - 通过writableComparator来直接比较字节流，避免反序列化



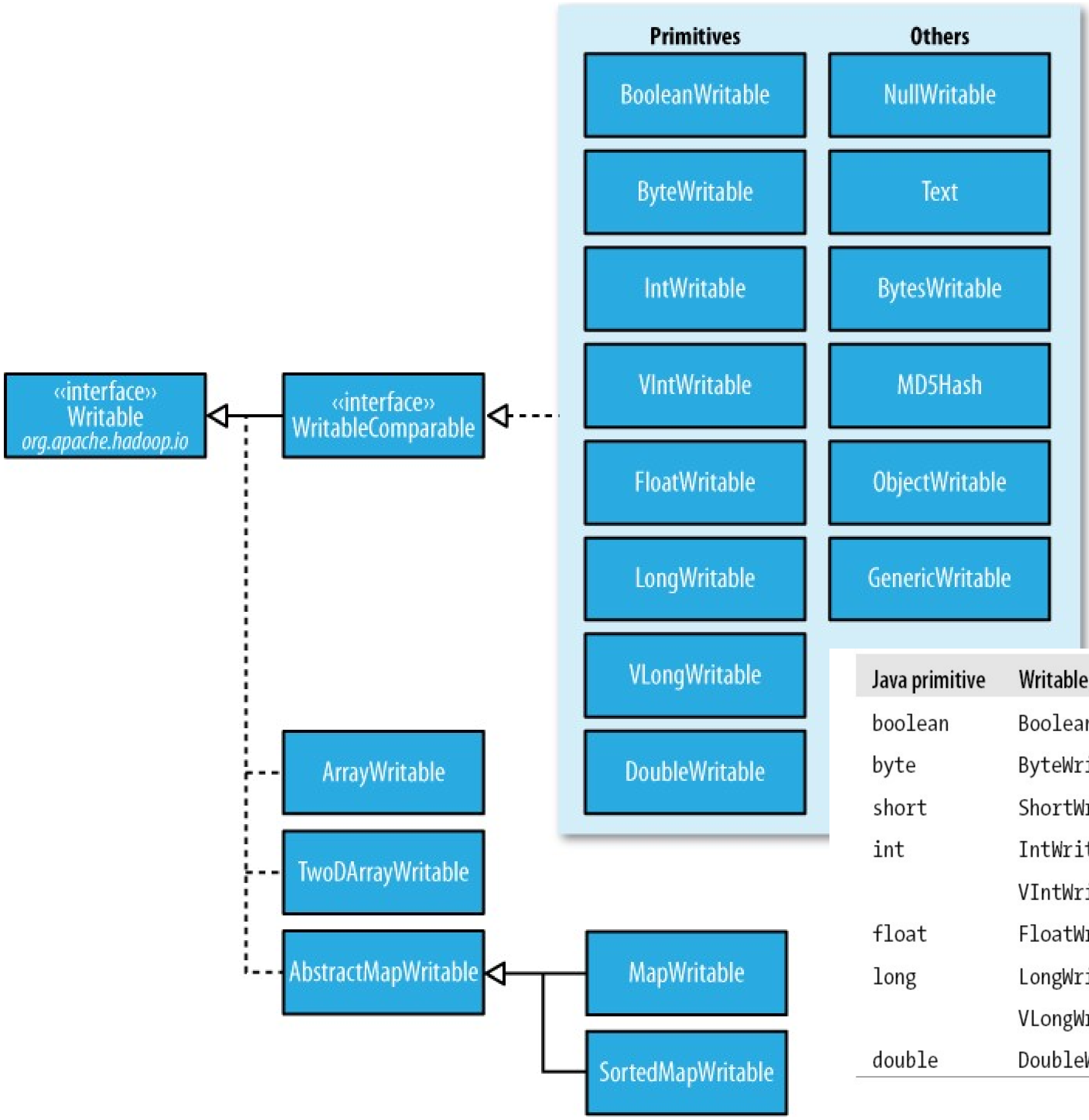
Nested Class Summary	
static class	IntWritable.Comparator A Comparator optimized for IntWritable.

Constructor Summary
IntWritable()
IntWritable(int value)

Method Summary	
int	compareTo(Object o) Compares two IntWritables.
boolean	equals(Object o) Returns true iff o is a IntWritable with the same value.
int	get() Return the value of this IntWritable.
int	hashCode()
void	readFields(DataInput in) Deserialize the fields of this object from in.
void	set(int value) Set the value of this IntWritable.
String	toString()
void	write(DataOutput out) Serialize the fields of this object to out.

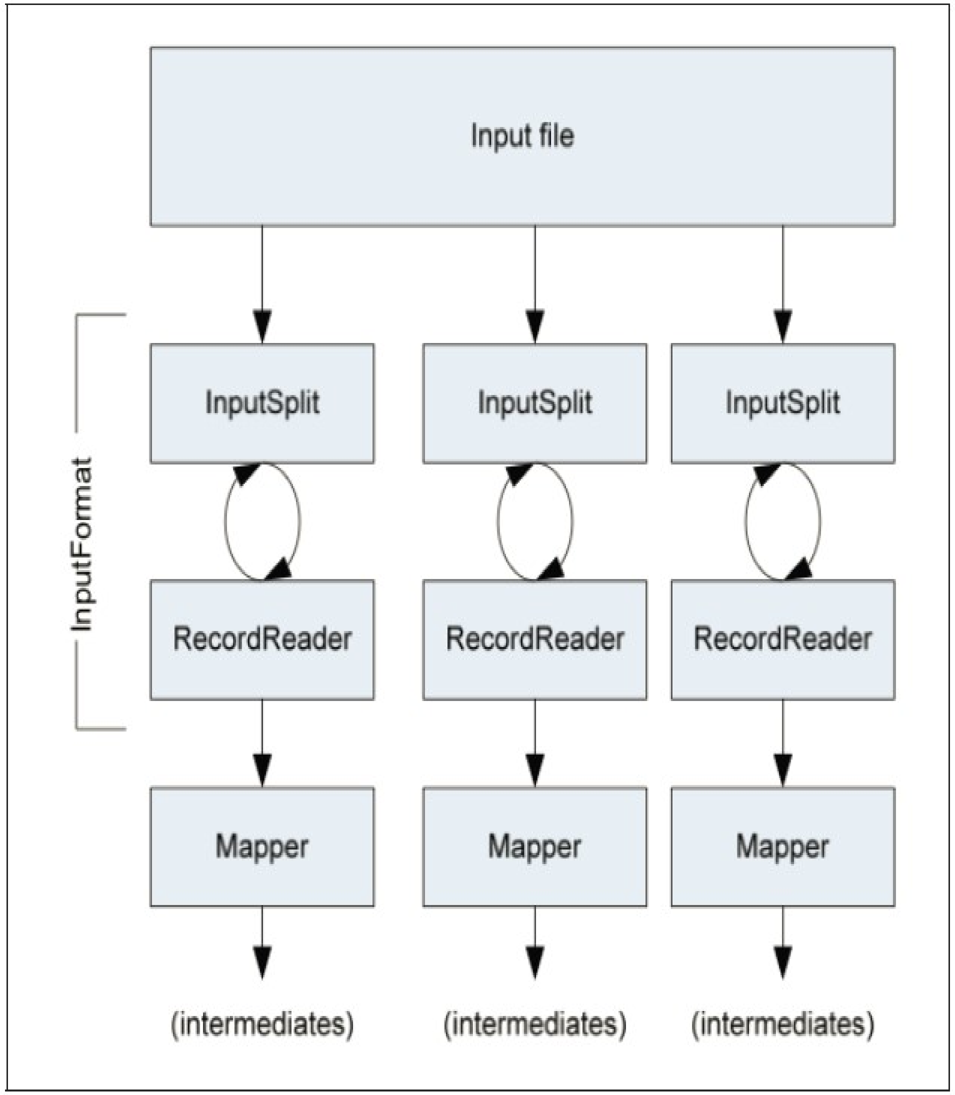
```
Package org.apache.hadoop.io;
Import  java.io.DataOutput;
Import  java.io.DataInput;
Import  java.io.IOException;

Public interface Writable {
void write(DataOutput out) throws IOException;
void readFields(DataInput in) throws IOException;
}
```

```
IntWritable writable = new IntWritable();
Writable.set(163);

IntWritable writable = new IntWritable(163);
```



Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Map输入



- 输入分片表示为InputSplit
  - getLength() : Get the size of the split, so that the input splits can be sorted by size.
  - getLocations(): Get the list of nodes by name where the data for the split would be local.
- 生成记录键值对RecordReader
  - nextKeyValue()： Read the next key, value pair.
  - getCurrentKey()： Get the current key
  - getCurrentValue()： Get the current value.

Split(分片)计算

属性名称	类型	默认值	描述
Mapred.min.split.size	int	1	一个文件分片最小的有效字节数
Mapred.max.split.size	long	Long.MAX-VLAUE	一个文件分片最大的有效字节数
Dfs.block.size	long	64MB	HDFS中块的大小

- SplitSize = max(minimumSize,min(maxiumSize,blockSize))

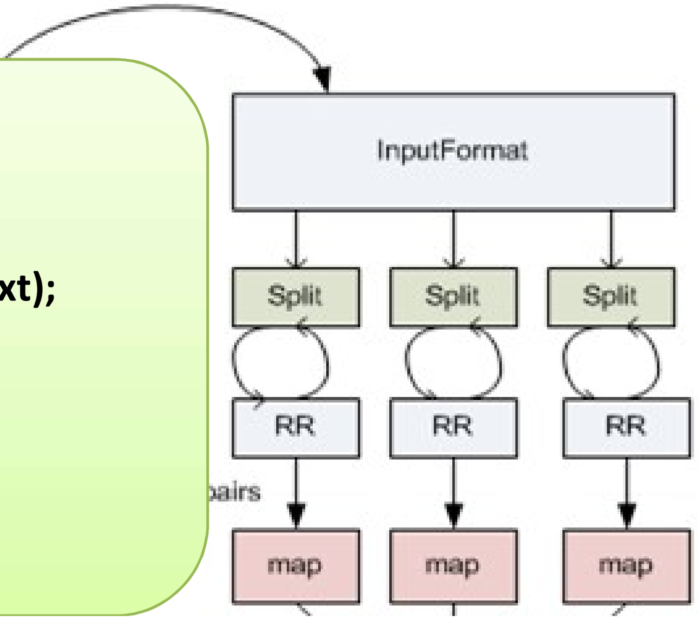
最小分片大小	最大分片大小	块大小	分片大小
1	Long.MAX-VLAUE	64MB	64MB
1	Long.MAX-VLAUE	128MB	128MB
128MB	Long.MAX-VLAUE	64MB	128MB
1	32MB	64MB	32MB

InputFormat类

- 负责产生输入分片并将他们分割成记录
  - getSplits()： Logically split the set of input files for the job.将输入文件切分为map处理所需要的split
  - createRecordReader()： Create a record reader for a given split.创建RecordReader类，它将从一个split生成键值对序列
- 每个InputFormat类都有对应的RecordReader方法
- 通过Job.setInputFormatClass设置文件输入的格式

数据本地化

```
Setup(context);
While (context. nextKeyValue()) {
      map(context.getCurrentKey(),context.getCurrentValue(),context);
}

Cleanup(context);
}
```



InputFormat类

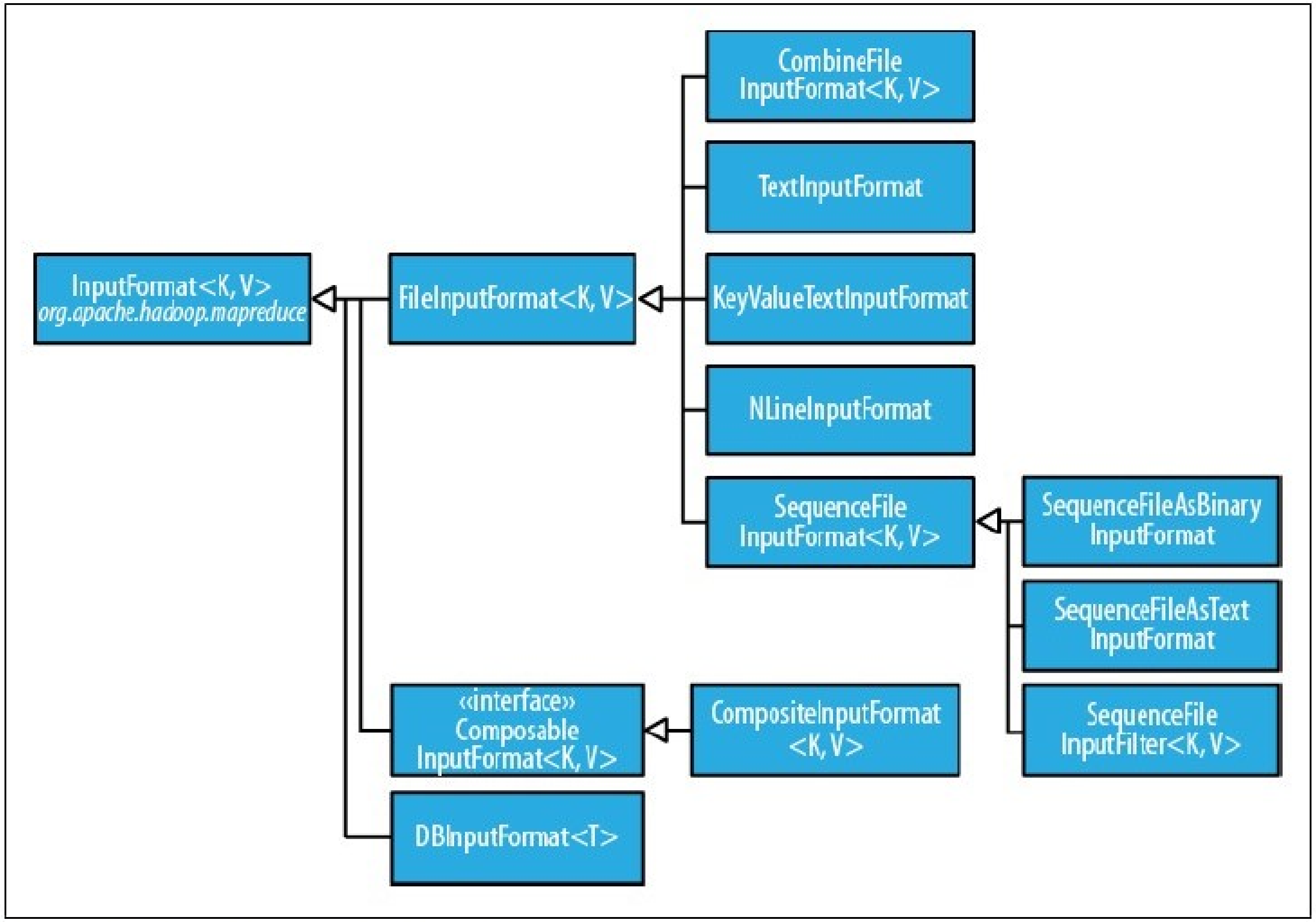

InputFormat<K, V>
org.apache.hadoop.mapreduce
FileInputFormat<K, V>
CombineFile InputFormat<K, V>
TextInputFormat
KeyValueTextInputFormat
NLineInputFormat
SequenceFile InputFormat<K, V>
SequenceFileAsBinary InputFormat
SequenceFileAsText InputFormat
SequenceFile InputFilter<K, V>
«interface» Composable InputFormat<K, V>
CompositeInputFormat <K, V>
DBInputFormat<T>

Figure 7-2. InputFormat class hierarchy

Hadoop内置的文件输入格式

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueTextInputFormat	Parses lines into key-val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

Hadoop内置的RecordReader

RecordReader:	InputFormat	Description:
LineRecordReader	default reader for TextInputFormat	reads lines of text files
KeyValueLineRecordReader	default reader for KeyValueTextInputFormat	parses lines into key-val pairs
SequenceFileRecordReader	default reader for SequenceFileInput Format	User-defined methods to create keys and values

Example：TextInputFormat，LineRecordReader

On the top of the Crumpetty Tree !	(0,On the top of the Crumpetty Tree !)
The Quangle Wangle sat, !	(33,The Quangle Wangle sat, !)
But his face you could not see, !	(57,But his face you could not see, !)
On account of his Beaver Hat. !	(89,On account of his Beaver Hat. !)

Example：KeyValueInputFormat，KeyValueRecordReader

H1 -> On the top of the Crumpetty Tree !	(H1,On the top of the Crumpetty Tree !)
H2 -> The Quangle Wangle sat, !	(H2,The Quangle Wangle sat, !)
H3 -> But his face you could not see, !	(H3,But his face you could not see, !)
H4 -> On account of his Beaver Hat. !	(H4,On account of his Beaver Hat. !)

Example： NlineInputFormat， LineRecordReader

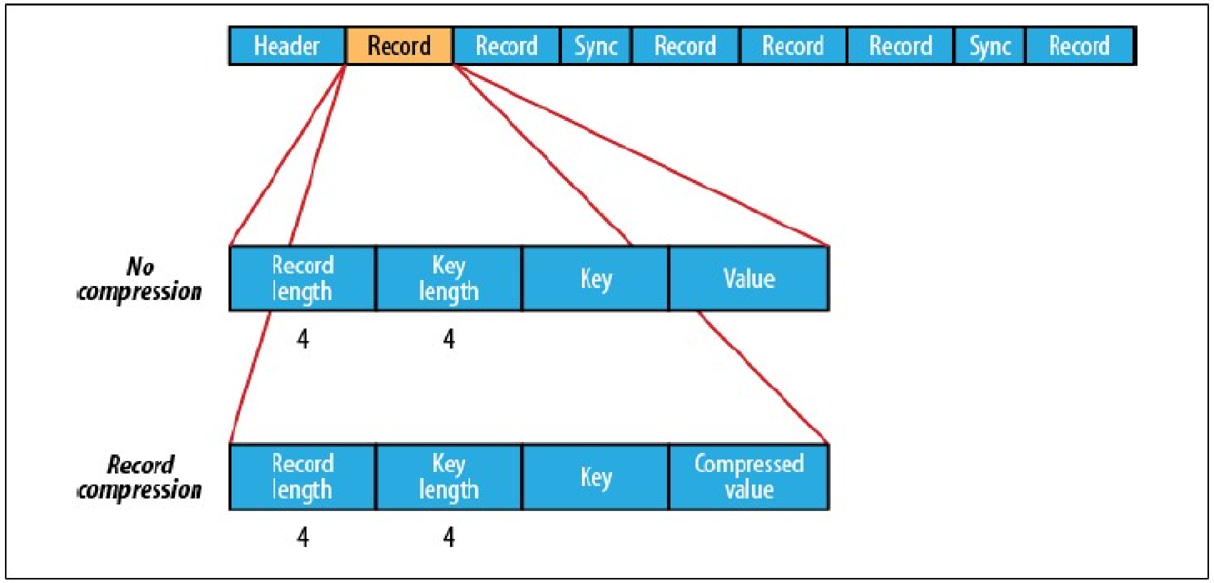
On the top of the Crumpetty Tree !
The Quangle Wangle sat, !
But his face you could not see, !
On account of his Beaver Hat. !

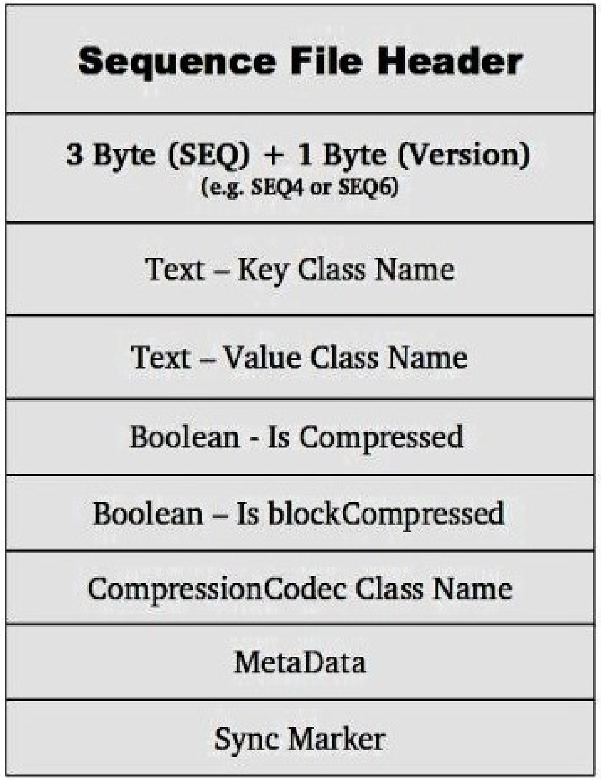
(0,On the top of the Crumpetty Tree !)
(33,The Quangle Wangle sat, !)

(57,But his face you could not see, !)
(89,On account of his Beaver Hat. !)

SequenceFile

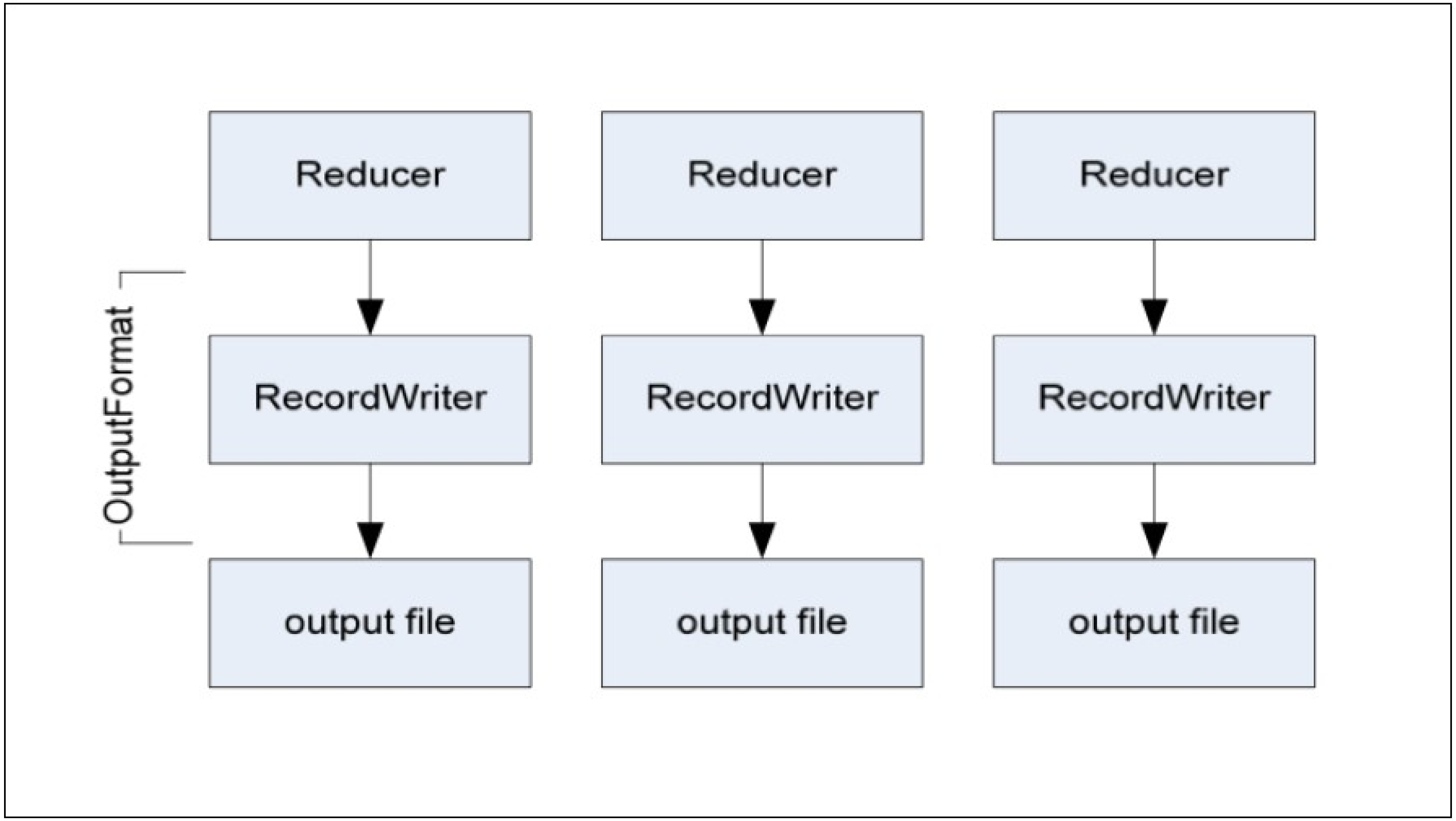
- SequenceFile提供了高效的二进制文件格式，经常用于MapReduce的输出
  - 作为二进制文件，比文本文件更紧凑
  - 可以对每条记录或整个split进行压缩
  - 文件可被并行切分和处理





Sequence File Header
3 Byte (SEQ) + 1 Byte (Version) (e.g. SEQ4 or SEQ6)
Text – Key Class Name
Text – Value Class Name
Boolean - Is Compressed
Boolean – Is blockCompressed
CompressionCodec Class Name
MetaData
Sync Marker

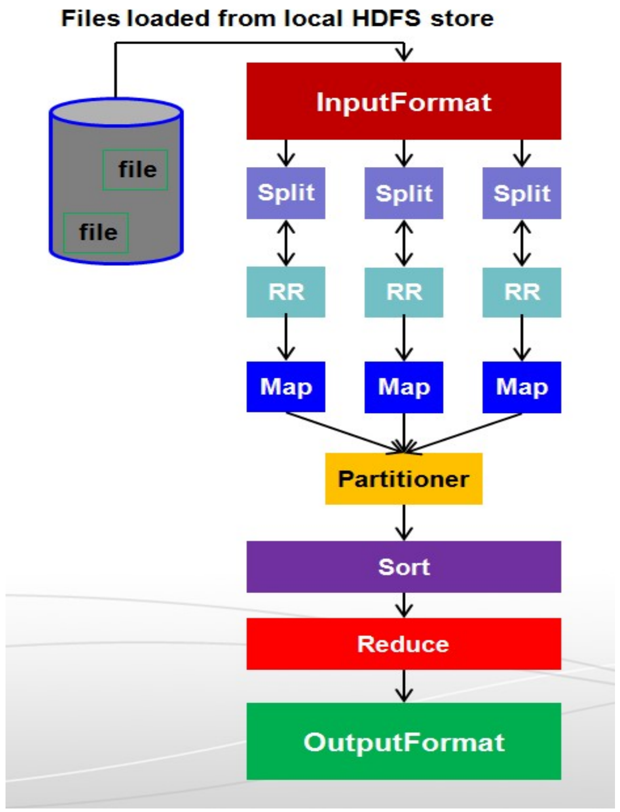
Writing the Output



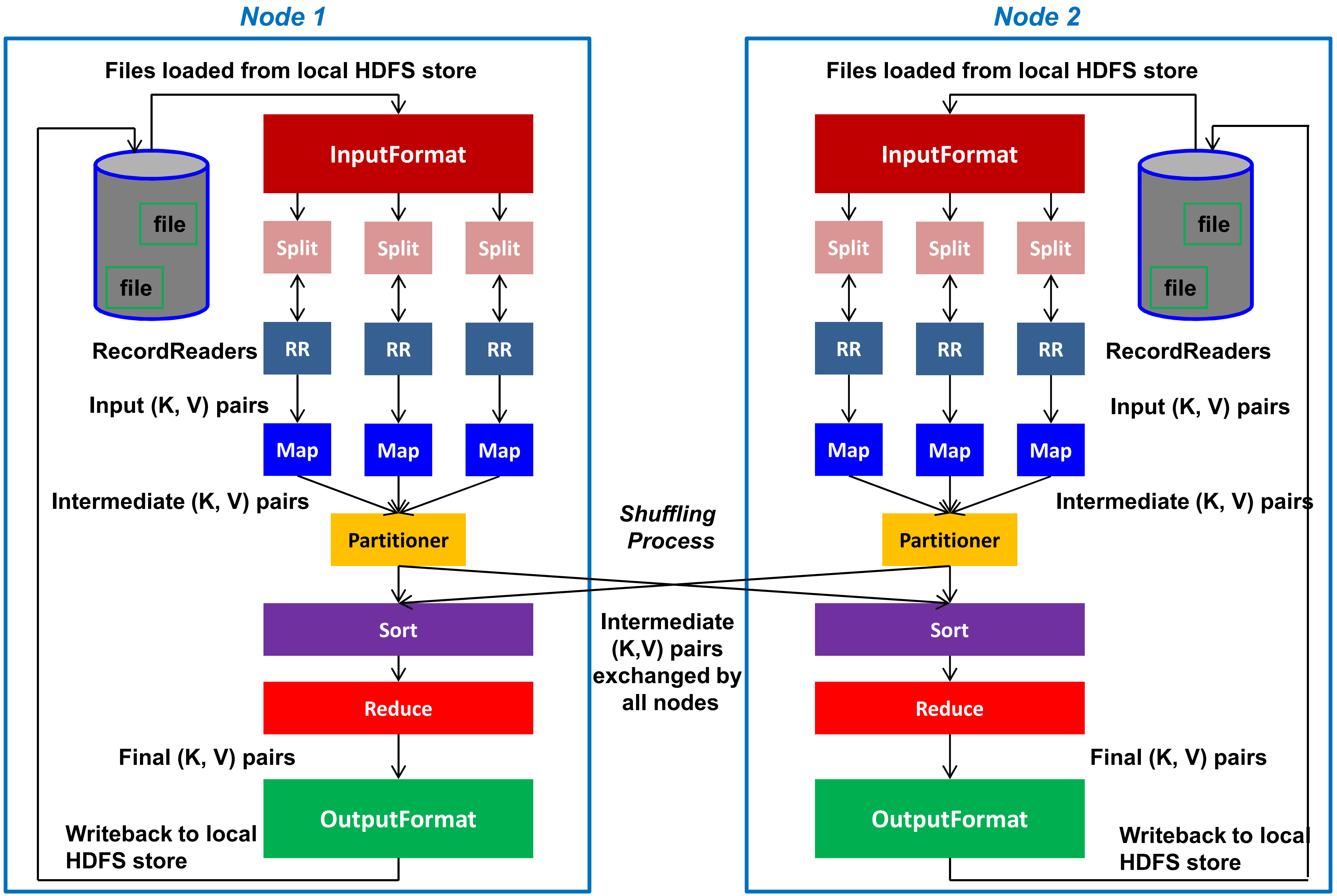
OutputFormat

- OutputFormat 类定义了Reduce产生的(K,V) 对写到对应文件中的格式
- 有以下几种格式：

OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files



Hadoop MapReduce: 具体流程



作业

- 给定一个英文文档集，基于学过的wordcount，编写伪代码，计算该文档集中每行文本平均有多少个单词（忽略标点符号等字符的处理）。

END