MapReduce高级程序设计

摘要

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

复合键值对的使用

- 思路：用复合键让系统完成排序
  - 问题：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。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。
  - 改进方法：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。
  - 代价：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。

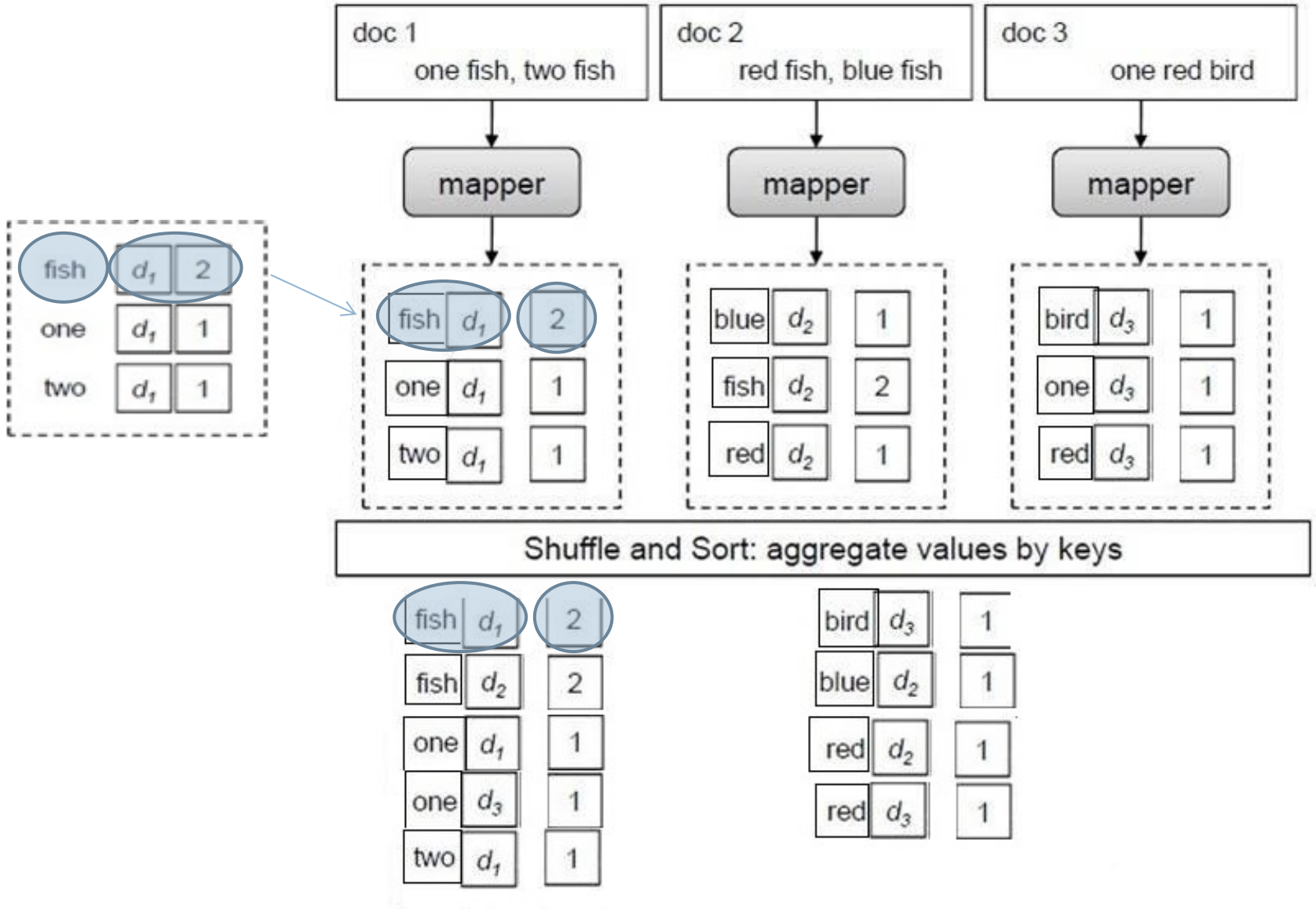
复合键值对的使用

带频率的倒排索引示例

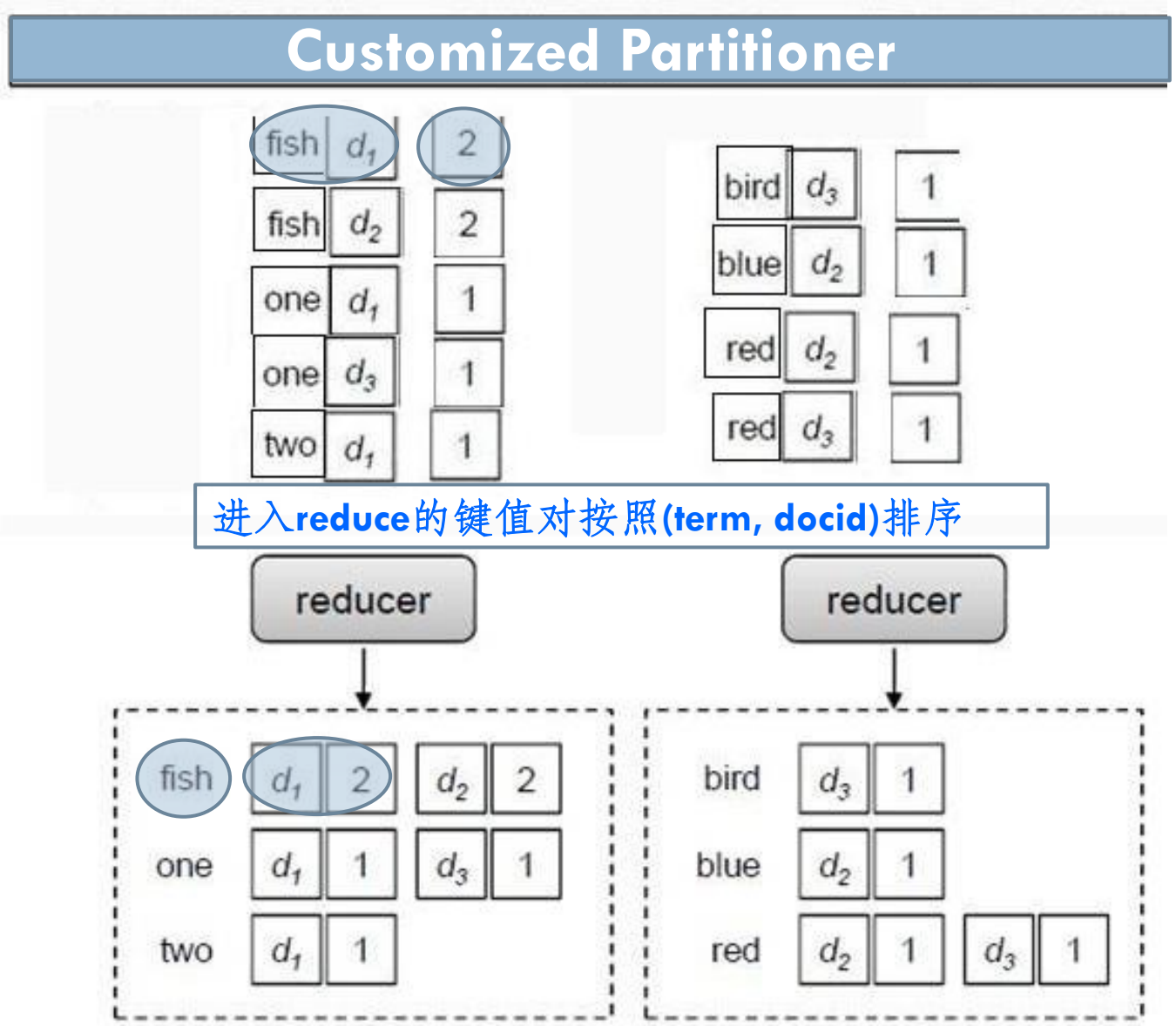
```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting <n, H{t}>)
1: class Reducer
2:     procedure Reduce(term t, postings [<n1, f1>, <n2, f2>…])
3:         P ← new List
4:         for all posting <a, f> ∈ postings [<n1, f1>, <n2, 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)
```

复合键值对的使用

- 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。


doc 1
one fish, two fish
doc 2
red fish, blue fish
doc 3
one red bird
mapper
mapper
mapper
fish d1 2
one d1 1
two d1 1
blue d2 1
fish d2 2
red d2 1
bird d3 1
one d3 1
red d3 1
Shuffle and Sort: aggregate values by keys
fish d1 2
fish d2 2
one d1 1
one d3 1
two d1 1
bird d3 1
blue d2 1
red d2 1
red d3 1

Customized Partitioner



复合键值对的使用

- 思路：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->stripes)。

复合键值对的使用

- 例如：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) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

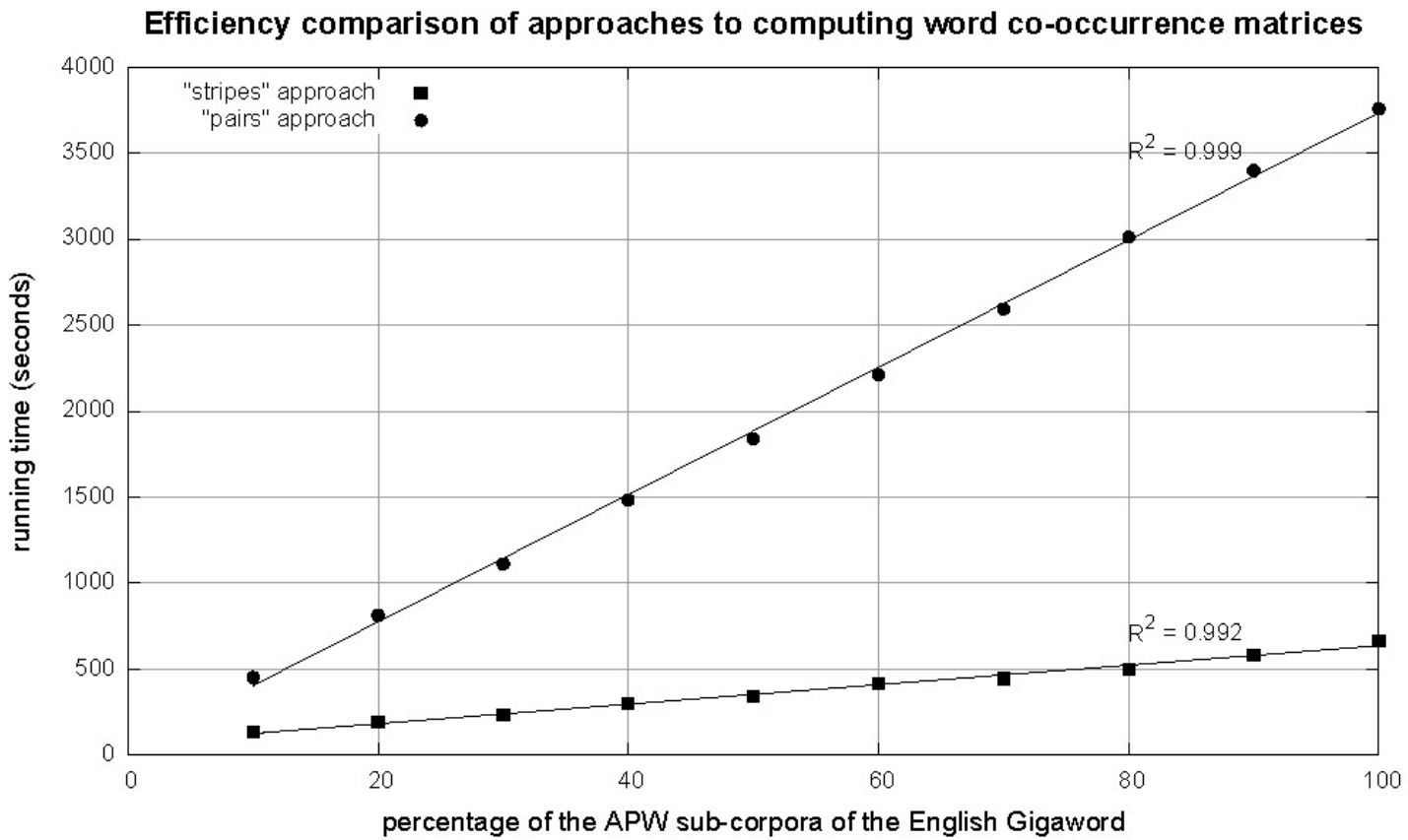
a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

  - 然后，在Reduce阶段，把每个单词a的键值对(条)进行累加：

```
    a → { b: 1,         d: 5, e: 3 }
+   a → { b: 1, c: 2, d: 2,        f: 2 }
-------------------------------------------
    a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

复合键值对的使用

- 单词同现矩阵算法



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)

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
BooleanWritable	Wrapper for a standard Boolean variable
ByteWritable	Wrapper for a single byte
DoubleWritable	Wrapper for a Double
FloatWritable	Wrapper for a Float
IntWritable	Wrapper for a Integer
LongWritable	Wrapper for a Long
NullWritable	Placeholder when the key or value is not needed
Text	Wrapper to store text using the UTF-8 format

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(小于), 0(等于), 1(大于)
    }
}
```

用户自定义数据类型


```
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);
     }
}
```

用户自定义输入输出格式

- 数据输入格式（InputFormat）用于描述MapReduce作业的数据输入规范。
- MapReduce框架依靠数据输入格式完成输入规范检查（比如输入文件目录的检查）、对数据文件进行输入分片（InputSplit），以及提供从输入分块中将数据记录逐一读出，并转换为Map过程的输入键值对等功能。
- TextInputFormat是系统缺省的数据输入格式。

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
KeyValueTextInput Format	Parses lines into key-val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInput Format	A Hadoop-specific high-performance binary format	user-defined	user-defined

用户自定义输入输出格式

- Hadoop内置的文件输入格式
- AutoInputFormat, CombineFileInutFormat, CompositeInputFormat, DBInputFormat, FileInputFormat, KeyValueTextInputFormat, LineDocInputFormat, MultiFileInputFormat, NLineInputFormat, SequenceFileAsBinaryInputFormat, SequenceFileAsTextInputFormat, SequenceFileInputFilter, SequenceFileInputFormat, StreamInputFormat, TextInputFormat

用户自定义输入输出格式

- 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

用户自定义输入输出格式

- Hadoop内置的RecordReader
  - CombineFileRecordReader, DBInputFormat.DBRecordReader, InnerJoinRecordReader, JoinRecordReader, KeyValueLineRecordReader, LineDocRecordReader, MultiFilterRecordReader, OuterJoinRecordReader, OverrideRecordReader, SequenceFileAsBinaryInputFormat.SequenceFileAsBinaryRecordReader, SequenceFileAsTextRecordReader, SequenceFileRecordReader, StreamBaseRecordReader, StreamXmlRecordReader, WrappedRecordReader

用户自定义输入输出格式

- 用户自定义InputFormat和RecordReader

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

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

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

用户自定义输入输出格式


```
public class FileNameLocRecordReader 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();
   }
}
```

用户自定义输入输出格式


```
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(FileNameLocInputFormat.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();     }
    }
}
```

用户自定义输入输出格式


```
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: FileNameLocInputFormat
        // RecordReader: FileNameLocRecordReader
        // 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

- 数据输出格式（OutputFormat）用于描述MapReduce作业的数据输出规范。
- MapReduce框架依据数据输出格式完成输出规范检查（如检查输出目录是否存在）以及提供作业结果数据输出等功能。
- TextOutputFormat是系统缺省的数据输出格式。

用户自定义输入输出格式

- 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 outputs

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

用户自定义输入输出格式

- 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

用户自定义输入输出格式

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结合。
  - 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(org.apache.hadoop.mapred.lib.MultipleOutputFormat)来快速完成这一处理功能。在Reduce进行数据输出前， MultipleOutputFormat将调用一个内部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由程序产生并返回一个输出文件路径：

    protected String generateFileNameForKeyValue(K key, V value, String name)

用户自定义输入输出格式

- 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", "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,,,,,,,,

用户自定义输入输出格式

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;
    }
}
```

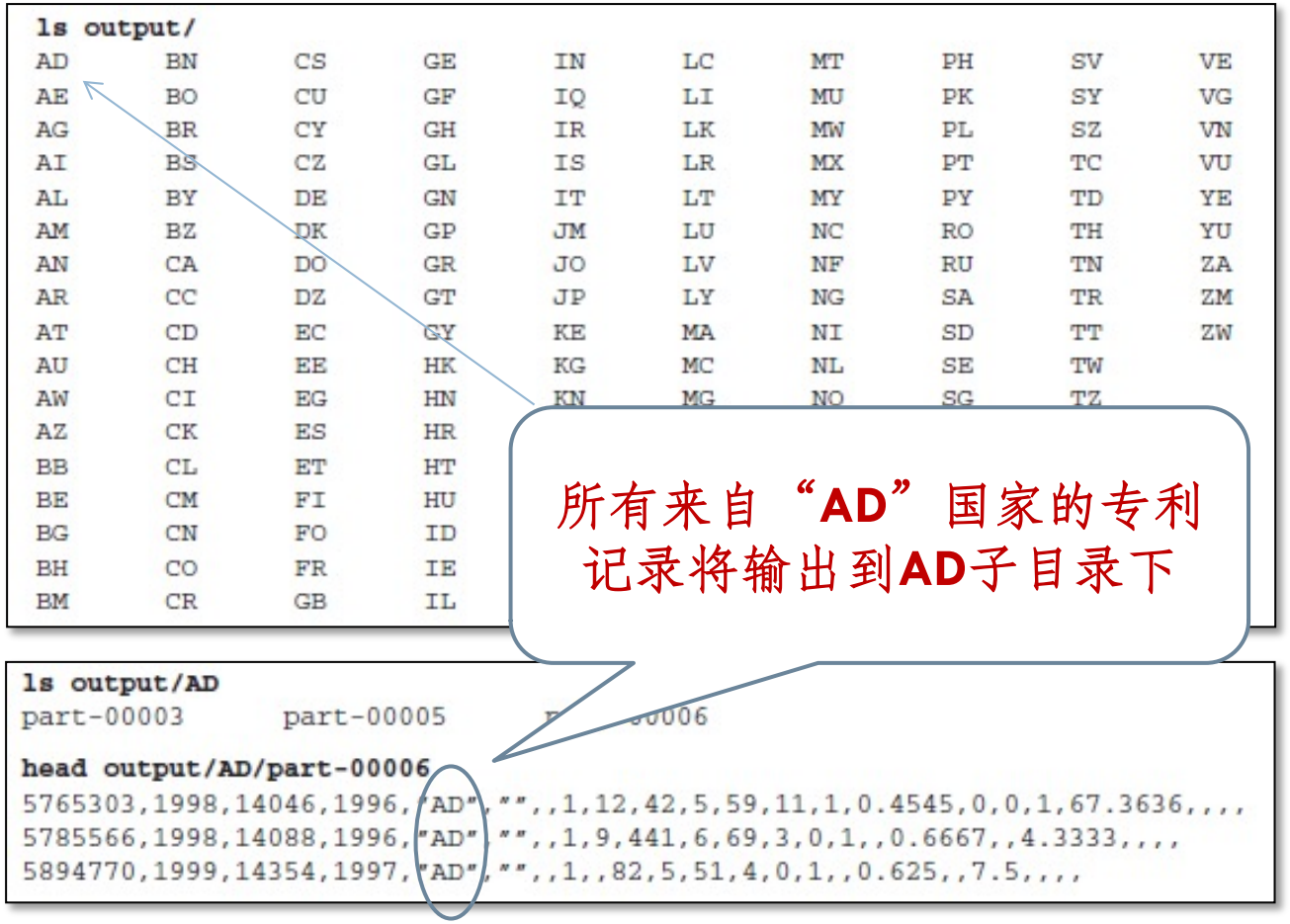
用户自定义输入输出格式

- 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); }
}
```

用户自定义输入输出格式

- 执行结果



用户自定义输入输出格式

- MultipleOutputFormat (org.apache.hadoop.mapred.lib.MultipleOutputFormat)
  - MultipleOutputFormat是Hadoop的OutputFormat的一个扩展，用于处理多个输出文件。
  - 它允许你为每个Reducer任务定义一个不同的OutputFormat。这意味着你可以为每个Reducer任务指定不同的输出目录和输出文件格式。
  - 这对于根据特定的数据或逻辑将数据分发到不同的输出目录非常有用，更适合在整个MapReduce作业级别控制多个输出文件的格式和目录。
- MultipleOutputs (org.apache.hadoop.mapreduce.lib.output.MultipleOutputs)
  - MultipleOutputs是一个更高级别的API，用于在Mapper或Reducer内部根据某些条件将数据输出到多个文件或目录。
  - 它允许你在Mapper或Reducer内部为不同的输出文件指定不同的键值对，而不需要为每个Reducer任务创建不同的OutputFormat。
  - 这对于根据数据的属性或业务逻辑将数据分发到多个输出目录非常有用，而无需创建多个Reducer任务，更适合在Mapper或Reducer内部根据条件动态控制多个输出文件。

用户自定义输入输出格式

- MultipleOutputFormat (org.apache.hadoop.mapred.lib.MultipleOutputFormat)
  - MultipleOutputFormat是Hadoop的OutputFormat的一个扩展，用于处理多个输出文件。
  - 它允许你为每个Reducer任务定义一个不同的OutputFormat。这意味着你可以为每个Reducer任务指定不同的输出目录和输出文件格式。
  - 这对于根据特定的数据或逻辑将数据分发到不同的输出目录非常有用，更适合在整个MapReduce作业级别控制多个输出文件的格式和目录。
- MultipleOutputs (org.apache.hadoop.mapreduce.lib.output.MultipleOutputs)
  - MultipleOutputs是一个更高级别的API，用于在Mapper或Reducer内部根据某些条件将数据输出到多个文件或目录。
  - 它允许你在Mapper或Reducer内部为不同的输出文件指定不同的键值对，而不需要为每个Reducer任务创建不同的OutputFormat。
  - 这对于根据数据的属性或业务逻辑将数据分发到多个输出目录非常有用，而无需创建多个Reducer任务，更适合在Mapper或Reducer内部根据条件动态控制多个输出文件。

用户自定义输入输出格式

- org.apache.hadoop.mapred.* vs org.apache.hadoop.mapreduce.*
- 历史演变：
  - mapred包是Hadoop的早期版本中使用的包，用于实现MapReduce编程模型。
  - mapreduce包是Hadoop 0.20版本之后引入的，用于替代mapred包。这一改变是为了改进和优化MapReduce的性能以及提供更灵活的编程接口。
- 性能优化：
  - mapreduce包相对于mapred包进行了许多性能优化，包括更好的资源管理、任务调度、错误处理等方面的改进，以提高MapReduce作业的执行效率。
- API灵活性：
  - mapreduce包引入了一种新的API，使得编写MapReduce作业更加灵活和容易。这一API的设计更加现代化，与标准Java编程实践更加一致，因此编写和维护MapReduce作业变得更容易。

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.class)
```

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

例如，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,,,,,,

用户自定义Partitioner和Combiner

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：如每碰到一个<year, country >，即emit(year, 1)有问题吗?

答案：有问题。因为可能会有从不同Map节点发来的同样的<year, country>，因此会出现对同一国家的重复计数

解决办法：在Reduce中仅计数同一年份下不同的国家个数

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

用户自定义Partitioner和Combiner

- 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,...])后部大量重复的[1,1,1,...],
        // 归并为<year, country>的1次出现
          context.write(key, new IntWritable(1));
     } // 输出key: <year, country>; value: 1
}
```

- 并在job中设置Combiner：

```
job. setCombinerClass(NewCombiner.class)
```

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。
- 例如：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

迭代MapReduce计算

- 页面排序算法PageRank

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

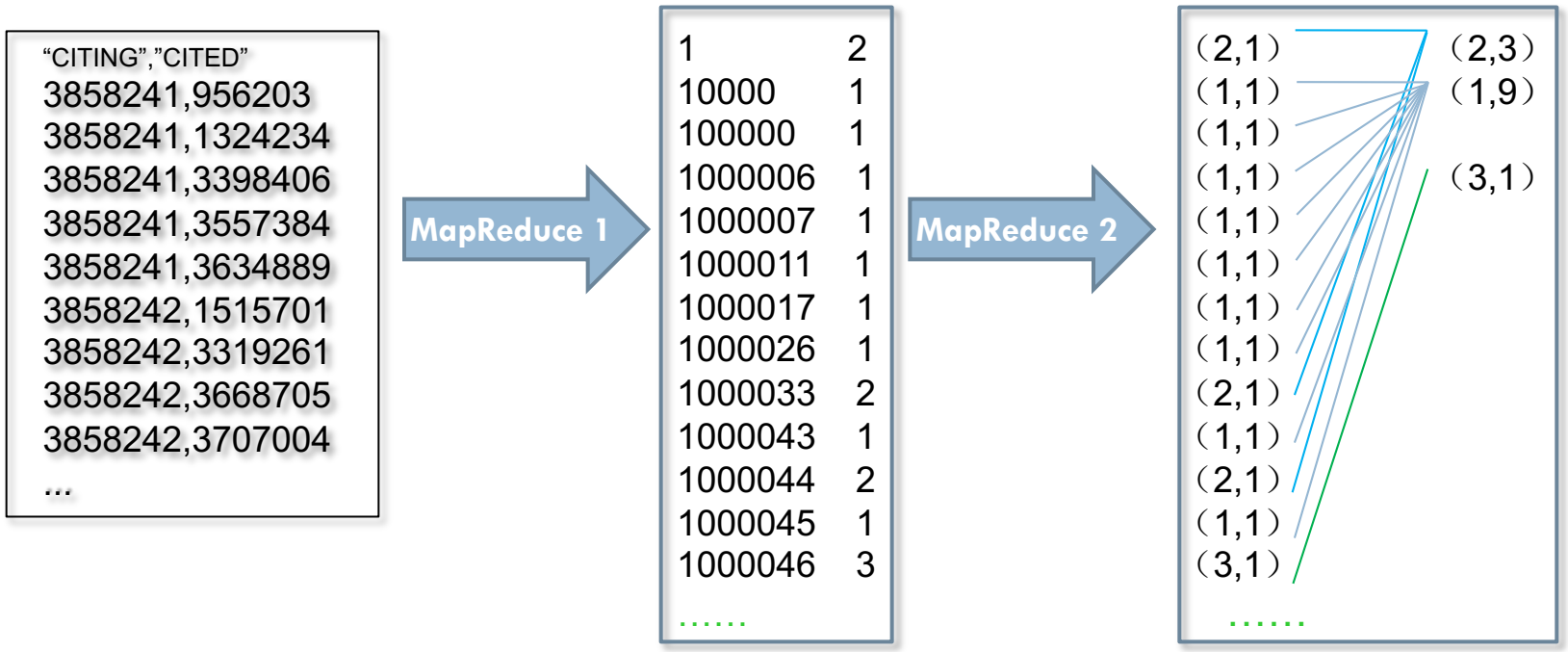
- 问题是在求解$PR(p_i)$时，需要递归调用$PR(p_j)$，而$PR(p_j)$本身也是待求解的。因此，我们只能先给每个网页赋一个假定的PR值，如0.5。但这样求出的$PR(p_i)$肯定不准确。然而，当用求出的PR值反复进行迭代计算时，会越来越趋近于最终的准确结果。
- 因此，需要用迭代方法循环运行MapReduce过程，直至第n次迭代后的结果与第n-1次的结果小于某个指定的阈值时结束，或者通过经验控制循环固定的次数。

迭代MapReduce计算


```
public class PageRankDriver {
    private static int times = 10;
    public static void main(String args[]) throws Exception{
        String[] forGB = {"", args[1]+"/Data0"};
        forGB[0] = args[0];
        GraphBuilder.main(forGB);
        String[] forItr = {"Data","Data"};
        for (int i=0; i<times; i++) {
            forItr[0] = args[1]+"/Data"+(i);
            forItr[1] = args[1]+"/Data"+(i+1);
            PageRankIter.main(forItr);
        }
        String[] forRV = {args[1]+"/Data"+times, args[1]+"/FinalRank"};
        PageRankViewer.main(forRV);
    }
}
```

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



组合式MapReduce程序设计

- 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穿起来，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);
```

组合式MapReduce程序设计

- MapReduce子任务的顺序化执行
  - 同样，链式MapReduce中的每个子任务需要穿件独立的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。

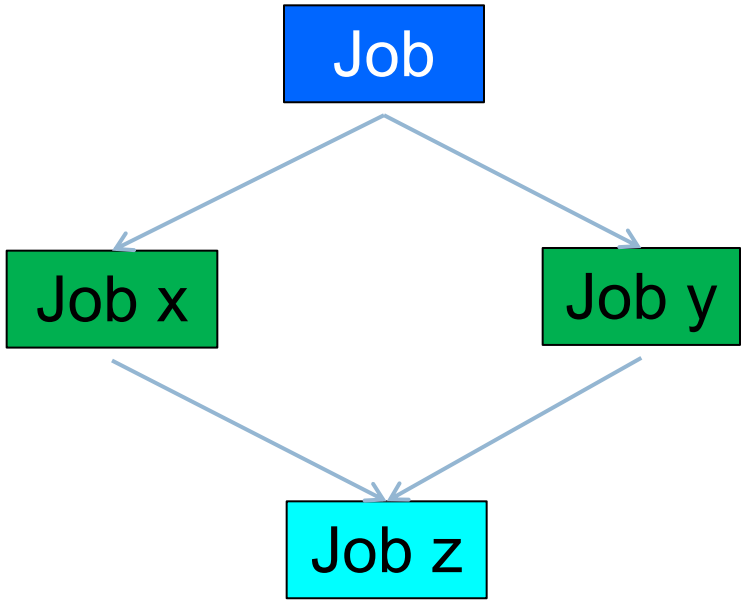
```
Configuration jobconf1= new Configuration();
job1 = new Job(jobconf1, “Job1");
job1.setJarByClass(jobclass1);
……
FileInputFormat.addInputPath(job1,  inpath1);
FileOutputFormat.setOutputPath(job1, outpath1);
job1.waitForCompletion(true);
```

```
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);
```

组合式MapReduce程序设计

- 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Dx，y处理另一个数据集Dy，然后z需要将Dx和Dy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。



组合式MapReduce程序设计

- 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
```

组合式MapReduce程序设计

- 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();
```

组合式MapReduce程序设计

- 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，但将增加很多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，将这些辅助的Map和Reduce过程与核心的Map和Reduce过程合并为一个过程链，从而完成整个作业。

组合式MapReduce程序设计

- MapReduce前处理和后处理步骤的链式执行
  - Hadoop提供了链式Mapper(org.apache.hadoop.mapreduce.lib.chain.ChainMapper)和链式Reducer (org.apache.hadoop.mapreduce.lib.chain.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> kclass,    //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
Configuration mapperConf           //待加入的map的conf
) throws IOException
```

组合式MapReduce程序设计

- 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);
Configuration  map1Conf = new Configuration(false);
ChainMapper.addMapper(job,  Map1.class,  LongWritable.class,  Text.class, Text.class,  Text.class, map1Conf);
Configuration map2Conf = new Configuration(false);
ChainMapper.addMapper(job, Map2.class, Text.class, Text.class, LongWritable.class, Text.class, map2Conf);
```

组合式MapReduce程序设计

- MapReduce前处理和后处理步骤的链式执行

```
Configuration reduceConf = new Configuration(false);
ChainReducer.setReducer(job, Reduce.class, LongWritable.class, Text.class,
                        Text.class, Text.class, true, reduceConf);
Configuration map3Conf = new Configuration(false);
ChainReducer.addMapper(job, Map3.class, Text.class, Text.class,
                        LongWritable.class, Text.class, true,  map3Conf);
Configuration map4Conf = new Configuration(false);
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class,
        LongWritable.class, Text.class, true, map4Conf);
job.waitForCompletion(true);
```

ChainReducer.setReducer()方法必须在ChainReducer最开始的地方使用，其后方可加入后续的辅助处理Mapper；另一个需要注意的问题是，这些链式Mapper和Reducer之间传递的键值对数据类型必须保持前后一致。

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。

本例引自Chuck Lam，Hadoop in Action，2010，Manning Publications

多数据源的连接

设有两个数据集，一个是顾客数据集：

Customer ID, Name, and Phone Number

1, Stephanie Leung,555-555-5555

2, Edward Kim,123-456-7890

3, Jose Madriz,281-330-8004

4, David Stork,408-555-0000

第二个是顾客的订单数据集：

Customer ID, Order ID, Price, Purchase Date

3, A, 12.95, 02-Jun-2008

1, B, 88.25, 20-May-2008

2, C, 32.00, 30-Nov-2007

3, D, 25.02, 22-Jan-2009

以CustomerID进行内连接(inner join)后的数据记录将是：

Customer ID, Name, and Phone Number, Order ID, Price, Purchase Date

1, Stephanie Leung,555-555-5555, B, 88.25,20-May-2008

2, Edward Kim,123-456-7890, C, 32.00,30-Nov-2007

3, Jose Madriz,281-330-8004, A, 12.95,02-Jun-2008

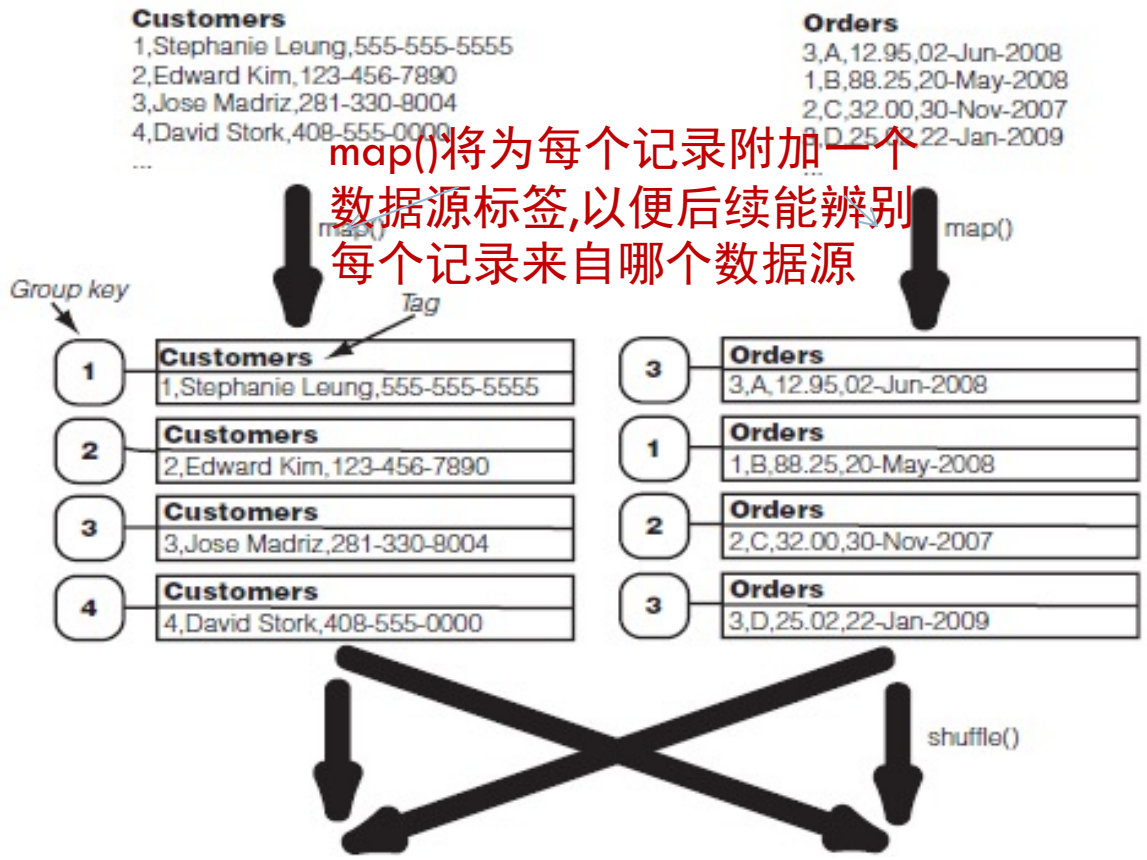
3, Jose Madriz,281-330-8004, D, 25.02,22-Jan-2009

多数据源的连接

- **Reduce-side Join是最常见的Join实现方式**，适用于两个或多个大型数据集之间的关联操作。
- 核心思路：
  - Mapper阶段
    - 为每条数据打标签（标记数据来源，例如Customers或Orders）
    - 输出<JoinKey, TaggedValue>，其中JoinKey是关联键（如CustomerID）
  - Reducer阶段
    - 将相同JoinKey的不同数据来源的值分组
    - 对不同来源的数据进行笛卡尔积组合

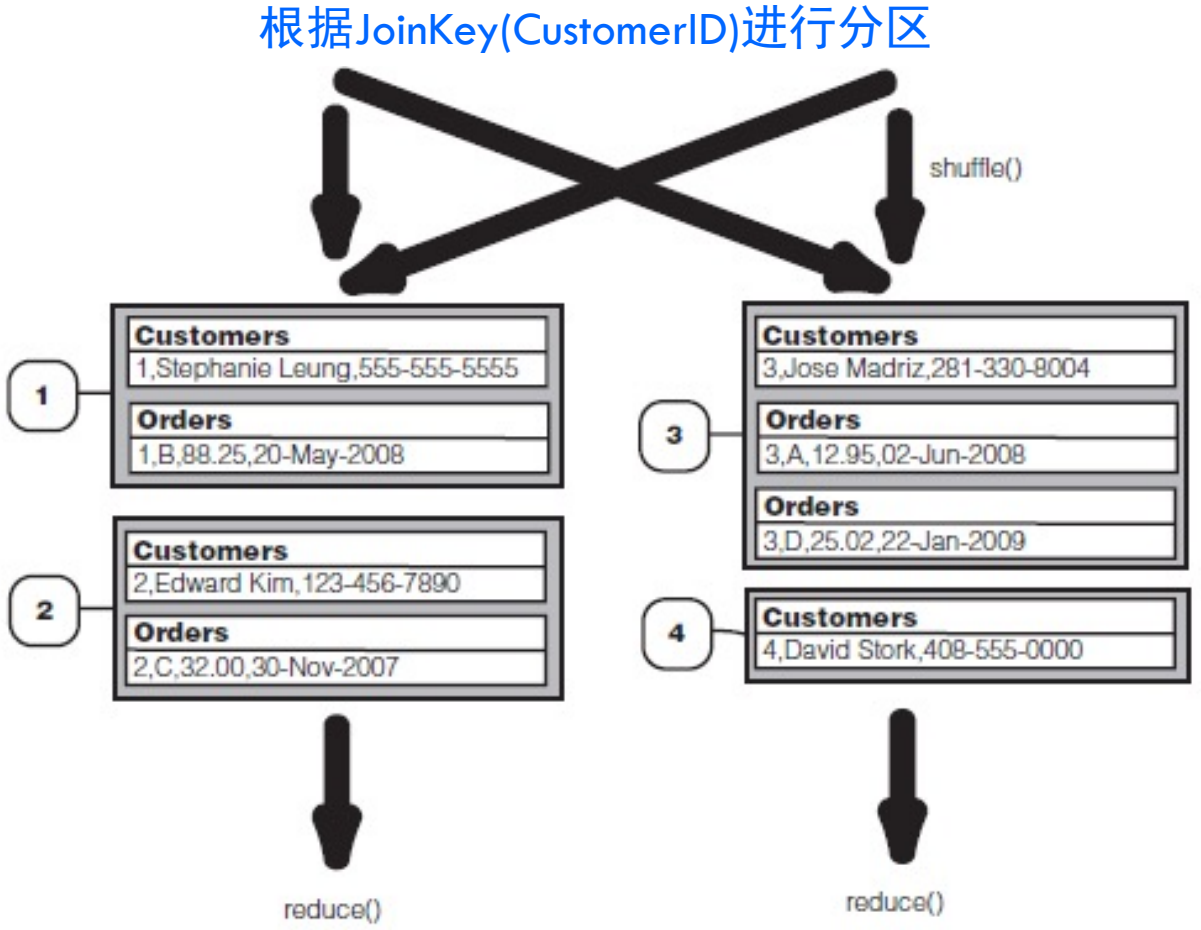
多数据源的连接

- Mapper阶段

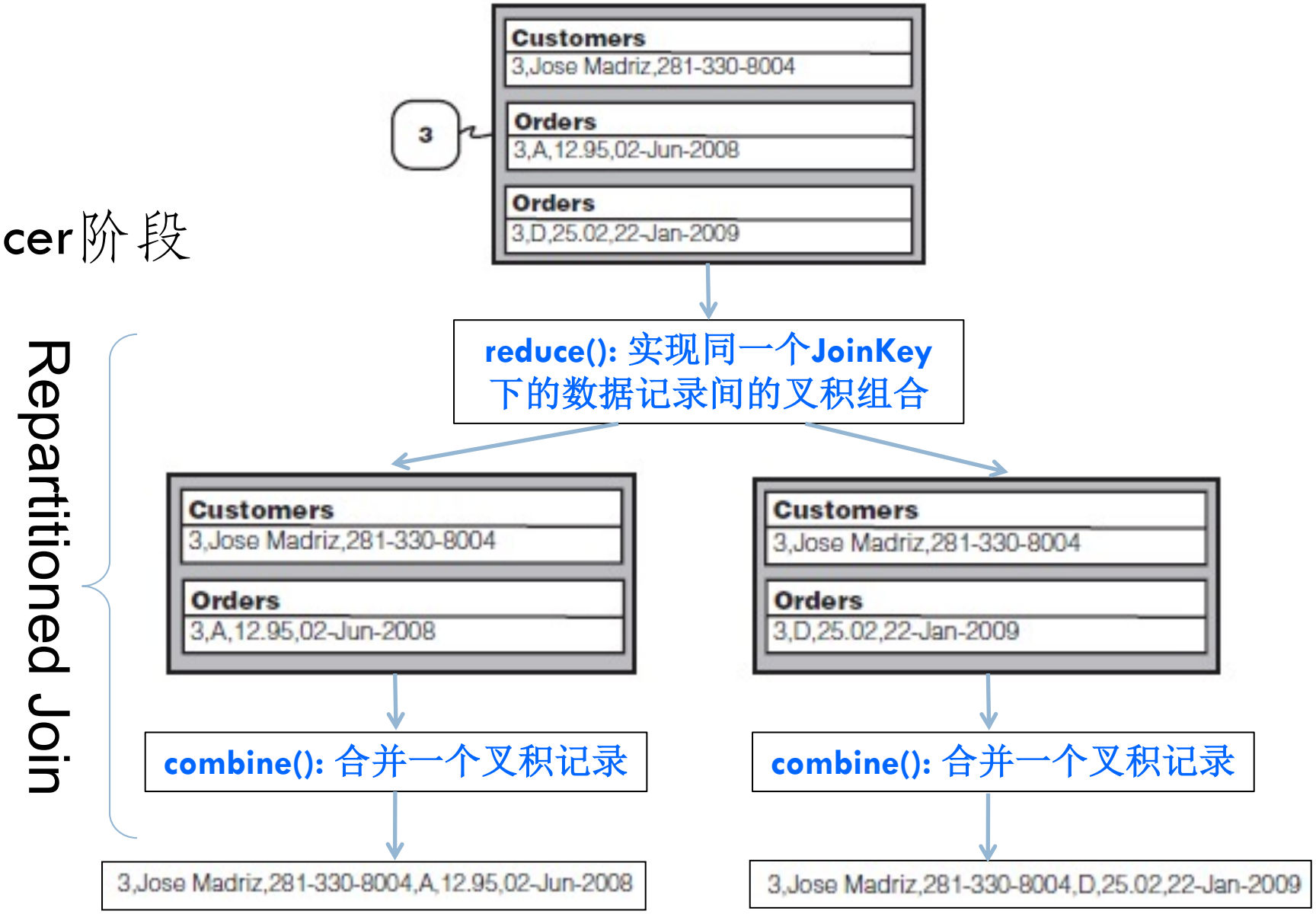


多数据源的连接

- Shuffle阶段



- Reducer阶段


Repartitioned Join
Customers
3,Jose Madriz,281-330-8004
3
Orders
3,A,12.95,02-Jun-2008
Orders
3,D,25.02,22-Jan-2009
reduce(): 实现同一个JoinKey下的数据记录间的叉积组合
Customers
3,Jose Madriz,281-330-8004
Orders
3,A,12.95,02-Jun-2008
Customers
3,Jose Madriz,281-330-8004
Orders
3,D,25.02,22-Jan-2009
combine(): 合并一个叉积记录
combine(): 合并一个叉积记录
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009

多数据源的连接

- 优化建议
  - 过滤空值：在Mapper中提前过滤无效的JoinKey
  - Combiner：如果数据倾斜严重，可尝试自定义Combiner预处理
  - Map-Side Join：若其中一个数据集很小，使用 DistributedCache 实现 Map 端 Join 更高效
  - 使用Hive：直接通过SQL或脚本语言实现Join，避免手写MapReduce

多数据源的连接

用文件复制方法实现Map-side Join

- 前述Reduce端Join方法中，Join操作直到Reduce阶段才能处理，很多无效的连接组合数据在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、能够存在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分布复制到所有节点上。
  - 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

多数据源的连接

- 用文件复制方法实现 **Map-side 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);
```

多数据源的连接

- 用文件复制方法实现Map-side 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保存在内存中处理。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地区的顾客的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。

多数据源的连接

- 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

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，避免作业参数在程序中的硬编码，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join时程序用硬编码方式指定第一个数据列为join的主键(CustomerID)。但为了要实现一个具有一定通用性的程序，可以任意指定一个列为join主键的话，就需要在程序运行时在命令行中指定join主键所在的数据列。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。

全局参数/数据文件的传递

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“,”隔开的一个长字符串，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。

全局参数/数据文件的传递

- 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
job.setInt("JoinKeyColIdx", Integer.parseInt(args[2]));
……
Job.waitForCompletion(true);
```

全局参数/数据文件的传递

- 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;
    protected 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;
      ……
    }
}
```

全局参数/数据文件的传递

- 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;
    protected void setup(Reducer.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;
    ……
    }
}
```

全局参数/数据文件的传递

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中将这些文件并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

全局参数/数据文件的传递

- 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]);
……
```

- 示例代码可以参考Word Count 2.0

其它处理技术

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

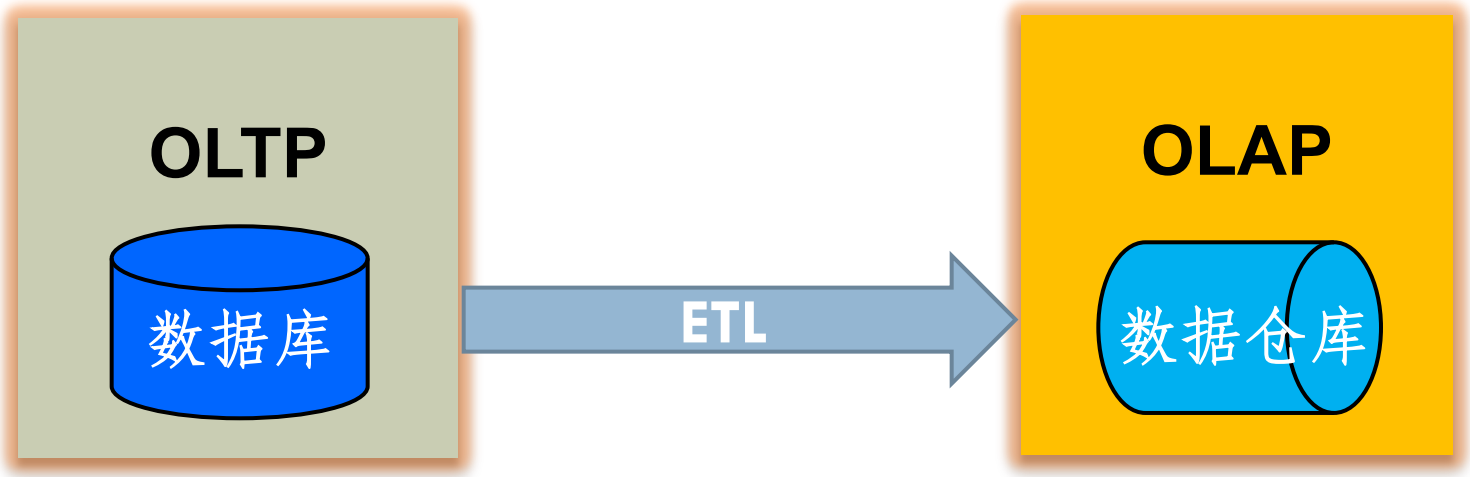
其它处理技术

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。

其它处理技术

企业数据库应用系统

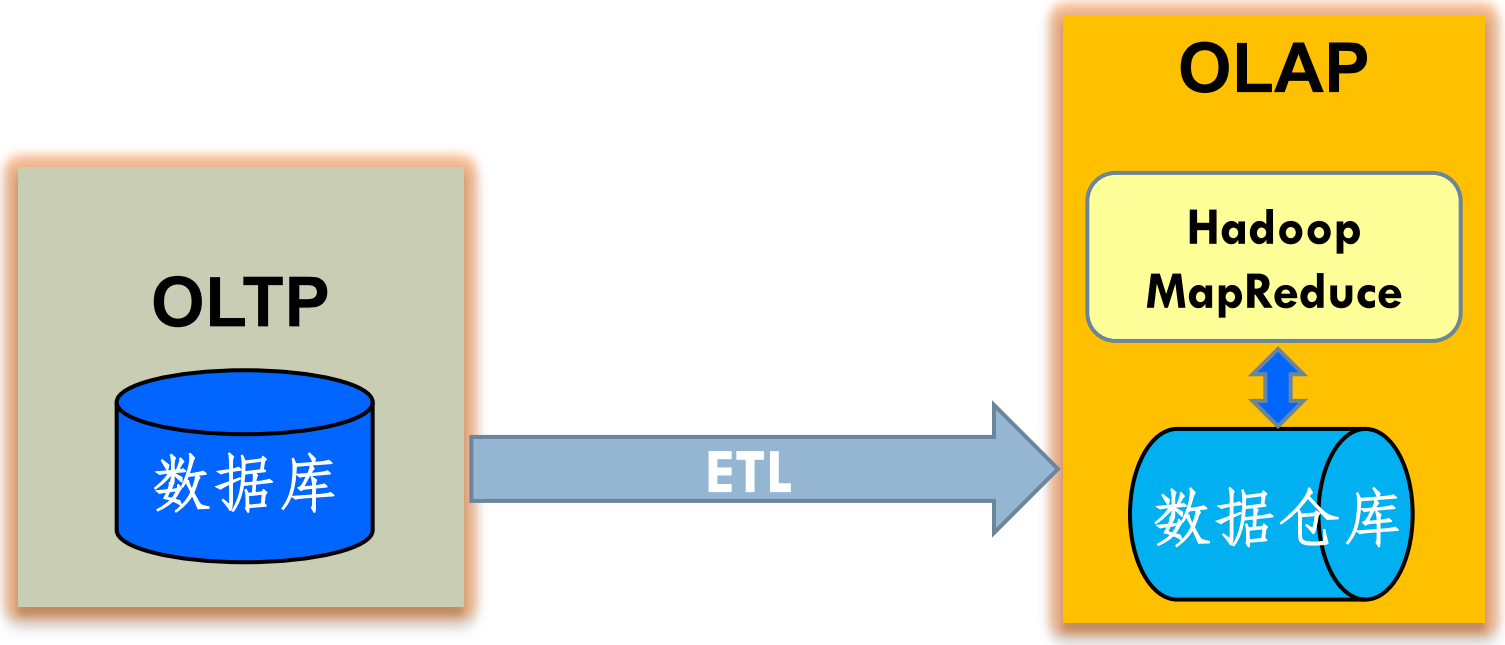


Extract：从OLTP数据库中抽取事务数据
Transform：转换为数据仓库中的数据格式
Load：装载到数据仓库中

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

其它处理技术

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？

其它处理技术

- 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库查询和读取数据的接口 (org.apache.hadoop.mapreduce.lib.db.*)
    - 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。

其它处理技术

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
- 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)

其它处理技术

- 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
```

其它处理技术

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
}
```

THANK YOU

南京大学
NANJING UNIVERSITY

南京大学计算机软件研究所
Institute of Computer Software, Nanjing University