

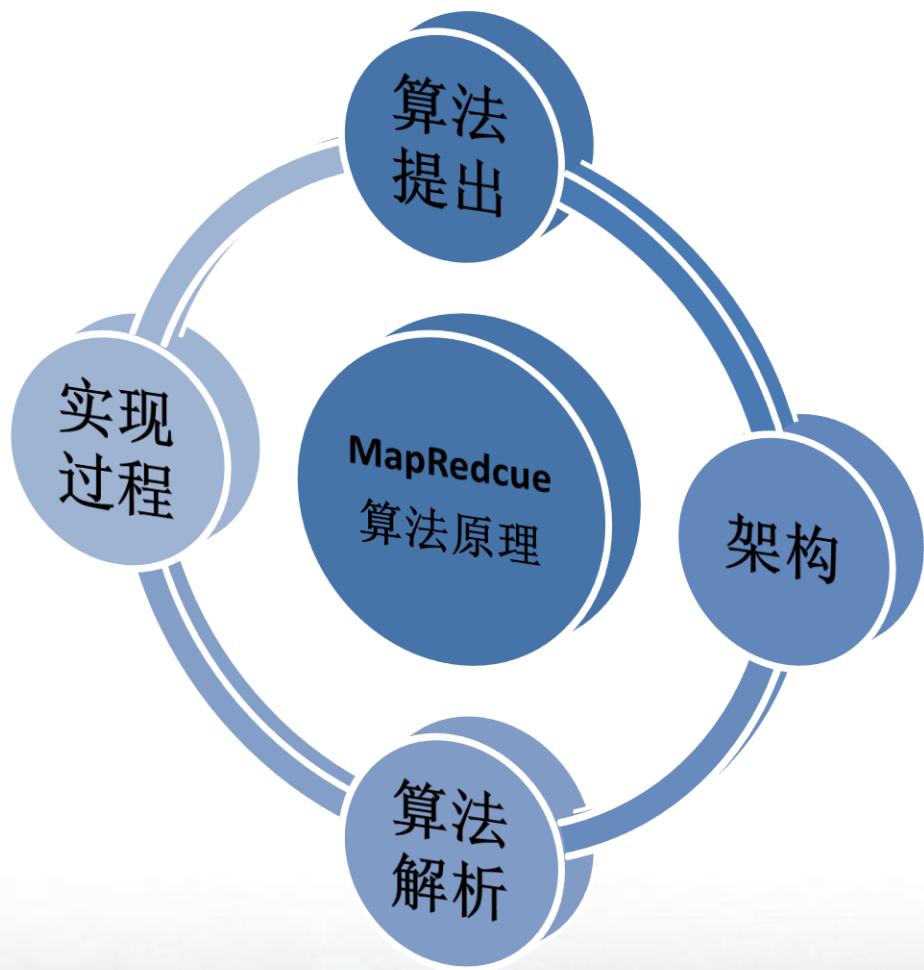


第7讲 MapReduce算法应用

莫同



上一讲回顾



- 算法提出
 - 并行时代
- 架构
 - 架构、容错、优化
- 算法解析
 - 算法步骤
 - 案例代码分析
- 实现过程
 - 数据流转



1

• 应用案例

2

• 算法调优

3

• 应用设计

4

• 应用作业



- 1 • 应用案例
- 2 • 算法调优
- 3 • 应用设计
- 4 • 应用作业

Case1: 词频

- $(a, b) \rightarrow 1; (a, c) \rightarrow 2; (a, d) \rightarrow 5; (a, e) \rightarrow 3; (a, f) \rightarrow 2; (a, b) \rightarrow 1; (a, d) \rightarrow 2; (b, c) \rightarrow 4; (b, d) \rightarrow 2; (b, e) \rightarrow 6; (b, f) \rightarrow 3; (b, c) \rightarrow 9; (a, f) \rightarrow 2; (a, d) \rightarrow 7; (a, d) \rightarrow 2; (b, f) \rightarrow 4; (b, d) \rightarrow 8; (c, f) \rightarrow 7; (a, f) \rightarrow 6; (a, b) \rightarrow 8\dots$
- 如何统计词频?
 - 比如如何统计 $P(b|a)$

- Wordcount' —— 统计各个词对的次数
- 先进行wordcount'，然后按词对的第一个词进行汇总（又一次mapreduce），然后分别除以总数
- 有没有一次完成的方案？

- 算法分析：
 - 如何计算词频？

$$P(B | A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')}$$

- Count(A, B) 容易计算 (wordcount)
- 关键问题在于mapreduce的同时也计算出Count(A)



Case1: 词频

- 增加一个词对(a, *)

(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 }

- a → {*:13, b:1, c:2, d:5, e:3, f:2}



- 统计词频

- 在mapper中增加(a, *)
- 通过partitioner将所有a的送到同一个reducer
- 通过排序将(a, *)排在第一位
- 增加计算 $P(x|a)$

Case2: inverted index

- inverted index——倒排索引
 - 每个文件中有若干词——可以建立文件-词的索引
 - 倒排索引——词-文件
 - 广泛应用于搜索等



Case2: inverted index

- 倒排索引算法思想示意图

Term	Doc 1	Doc 2	Doc 3	Doc 4	Doc 5	Doc 6	Doc 7	Doc 8
aid	0	0	0	1	0	0	0	1
all	0	1	0	1	0	1	0	0
back	1	0	1	0	0	0	1	0
brown	1	0	1	0	1	0	1	0
come	0	1	0	1	0	1	0	1
dog	0	0	1	0	1	0	0	0
fox	0	0	1	0	1	0	1	0
good	0	1	0	1	0	1	0	1
jump	0	0	1	0	0	0	0	0
lazy	1	0	1	0	1	0	1	0
men	0	1	0	1	0	0	0	1
now	0	1	0	0	0	1	0	1
over	1	0	1	0	1	0	1	1
party	0	0	0	0	0	1	0	1
	0	1	0	0	0	0	0	0
	0	0	0	1	0	1	0	0



Term	Postings
aid	→ 4 → 8
all	→ 2 → 4 → 6
back	→ 1 → 3 → 7
brown	→ 1 → 3 → 5 → 7
come	→ 2 → 4 → 6 → 8
dog	→ 3 → 5
fox	→ 3 → 5 → 7
good	→ 2 → 4 → 6 → 8
jump	→ 3
lazy	→ 1 → 3 → 5 → 7
men	→ 2 → 4 → 8
now	→ 2 → 6 → 8
over	→ 1 → 3 → 5 → 7 → 8
party	→ 6 → 8
quick	→ 1 → 3
their	→ 1 → 5 → 7

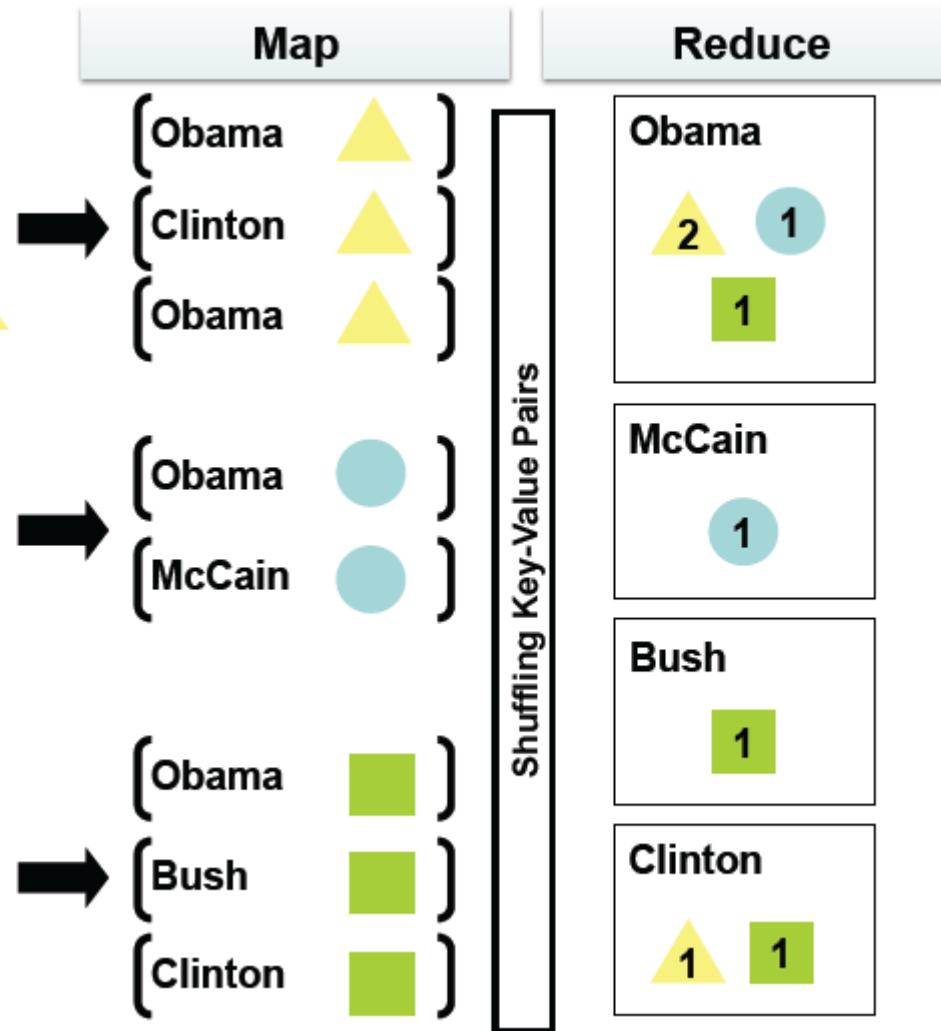
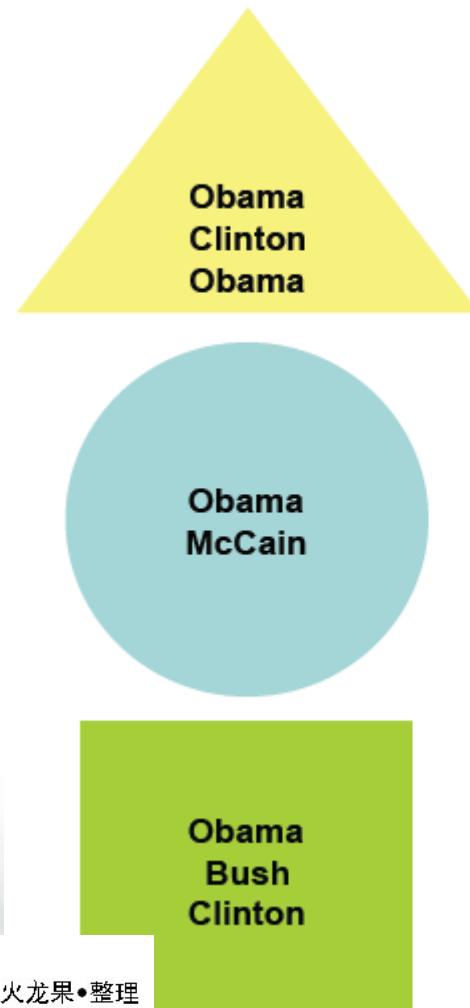
Case2: inverted index

- 算法输入：若干文本文件
- 算法输出：倒排索引结果
- 如何实现？



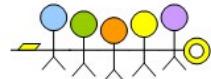
Case2: inverted index

- 实现思想



Case2: inverted index

- import java.io.IOException;
- import java.util.StringTokenizer;
- import java.io.DataInput;
- import java.io.DataOutput;
- import org.apache.hadoop.conf.Configuration;
- import org.apache.hadoop.fs.Path;
- import org.apache.hadoop.io.WritableComparable;
- import org.apache.hadoop.io.IntWritable;
- import org.apache.hadoop.io.LongWritable;
- import org.apache.hadoop.io.Text;
- import org.apache.hadoop.mapreduce.Job;
- import org.apache.hadoop.mapreduce.Mapper;
- import org.apache.hadoop.mapreduce.Reducer;
- import org.apache.hadoop.mapreduce.Partitioner;
- import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
- import org.apache.hadoop.mapreduce.lib.input.FileSplit;
- import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
- import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
- import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
- import org.apache.hadoop.util.GenericOptionsParser;



Case2: inverted index

```
• public class InvertedIndex {  
•     public static class Elem implements WritableComparable {  
•         private Text word;  
•         private Text docno;  
•         public Elem(String word, String docno) {  
•             this.word = new Text(word);  
•             this.docno = new Text(docno);}  
•         public Elem() {  
•             word = new Text();  
•             docno = new Text();}  
•         public void readFields(DataInput in) throws IOException {  
•             word.readFields(in);  
•             docno.readFields(in);}  
•         public void write(DataOutput out) throws IOException {  
•             word.write(out);  
•             docno.write(out);}  
•         public String getWord() {  
•             return word.toString();}  
•         public String getDocno() {  
•             return docno.toString();}  
•         public int compareTo(Object o) {  
•             Elem e = (Elem)o;  
•             if (!this.getWord().equals(e.getWord()))  
•                 return this.getWord().compareTo(e.getWord());  
•             if (!this.getDocno().equals(e.getDocno()))  
•                 return this.getDocno().compareTo(e.getDocno());  
•             return 0;}
```



Case2: inverted index

```
public static class InvertedIndexMapper extends Mapper<LongWritable, Text,  
Elem, IntWritable> {  
  
    private IntWritable one = new IntWritable(1);  
    /*  
     * input: (line-offset, line)  
     * output: (word docno, 1)  
     */  
    public void map(LongWritable lineOffset, Text line, Context context) throws  
IOException, InterruptedException {  
        FileSplit split = (FileSplit)(context.getInputSplit());  
        String fileName = split.getPath().getName();  
        StringTokenizer itr = new StringTokenizer(line.toString());  
        while (itr.hasMoreTokens()) {  
            String token = itr.nextToken().toLowerCase();  
            context.write(new Elem(token, fileName), one);  
        }  
    }  
}
```



Case2: inverted index

```
public static class InvertedIndexCombiner extends Reducer<Elem,  
IntWritable, Elem, IntWritable> {
```

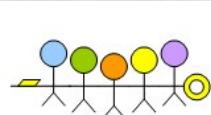
```
    IntWritable sum = new IntWritable();
```

```
    public void reduce(Elem e, Iterable<IntWritable> values,  
Context context) throws IOException, InterruptedException {
```

```
        int count = 0;  
        for(IntWritable val : values)  
            count += val.get();  
        sum.set(count);  
        context.write(e, sum);
```

```
}
```

```
}
```



Case2: inverted index

```
public static class InvertedIndexPartitioner extends  
Partitioner<Elem, IntWritable> {  
  
    /*  
     * partitioned by word  
     */  
    public int getPartition(Elem key, IntWritable value,  
int numPartition) {  
        return (key.getWord().hashCode() &  
0x7fffffff) % numPartition;  
    }  
}
```



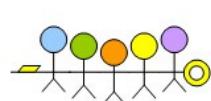
Case2: inverted index

```
public static class InvertedIndexReducer extends Reducer<Elem, IntWritable, Text, Text> {  
  
    private String word = null;  
    private int num = 0;  
    private String result = "";  
  
    public void cleanup(Context context) throws IOException, InterruptedException {  
        if (word != null)  
            output(context);  
    }  
    private void output(Context context) throws IOException, InterruptedException {  
        context.write(new Text(word), new Text(String.format(": %d :%s", num, result)));  
        result = "";  
        num = 0;  
    }  
    public void reduce(Elem e, Iterable<IntWritable> values, Context context) throws IOException,  
    InterruptedException {  
        int count = 0;  
        for (IntWritable iw : values)  
            count += iw.get();  
        if (word != null && !e.getWord().equals(word))  
            output(context);  
        word = e.getWord();  
        result += String.format(" (%os, %d)", e.getDocno(), count);  
        num++;  
    }  
}
```



Case2: inverted index

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    args = new GenericOptionsParser(conf, args).getRemainingArgs();  
    if (args.length != 2) {  
        System.err.println("Usage: InvertedIndex <input> <output>");  
        System.exit(2);  
    }  
  
    String inputDirName = args[0];  
    String outputDirName = args[1];  
  
    Job job = new Job(conf, "Inverted Index");  
    job.setJarByClass(InvertedIndex.class);  
    job.setMapperClass(InvertedIndexMapper.class);  
    job.setCombinerClass(InvertedIndexCombiner.class);  
    job.setPartitionerClass(InvertedIndexPartitioner.class);  
    job.setReducerClass(InvertedIndexReducer.class);  
    job.setMapOutputKeyClass(Elem.class);  
    job.setMapOutputValueClass(IntWritable.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(Text.class);  
    job.setInputFormatClass(TextInputFormat.class);  
    job.setOutputFormatClass(TextOutputFormat.class);  
    FileInputFormat.addInputPath(job, new Path(inputDirName));  
    FileOutputFormat.setOutputPath(job, new Path(outputDirName));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```



Case3: similarity comparison

- 如何计算两个文本的相似度?
 - 对比字符串?
 - 例子:
 - 朝鲜要发射卫星，美国和日本表示不淡定
 - 美国跟日本对朝鲜发射卫星表示不淡定
 - 相似or不相似?



Case3: similarity comparison

- 相似度比较
 - 分词
 - 词的重复程度
 - 当重复程度较高时，则认为相似



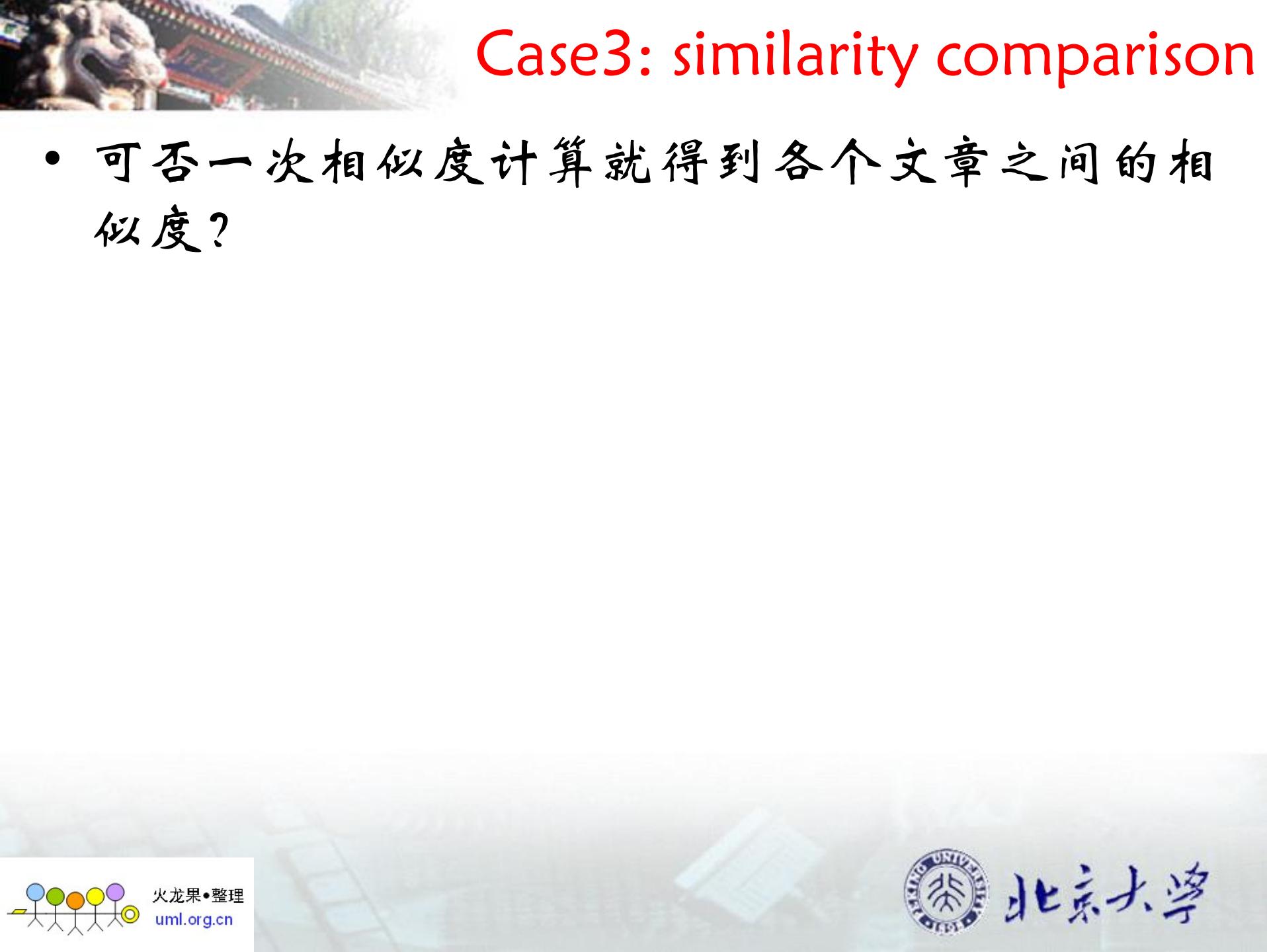
Case3: similarity comparison

- 如何求解多个文本的相似度?
 - 文本很多 (>2)
 - 通过尽可能少的计算次数来得出各个文本之间的相似程度
 - 如何计算?
 - 传统方式:
 - a和bcdefg比较
 - b和cdefg比较
 - c和defg比较
 - ...



Case3: similarity comparison

- 可否一次相似度计算就得到各个文章之间的相似度？



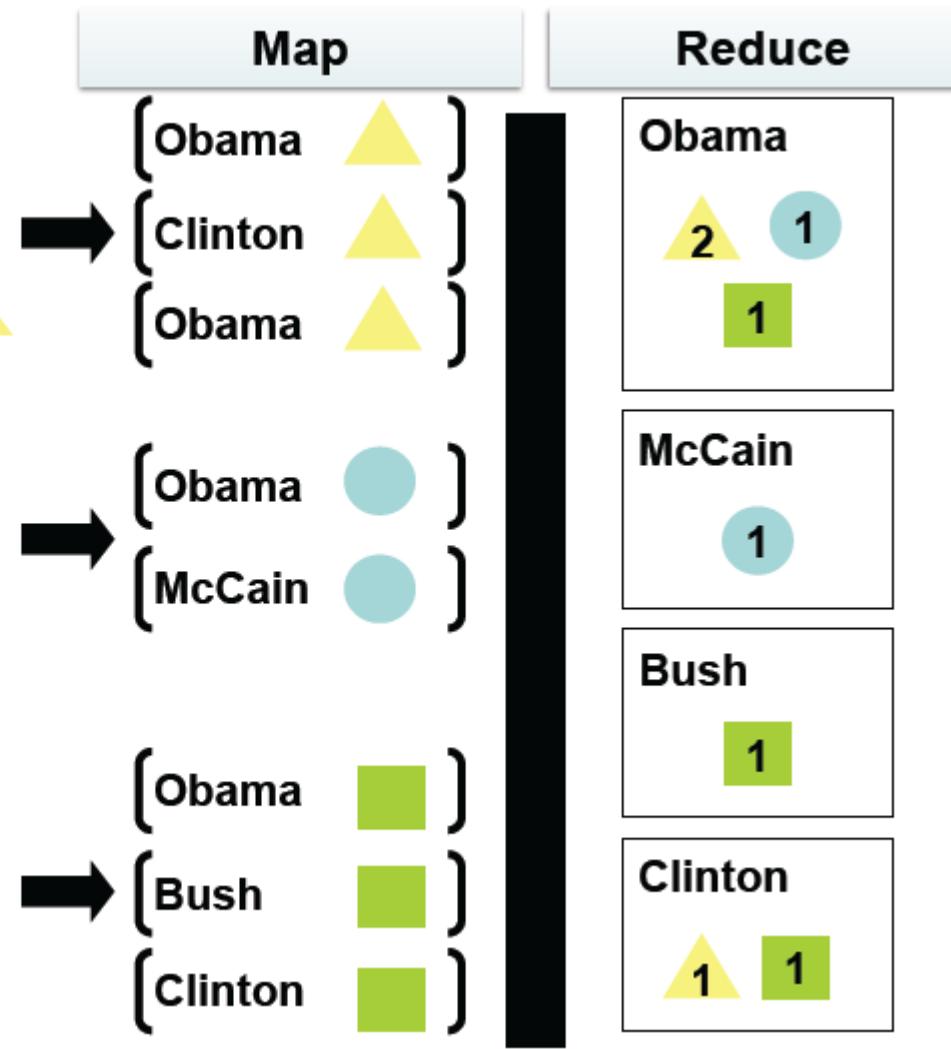
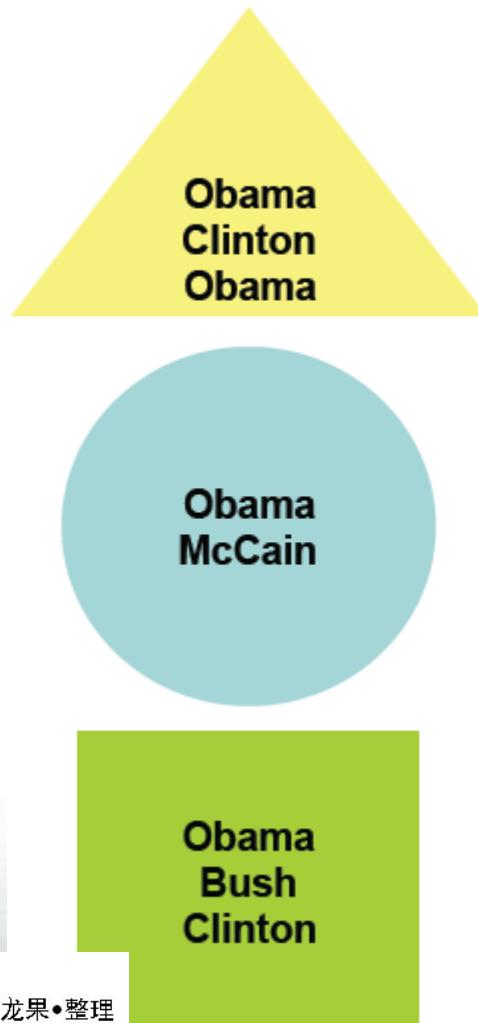
Case3: similarity comparison

- 相似度计算
 - 进行倒排索引
 - 计算文章对相似性（两两相似）
 - 统计文章相似度



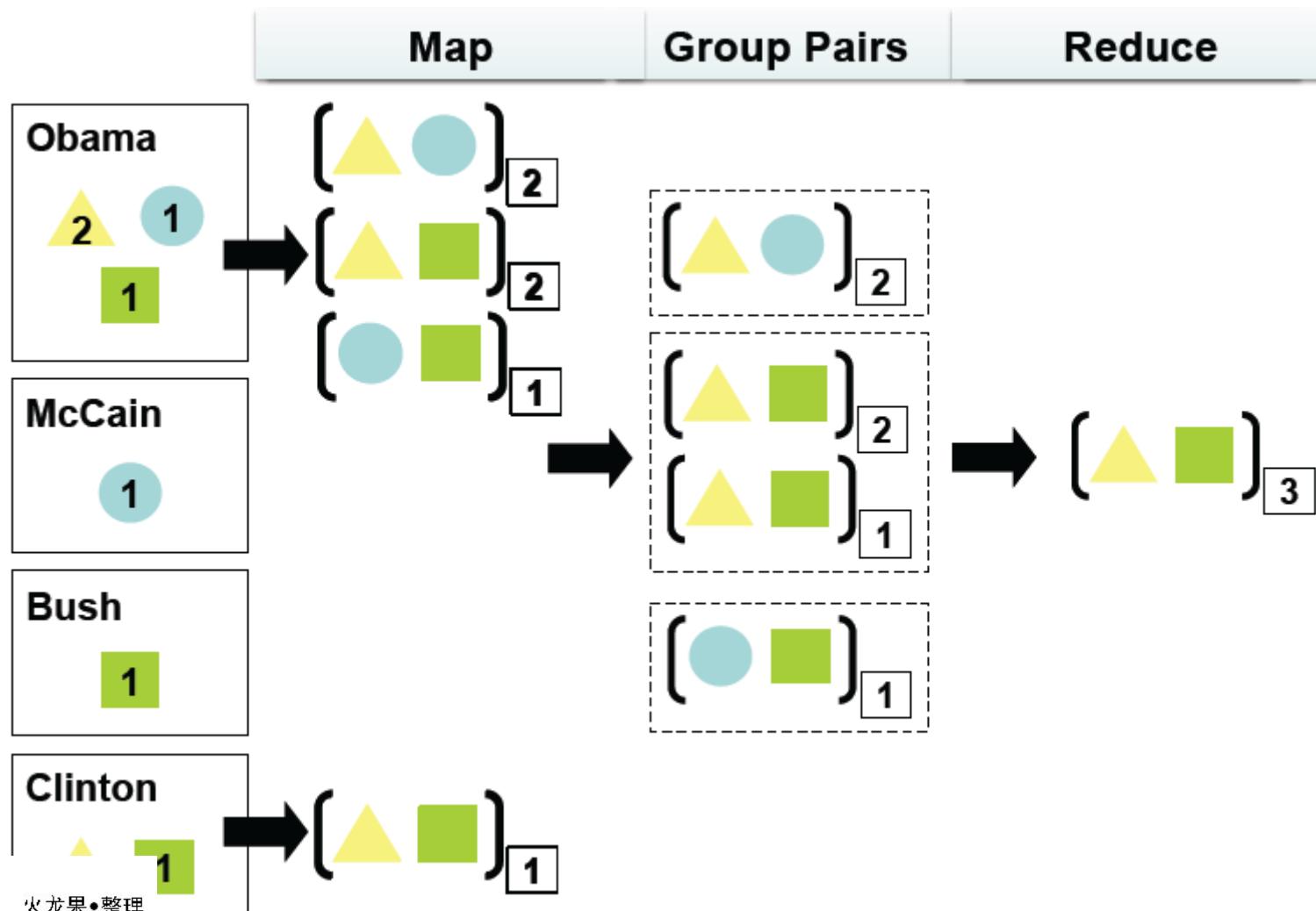
Case3: similarity comparison

- Step1:



Case3: similarity comparison

- Step2:

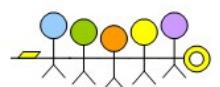


Case3: similarity comparison

- Step3:
 - 各种统计算法
 - 阈值



- 1 • 应用案例
- 2 • 算法调优
- 3 • 应用设计
- 4 • 应用作业



Mapreduce算法调优

- 算法级调优

- 更加优化的key-value对设置
- Map算法
- Combiner算法
- Partition算法
- Reduce算法



Mapreduce算法调优

- 更加优化的key-value对设置
 - Key
 - Value
 - 新增加key-value对（计算频率用求和）
- Map算法
 - Key-value对生成



Mapreduce算法调优

- Combiner算法

- 常规使用reduce算法
- 自定义combiner
- 对key的解析
- 对value的解析
- 对value的计算
- 自定义combiner需要扩展自reducer对象



Mapreduce算法调优

- Partition 算法
 - 控制分发策略
 - 原则上按照key分发
 - 如果key是多元key， 需要对key进行解析之后分发
- Reduce 算法
 - 对key的解析
 - 对value的解析
 - 对value的计算



Mapreduce算法调优

- 参数级调优
 - 调优属性
 - Map属性
 - Reduce属性



Mapreduce算法调优

- Mapreduce在hadoop中的执行过程
 - Map
 - 从磁盘上读数据
 - 执行map函数
 - Combine结果
 - 将结果写到本地磁盘上
 - Reduce
 - 从各个map task的磁盘上读相应的数据 (shuffle)
 - 排序sort
 - 执行reduce函数
 - 将结果写回HDFS



Mapreduce算法调优

- Mapreduce在hadoop中的执行过程
 - Map

- 从磁盘上读数据
- 执行map函数
- Combine结果
- 将结果写到本地磁盘上

- Reduce

- 从各个map task的磁盘上读相应的数据 (shuffle)
- 排序sort
- 执行reduce函数
- 将结果写回HDFS

Mapreduce算法调优

- 将结果写到本地磁盘上
 - map阶段产生的中间结果要写到磁盘上
 - 提高系统的可靠性
 - 代价是降低了系统的性能
- 从各个map task的磁盘上读相应的数据(shuffle)
 - 采用HTTP协议从各个map task上远程拷贝结果



Mapreduce算法调优

- Mapreduce确保每个reducer的输入都按键排序，系统执行排序的过程——将map输出作为输入传给reduer——称为shuffle
- 从许多方面看，shuffle是mapreduce的心脏，是奇迹发生的地方（调优的重点）



Mapreduce算法调优

- Map
 - Map任务交给JVM处理
 - JVM内存大小: mapred.child.java.opts



Mapreduce算法调优

- Map

- Map函数开始产生输出时，并不是直接写入磁盘，首先通过缓冲方式写入内存，并进行预排序

- 缓冲区大小：io.sort.mb，默认100MB
 - 缓冲区容量阈值：io.sort.spill.percent，默认0.80
 - 缓冲区达到阈值后写入mapred.local.dir指定路径
 - 当缓冲区被填满，map会阻塞直到写过程完成



Mapreduce算法调优

- Map

- 在写磁盘之前，线程根据最终要传送到的reducer对数据进行partition，每个partition内数据按照key进行键内排序，如果有combiner，则在排序后的输出上运行

- 任务完成前，多溢出写文件需要合并进行分区和排序，一次最多合并流数：io.sort.factor，默认10
 - 如果指定combiner，溢出写次数最小值：min.num.spills.for.combine，默认3，被满足时，combiner在输出文件写到磁盘之前运行



Mapreduce算法调优

- Map

- 写磁盘时如果压缩map输出则效率更高

- 压缩标志: mapred.compress.map.output, 默认false
 - 压缩方式: mapred.map.output.compression.codec, 默认org.apache.hadoop.io.compress.DefaultCodec



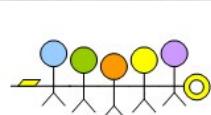
Mapreduce算法调优

- Map
 - Reducer通过HTTP方式从map处得到输出文件的分区，文件分区的工作线程数由tasktracker控制，而不属于哪一个具体的map任务
 - Tasktracker的工作线程数：tracker.http.threads，默认40



Mapreduce算法调优

- Map结束后，其输出作为reduce的输入，由reduce进一步执行
 - Map结束后通知tasktracker更新状态
 - Tasktracker通过心跳通知jobtracker
 - 对于指定作业，jobtracker知道map输出与tasktracker的映射关系
 - Reducer的一个线程定期询问jobtracker以便获得map输出的位置，直到它获得所有输出
 - 由于reducer可能失败，因此tasktracker不会在第一reducer获得map输出后就从本地磁盘删除它们，而是直到jobtracker得到reducer的完成报告后告知



Mapreduce算法调优

- Reduce
 - Reduce任务交给JVM处理
 - JVM内存大小: mapred.child.java.opts



Mapreduce算法调优

- Reduce

- Map输出文件位于运行map任务的tasktracker的本地磁盘，reduce任务需要集群上若干map任务的输出作为输入，每个map任务完成时间可能不同，因此只要有一个map任务完成，reduce任务就开始复制其输出

- reduce任务复制线程：mapred.reduce.parallel.copies，默认5
 - Reduce获取一个map输出最大时间：mapred.reduce.copy.backoff，默认300s，在声明失败之前，如果超时，可以尝试重传



Mapreduce算法调优

- Reduce

- 如果map输出很小，则复制到reduce的tasktracker 的内存中，否则则先写入内存缓冲区，当超过缓冲区溢出阈值，或达到map输出阈值时，再合并后溢出写到磁盘中

- Map输出内存缓冲区占堆空间的百分比：
mapred.job.shuffle.input.buffer.percent， 默认0.70
 - 缓冲区溢出阈值： mapred.job.shuffle.merge.percent，
默认0.66
 - Map输出阈值： mapred.inmem.merge.threshold， 默认
1000



Mapreduce算法调优

- Reduce

- 随着磁盘上的副本的增多，后台线程会根据合并因子将它们合并成更大的排好序的文件，压缩的map输出都在内存中解压缩
 - 合并因子：io.sort.factor， 默认10
- 合并策略目标是合并最小数量的文件一般满足最后一趟的合并系数，比如有40个文件，并不会合并4次，每次10个得到4个文件然后再合并。而是第一次只合并4个，然后后三次合并10个，最后一次，4个已合并文件和剩余6个再合并

Mapreduce算法调优

- Reduce

- 从map处获取数据之后，则开始reduce过程，reduce开始时，内存中map输出大小不能超过输入内存阈值，以便为reduce提供尽可能多的内存，如果reduce需要内存较少，可以增加此值来减少访问磁盘次数

- 输入内存阈值：mapred.job.reduce.input.buffer.percent，默认0.0



Mapreduce算法调优

- Reduce
 - Reduce的输出结果直接写入HDFS系统
 - Hadoop文件缓冲区：io.file.buffer.size， 默认4KB



Mapreduce算法调优

- 调优原则

- 给shuffle过程尽可能多的内存空间
- Map和reduce函数尽量少用内存
- 运行map和reduce任务的JVM的内存尽量大
- Map端尽量估算map输出的大小，减少溢出写磁盘的次数
- Reduce端的中间数据尽可能的多驻留在内存
- 增加Hadoop的文件缓冲区



- 1 • 应用案例
- 2 • 算法调优
- 3 • 应用设计
- 4 • 应用作业

Mapreduce算法设计

- 数学运算
 - 加法
 - 减法——加负数
 - 乘法——同一数的多次加法
 - 除法——同一数的多次减法
 - 平方、开方、幂运算…
- 同理，mapreduce算法也可看做是一种并行化解决问题的思想，可以用于许多问题的求解

Mapreduce算法设计

- Mapreduce作用-如何通过并行计算提高效率？
- 例子1：sum
 - 分段求和
 - 结果再求和
- 例子2：wordcount
 - 分别统计各文章中每个词的次数
 - 按照词对结果进行汇总



Mapreduce算法设计

- Mapreduce 算法
 - 设置key-value对
 - Map——生成key-value对
 - Combiner——对map的结果进行初步合并
 - Partition——对map的结果进行汇总排序分发
 - Reduce——合并key-value得到结果



Mapreduce算法设计

- 如何对这样一些数据进行求和?
 - wordcount'
 - $(a, b) \rightarrow 1; (a, c) \rightarrow 2; (a, d) \rightarrow 5; (a, e) \rightarrow 3;$
 $(a, f) \rightarrow 2; (a, b) \rightarrow 1; (a, d) \rightarrow 2; (b, c) \rightarrow 4;$
 $(b, d) \rightarrow 2; (b, e) \rightarrow 6; (b, f) \rightarrow 3; (b, c) \rightarrow 9;$
 $(a, f) \rightarrow 2; (a, d) \rightarrow 7; (a, d) \rightarrow 2; (b, f) \rightarrow 4;$
 $(b, d) \rightarrow 8; (c, f) \rightarrow 7; (a, f) \rightarrow 6; (a, b) \rightarrow 8\dots$
 - 如何设置key-value?



Mapreduce算法设计

- 改变一下数据结构

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5$

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

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

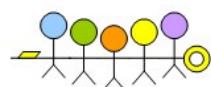
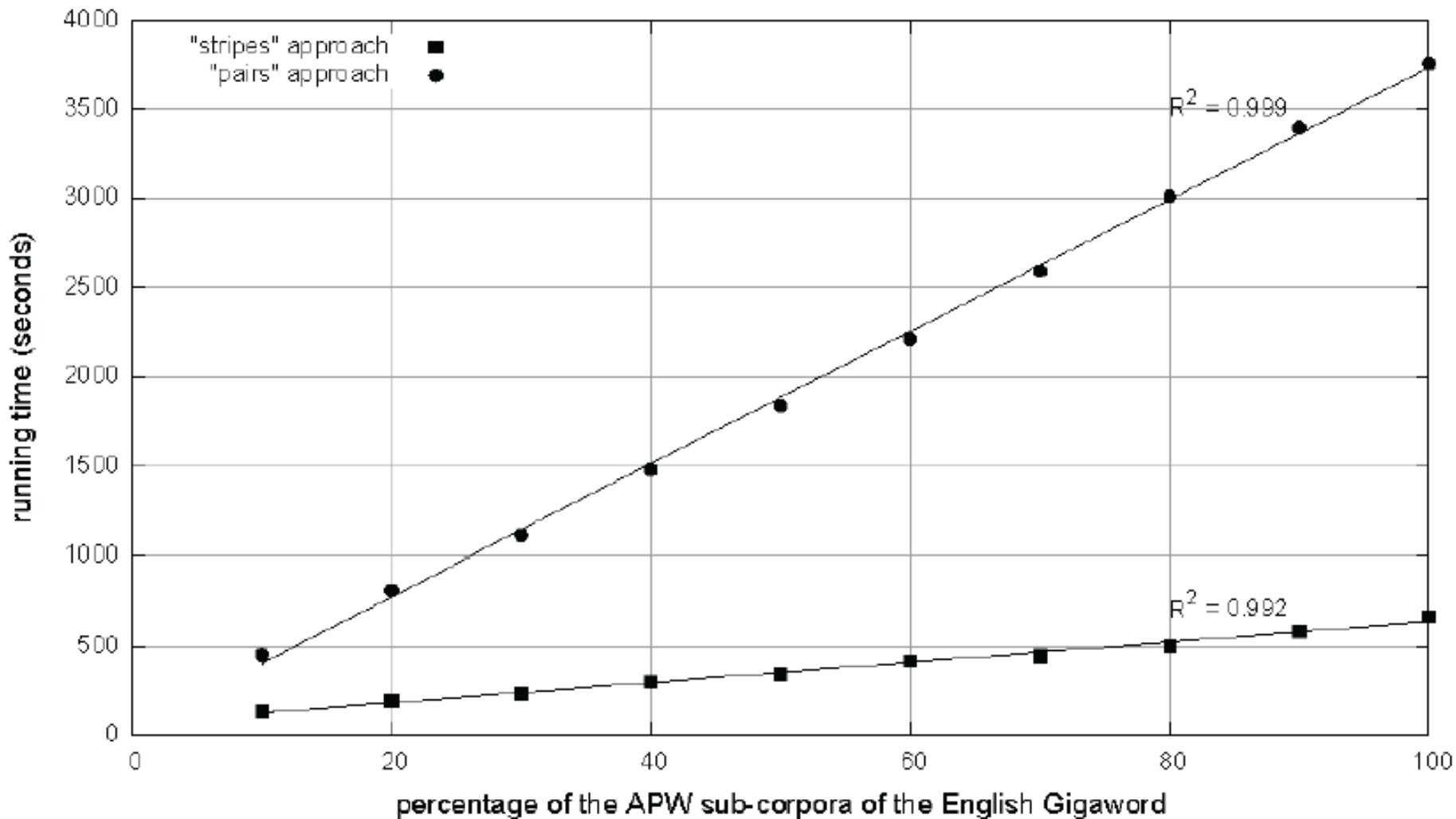
- 改变一下求和方式

$$\begin{array}{r} a \rightarrow \{ b: 1, \quad d: 5, e: 3 \} \\ + \quad a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \} \\ \hline a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \} \end{array}$$



Mapreduce 算法设计

Efficiency comparison of approaches to computing word co-occurrence matrices



Mapreduce算法设计

- 核心思想
 - 并行化
 - Mapper
 - Key-value设置
 - Reducer
- 进阶
 - Combiner
 - Partition
 - Sort



Mapreduce算法设计

- 小技巧

- reducer内部可以做synchronization
- 通过key的设置，控制谁先被做，或者谁先被reduce
- partition可以将相同的key分配到一起，以及按照一定的顺序进行排序，交给各个reduce
- 把各种相关信息嵌入在key中，或者value中



- 1 • 应用案例
- 2 • 算法调优
- 3 • 应用设计
- 4 • 应用作业

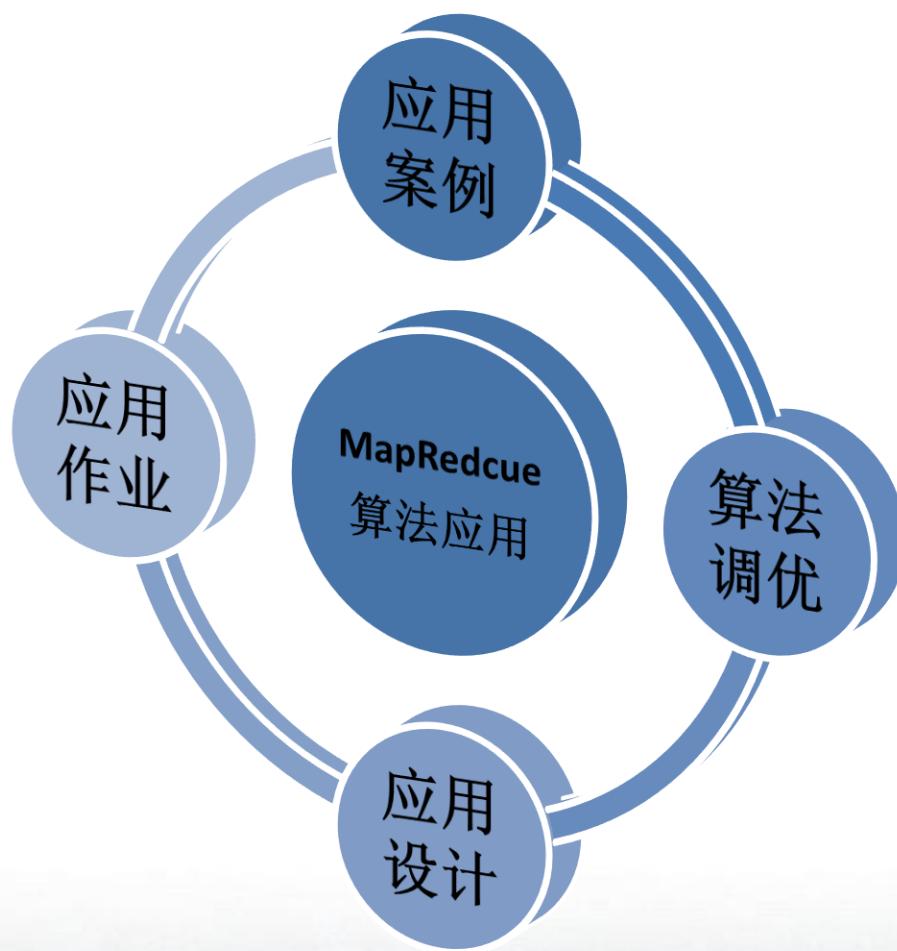
应用作业

- 提交物——报告

- 业务背景
- 业务数据格式
- 计算目的
- MapReduce算法（伪代码）
- 计算结果及分析**
- 提交时间：第11讲，小组形式宣讲报告



小结



- 应用案例
 - 词频
 - 倒排索引
 - 相似比较
- 算法调优
 - Map
 - Reduce
- 算法设计
 - Key-value设置
 - Map/reduce设置
- 应用作业

- 思考题：

- Mapreduce算法设计思想
- 运用mapreduce算法解决实际问题
- 算法调优
- Mapreduce运行过程中的各种参数及其作用
- 参数调优
- 案例的mapreduce算法