

深入理解大数据-大数据处理与编程实践

Ch.7. 高级MapReduce 编程技术

南京大学计算机科学与技术系

主讲人：黄宜华，顾荣

鸣谢：本课程得到Google（北京）与Intel公司
中国大学合作部精品课程计划资助

Ch.7. 高级MapReduce编程技术

1. 复合键值对的使用
2. 用户自定义数据类型
3. 用户自定义输入输出格式
4. 用户自定义Partitioner和Combiner
5. 迭代完成MapReduce计算
6. 组合式MapReduce程序设计
7. 多数据源的连接
8. 全局参数/数据文件的传递
9. 其它处理技术

1. 复合键值对的使用

用复合键让系统完成排序

map计算过程结束后进行Partitioning处理时，系统自动按照map的输出键进行排序，因此，进入Reduce节点的(key, {value})对将保证是按照key进行排序的，而{value}则不保证是排好序的。但有些应用中恰恰需要{value}列表中的value也是排好序的。为了解决这个问题，可以在Reduce过程中对{value}列表中的各个value进行本地排序。但当{value}列表数据量巨大、无法在本地内存中进行排序时，将出现问题。

一个更好的办法是，将value中需要排序的部分加入到key中形成复合键，这样将能利用MapReduce系统的排序功能完成排序。但需要实现一个新的Partitioner，保证原来同一key值的键值对最后分区到同一个Reduce节点上。

用复合键让系统完成排序

带频率的倒排索引示例

```
1: class Mapper
2: procedure Map(docid dn, doc d)
3:   F ← new AssociativeArray
4:   for all term t ∈ doc d do
5:     F{t} ← F{t} + 1
6:   for all term t ∈ F do
7:     Emit(term t, posting <dn, F{t}>)

1: class Reducer
2: procedure Reduce(term t, postings [<dn1, f1>, <dn2, f2>...])
3:   P ← new List
4:   for all posting <a, f> ∈ postings [<dn1, f1>, <dn2, f2>...] do
5:     Append(P, <a, f>)
6:   Sort(P); // 进入Reduce节点的postings不保证按照文档序
              //号排序,因而需要对postings进行一个本地排序
7:   Emit(term t; postings P)
```

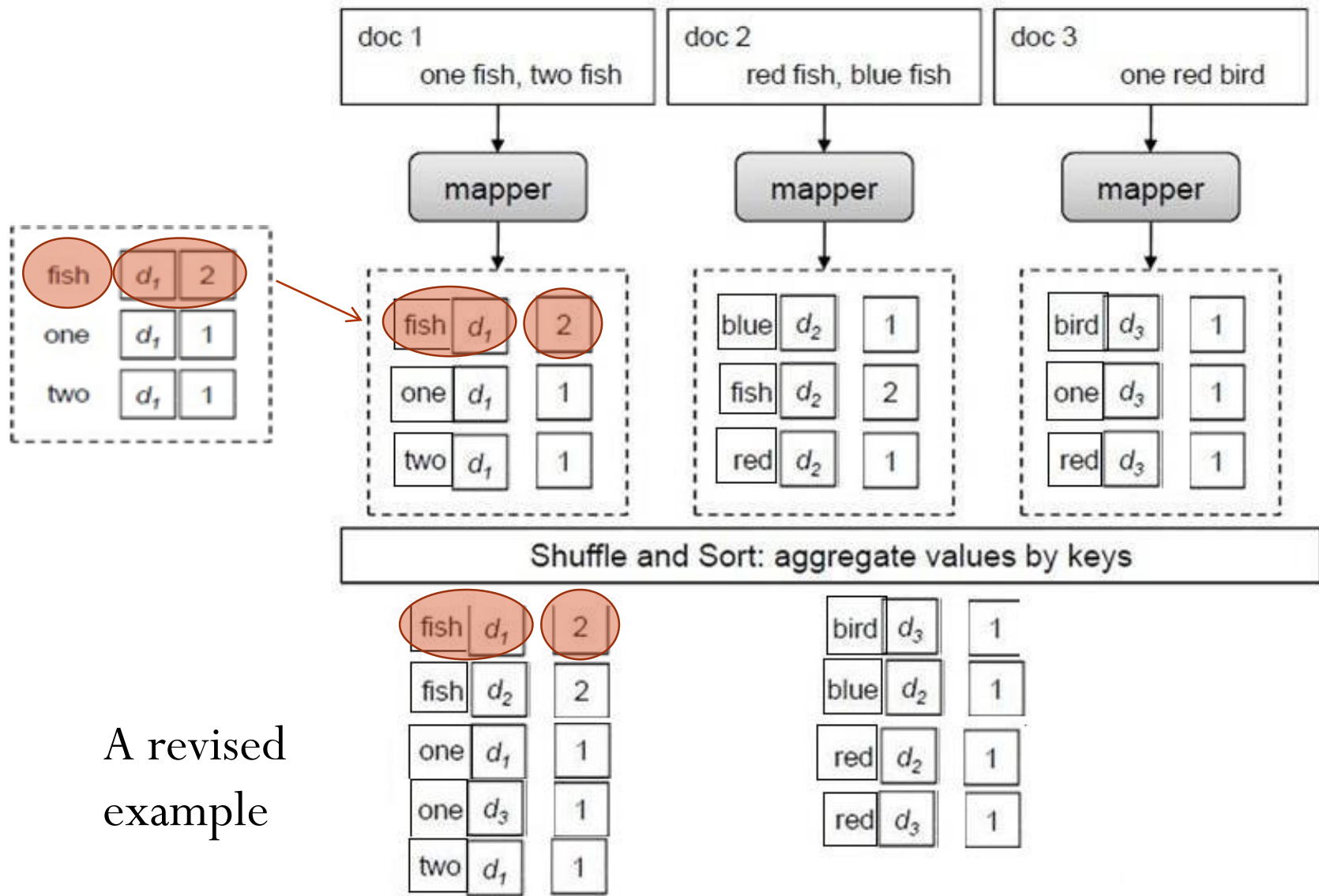
用复合键让系统完成排序

带频率的倒排索引示例

为了能利用系统自动对docid进行排序，解决方法是：代之以生成 (term, <docid, tf>)键值对，map时将term和docid组合起来形成复合键<term,docid>。

但会引起新的问题，同一个term下的所有posting信息无法被分区到同一个Reduce节点，为此，需要实现一个新的Partitioner：从<term, docid>中取出term，以term作为key进行分区。

复合键值对的使用



复合键值对的使用

Customized Partitioner

ues by keys

A revised example
(cont.)

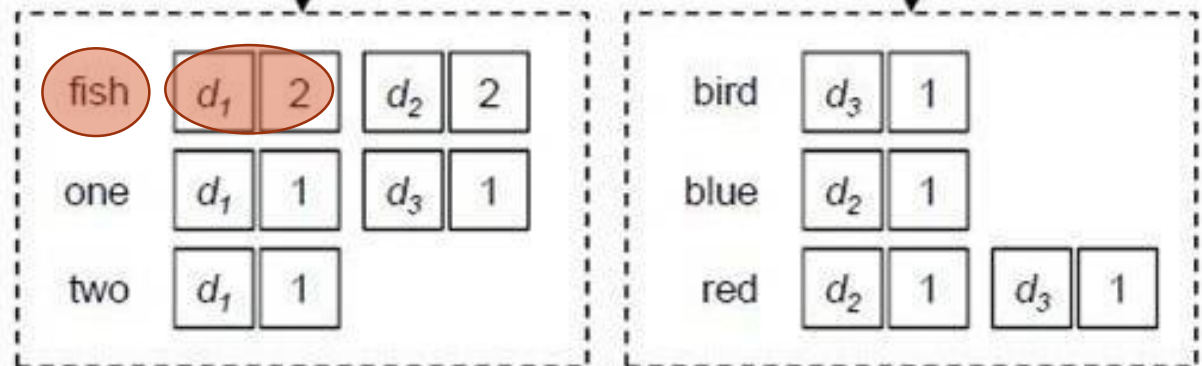
| | | |
|------|-------|---|
| fish | d_1 | 2 |
| fish | d_2 | 2 |
| one | d_1 | 1 |
| one | d_3 | 1 |
| two | d_1 | 1 |

| | | |
|------|-------|---|
| bird | d_3 | 1 |
| blue | d_2 | 1 |
| red | d_2 | 1 |
| red | d_3 | 1 |

进入reduce的键值对按照(term, docid)排序

reducer

reducer




把小的键值对合并成大的键值对

通常一个计算问题会产生大量的键值对，为了减少键值对传输和排序的开销，一些问题中的大量小的键值对可以被合并成一些大的键值对 (pairs \rightarrow stripes)。

单词同现矩阵算法

一个Map可能会产生单词a与其它单词间的多个键值对，这些键值对可以在Map过程中合并成右侧的一个大的键值对(条)：

| | | |
|------------------------|---|--|
| (a, b) \rightarrow 1 | | |
| (a, c) \rightarrow 2 | | |
| (a, d) \rightarrow 5 |  | |
| (a, e) \rightarrow 3 | | |
| (a, f) \rightarrow 2 | | |
| | | $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$ |

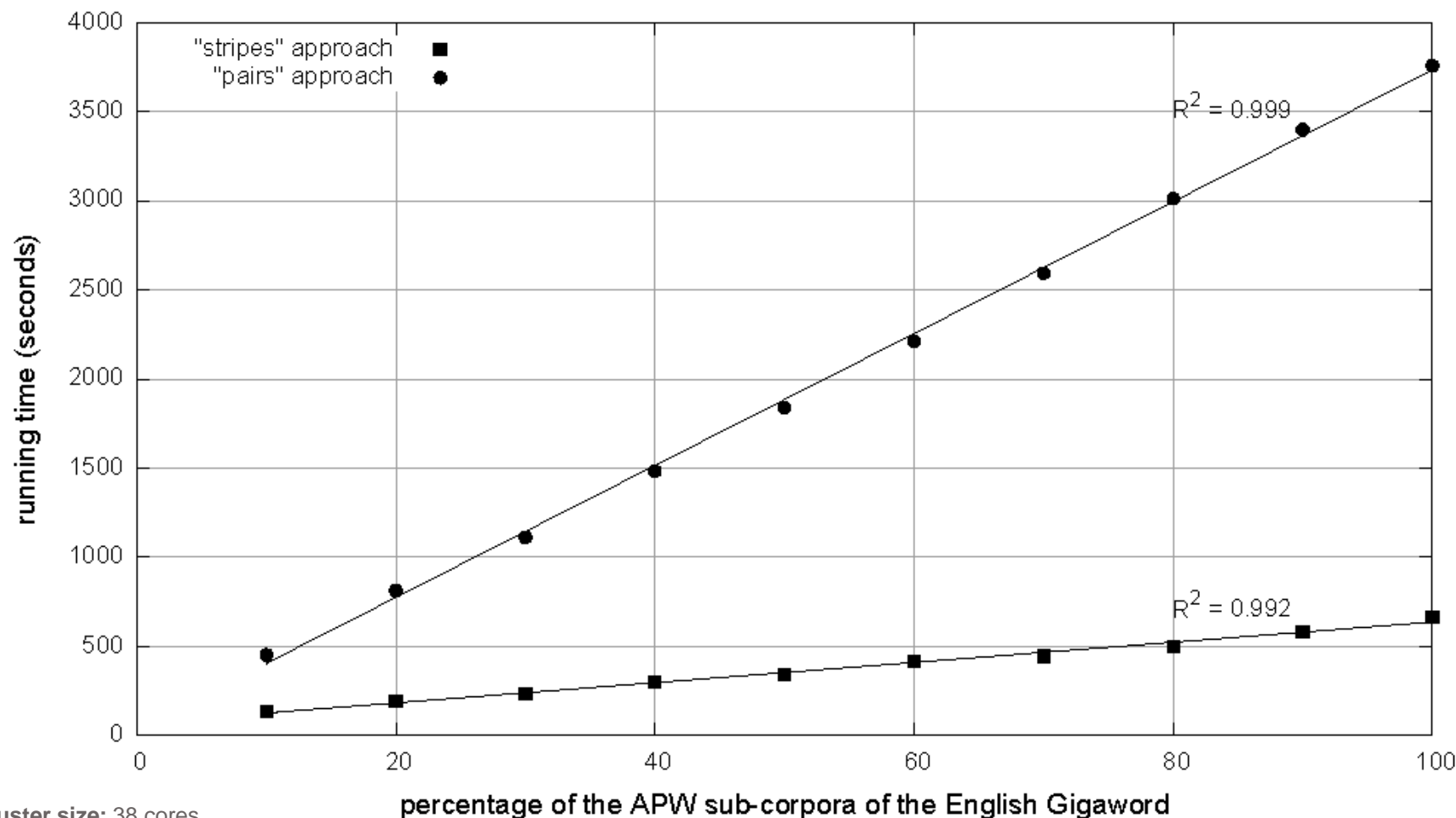
然后，在Reduce阶段，把每个单词a的键值对(条)进行累加：

| | |
|-------|--|
| | $a \rightarrow \{ b: 1, \quad d: 5, e: 3 \}$ |
| + | $a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \}$ |
| <hr/> | |
| | $a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}$ |

把小的键值对合并成大的键值对

单词同现矩阵算法

Efficiency comparison of approaches to computing word co-occurrence matrices



Cluster size: 38 cores

Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

把小的键值对合并成大的键值对

用JobTracker的Web GUI可以观察优化执行后各项统计数据

| | Counter | Map | Reduce | Total |
|----------------------|------------------------|-------------|--------|-------------|
| Job Counters | Data-local map tasks | 0 | 0 | 4 |
| | Launched reduce tasks | 0 | 0 | 2 |
| | Launched map tasks | 0 | 0 | 4 |
| Map-Reduce Framework | Reduce input records | 0 | 151 | 151 |
| | Map output records | 1,984,055 | 0 | 1,984,055 |
| | Map output bytes | 18,862,764 | 0 | 18,862,764 |
| | Combine output records | 1,063 | 151 | 1,214 |
| | Map input records | 2,923,923 | 0 | 2,923,923 |
| | Reduce input groups | 0 | 151 | 151 |
| | Combine input records | 1,984,625 | 493 | 1,985,118 |
| | Map input bytes | 236,903,179 | 0 | 236,903,179 |
| | Reduce output records | 0 | 151 | 151 |
| | HDFS bytes written | 0 | 2,658 | 2,658 |
| File Systems | Local bytes written | 20,554 | 2,510 | 23,064 |
| | HDFS bytes read | 236,915,470 | 0 | 236,915,470 |
| | Local bytes read | 21,112 | 2,510 | 23,622 |
| | | | | |

2. 用户自定义数据类型

Hadoop内置的数据类型

这些数据类型都实现了WritableComparable接口，以便进行网络传输和文件存储，以及进行大小比较。

| Class | Description |
|------------------------------|---|
| <code>BooleanWritable</code> | Wrapper for a standard Boolean variable |
| <code>ByteWritable</code> | Wrapper for a single byte |
| <code>DoubleWritable</code> | Wrapper for a Double |
| <code>FloatWritable</code> | Wrapper for a Float |
| <code>IntWritable</code> | Wrapper for a Integer |
| <code>LongWritable</code> | Wrapper for a Long |
| <code>Text</code> | Wrapper to store text using the UTF8 format |
| <code>NullWritable</code> | Placeholder when the key or value is not needed |

用户自定义数据类型

需要实现Writable接口，作为key或者需要比较大小时则需要实现WritableComparable接口

```
public class Point3D implements WritableComparable <Point3D>
{
    private int x, y, z;
    public int getX() { return x; }
    public int getY() { return y; }
    public int getZ() { return z; }
    public void write(DataOutput out) throws IOException
    {
        out.writeFloat(x);
        out.writeFloat(y);
        out.writeFloat(z);
    }
    public void readFields(DataInput in) throws IOException
    {
        x = in.readFloat();
        y = in.readFloat();
        z = in.readFloat();
    }
    public int compareTo(Point3D p)
    {
        //compares this(x, y, z) with p(x, y, z)
        and outputs -1, 0, 1
    }
}
```

若用户需要自定义数据类型, 实现WritableComparable接口

```
public class Edge implements WritableComparable<Edge>
{
    private String departureNode;
    private String arrivalNode;
    public String getDepartureNode() { return departureNode;}
    @Override
    public void readFields(DataInput in) throws IOException
    {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }
    @Override
    public void write(DataOutput out) throws IOException
    {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }
    @Override
    public int compareTo(Edge o)
    {
        return (departureNode.compareTo(o.departureNode)!=0)
            ?departureNode.compareTo(o.departureNode):arrivalNode.compareTo(o.arrivalNode);
    }
}
```

3. 用户自定义输入输出格式

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内置的文件输入格式

AutoInputFormat, CombineFileInputFormat, CompositeInputFormat,
DBInputFormat, FileInputFormat, KeyValueTextInputFormat,
LineDocInputFormat, MultiFileInputFormat, NLineInputFormat,
SequenceFileAsBinaryInputFormat, SequenceFileAsTextInputFormat,
SequenceFileInputFilter, SequenceFileInputFormat, StreamInputFormat,
TextInputFormat

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 |

Hadoop内置的RecordReader

CombineFileRecordReader, DBInputFormat.DBRecordReader,
InnerJoinRecordReader, JoinRecordReader, KeyValueLineRecordReader,
LineDocRecordReader, MultiFilterRecordReader, OuterJoinRecordReader,
OverrideRecordReader,
SequenceFileAsBinaryInputFormat.SequenceFileAsBinaryRecordReader,
SequenceFileAsTextRecordReader, SequenceFileRecordReader,
StreamBaseRecordReader, StreamXmlRecordReader, WrappedRecordReader

用户自定义InputFormat和RecordReader示例

简单的文档倒排索引

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class InvertedIndexMapper extends Mapper<Text, Text, Text, Text>
{
    @Override
    protected void map(Text key, Text value, Context context)
        throws IOException, InterruptedException
    // default RecordReader: LineRecordReader; key: line offset; value: line string
    {
        Text word = new Text();
        FileSplit fileSplit = (FileSplit)context.getInputSplit();
        String fileName = fileSplit.getPath().getName();
        Text fileName_lineOffset = new Text(fileName+"#" +key.toString());
        StringTokenizer itr = new StringTokenizer(value.toString());
        for(; itr.hasMoreTokens(); )
        {
            word.set(itr.nextToken());
            context.write(word, fileName_lineOffset);
        }
    }
}
```

由于采用了缺省的
TextInputFormat和
LineRecordReader,
需要增加此段代码
完成特殊处理

用户自定义输入输出格式

用户自定义InputFormat和RecordReader示例

简单的文档倒排索引

可以自定义一个InputFormat和RecordReader实现同样的效果

```
public class FileNameOffsetInputFormat extends FileInputFormat<Text, Text>
{
    @Override
    public RecordReader<Text, Text> createRecordReader(InputSplit split,
                                                         TaskAttemptContext context)
    {
        FileNameOffsetRecordReader fnrr = new FileNameOffsetRecordReader();
        try
        {
            fnrr.initialize(split, context);
        }
        catch (IOException e) { e.printStackTrace(); }
        catch (InterruptedException e) { e.printStackTrace(); }
        return fnrr;
    }
}
```

用户自定义 InputFormat 和 RecordReader 示例

简单的文档倒排索引

```
public class FileNameOffsetRecordReader extends RecordReader<Text, Text>
{
    String fileName;
    LineRecordReader lrr = new LineRecordReader();
    .....
    @Override
    public Text getCurrentKey() throws IOException, InterruptedException
    {
        return new Text("(" + fileName + "#" + lrr.getCurrentKey() + ")");
    }
    @Override
    public Text getCurrentValue() throws IOException, InterruptedException
    {
        return lrr.getCurrentValue();
    }
    @Override
    public void initialize(InputSplit arg0, TaskAttemptContext arg1)
        throws IOException, InterruptedException
    {
        lrr.initialize(arg0, arg1);
        fileName = ((FileSplit)arg0).getPath().getName();
    }
}
```

用户自定义InputFormat和RecordReader示例

简单的文档倒排索引

```
public class InvertedIndexer
{
    public static void main(String[] args)
    {
        try {
            Configuration conf = new Configuration();
            job = new Job(conf, "invert index");
            job.setJarByClass(InvertedIndexer.class);
            job.setInputFormatClass(FileNameOffsetInputFormat.class);
            job.setMapperClass(InvertedIndexMapper.class);
            job.setReducerClass(InvertedIndexReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            System.exit(job.waitForCompletion(true) ? 0 : 1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

用户自定义InputFormat和RecordReader示例

简单的文档倒排索引

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class InvertedIndexMapper extends Mapper<Text, Text, Text, Text>
{   @Override
    protected void map(Text key, Text value, Context context)
                                throws IOException, InterruptedException
    // InputFormat: FileNameOffsetInputFormat
    // RecordReader: FileNameOffsetRecordReader; key: filename#lineoffset; value: line string
    {   Text word = new Text();
        StringTokenizer itr = new StringTokenizer(value.toString());
        for(; itr.hasMoreTokens(); )
        {   word.set(itr.nextToken());
            context.write(word, key);
        }
    }
}
```


用户自定义输入输出格式

用户自定义OutputFormat和RecordWriter

Hadoop内置的OutputFormat和RecordWriter

| OutputFormat: | Description |
|--------------------------|---|
| TextOutputFormat | Default; writes lines in "key \t value" form |
| SequenceFileOutputFormat | Writes binary files suitable for reading into subsequent MapReduce jobs |
| NullOutputFormat | Disregards its inputs |

DBOutputFormat, FileOutputFormat, FilterOutputFormat, IndexUpdateOutputFormat, LazyOutputFormat, MapFileOutputFormat, MultipleOutputFormat, MultipleSequenceFileOutputFormat, MultipleTextOutputFormat, NullOutputFormat, SequenceFileAsBinaryOutputFormat, SequenceFileOutputFormat, TextOutputFormat

用户自定义输入输出格式

用户自定义OutputFormat和RecordWriter

Hadoop内置的OutputFormat和RecordWriter

| RecordWriter: | Description |
|----------------------------------|--|
| LineRecordWriter | Default RecordWriter for TextOutputFormat writes lines in "key \t value" form |

[DBOutputFormat.DBRecordWriter](#),
[FilterOutputFormat.FilterRecordWriter](#),
[TextOutputFormat.LineRecordWriter](#)

与InputFormat和RecordReader类似，用户可以根据需要定制OutputFormat和RecordWriter

4. 用户自定义Partitioner和Combiner

定制Partitioner

程序员可以根据需要定制Partitioner来改变Map中间结果到Reduce节点的分区方式，并在Job中设置新的Partitioner

Class **NewPartitioner** extends HashPartitioner<K,V>

```
{ // override the method
```

```
    getPartition(K key, V value, int numReduceTasks)
```

```
    { term = key. toString().split(",")[0]; //<term, docid>=>term
```

```
        super.getPartition(term, value, numReduceTasks);
```

```
    }
```

```
}
```

并在Job中设置新的Partitioner：

```
Job. setPartitionerClass(NewPartitioner)
```

用户自定义Partitioner和Combiner

定制Combiner

程序员可以根据需要定制Combiner来减少网络数据传输量，提高系统效率，并在Job中设置新的Combiner

每年申请美国专利的国家数统计

Patent description data set “apat63_99.txt”

“PATENT”, “**GYEAR**”, “GDATE”, “APPYEAR”, “**COUNTRY**”, “POSTATE”, “ASSIGNEE”, “ASSCODE”, “CLAIMS”, “NCLASS”, “CAT”, “SUBCAT”, “CMADE”, “CRECEIVE”, “RATIOCIT”, “GENERAL”, “ORIGINAL”, “FWDAPLAG”, “BCKGTLAG”, “SELFCTUB”, “SELFCTLB”, “SECDUPBD”, “SECDLWBD”

3070801, **1963**, 1096, “**BE**”, “”, 1, 269, 6, 69, 1, 0, , , , , , , ,

3070802, **1963**, 1096, “**US**”, “TX”, 1, 2, 6, 63, 0, , , , , , , ,

3070803, **1963**, 1096, “**US**”, “IL”, 1, 2, 6, 63, 9, 0.3704, , , , , , , ,

3070804, **1963**, 1096, “**US**”, “OH”, 1, 2, 6, 63, 3, 0.6667, , , , , , , ,

3070805, **1963**, 1096, “**US**”, “CA”, 1, 2, 6, 63, 1, 0, , , , , , , ,

.....

定制Combiner

每年申请美国专利的国家数统计

1. Map中用<year, country>作为key输出, Emit(<year, country>,1)
(<1963, BE>, 1), (<1963, US>, 1), (<1963, US>, 1), ...
2. 实现一个定制的Partitioner, 保证同一年份的数据划分到同一个Reduce节点
3. Reduce中对每一个(<year, country>, [1, 1,1,...])输入, 忽略后部的出现次数, 仅考虑key部分: <year, country>

问题: Map结果(<year, country>, [1, 1,1,...])数据通信量较大

解决办法: 实现一个Combiner将[1, 1,1,...]合并为1

定制Combiner

每年申请美国专利的国家数统计

```
public static class NewCombiner extends Reducer  
    < Text, IntWritable, Text, IntWritable >
```

```
{  
    public void reduce(Text key, Iterable<IntWritable> values,    Context context)  
        throws IOException, InterruptedException  
    {  
        // 忽略(<year, country>, [1, 1, 1,...])后部很长的数据串,  
        // 归并为<year, country>的1次出现  
        context.write(key, new IntWritable(1));  
    } // 输出key: <year, country>; value: 1  
}
```

5. 迭代MapReduce计算

基本问题

一些求解计算需要用迭代方法求得逼近结果(求解计算必须是收敛性的)。当用MapReduce进行这样的问题求解时，运行一趟MapReduce过程无法完成整个求解过程，因此，需要采用迭代方法循环运行该MapReduce过程，直到达到一个逼近结果。

页面排序算法PageRank

- 随机浏览模型：假设一位上网者随机地浏览一些网页
 - 有可能从当前网页点击一个链接继续浏览（概率为d）；
 - 有可能随机跳转到其它N个网页中的任一个（概率为1-d）。
- 每个网页的PageRank值可以看成该网页被随机浏览的概率：

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

$L(p_j)$ 为网页 p_j 上的超链个数

页面排序算法PageRank

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

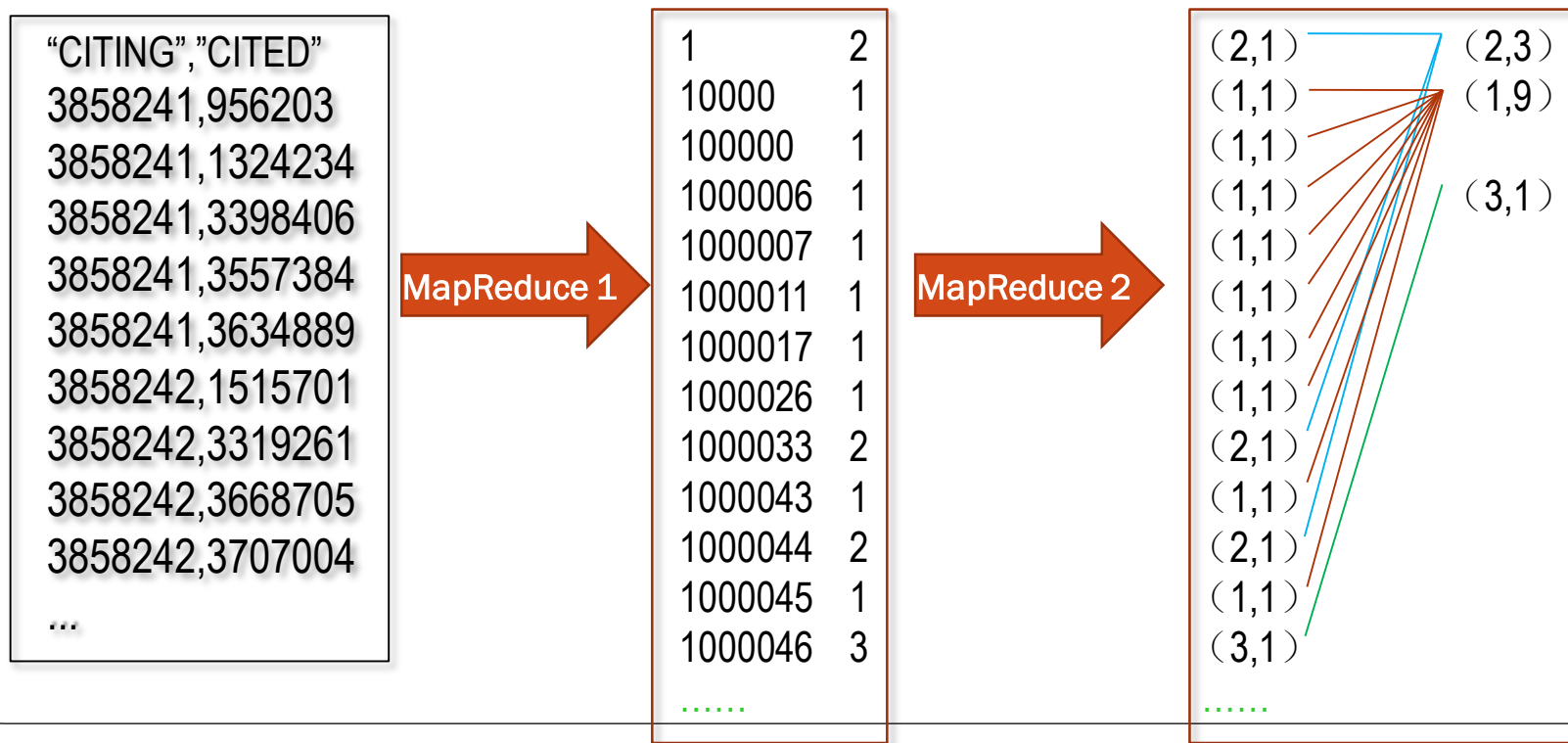
问题是在求解 $PR(p_i)$ 时，需要递归调用 $PR(p_j)$ ，而 $PR(p_j)$ 本身也是待求解的。因此，我们只能先给每个网页赋一个假定的 PR 值，如0.5。但这样求出的 $PR(p_i)$ 肯定不准确。然而，当用求出的 PR 值反复进行迭代计算时，会越来越趋近于最终的准确结果。因此，需要用迭代方法循环运行MapReduce过程，直至第 n 次迭代后的结果与第 $n-1$ 次的结果小于某个指定的阈值时结束。

6. 组合式MapReduce程序设计

基本问题

一些复杂任务难以用一趟MapReduce处理过程来完成，需要将其分为多趟简单些的MapReduce子任务组合完成。如：

- 专利文献引用直方图统计，需要先进行被引次数统计，然后在被引次数上再进行被引直方图统计



MapReduce子任务的顺序化执行

多个MapReduce子任务可以用手工逐一执行，但更方便的做法是
将这些子任务串起来，前面MapReduce任务的输出作为后面
MapReduce的输入，自动地完成顺序化的执行，如：

mapreduce-1=> mapreduce-2 => mapreduce-3 => ...

单个MapReduce作业控制执行代码：

```
Configuration jobconf = new Configuration();
```

```
job = new Job(jobconf, "invert index");
```

```
job.setJarByClass(InvertedIndexer.class);
```

```
.....
```

```
FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

```
job.waitForCompletion(true);
```

MapReduce子任务的顺序化执行

同样，链式MapReduce中的每个子任务需要提供独立的jobconf，并按照前后子任务间的输入输出关系设置输入输出路径，而任务完成后所有中间过程的输出结果路径都可以删除掉。

```
Configuration jobconf1 = new Configuration();  
job1 = new Job(jobconf1, "Job1");  
job1.setJarByClass(jobclass1);  
  
.....  
FileInputFormat.addInputPath(job1, inpath1);  
FileOutputFormat.setOutputPath(job1, outpath1);  
job1.waitForCompletion(true);
```



```
Configuration jobconf2 = new Configuration();  
job2 = new Job(jobconf2, "Job2");  
job2.setJarByClass(jobclass2);  
  
.....  
FileInputFormat.addInputPath(job2, outpath1);  
FileOutputFormat.setOutputPath(job2, outpath2);  
job2.waitForCompletion(true);
```

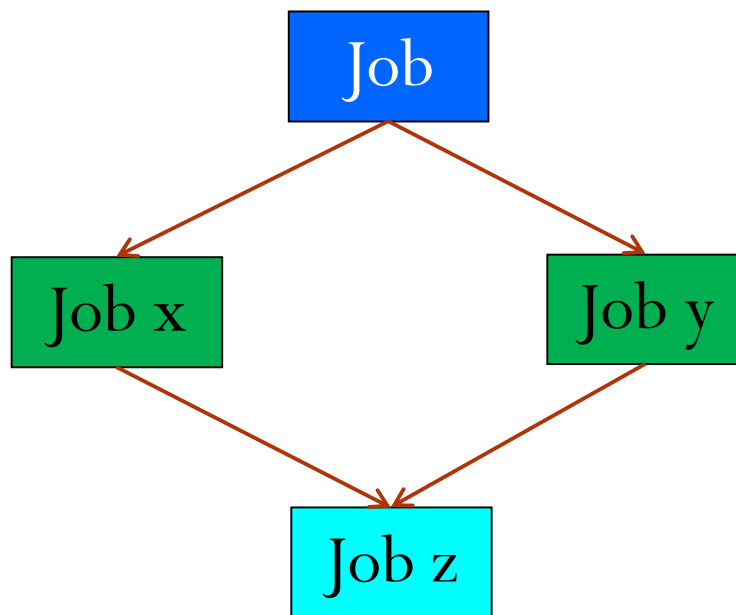
```
Configuration jobconf3 = new Configuration();  
job3 = new Job(jobconf3, "Job3");  
job3.setJarByClass(jobclass3);  
  
.....  
FileInputFormat.addInputPath(job3, outpath2);  
FileOutputFormat.setOutputPath(job3, outpath3);  
job3.waitForCompletion(true);
```



具有数据依赖关系的MapReduce子任务的执行

设一个MapReduce任务由子任务x、y和z构成，其中x和y是相互独立的，但z依赖于x和y，因此，x，y和z不能按照顺序执行。

如：x处理一个数据集D_x，y处理另一个数据集D_y，然后z需要将D_x和D_y进行一个join处理，则z一定要等到x和y执行完毕才能开始执行。



具有数据依赖关系的MapReduce子任务的执行

Hadoop通过Job和JobControl类提供了一种管理这种具有数据依赖关系的子任务的MapReduce作业的执行。Job除了维护Conf信息外，还能维护job间的依赖关系。

```
jobx = new Job(jobxconf, "Jobx");
```

```
.....
```

```
joby = new Job(jobyconf, "Joby");
```

```
.....
```

```
jobz = new Job(jobzconf, "Jobz");
```

```
jobz.addDependingJob(jobx); // jobz将等待jobx执行完毕
```

```
jobz.addDependingJob(joby); // jobz将等待joby执行完毕
```

具有数据依赖关系的MapReduce子任务的执行

JobControl类用来控制整个作业的执行。把所有子任务的作业加入到JobControl中，执行JobControl的run（）方法即可开始整个作业的执行

```
jobx = new Job(jobxconf, "Jobx");  
.....  
joby = new Job(jobyconf, "Joby");  
.....  
jobz = new Job(jobzconf, "Jobz");  
jobz.addDependingJob(jobx); // jobz将等待jobx执行完毕  
jobz.addDependingJob(joby); // jobz将等待joby执行完毕  
JobControl JC = new JobControl ("XYZJob") ;  
JC.addJob(jobx);  
JC.addJob(joby);  
JC.addJob(jobz);  
JC.run();
```


MapReduce前处理和后处理步骤的链式执行

一个MapReduce作业可能会有一些前处理和后处理步骤，比如，文档倒排索引处理前需要一个去除Stop-word的前处理，倒排索引处理后需要一个变形词后处理步骤(making,made->make)。将这些前后处理步骤实现为单独的MapReduce任务可以达到目的，但多个独立的MapReduce作业的执行开销较大，且增加很多I/O操作，因而效率不高。

一个办法是在核心的Map和Reduce过程之外，把这些前后处理步骤实现为一些辅助的Map和Reduce过程，将这些辅助Map和Reduce过程与核心Map和Reduce过程合并为一个过程链，从而完成整个作业的执行。

MapReduce前处理和后处理步骤的链式执行

Hadoop提供了链式Mapper(ChainMapper)和链式Reducer(ChainReducer)来完成这种处理。

ChainMapper和ChainReducer分别提供了addMapper方法加入一系列Mapper:

ChainMapper.addMapper (.....)

ChainReducer.addMapper (.....)

```
public static void addMapper
(Job job,                               // 主作业
 Class<? extends Mapper> mclass,       // 待加入的map class
 Class<?> inputKeyClass,               // 待加入的map输入键class
 Class<?> inputValueClass,            // 待加入的map输入键值class
 Class<?> outputKeyClass,              // 待加入的map输出键class
 Class<?> outputValueClass,           // 待加入的map输出键值class
 org.apache.hadoop.conf.Configuration mapperConf // 待加入的map的conf
) throws IOException
```

MapReduce前处理和后处理步骤的链式执行

设有一个完整的MapReduce作业，由Map1 , Map2 , Reduce, Map3, Map4构成。

```
Configuration conf = new Configuration();
```

```
Job job = new Job(conf);
```

```
job.setJobName("ChainJob");
```

```
job.setInputFormat(TextInputFormat.class);
```

```
job.setOutputFormat(TextOutputFormat.class);
```

```
FileInputFormat.setInputPaths(job, in);
```

```
FileOutputFormat.setOutputPath(job, out);
```

```
JobConf map1Conf = new JobConf(false);
```

```
ChainMapper.addMapper(job, Map1.class, LongWritable.class, Text.class,  
                        Text.class, Text.class, true, map1Conf);
```

```
JobConf map2Conf = new JobConf(false);
```

```
ChainMapper.addMapper(job, Map2.class, Text.class, Text.class, LongWritable.class,  
                        Text.class, true, map2Conf);
```

MapReduce前处理和后处理步骤的链式执行

(接前页)

```
JobConf reduceConf = new JobConf(false);
```

```
ChainReducer.setReducer(job, Reduce.class, LongWritable.class, Text.class,  
                        Text.class, Text.class, true, reduceConf);
```

```
JobConf map3Conf = new JobConf(false);
```

```
ChainReducer.addMapper(job, Map3.class, Text.class, Text.class,  
                       LongWritable.class, Text.class, true, map3Conf);
```

```
JobConf map4Conf = new JobConf(false);
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class,  
                       LongWritable.class, Text.class, true, map4Conf);
```

```
JobClient.runJob(job);
```

7. 多数据源的连接

基本问题

一个MapReduce任务很可能需要访问和处理两个甚至多个数据集，比如，专利文献数据分析中，当需要统计每个国家的专利引用率时，将需要同时访问“专利引用数据集cite75_99.txt”和专利描述数据集“apat63_99.txt”。

在关系数据库中，这将是两个或多个表的连接(join)处理，且join操作完全由数据库系统负责处理。但Hadoop系统没有关系数据库中那样强大的join处理功能，因此多数据源的连接处理比关系数据库中要复杂一些。根据不同的需要和权衡，可以有几种不同的连接方法。

设有两个数据集，一个是顾客数据集：

Customer ID, Name, PhoneNumber

- 1, 王二, 025-1111-1111
- 2, 张三, 021-2222-2222
- 3, 李四, 025-3333-3333
- 4, 孙五, 010-4444-4444

第二个是顾客的订单数据集为：

Customer ID, Order ID, Price, Purchase Date

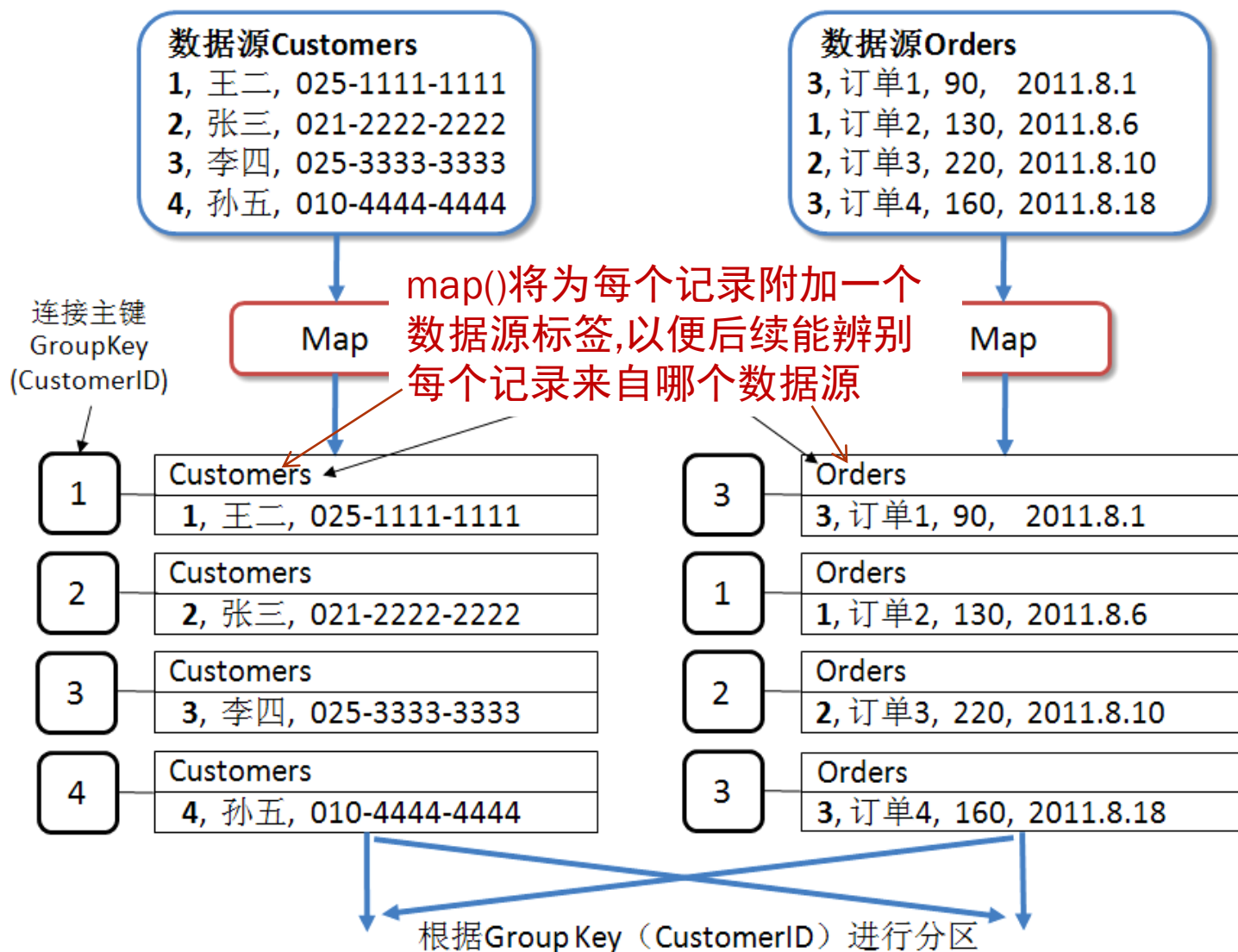
- 3, 订单1, 90, 2011.8.1
- 1, 订单2, 130, 2011.8.6
- 2, 订单3, 220, 2011.8.10
- 3, 订单4, 160, 2011.8.18

以CustomerID进行内连接（inner join）后的数据记录是：

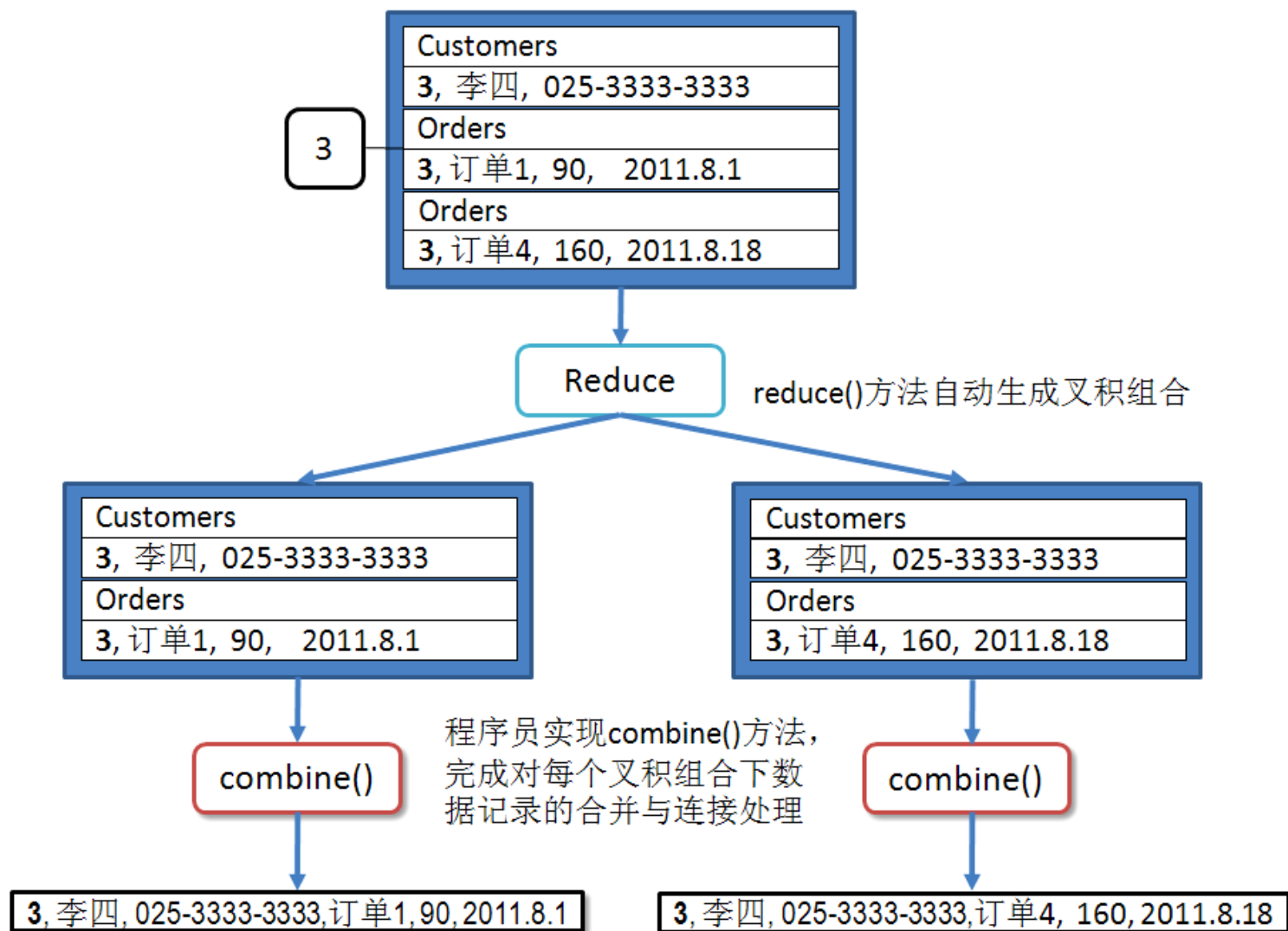
Customer ID, Name, PhoneNumber, Order ID, Price, Purchase Date

- 1, 王二, 025-1111-1111, 订单2, 90, 2011.8.6
- 2, 张三, 021-2222-2222, 订单3, 220, 2011.8.10
- 3, 李四, 025-3333-3333, 订单1, 100, 2011.8.1
- 3, 李四, 025-3333-3333, 订单4, 160, 2011.8.18

用DataJoin类实现Reduce端Join



用DataJoin类实现Reduce端Join



用DataJoin类实现Reduce端Join

Hadoop提供了一个实现多数据源连接的基本框架DataJoin类，帮助程序员完成一些必要的连接处理；DataJoin框架包含以下三个抽象类，使用DataJoin框架后，程序员仅需要实现几个抽象方法：

- **DataJoinMapperBase**: 由程序员实现的Mapper类继承,该基类已实现了map()方法用以完成标签化数据记录的生成和输出,因此程序员仅需实现三个产生数据源标签、GroupKey(key)和标签化记录(value)所需要的抽象方法
- **DataJoinReducerBase**: 由程序员实现的Reducer类继承,该基类已实现了reduce()方法用以完成多数据源记录的叉积组合生成
- **TaggedMapOutput**: 描述一个标签化数据记录，实现了getTag(), setTag()方法；作为Mapper的key-value输出中的value的数据类型，由于需要进行I/O，程序员需要继承并实现Writable接口,并实现抽象的getData() 方法用以读取记录数据。

用DataJoin类实现Reduce端Join

Mapper需要实现的抽象方法

- `abstract Text generateInputTag(String inputFile)`

由程序员决定如何产生记录的数据源标签

如：使用数据集的文件名为数据源标签时(如Customers):

```
protected Text generateInputTag(String inputFile)
{
    return new Text(inputFile);
}
```

当一个数据集包含多个文件时(如part-0000, part-0001,等)、以致无法直接用文件名时，可以从这些文件名定制一个标签名，比如

```
protected Text generateInputTag(String inputFile)
{ // 取 “-” 前的 “part” 作为标签名
    String datasource = inputFile.split('-')[0];
    return new Text(datasource);
}
```

用DataJoin类实现Reduce端Join

Mapper需要实现的抽象方法

- `abstract TaggedMapOutput generateTaggedMapOutput(Object value)`

用于给数据源中的原始数据记录贴上inputTag指定的标签

例如：

```
protected TaggedMapOutput generateTaggedMapOutput(Object value)
{ // 设程序员继承实现的TaggedMapOutput子类为TaggedWritable
    TaggedWritable retv = new TaggedWritable((Text) value);
    // 将generateInputTag()方法计算出来存储在Mapper.inputTag中
    // 的标签设为数据源标签
    retv.setTag(this.inputTag);
    return retv;
}
```

每个记录的数据源标签可以由generateInputTag() 所产生的标签，需要时也可以通过setTag()方法为同一数据源的不同数据记录设置不同的标签

用DataJoin类实现Reduce端Join

Mapper需要实现的抽象方法

protected Text **generateGroupKey**(TaggedMapOutput aRecord)

用于指定一个记录的GroupKey

例如：

```
protected Text generateGroupKey(TaggedMapOutput aRecord)
```

```
{ String line = ((Text) aRecord.getData()).toString();
```

```
    String[] tokens = line.split(",");
```

```
    String groupKey = tokens[0];
```

```
    return new Text(groupKey); // 取CustomerID作为GroupKey (key)
```

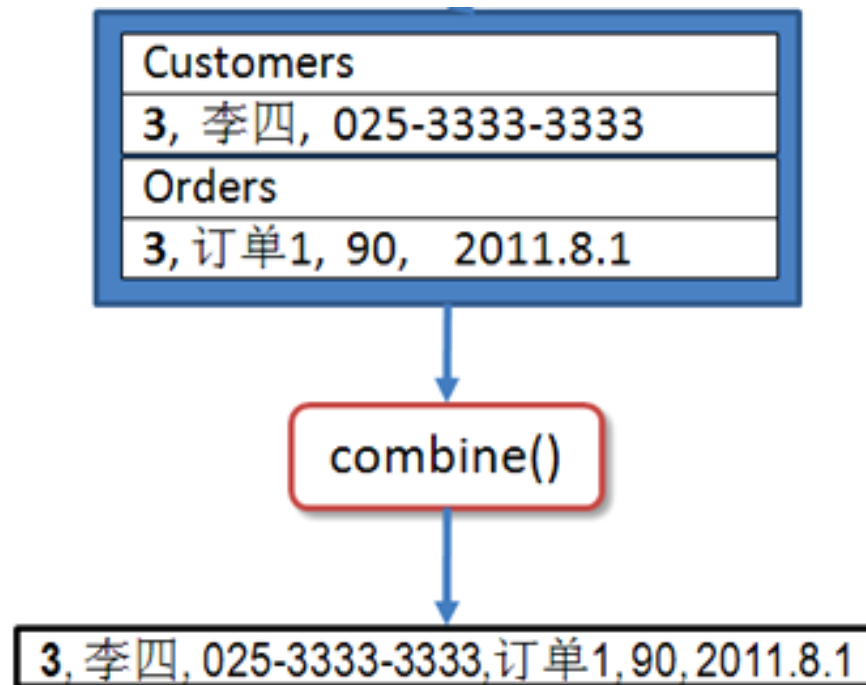
```
}
```

用DataJoin类实现Reduce端Join

Reducer需要实现的抽象方法

- `abstract TaggedMapOutput combine(Object[] tags, Object[] values)`

用于把DataJoinReducerBase类的reduce()方法中由系统所自动完成的多数据源数据记录的叉积进行某种连接处理。



用DataJoin类实现Reduce端Join

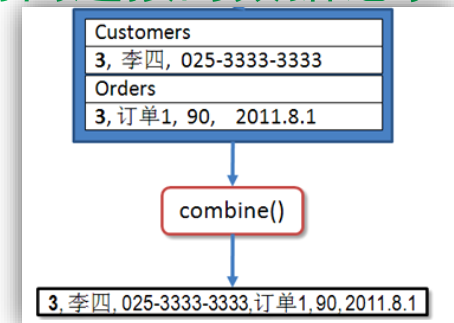
实现Customers和Orders连接处理的程序实现

```
public static class MapClass extends DataJoinMapperBase
{
    protected Text generateInputTag(String inputFile)
    {
        String datasource = inputFile.split("-")[0]; // 使用输入文件名作为标签
        return new Text(datasource); // 该数据源标签将被map()保存在inputTag中
    }
    protected Text generateGroupKey(TaggedMapOutput aRecord)
    {
        String line = ((Text) aRecord.getData()).toString();
        String[] tokens = line.split(",");
        String groupKey = tokens[0];
        return new Text(groupKey); // 取CustomerID作为GroupKey (key)
    }
    protected TaggedMapOutput generateTaggedMapOutput(Object value)
    {
        TaggedWritable retv = new TaggedWritable((Text) value);
        retv.setTag(this.inputTag); // 给一个原始数据记录贴上指定的标签
        return retv;
    }
}
```

用DataJoin类实现Reduce端Join

实现Customers和Orders连接处理的程序实现

```
public static class ReduceClass extends DataJoinReducerBase
{
    protected TaggedMapOutput combine(Object[] tags, Object[] values)
    {
        if (tags.length < 2) return null; // 一个以下数据源，没有需连接的数据记录
        String joinedStr = "";
        for (int i=0; i<values.length; i++)
        {
            if (i > 0) joinedStr += ",";
            TaggedWritable tw = (TaggedWritable) values[i];
            String line = ((Text) tw.getData()).toString();
            String[] tokens = line.split(",", 2); // 把CustomerID与后部的字段分为两段
            if(i==0) joinedStr += tokens[0]; // 拼接一次CustomerID
            joinedStr += tokens[1]; // 拼接每个数据源记录后部的字段
        }
        TaggedWritable retv = new TaggedWritable(new Text(joinedStr));
        retv.setTag((Text) tags[0]); // 把第一个数据源标签设为join后记录的标签
        return retv; // join后的该数据记录将在reduce()中与GroupKey一起输出
    }
}
```



用DataJoin类实现Reduce端Join

实现Customers和Orders连接处理的程序实现

此外还需要实现TaggedWritable对象

```
public static class TaggedWritable extends TaggedMapOutput
{
    private Writable data;
    public TaggedWritable(Writable data)
    {
        this.tag = new Text("");
        this.data = data;
    }
    @Override
    public Writable getData()
    {
        return data;
    }
    public void write(DataOutput out) throws IOException
    {
        this.tag.write(out);
        this.data.write(out);
    }
    public void readFields(DataInput in) throws IOException
    {
        this.tag.readFields(in);
        this.data.readFields(in);
    }
}
```


用DataJoin类实现Reduce端Join

实现Customers和Orders连接处理的程序实现 实现主控程序

```
public Class DataJoin
{
    public static void main(String[] args) throws Exception
    {
        Configuration conf = getConf();
        JobConf job = new JobConf(conf, DataJoin.class);
        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        job.setJobName("DataJoin");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormat(TextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(TaggedWritable.class);
        job.set("mapred.textoutputformat.separator", ",");
        Job.waitForCompletion(true);
    }
}
```

用文件复制方法实现Map端Join

前述用DataJoin类实现的Reduce端Join方法中，Join操作直到Reduce阶段才能处理，很多无效的连接数据组合在Reduce阶段才能去除，而这时这些数据已经通过网络从Map阶段传送到了Reduce阶段，占据了很多的通信带宽。因此这个方法的效率不是很高。

当一个数据源的数据量较小、能够存放在单个节点的内存中时，我们可以使用一个称为“Replicated Join”的方法，把较小的数据源文件复制到每个Map节点，然后在Map阶段完成Join操作。Hadoop提供了一个distributed cache机制用于将一个或多个文件复制到所有节点上。

- Job类中：public void **addCacheFile**(URI uri)：将一个文件放到distributed cache file中
- Mapper或Reducer的context类中：
public Path[] **getLocalCacheFiles**()：获取设置在distributed cache files中的文件路径，以便能将这些文件读入到每个节点内存中

用文件复制方法实现Map端Join

```
Configuration conf = getConf();
Job job = new Job(conf, DataJoinDC.class);
// 将第一个数据源(假定是较小的那个)放置到distributed cache file中
Job.addCacheFile(new Path(args[0]).toUri());
Path in = new Path(args[1]);
Path out = new Path(args[2]);
FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);
job.setJobName("DataJoin with DistributedCache");
job.setMapperClass(MapClass.class);
job.setNumReduceTasks(0);
job.setInputFormat(KeyValueTextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);
job.set("key.value.separator.in.input.line", ",");
Job.waitForCompletion(true);
```

用文件复制方法实现Map端Join

```
public static class MapClass extends Mapper<Text, Text, Text, Text>
{
    private Hashtable<String, String> joinData = new Hashtable<String, String>();
    public void setup(Mapper.Context context) // override setup()
    {
        try // 将distributed cache file装入各个Map节点本地的内存数据joinData中
        {
            Path [] cacheFiles = context.getLocalCacheFiles();
            if (cacheFiles != null && cacheFiles.length > 0)
            {
                String line;
                String[] tokens;
                BufferedReader joinReader = new BufferedReader(
                    new FileReader(cacheFiles[0].toString()));
                try // 以CustomerID作为key，将后部的字段数据存入一个
                    // Hashtable，以便后面使用
                {
                    while ((line = joinReader.readLine()) != null)
                    {
                        tokens = line.split(",", 2);
                        joinData.put(tokens[0], tokens[1]);
                    }
                }
                finally { joinReader.close(); }
            }
        }
        catch (IOException e)
        {
            System.err.println("Exception reading DistributedCache: " + e);
        }
    }
}
```

用文件复制方法实现Map端Join

```
public static class MapClass extends Mapper<Text, Text, Text, Text>
{
    (.....接上)
    public void map(Text key, Text value, Context context)
        throws IOException, InterruptedException
    {
        // 第二个数据源的数据记录将作为Map方法的输入键值对
        // 将value与joinData中的相应记录进行join
        String joinValue = joinData.get(key);
        if (joinValue != null)
        {
            output.collect(key,
                new Text(value.toString() + "," + joinValue));
        }
    }
}
```

由于在Map端即实现了join，因而不需要实现Reducer。

用文件复制方法实现Map端Join

Replicated Joins方法的一个变化使用

即使较小的数据源文件，也可能仍然无法全部存放在内存中处理。但如果计算问题本身仅需要使用较小数据源中的部分记录，如仅仅需要查询位于电话区号为025地区顾客的Orders信息，此时可先将Customers的数据记录进行过滤，仅保留电话区号为025的顾客记录，并保存为一个临时文件(如Customers025)；当这个临时文件数据记录能存放在内存中时，即可使用Replicated Joins方法进行处理。

需要做的额外处理是实现一个方法，根据一定的条件过滤Customers数据记录、并保存为一个临时文件。

设有两个数据集S和R，较小的数据集R可以被分为R1, R2, R3,的子集，且每个子集都足以存放在内存中处理，则可以先对每个Ri用Replicated Join进行与S的Join处理，最后将处理结果合并起来(Union)，得到S Join R。

以上多数据源连接解决方法的限制

以上的多数据源Join只能是具有相同主键/外键的数据源间的连接，如果数据源两两之间具有多个不同的主键/外键的连接，则需要使用多次MapReduce过程完成不同主/外键间的连接。

如，有三个数据源：Customers (CustomerID) ，
Orders (CustomerID, ItemID) ， 以及Products(ItemID)

关系数据库中的Join：

```
Select ...  
from Customers C  
join Orders O on C.CustomerID=O.CustomerID  
join Products P on O.ItemID=P.ItemID
```

但在MapReduce中将需要分两个MapReduce作业来完成三个数据源的Join:第一个MapReduce作业完成Customers与Orders的Join，然后，Join后的结果再通过第二个MapReduce作业完成与Products的Join

8. 全局参数/数据文件的传递

全局作业参数的传递

为了能让用户灵活设置某些作业参数，避免用硬编码方式在程序中设置作业参数，一个MapReduce计算任务可能需要在执行时从命令行输入这些作业参数，并将这些参数传递给各个计算节点。

比如，上一节中两个数据集连接时程序用硬编码方式指定第一个数据字段为连接主键(CustomerID)。但如果要实现一个具有一定通用性的程序、允许任意指定一列字段为连接主键的话，就需要在程序运行时在命令行中指定连接主键所在的数据列或字段名称。然后该输入参数可以作为一个属性保存在Configuration对象中，并允许Map和Reduce节点从Configuration对象中获取和使用该属性值。

全局作业参数的传递

Configuration类专门提供以下用于保存和获取属性的方法：

- `public void set(String name, String value)` //设置字符串属性
- `public String get(String name)` // 读取字符串属性
- `public String get(String name, String defaultValue)` // 读取字符串属性
- `public void setBoolean(String name, boolean value)` //设置布尔属性
- `public boolean getBoolean(String name, boolean defaultValue)` //读取布尔属性
- `public void setInt(String name, int value)` //设置整数属性
- `public int getInt(String name, int defaultValue)` // 读取整数属性
- `public void setLong(String name, long value)` //设置长整数属性
- `public long getLong(String name, long defaultValue)` // 读取长整数属性
- `public void setFloat(String name, float value)` //设置浮点数属性
- `public float getFloat(String name, float defaultValue)` //读取浮点数属性
- `public void setStrings(String name, String... values)` //设置一组字符串属性
- `public String[] getStrings(String name, String... defaultValue)` //读取一组字符串属性

需要说明的是，setStrings方法将把一组字符串转换为用“，”隔开的长字符串，然后getStrings时自动再根据“，”split成一组字符串，因此，在该组中的每个字符串都不能包含“，”，否则会出错。

全局作业参数的传递

例：专利文献数据集Join时主键所在数据列参数的设置

```
Configuration jobconf = new Configuration();
```

```
Job job = new Job(jobconf, MyJob.class);
```

```
...
```

```
// 将第三个输入参数设置为JoinKeyColIdx属性
```

```
jobconf.setInt("JoinKeyColIdx", Integer.parseInt(args[2]));
```

```
.....
```

```
Job.waitForCompletion(true);
```

全局作业参数的传递

在mapper类的初始化方法setup()中从configuration对象中读出属性

```
public static class MapClass extends Mapper <Text, Text, Text, Text>
{
    int join_key_col_idx;
    public void setup(Mapper.Context context)
    {
        Configuration jobconf = context.getConfiguration();
        join_key_col_idx = jobconf.getInt("JoinKeyColIdx", -1);
        // 无值时置为-1
    }
    protected void map(Text key, Text value, Context context)
        throws IOException, InterruptedException
    {
        //使用join_key_col_idx完成数据处理;
        .....
    }
}
```

全局作业参数的传递

同样需要时在reducer类的初始化方法setup()中从configuration对象中读出属性

```
public static class ReduceClass extends Reducer <Text, Text, Text, Text>
{
    int join_key_col_idx;
    public void setup(Mapper.Context context)
    {
        Configuration jobconf = context.getConfiguration();
        join_key_col_idx = jobconf.getInt("JoinKeyColIdx", -1);
        // 无值时置为-1
    }
    protected void reduce(Text key, Text value, Context context)
        throws IOException, InterruptedException
    {
        //使用join_key_col_idx完成数据处理;
        .....
    }
}
```

全局数据文件的传递

有时候一个MapReduce作业可能会使用一些较小的并且需要复制到各个节点的数据文件。为此，可以使用DistributedCache文件传递机制，先将这些文件传送到DistributedCache中，然后各个节点从DistributedCache中将这些文件复制到本地的文件系统中使用。具体使用时，为提供访问速度，可将这些较小的文件数据读入内存。

- Job类中： `public void addCacheFile(URI uri)`： 将一个文件放到 distributed cache file 中
- Mapper或Reducer的context类中：
`public Path[] getLocalCacheFiles()`： 获取设置在distributed cache files中的文件路径，以便能将这些文件读入到每个节点内存中

全局数据文件的传递

在作业Configuration时将文件存入Distributed Cache：

```
Configuration conf = getConf();  
Job job = new Job(conf, DataJoinDC.class);  
// 将第一个数据源(假定是较小的那个)放置到distributed cache file中  
Job.addCacheFile(new Path(args[0]).toUri());  
Path in = new Path(args[1]);  
.....
```

全局数据文件的传递

```
public static class MapClass extends Mapper<Text, Text, Text, Text>
{
    private Hashtable<String, String> joinData = new Hashtable<String, String>();
    public void setup(Mapper.Context context) // override setup()
    {
        try // 将distributed cache file装入各个Map节点本地的内存数据joinData中
        {
            Path [] cacheFiles = context.getLocalCacheFiles();
            if (cacheFiles != null && cacheFiles.length > 0)
            {
                String line;
                String[] tokens;
                BufferedReader joinReader = new BufferedReader(
                    new FileReader(cacheFiles[0].toString()));
                try // 以CustomerID作为key，将后部的字段数据存入一个
                    // Hashtable，以便后面使用
                {
                    while ((line = joinReader.readLine()) != null)
                    {
                        tokens = line.split(",", 2);
                        joinData.put(tokens[0], tokens[1]);
                    }
                } finally { joinReader.close(); }
            }
        } catch (IOException e)
        {
            System.err.println("Exception reading DistributedCache: " + e);
        }
    }
}
```

9. 其他处理技术

查询任务相关信息

可以通过Configuration对象，使用预定义的属性名称查询计算作业相关的信息。

| Property | Type | Description |
|-------------------------------------|---------|---|
| <code>mapred.job.id</code> | String | The job ID |
| <code>mapred.jar</code> | String | The jar location in job directory |
| <code>job.local.dir</code> | String | The job's local scratch space |
| <code>mapred.tip.id</code> | String | The task ID |
| <code>mapred.task.id</code> | String | The task attempt ID |
| <code>mapred.task.is.map</code> | boolean | Flag denoting whether this is a map task |
| <code>mapred.task.partition</code> | int | The ID of the task within the job |
| <code>map.input.file</code> | String | The file path that the mapper is reading from |
| <code>map.input.start</code> | long | The offset into the file of the start of the current mapper's input split |
| <code>map.input.length</code> | long | The number of bytes in the current mapper's input split |
| <code>mapred.work.output.dir</code> | String | The task's working (i.e., temporary) output directory |

划分多个输出文件集合

缺省情况下，MapReduce将产生包含一至多个文件的单个输出数据文件集合。但有时候作业可能需要输出多个文件集合。

比如：在处理巨大的访问日志文件时，由于文件太大我们可能希望按每天的日期将访问日志记录输出为每天日期下的文件。在处理专利数据集时，我们希望根据不同国家，将每个国家的专利数据记录输出到不同国家的文件目录中。

Hadoop提供了MultipleOutputFormat类来快速完成这一处理功能。

在Reduce进行数据输出前， MultipleOutputFormat将调用一个方法以决定输出的文件名是什么。通常需要继承并实现

MultipleOutputFormat的一个子类并实现其中的

generateFileNameForKeyValue()方法以根据当前的键值对h和当前数据文件名由程序产生并返回一个输出文件路径：

[illegible]

划分多个输出文件集合

例如：将专利描述文件数据集按照国家进行多文件集合输出

“PATENT”, “GYEAR”, “GDATE”, “APPYEAR”, “COUNTRY”, “POSTATE”, “ASSIGNEE”, “
ASSCODE”, “CLAIMS”, “NCLASS”, “CAT”, “SUBCAT”, “CMADE”, “CRECEIVE”, “RATIOCI
T”, “GENERAL”, “ORIGINAL”, “FWDAPLAG”, “BCKGTLAG”, “SELFCTUB”, “SELFCTLB”,
“SECDUPBD”, “SECDLWBD”

3070801, 1963, 1096, “BE”, “”, 1, 269, 6, 69, 1, 0, , , , , , , ,

3070802, 1963, 1096, “US”, “TX”, 1, 2, 6, 63, 0, , , , , , , ,

划分多个输出文件集合

例如：将专利描述文件数据集按照国家进行多文件集合输出

```
public static class MapClass extends Mapper<LongWritable, Text, NullWritable, Text>
{   public void map(LongWritable key, Text value, Context context)
                                throws IOException, InterruptedException
    {   context.write (NullWritable.get(), value);
        // NullWritable.get() 返回singleton单一实例
    }

    public static class SaveByCountryOutputFormat
                                extends MultipleTextOutputFormat<NullWritable,Text>
    {
        protected String generateFileNameForKeyValue
                                (NullWritable key, Text value, String filename)
        {   String[] arr = value.toString().split(",", -1);
            String country = arr[4].substring(1,3);
            return country + "/" + filename;
        }
    }
}
```

划分多个输出文件集合

```
public class MultiFileDemo
{
    public static void main(String[] args) throws Exception
    {
        Configuration conf = new Configuration();
        Job job = new Job(conf, MultiFileDemo.class);
        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        job.setJobName("MultiFileDemo");
        job.setMapperClass(MapClass.class);
        job.setInputFormat(TextInputFormat.class);
        job.setOutputFormat(SaveByCountryOutputFormat.class);
        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Text.class);
        job.setNumReduceTasks(0);
        Job.waitForCompletion(true);
    }
}
```

划分多个输出文件集合

执行结果：

```
ls output/
```

| | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|
| AD | BN | CS | GE | IN | LC | MT | PH | SV | VE |
| AE | BO | CU | GF | IQ | LI | MU | PK | SY | VG |
| AG | BR | CY | GH | IR | LK | MW | PL | SZ | VN |
| AI | BS | CZ | GL | IS | LR | MX | PT | TC | VU |
| AL | BY | DE | GN | IT | LT | MY | PY | TD | YE |
| AM | BZ | DK | GP | JM | LU | NC | RO | TH | YU |
| AN | CA | DO | GR | JO | LV | NF | RU | TN | ZA |
| AR | CC | DZ | GT | JP | LY | NG | SA | TR | ZM |
| AT | CD | EC | GY | KE | MA | NI | SD | TT | ZW |
| AU | CH | EE | HK | KG | MC | NL | SE | TW | |
| AW | CI | EG | HN | KN | MG | NO | SG | TZ | |
| AZ | CK | ES | HR | KP | MH | NZ | SI | UA | |
| BB | CL | ET | HT | KR | ML | OM | SK | UG | |
| BE | CM | FI | HU | | | PA | SM | US | |
| BG | CN | FO | ID | | | PE | SN | UY | |
| BH | CO | FR | IE | | | PF | SR | UZ | |
| BM | CR | GB | IL | | | PG | SU | VC | |

所有来自
“AD” 国家
的专利记录将
输出到AD子
目录下

```
ls output/AD
```

```
part-00003      part-00005      part-00006
```

```
head output/AD/part-00006
```

```
5765303,1998,14046,1996,"AD",,,1,12,42,5,59,11,1,0.4545,0,0,1,67.3636,,,
5785566,1998,14088,1996,"AD",,,1,9,441,6,69,3,0,1,,0.6667,,4.3333,,,
5894770,1999,14354,1997,"AD",,,1,,82,5,51,4,0,1,,0.625,,7.5,,,
```

输入输出到关系数据库

MapReduce用于处理存储在HDFS中的大规模数据，但现实环境中有很多应用数据保存在关系数据库中，因此，Hadoop提供了访问关系数据库的能力以便在需要时能用MapReduce技术处理关系数据库中的数据。这在基于MapReduce进行联机数据分析处理时尤为有用。

- OLTP (online transaction processing)

联机事务处理：主要是关系数据库应用系统中前台常规的各种事务处理

- OLAP (online analytical processing)

联机分析处理：主要是进行基于数据仓库的后台数据分析和挖掘，提供优化的客户服务和运营决策支持

- OLTP与OLAP一般采用分离的数据库,前者数据库负责大量的常规的事务处理,后者用数据仓库应对大量的数据分析处理负载。

输入输出到关系数据库

企业数据库应用系统



Extract: 从OLTP数据库中抽取事务数据

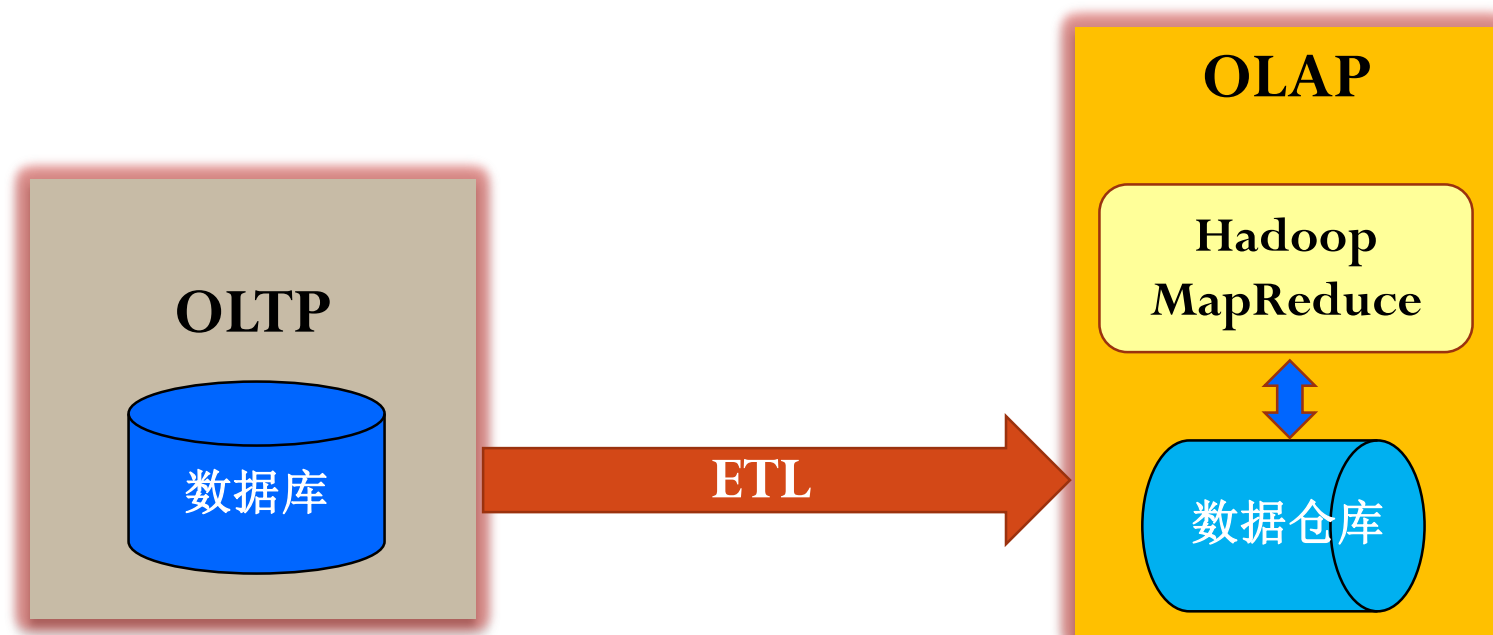
Transform: 转换为数据仓库中的数据格式

Load: 装载到数据仓库中

问题：OLAP端基于关系数据库的数据仓库解决方案，在数据量巨大的情况下，复杂数据分析和挖掘处理的负载很大，速度性能跟不上

输入输出到关系数据库

企业数据库应用系统



解决方案：提供基于MapReduce大规模数据并行处理的OLAP！

问题：如何从MapReduce访问关系数据库？

输入输出到关系数据库

从数据库中输入数据

Hadoop提供了相应的从关系数据库查询和读取数据的接口

- DBInputFormat: 提供从数据库读取数据的格式
- DBRecordReader: 提供读取数据记录的接口

虽然Hadoop允许用以上接口从数据库中直接读取数据记录作为MapReduce的输入，但处理效率不理想，因此，仅适合读取小量数据记录的计算和应用，不适合OLAP数据仓库大量数据的读取处理。

读取大量数据记录一个更好的解决办法是，用数据库中的Dump工具将大量待分析数据输出为文本数据文件，并上载到HDFS中进行处理。

输入输出到关系数据库

向数据库中输出计算结果

基于数据仓库的数据分析和挖掘输出结果的数据量一般不会太大，因而可能适合于直接向数据库写入。Hadoop提供了相应的向关系数据库直接输出计算结果的编程接口

- DBOutputFormat：提供向数据库输出数据的格式
- DBRecordWriter：提供向数据库写入数据记录的接口
- DBConfiguration提供数据库配置和创建连接的接口

创建数据库连接

DBConfiguration 类中提供了一个静态方法创建数据库连接：

- `public static void configureDB(Job job, String driverClass, String dbUrl, String userName, String passwd)`

指定写入的数据表和字段

DBOutputFormat中提供了一个静态方法完成这一工作：

- `public static void setOutput(Job job, String tableName, String... fieldNames)`

输入输出到关系数据库

向数据库中输出计算结果

Configuration示例

```
Configuration conf = new Configuration();  
Job job = new Job(conf, JobClass.class);  
job.setOutputFormat(DBOutputFormat.class);  
DBConfiguration.configureDB(job, "com.mysql.jdbc.Driver",  
    "jdbc:mysql://db.host.com/mydb", "username", "password")  
DBOutputFormat.setOutput(job, "Events", "event_id", "time");  
    // 向Events表输出event_id和time字段
```

输入输出到关系数据库

向数据库中输出计算结果

实现DBWritable

为了实际完成向数据库中数据写入，程序员要实现DBWritable：

```
public class EventsDBWritable implements Writable, DBWritable
{
    private int id;
    private long timestamp;
    public void write(DataOutput out) throws IOException
    {
        out.writeInt(id); out.writeLong(timestamp);
    }
    public void readFields(DataInput in) throws IOException
    {
        id = in.readInt(); timestamp = in.readLong();
    }
    public void write(PreparedStatement statement) throws SQLException
    {
        statement.setInt(1, id); statement.setLong(2, timestamp);
    }
    public void readFields(ResultSet resultSet) throws SQLException
    {
        id = resultSet.getInt(1); timestamp = resultSet.getLong(2);
    }
    // 除非使用DBInputFormat直接从数据库输入数据,否则readFields方法不会被调用
}
```

Thanks!