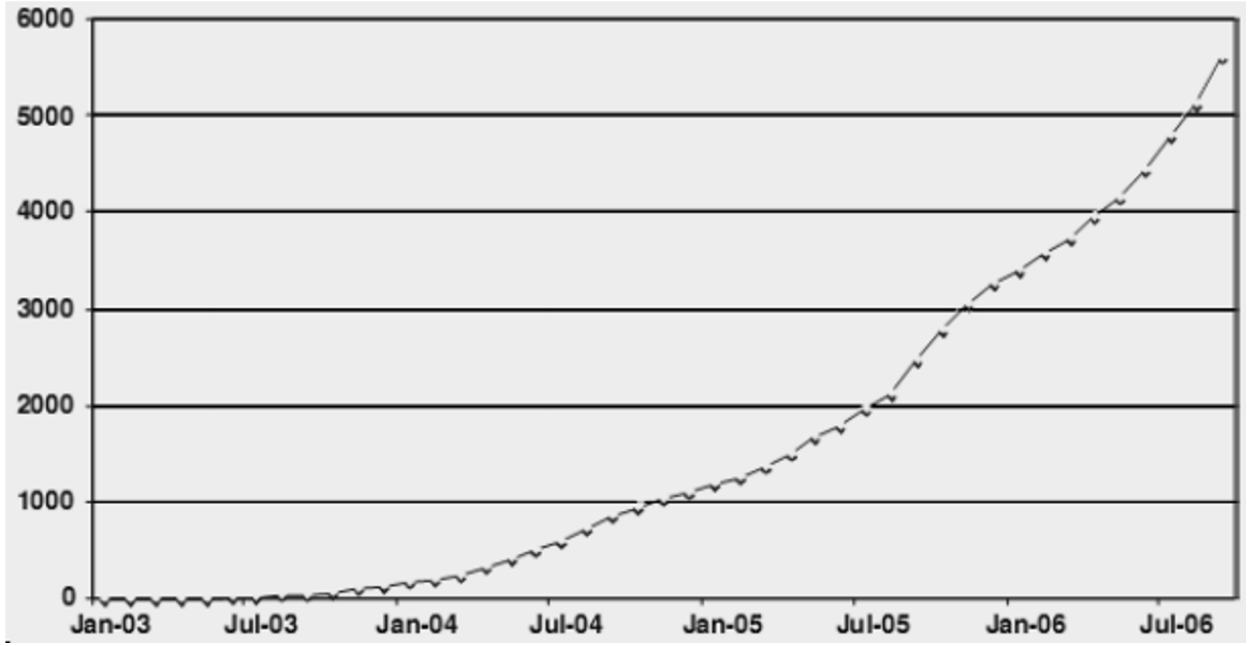
MapReduce基础算法程序设计 (I)

摘要

- MapReduce可解决哪些算法问题?
- 回顾：MapReduce流水线
- MapReduce WordCount1.0
- MapReduce WordCount2.0
- MapReduce排序算法
- MapReduce二级排序

应用范围

- 自MapReduce发明后，Google大量用于各种海量数据处理。目前Google内部有7千以上的程序基于MapReduce实现。MapReduce可广泛应用于搜索引擎（文档倒排索引，网页链接图分析与页面排序等）、Web日志分析、文档分析处理、机器学习、机器翻译等各种大规模数据并行计算应用领域各类大规模数据并行处理算法。



基本算法

各种全局数据相关性小、能适当划分数据的计算任务，如：

- 分布式排序
- 分布式GREP(文本匹配查找)
- 关系代数操作

  如：选择，投影，求交集、并集，连接，成组，聚合…
- 矩阵向量相乘、矩阵相乘
- 词频统计(word count)，词频重要性分析(TF-IDF)
- 单词同现关系分析

  典型的应用如从生物医学文献中自动挖掘基因交互作用关系
- 文档倒排索引
- ……

复杂算法及应用

- Web搜索引擎
  - 网页爬取、倒排索引、网页排序、搜索算法
- Web访问日志分析
  - 分析和挖掘用户在Web上的访问、购物行为特征、以定制个性化用户界面或投放用户感兴趣的产品广告
- 数据/文本统计分析
  - 如科技文献引用关系分析和统计、专利文献引用分析和统计
- 图算法
  - 并行化宽度优先搜索(最短路径问题，可克服Dijkstra串行算法的不足)，最小生成树，子树搜索、比对
  - Web链接图分析算法PageRank，垃圾邮件连接分析
- 聚类(clustering)
  - 文档聚类、图聚类、其它数据集聚类

课程项目设计

- 梁亚澜,李杰,钮鑫涛:Hadoop平台下覆盖表生成遗传算法参数配置启发式演化工具
- 李袁奎,刘文杰,王姜: 使用Mapreduce框架进行软件代码分析
- 黄刚, 陈光鹏: 一种基于MapReduce的频繁闭项集挖掘算法研究及其实现
- 王苏琦,金龑,罗爱宝,王灵江: 基于模型的协同过滤并行化算法
- 胡昊然，冯子陵，窦文科，刘晶晶： 面向新浪微博的关注推荐系统
- 段轶： Netflix电影数据聚类分析
- 孙道平： 基于MapReduce 的数据关联分析
- 刘敏, 刘振兴, 鲁林: NBA球员数据分析工具
- 刘正,朱小虎,王俊,金杰: 基于MapReduce对社会网络分析算法并行化的研究

软件工程

机器学习
数据挖掘

社会网络
分析

课程项目设计

- 王尧,苏宗轩,张林,陈运海小组：利用MapReduce对小百合人际关系的分析实验
- 金惠益,刘友强,刘长辉：基于短语的统计机器翻译系统的短语抽取模型和调序模型的分布式设计
- 张旭，何良朋：P2P流媒体中的结点分簇与最短路径构造
- 陈虎，笪庆小组：基于内容的图像搜索引擎EagleEye
- 张航,杨琬琪,陶承恺：基于MapReduce的本体匹配技术
- 江凯,顾小东,陆瑶,王团团小组：基于Hadoop的SQL查询工具

机器翻译

网络通信

多媒体检索

Web本体

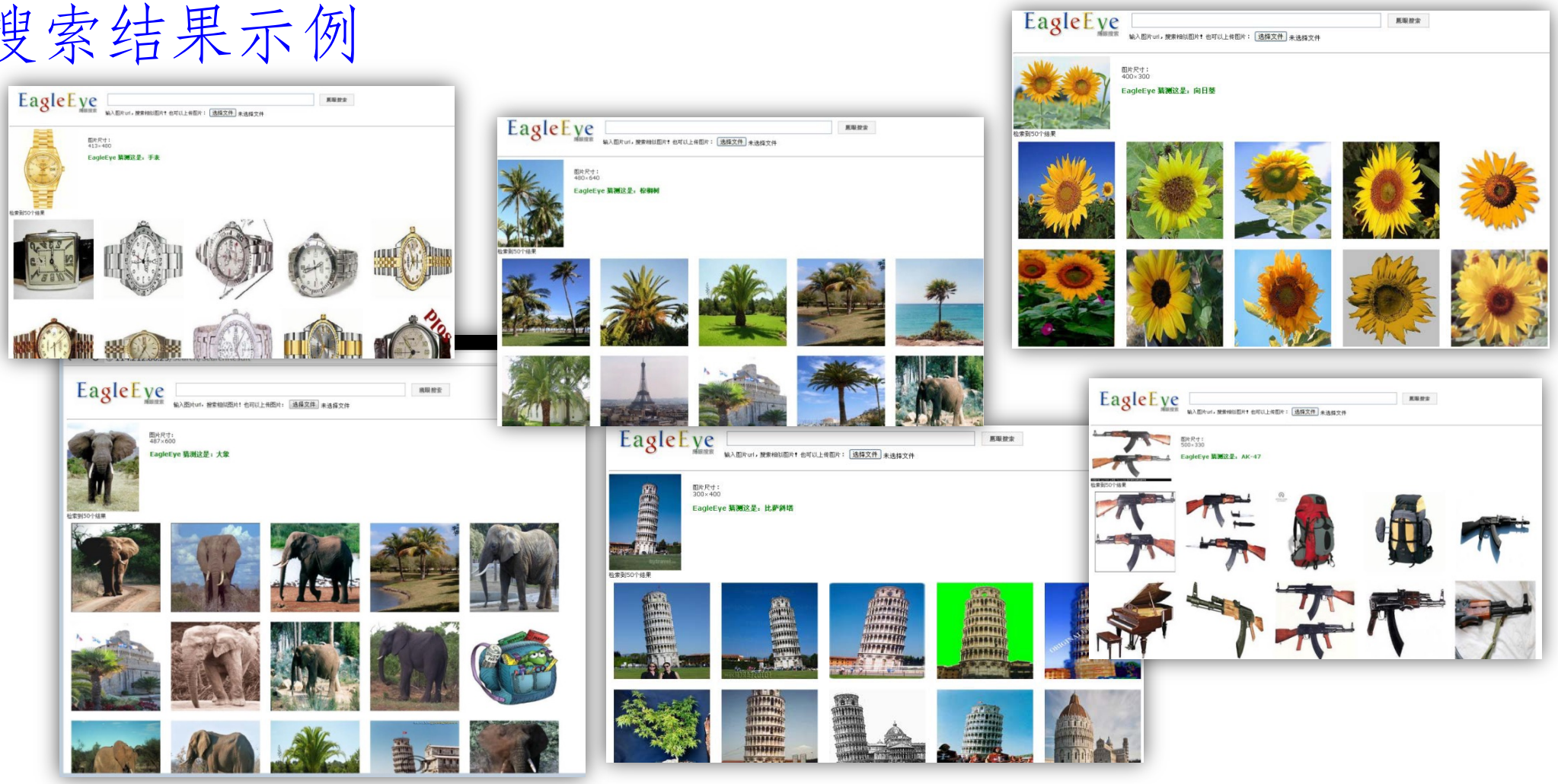
数据库

课程项目设计

- 陈虎，笪庆小组：基于内容的图像搜索引擎**EagleEye**
- 主要研究内容：
  - 1、研究解决了有效的图像特征表示和快速提取方法：表示和提取图像的特征使其在基于内容的图像检索中能够更准确地表征不同图像之间的相似程度。
  - 2、研究解决了基于MapReduce的海量图像特征索引和图像搜索算法
  - 3、完成了一个基于内容的图像搜索EagleEye原型系统的设计实现

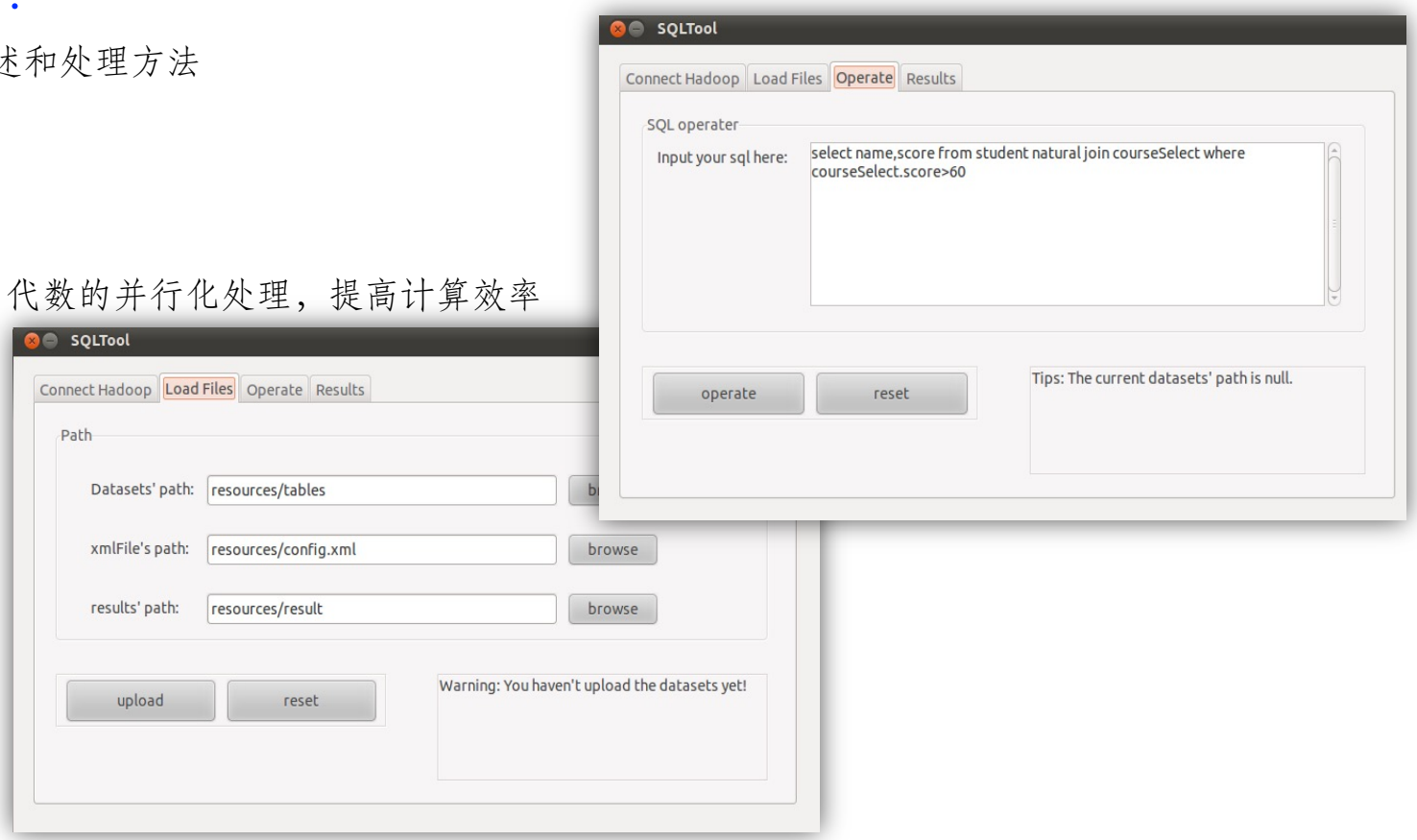
课程项目设计

- 搜索结果示例



课程项目设计

- 江凯,顾小东,陆瑶,王团团小组：基于Hadoop的SQL查询工具
- 主要研究了在Hadoop分布式文件系统环境下设计和模拟一个管理和查询结构化数据的原型数据库系统，主要技术内容包括：
  - 设计了基于XML的数据库Schema的描述和处理方法
  - 设计了基本的SQL查询语言
  - 完成SQL语句的解析处理
  - 完成SQL到关系代数的转换处理
  - 基于MapReduce并行计算框架完成关系代数的并行化处理，提高计算效率
  - 设计实现了一个原型的查询工具



课程项目设计

- 梁亚澜,李杰,钮鑫涛: Hadoop平台下覆盖表生成遗传算法参数配置启发式演化工具
- 主要研究内容:
  - 1.采用启发式演化方法对遗传算法的种群规模、进化机制、交叉概率、变异概率及其变种算法5个因素进行取值组合演化，系统地探索各个因素对遗传算法覆盖表生成效果的影响程度和性质，并以覆盖表规模和消耗时间为依据寻找出最佳配置。
  - 2.遗传算法生成覆盖表的计算量极大，设种群规模为100，进化代数为1000，则完整的进化过程需运行遗传算法100*1000＝100,000次，以一次生成覆盖表的时间为1分钟为例，采用串行计算共需100000分钟，约71天。课题研究实现了基于Hadoop MapReduce的并行化遗传算法生成覆盖表算法,大大缩短了计算时间。

表 3：各待测实例的最终最优配置和覆盖表生成结果

	Algorithm	m	T	Pc	Pm	Size	Time		Algorithm	m	T	Pc	Pm	Size	Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8
6^{10}	GA climb	6100	1100	0.2	0.2	58	402	$7^{6}6^{7}5^{6}$	GAr climb	100	1100	0.8	0.2	82	23.5
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1	$8^{27^{2}6^{2}5^{2}}$	GA- climb	2100	600	0.8	0.6	70	277
8^{10}	GA climb	2100	600	0.6	0.2	98	604	$6^{15^{1}4^{6}3^{8}2^{3}}$	GAr climb	4100	1100	0.8	0.4	36	568.1
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31	6^{4}	GAr climb	100	100	0.6	0.2	41	0.03
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9	$5^{13^{8}2^{2}}$	GAr climb	100	100	0.8	0.2	20	0.43
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4								

复杂算法及应用

- 相似性比较分析算法
  - 字符序列、文档、图、数据集相似性比较分析
- 基于统计的文本处理
  - 最大期望(EM)统计模型，隐马可夫模型(HMM)，……
- 机器学习
  - 监督学习、无监督学习、分类算法(决策树、SVM…)
- 数据挖掘
- 统计机器翻译
- 生物信息处理
  - DNA序列分析比对算法Blast：双序列比对、多序列比对
  - 生物网络功能模块(Motif)查找和比对
- 广告推送与推荐系统
- ……

MapReduce算法应用专著

1.Mining of Massive Datasets

2010, Jure Leskovec (Stanford Univ.), Anand Rajaraman (Kosmix, Inc), Jeffrey D. Ullman (Stanford Univ.)

主要介绍基于MapReduce的大规模数据挖掘相关的技术和算法，尤其是Web或者从Web导出的数据

Ch3. Similarity search, including the key techniques of minhashing and locality-sensitive hashing.

Ch4. Data-stream processing and specialized algorithms for dealing with data that arrives so fast it must be processed immediately or lost.

Ch5. The technology of search engines, including Google's PageRank, link-spam detection, and the hubs-and-authorities approach(a link analysis algorithm: Hyperlink-Induced Topic Search (HITS)).

Ch6. Frequent-itemset mining, including association rules, market-baskets, the A-Priori Algorithm and its improvements (a classic algorithm for learning association rules).

Ch7. Algorithms for clustering very large, high-dimensional datasets.

Ch8. Two key problems for Web applications: managing advertising and recommendation systems.

MapReduce算法应用专著

2. Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer，2010，University of Maryland, College Park

主要介绍基于MapReduce的大规模文档数据处理技术和算法

Ch4. Inverted Indexing for Text Retrieval

Ch5. Graph Algorithms

- Parallel Breadth-First Search
- PageRank

Ch6. EM Algorithms for Text Processing

- EM, HMM
- Case Study: Word Alignment for Statistical Machine Translation

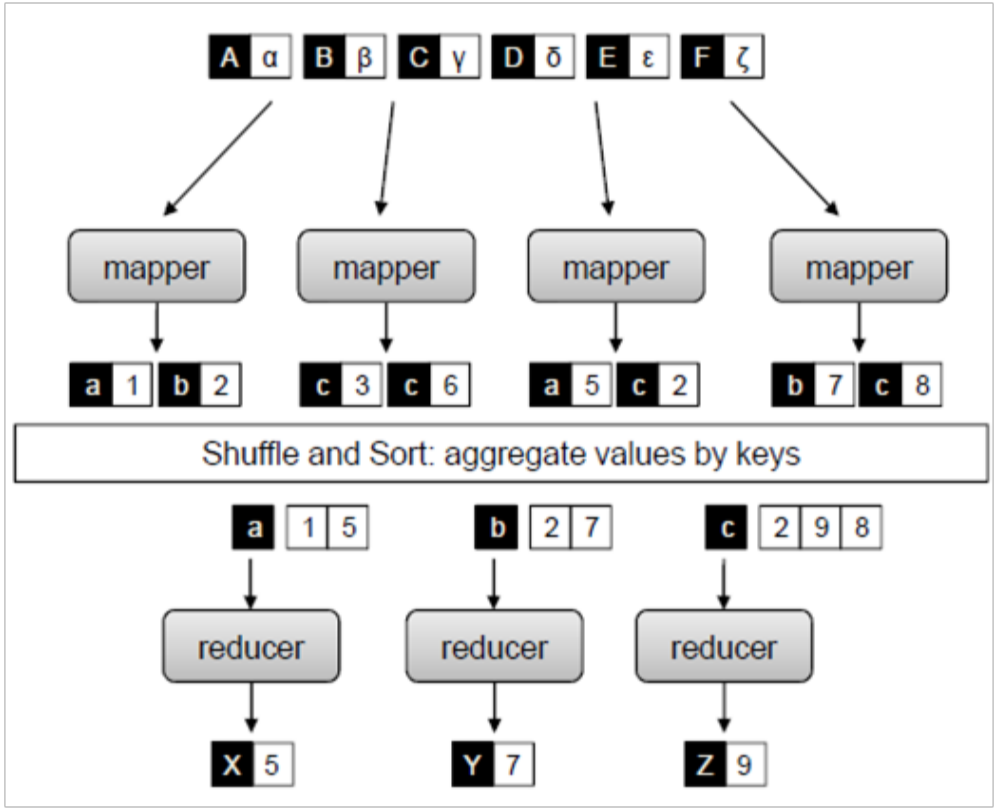
回顾：MapReduce流水线

MapReduce Pipeline

1. map(K1,V1)->[(K2,V2)]
2. shuffle and sort
3. reduce(K2,[V2]) -> [(K3,V3)]

([…] denotes a list)

Any algorithm that you wish to develop must be expressed in terms of such rigidly-defined components



回顾：MapReduce流水线

- Mapper
  - Initialize: setup()
  - map(): It is called once for each key/value pair in the input split. The default is the identity function.
  - Close: cleanup()
- Shuffle
  - Shuffle phase needs the *Partitioner* to route the output of mapper to reducer.
  - *Partitioner* controls the partitioning of the keys of the intermediate map-outputs. The key is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job.
  - *HashPartitioner* is the default Partitioner.

回顾：MapReduce流水线

- Sort
  - We can controls how the keys are sorted before they are passed to the Reducer by using a customized comparator.
- Reducer
  - Initialize: setup()
  - reduce(): It is called once for each key. The default implementation is an identity function.
  - Close: cleanup()

常用数据类型

- 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
Text	Wrapper to store text using the UTF8 format
NullWritable	Placeholder when the key or value is not needed

Hadoop API文档

- **Apache Hadoop Main API**
  - Common; HDFS; MapReduce; YARN
  - https://hadoop.apache.org/docs/stable/api/index.html

建议根据安装和使用的Hadoop版本，查看对应的API文档

MapReduce User Interface

- Mapper
  - map
    - How many mappers?
  - combine
- Reducer
  - shuffle
  - sort/secondary sort
  - Reduce
    - How many reducers?
- Partitioner
- Counter

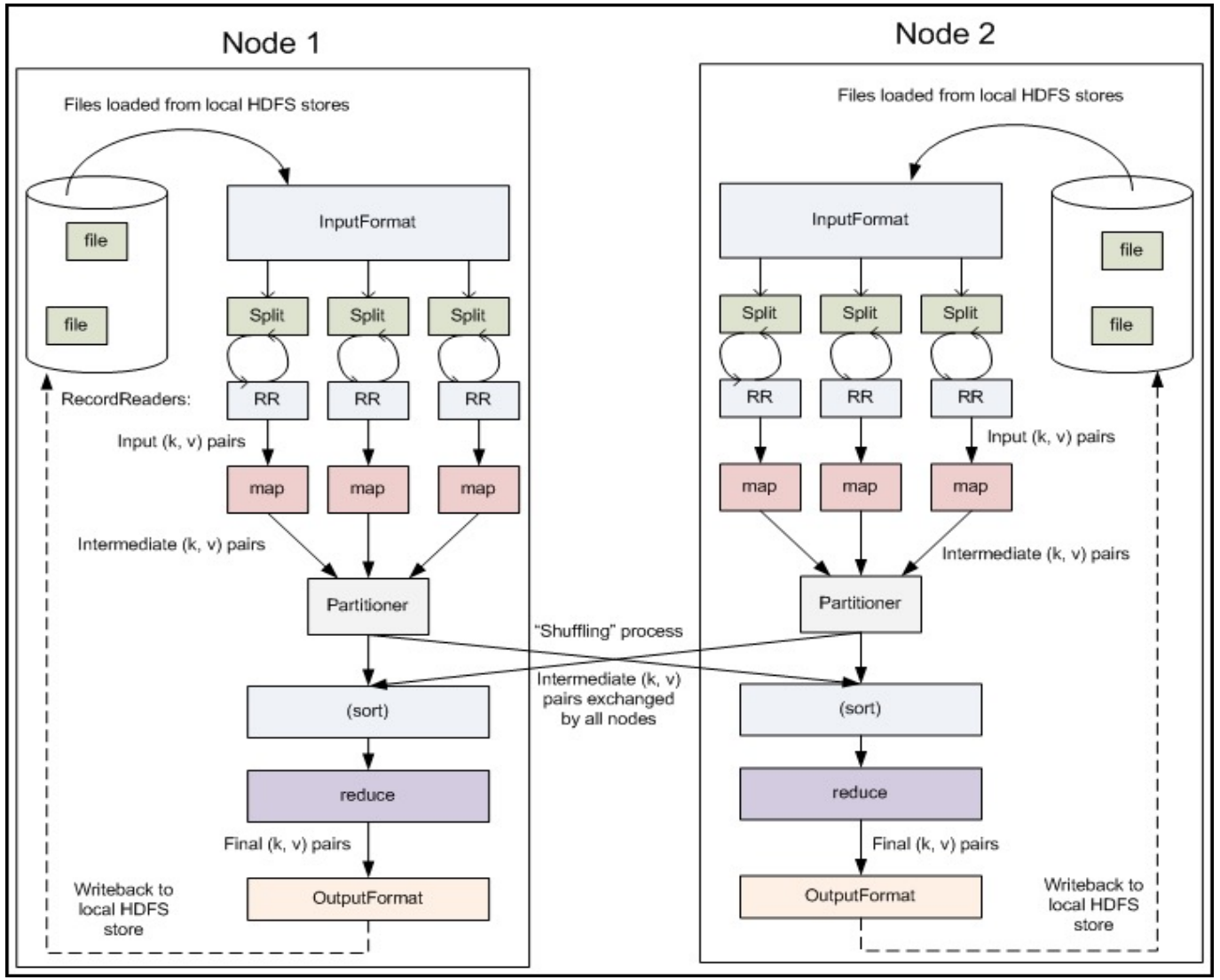
MapReduce User Interface

- Job Configuration
  - Job represents a MapReduce job configuration
- Task Execution & Environment
  - The MRAppMaster executes the Mapper/Reducer task as a child process in a separate JVM.
- Job Submission and Monitoring
  - job.submit() or job.waitForCompletion(boolean)
- Job Input
- Job Output

MapReduce User Interface

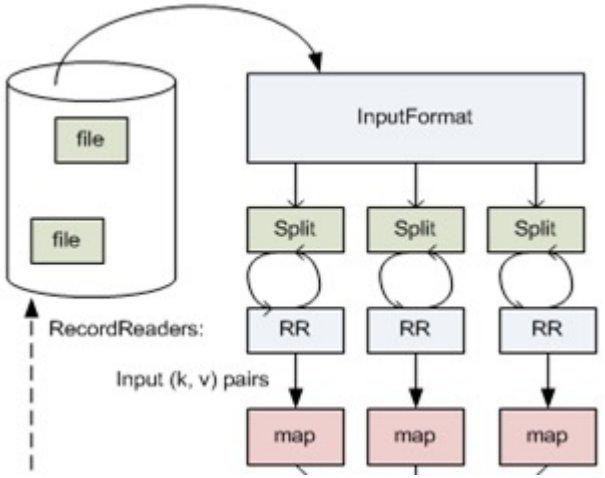
- Other Useful Features
  - Submitting Jobs to Queues
  - Counters: global counters
  - DistributedCache: distribute application-specific, large, read-only files efficiently
  - Profiling
  - Debugging
  - Data Compression
  - Skipping Bad Records
  - ……

MapReduce基本工作过程



主要组件

文件输入格式InputFormat



- 定义了数据文件如何分割和读取
- InputFormat提供了以下一些功能
  - 选择文件或者其它对象，用来作为输入
  - 定义InputSplits，将一个文件分开成为任务
  - 为RecordReader提供一个工厂，用来读取这个文件
- 有一个抽象的类FileInputFormat，所有的输入格式类都从这个类继承这个类的功能以及特性。当启动一个Hadoop任务的时候，一个输入文件所在的目录被输入到FileInputFormat对象中。FileInputFormat从这个目录中读取所有文件。然后FileInputFormat将这些文件分割为一个或者多个InputSplits。
- 通过job.setInputFormat()设置文件输入的格式

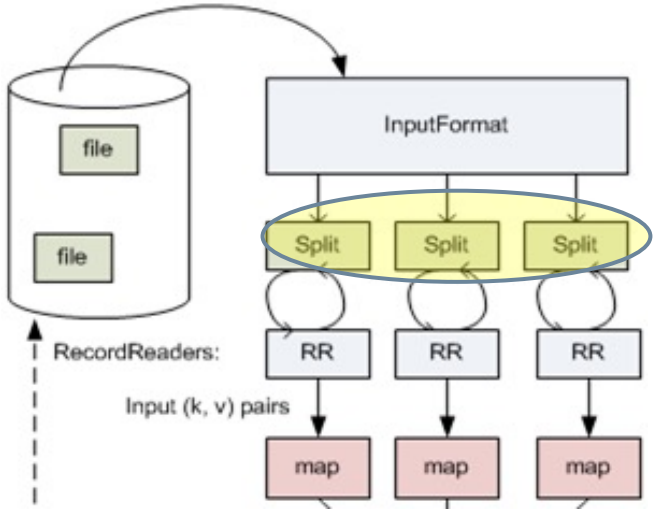
主要组件

- 文件输入格式InputFormat

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
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

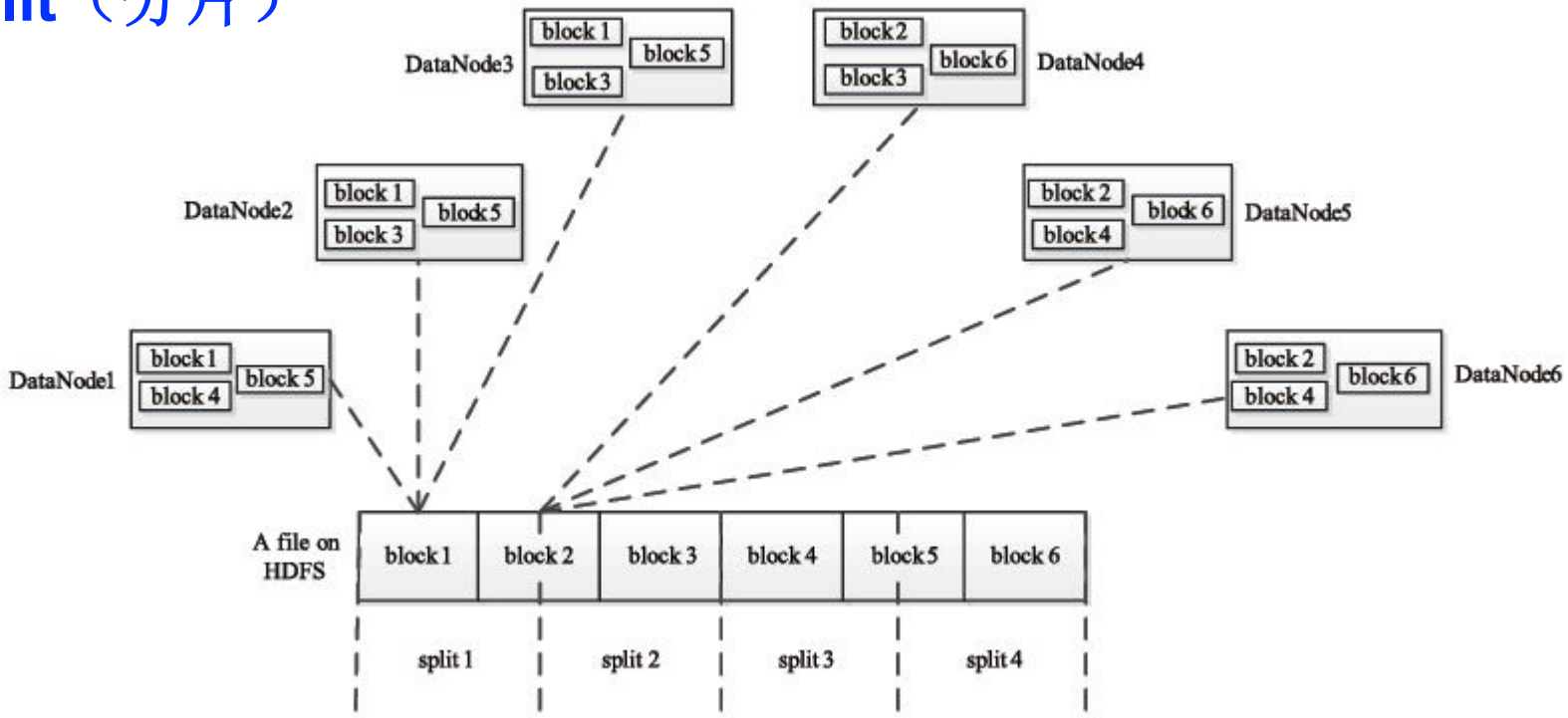
主要组件

- 输入数据分块InputSplits
  - InputSplit定义了输入到单个Map任务的输入数据
  - 一个MapReduce程序被统称为一个Job，可能有上百个任务构成
  - InputSplit将文件分为128MB的大小
    - 配置文件mapred-site.xml中的mapreduce.input.fileinputformat.split.minsize和mapreduce.input.fileinputformat.split.maxsize参数控制这个大小
  - mapreduce.tasktracker.map.tasks.maximum用来控制某一个节点上所有map任务的最大数目



主要组件

关于Split（分片）



HDFS 以固定大小的block 为基本单位存储数据（物理切片），而对于MapReduce 而言，其处理单位是split（逻辑切片）。split是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。

主要组件

Map任务的数量

- Hadoop为每个split创建一个Map任务，split 的多少决定了Map任务的数目。大多数情况下，理想的分片大小是一个HDFS块。
- 由于Mapper是基于虚拟机的，过多的Mapper创建和初始化及关闭虚拟机都会消耗大量的硬件资源；Mapper数太小，并发度过小，Job执行时间过长，无法充分利用分布式硬件资源。
- Mapper的个数通常由输入数据的大小、输入文件的切分策略、可用的集群资源等决定，也可以通过配置mapred-site.xml的参数（如mapreduce.job.maps）来手动设定（一般不需要认为设置）。尽管实际使用时，Hadoop会根据数据大小自动调整。

参数	含义
mapreduce.input.fileinputformat.split.minsize	#启动map最小的split size大小，默认0
mapreduce.input.fileinputformat.split.maxsize	#启动map最大的split size大小，默认256M
dfs.block.size	#block块大小，默认128M

splitSize = Math.max(minSize, Math.min(maxSize, blockSize));

主要组件

Reduce任务的数量

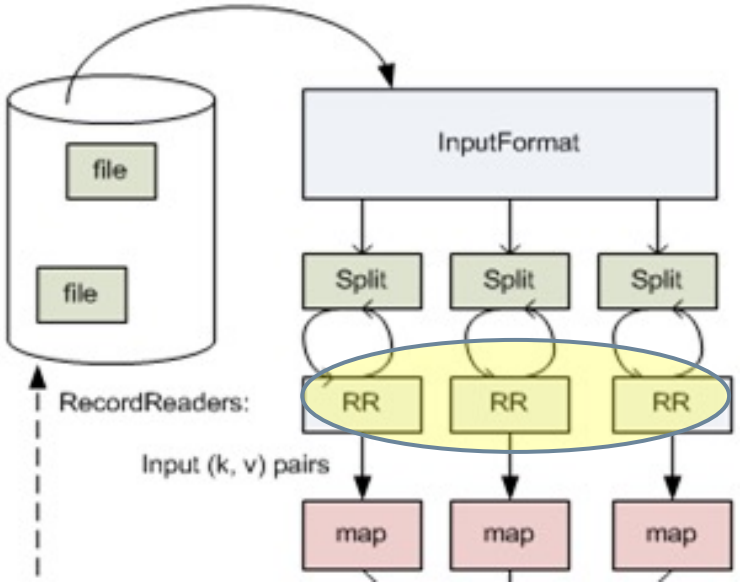
- 可以通过配置mapred-site.xml的mapreduce.job.reduces参数来设置，也可在程序中调用job.setNumReduceTasks(reduceNum)方法来设置reducer的个数。根据任务需求和集群资源，可以灵活调整这一数量。

Reducer的个数主要受以下因素影响

- 输入数据量：据量越大，通常需要更多的reducer来处理，以避免单个reducer成为瓶颈。
- 数据倾斜：如果数据分布不均，可能需要增加reducer数量，以确保负载均衡，避免某个reducer过载。
- 集群资源：集群的CPU、内存和网络带宽等资源限制会影响最优配置。可用资源越多，通常可以配置更多的reducer。
- 作业复杂度：处理逻辑的复杂性（如需要排序、聚合等）也会影响reducer的数量，以确保性能最优。
- 性能测试：通过实际运行和性能监控，可以确定最优的reducer数量，通常需要多次尝试和调优。

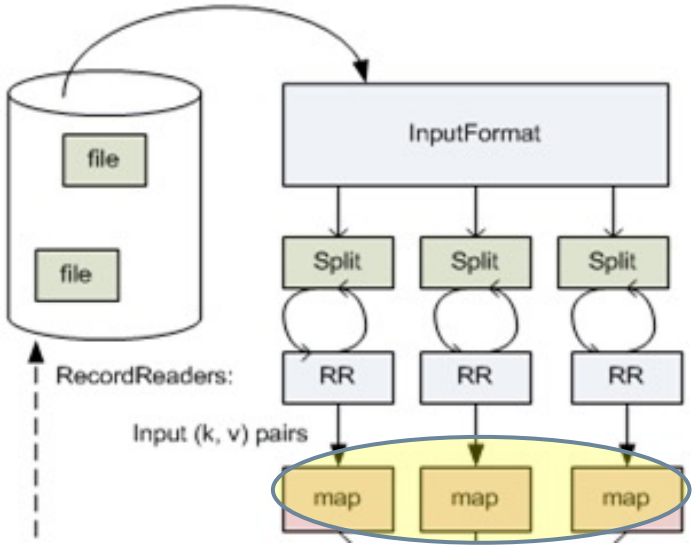
主要组件

- 数据记录读入RecordReader
  - InputSplit定义了一项工作的大小，但是没有定义如何读取数据
  - RecordReader实际上定义了如何从数据上转化为一个(key,value)对的详细方法，并将数据输出到Mapper类中
  - TextInputFormat提供了LineRecordReader



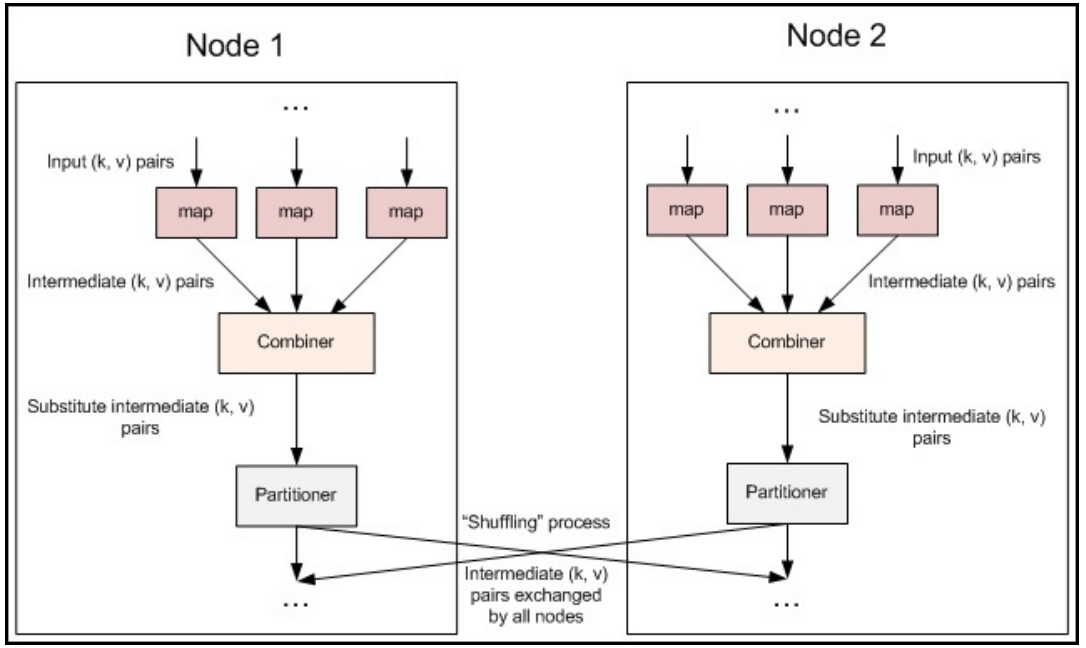
主要组件

- Mapper
  - 每一个Mapper类的实例生成了一个Java进程（在某一个InputSplit上执行）
  - org.apache.hadoop.mapreduce.Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>



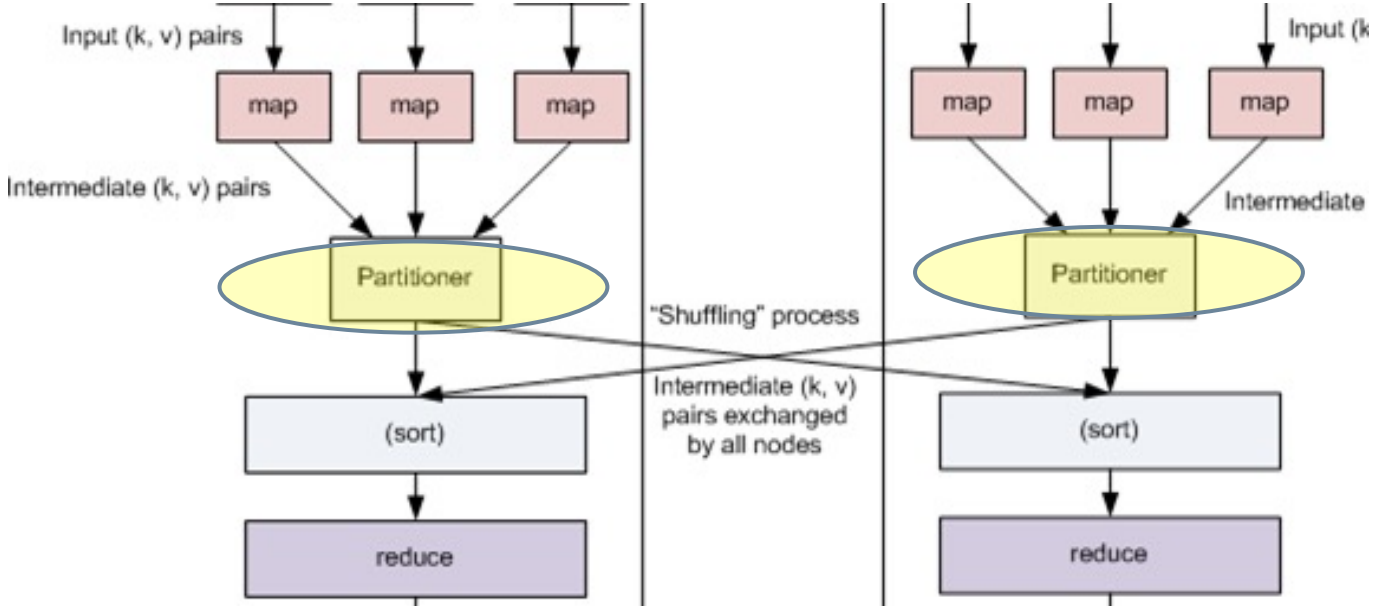
主要组件

- Combiner
  - 合并相同key的键值对，减少partition时候的数据通信开销；
  - 是在本地执行的一个Reducer，满足一定的条件才能够执行。
    - job.setCombinerClass(Reduce.class); //必须显式设置



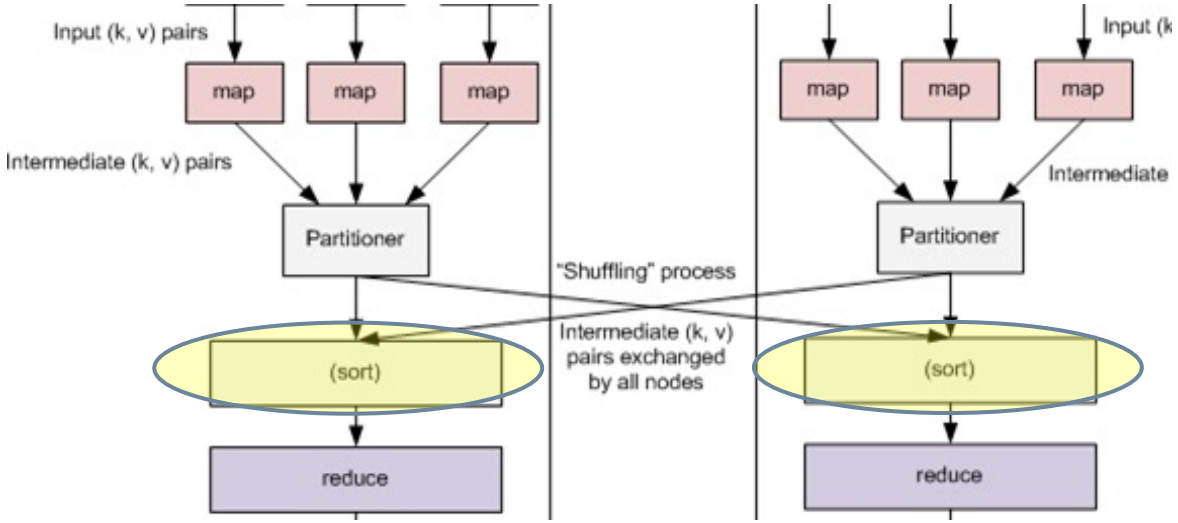
主要组件

- Partitioner & Shuffle
  - 在Map工作完成之后，每一个 Map函数会将结果传到对应的Reducer所在的节点，此时，用户可以提供一个Partitioner类，用来决定一个给定的(key,value)对传输的具体位置。



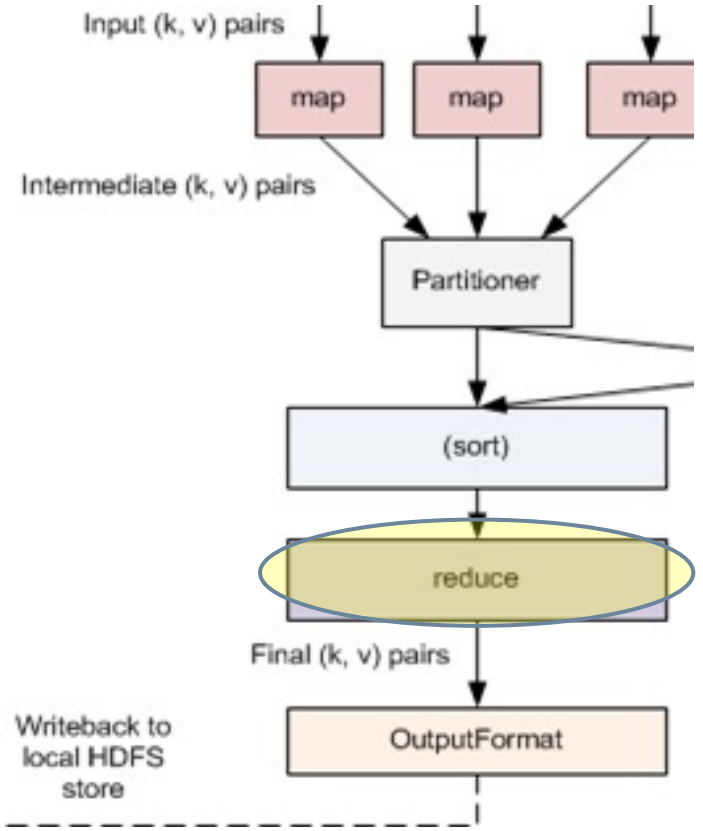
主要组件

- Sort
  - 传输到每一个节点上的所有的Reduce函数接收到的(key,value)都会被Hadoop自动排序（即Map生成的结果传送到某一个节点的时候，会被自动排序）



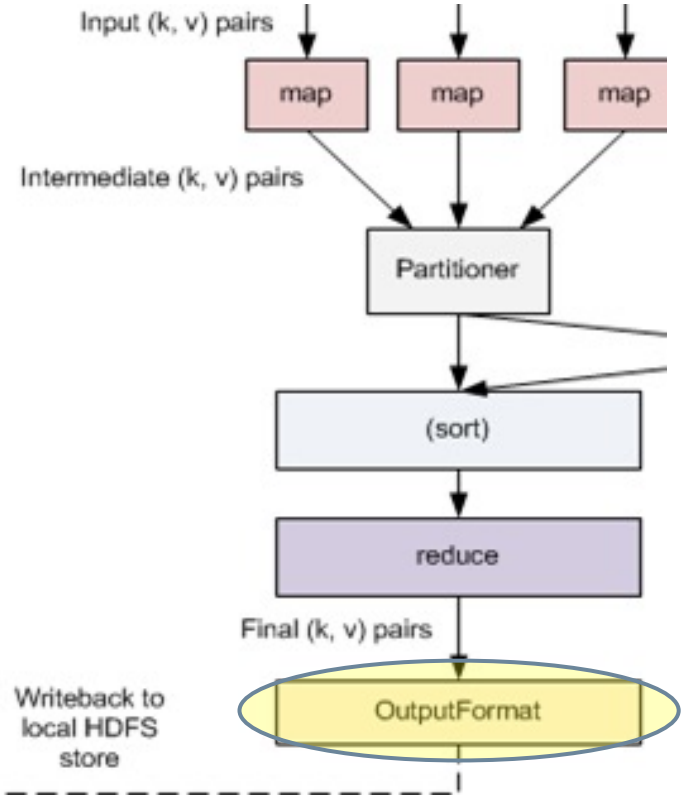
主要组件

- Reducer
  - 执行用户定义的Reduce操作
  - org.apache.hadoop.mapreduce.Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>



主要组件

- 文件输出格式OutputFormat
- 写入到HDFS的所有OutputFormat都继承自FileOutputFormat
- 每一个Reducer都写一个文件到一个共同的输出目录，文件名是part-r-nnnnn，其中nnnnn是与每一个reducer相关的一个号（partition id）
- FileOutputFormat.setOutputPath()
- job.setOutputFormat()



主要组件

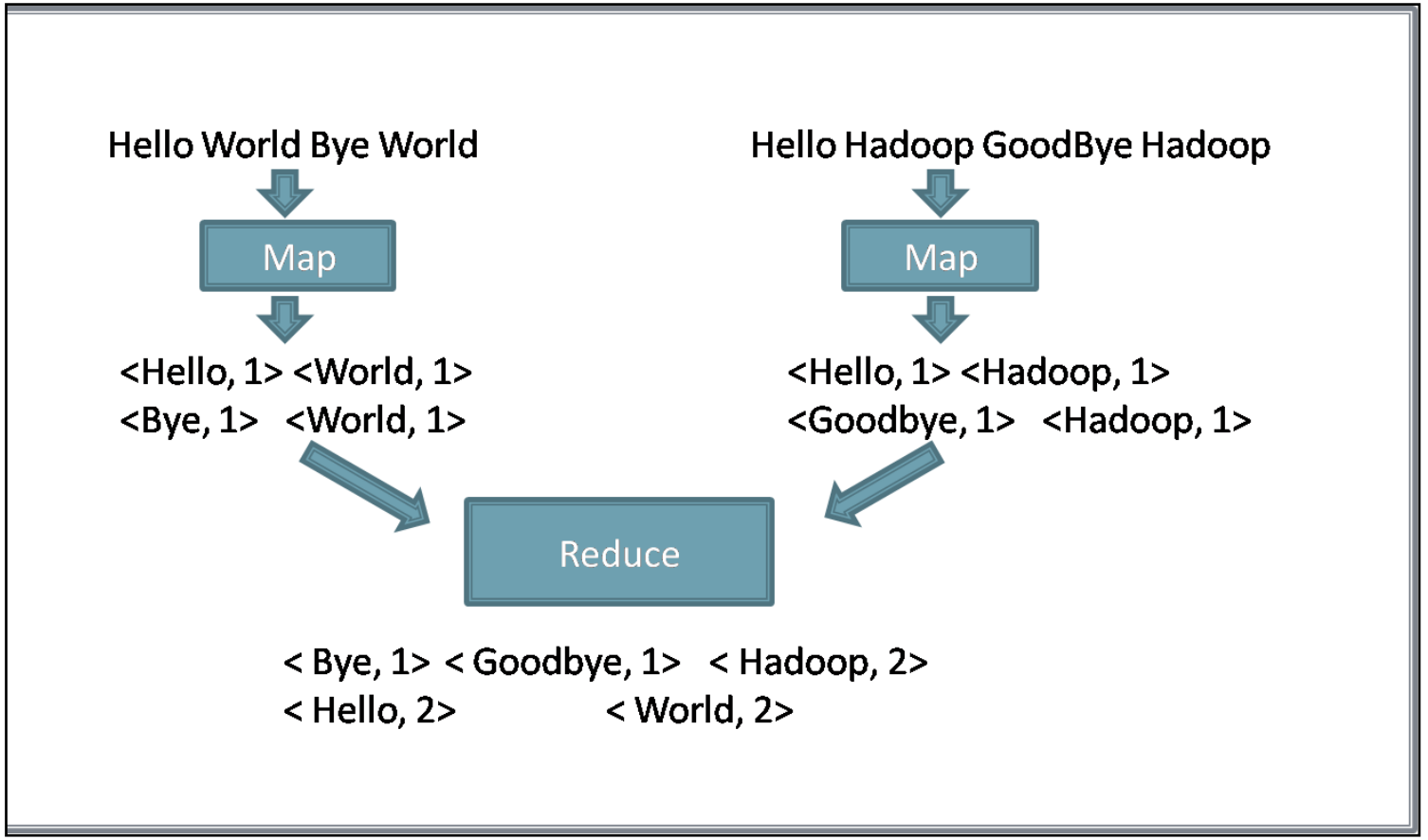
- 文件输出格式OutputFormat

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

- RecordWriter
  - TextOutputFormat实现了缺省的LineRecordWriter，以”key\t value”形式输出一行结果

MapReduce WordCount 1.0

- 基本数据处理流程



MapReduce WordCount 1.0

- 程序员主要的编码工作如下：
  - 实现Map类
  - 实现Reduce类
  - 实现main函数（运行Job）

MapReduce WordCount 1.0

- 实现Map类
  - 这个类实现 org.apache.hadoop.mapreduce.Mapper中的 map 方法，输入参数中的 value 是文本文件中的一行，利用 StringTokenizer 将这个字符串拆成单词，然后通过**context.write**收集<key, value>对。
  - 代码中 LongWritable, IntWritable, Text 均是 Hadoop 中实现的用于封装 Java 数据类型的类，这些类都能够被串行化从而便于在分布式环境中进行数据交换，可以将它们分别视为 long, int, String 的替代。

MapReduce WordCount 1.0

- **Map类代码**

public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

```
//定义Map类实现字符串分解
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    //实现map()函数
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //将字符串拆解成单词
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());          //将分解后的一个单词写入word
            context.write(word, one);           //收集<key, value>
        }
    }
}
```

MapReduce WordCount 1.0

- 实现Reduce类
  - 这个类实现org.apache.hadoop.mapreduce.Reducer中的 reduce 方法，输入参数中的(key, values) 是由 Map 任务输出的中间结果，values 是一个Iterator，遍历这个 Iterator，就可以得到属于同一个 key的所有value。
  - 此处key 是一个单词，value 是词频。只需要将所有的 value 相加，就可以得到这个单词的总的出现次数。

MapReduce WordCount 1.0

- **Reduce类代码**

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

```
//定义Reduce类规约同一key的value
public static class IntSumReducer  extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    //实现reduce()函数
    public void reduce(Text key, Iterable<IntWritable> values, Context context ) throws IOException, InterruptedException {
      int sum = 0;
      //遍历迭代器values，得到同一key的所有value
      for (IntWritable val : values) {  sum += val.get();      }
      result.set(sum);
      //产生输出对<key, value>
      context.write(key, result);
    }
}
```

MapReduce WordCount 1.0

- 实现main函数（运行Job）
  - 在 Hadoop 中一次计算任务称之为一个 Job，main函数主要负责新建一个Job对象并为之设定相应的Mapper和Reducer类，以及输入、输出路径等。

MapReduce WordCount 1.0

- **main函数代码**

```
public static void main(String[] args) throws Exception{
    //为任务设定配置文件
    Configuration conf = new Configuration();
    //命令行参数
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2){
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, “word count”);                    //新建一个用户定义的Job
    job.setJarByClass(WordCount.class);                       //设置执行任务的jar
    job.setMapperClass(TokenizerMapper.class);                //设置Mapper类
    job.setCombinerClass(IntSumReducer.class);                //设置Combine类
    job.setReducerClass(IntSumReducer.class);                 //设置Reducer类
    job.setOutputKeyClass(Text.class);                        //设置job输出的key
    //设置job输出的value
    job.setOutputValueClass(IntWritable.class);
    //设置输入文件的路径
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    //设置输出文件的路径
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    //提交任务并等待任务完成
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

源代码出处（官网tutorial）：
https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

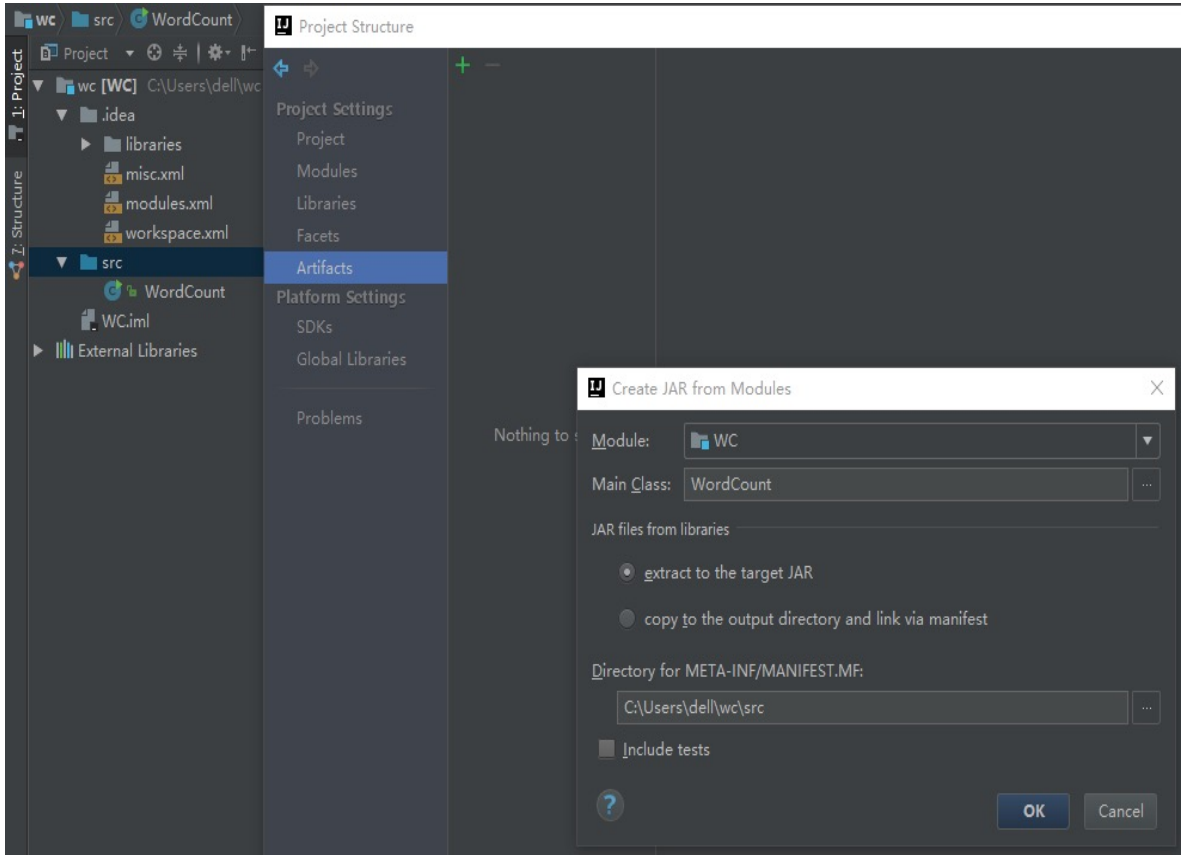
或者（github仓库）：
https://github.com/apache/hadoop/tree/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples

MapReduce WordCount 1.0

- 编译源代码
  - 完成编译
  - 导出jar文件
    - 导出wordcount程序的jar包
    - 导出jar文件的时候可以指定一个主类MainClass，作为默认执行的一个类

可以安装Big Data Tools插件

IntelliJ IDEA CE



MapReduce WordCount 1.0

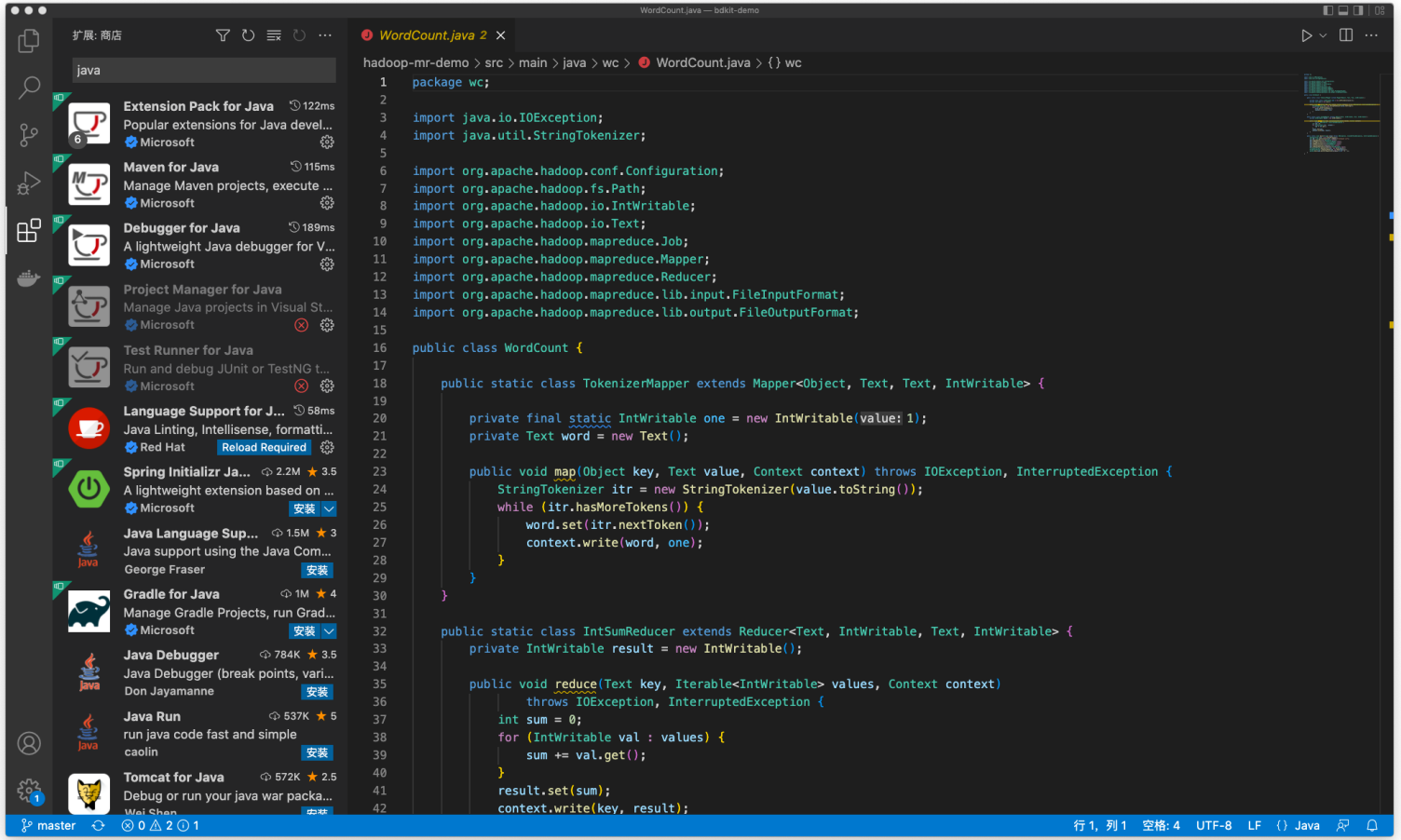
- 本地运行调试
  - 将程序复制到本地Hadoop系统的执行目录，并准备一个小的测试数据，即可通过hadoop的安装包进行运行调试

```
bin/hadoop fs -mkdir input
bin/hadoop fs -put docs/*.html input
bin/hadoop jar example.jar wordcount input output
```

  - 当需要用集群进行海量数据处理时，在本地程序调试正确运行后，可按照前述的远程作业提交步骤，将作业提交到远程hadoop集群上运行。

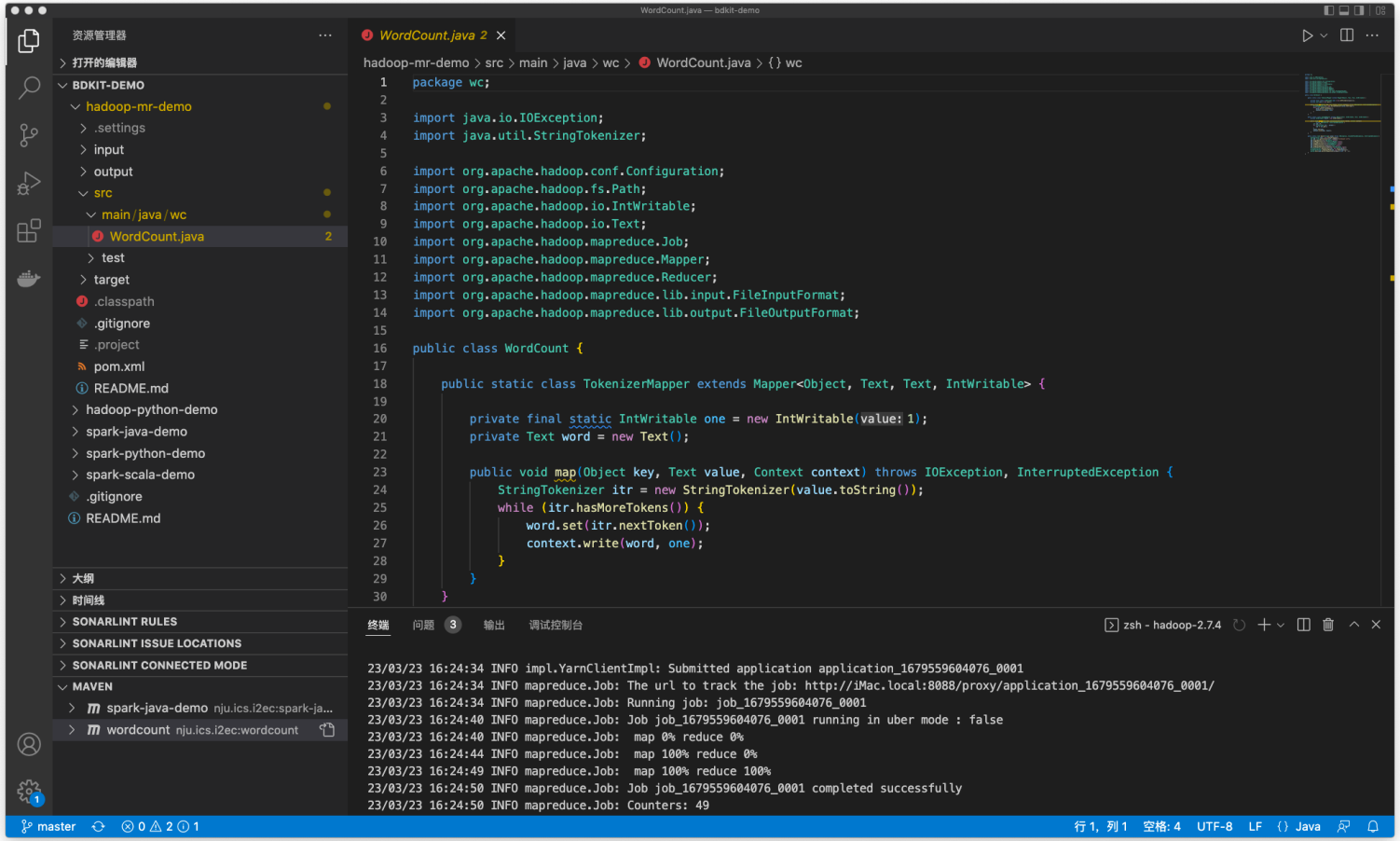
开发环境与工具：VS Code

安装Java开发插件和Maven插件



开发环境与工具：VS Code

编写、编译、打包、运行Java程序



MapReduce WordCount 2.0

- 1. 忽略大小写
- 2. 忽略标点符号

源代码出处：
https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

MapReduce WordCount 2.0

- Mapper (new)

```
public void setup(Context context) throws IOException,
    InterruptedException {
  conf = context.getConfiguration();
  caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
  if (conf.getBoolean("wordcount.skip.patterns", true)) {
    URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
    for (URI patternsURI : patternsURIs) {
      Path patternsPath = new Path(patternsURI.getPath());
      String patternsFileName = patternsPath.getName().toString();
      parseSkipFile(patternsFileName);
    }
  }
}
```

MapReduce WordCount 2.0

- Mapper (new)

```
private void parseSkipFile(String fileName) {
  try {
    fis = new BufferedReader(new FileReader(fileName));
    String pattern = null;
    while ((pattern = fis.readLine()) != null) {
      patternsToSkip.add(pattern);
    }
  } catch (IOException ioe) {
    System.err.println("Caught exception while parsing the cached file '"
        + StringUtils.stringifyException(ioe));
  }
}
```

MapReduce WordCount 2.0

- Mapper (updated)

```
@Override
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  String line = (caseSensitive) ?
      value.toString() : value.toString().toLowerCase();
  for (String pattern : patternsToSkip) {
    line = line.replaceAll(pattern, "");
  }
  StringTokenizer itr = new StringTokenizer(line);
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
    Counter counter = context.getCounter(CountersEnum.class.getName(),
        CountersEnum.INPUT_WORDS.toString());
    counter.increment(1);
  }
}
```

MapReduce WordCount 2.0

- Reducer

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

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

MapReduce WordCount 2.0

- main函数

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  GenericOptionsParser optionParser = new GenericOptionsParser(conf, args);
  String[] remainingArgs = optionParser.getRemainingArgs();
  if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
    System.err.println("Usage: wordcount <in> <out> [-skip skipPatternFile]");
    System.exit(2);
  }
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount2.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  List<String> otherArgs = new ArrayList<String>();
  for (int i=0; i < remainingArgs.length; ++i) {
    if ("-skip".equals(remainingArgs[i])) {
      job.addCacheFile(new Path(remainingArgs[++i]).toUri());
      job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
    } else {
      otherArgs.add(remainingArgs[i]);
    }
  }
  FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));

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

MapReduce WordCount 2.0

- Input
  - File01: Hello World Bye World
  - File02: Hello Hadoop Goodbye Hadoop
  - File03: Hello World, Bye World!
  - File04: Hello Hadoop, Goodbye to hadoop.
- 运行 $bin/hadoop jar wc2.jar input output2
- 结果:

```
Bye     2
Goodbye 2
Hadoop  2
Hadoop, 1
Hello   4
World   2
World!  1
World,  1
hadoop. 1
to      1
```

MapReduce WordCount 2.0

- 准备patterns文件

```
$ bin/hadoop fs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

- 再次运行
  - $ bin/hadoop jar wc2.jar -Dwordcount.case.sensitive=true input output3 -skip wordcount/patterns.txt
- 输出：

```
Bye     2
Goodbye 2
Hadoop  3
Hello   4
World   4
hadoop  1
```

- 再次运行
  - $ bin/hadoop jar wc2.jar -Dwordcount.case.sensitive=false input output4 -skip wordcount/patterns.txt
- 输出：

```
bye     2
goodbye 2
hadoop  4
hello   4
world   4
```

MapReduce WordCount 2.0

- Demonstrates how applications can access configuration parameters in the setup method of the Mapper (and Reducer) implementations.
- Demonstrates how the DistributedCache can be used to distribute read-only data needed by the jobs. Here it allows the user to specify word-patterns to skip while counting.
- Demonstrates the utility of the GenericOptionsParser to handle generic Hadoop command-line options.
- Demonstrates how applications can use Counters and how they can set application-specific status information passed to the map (and reduce) method.

思考

- How many Maps？
  - Map的数量通常由输入的总大小决定，即输入文件块的总数。
  - Map的合适并行度大概是每个节点10-100个。对于cpu非常轻的map任务，可以设置为300个。Task的初始化需要一段时间，因此执行map至少需要一分钟。
- How many Reduces？
  - Reduce的数量可以设置为0.95~1.75 * (节点的数量 * 每个节点的最大容器数量)
  - 增加reduce的数量会增加框架开销，但可以提高负载平衡并降低故障成本。
- Reducer NONE
  - 如果不需要reduce任务，则可以将reduce任务的数量设置为零。
  - map任务的输出直接进入设置的输出路径。在将map输出写入文件系统之前，框架不会对它们进行排序。

MapReduce排序算法

- 排序分类一
  - 1. 按照key排序
  - 2. 按照value排序
    - 可以改为按key排序，只需在map函数中将key和value对调，然后在reduce函数中再对调回去
    - 或者借助：org.apache.hadoop.mapreduce.lib.map.InverseMapper
- 排序分类二
  - 部分排序
  - 全局排序
  - 辅助排序
  - 二次排序
  - ……

MapReduce排序算法

- Data Size
  - 10MB? 10GB? 1000GB?
- Sort Algorithm in MapReduce
  - map(k1, *) -> (k1, *) // Identity function
  - shuffle and sort
    - (1) total-order partitioning
    - (2) local sorting
  - reduce(k1, *) -> (k1, *) // Identity function
- A customized total-order *Partitioner*
  - recall that shuffle phase needs a *Partitioner* to partition the key space
- InputFormat, OutputFormat
  - that depends on your data format

MapReduce排序算法

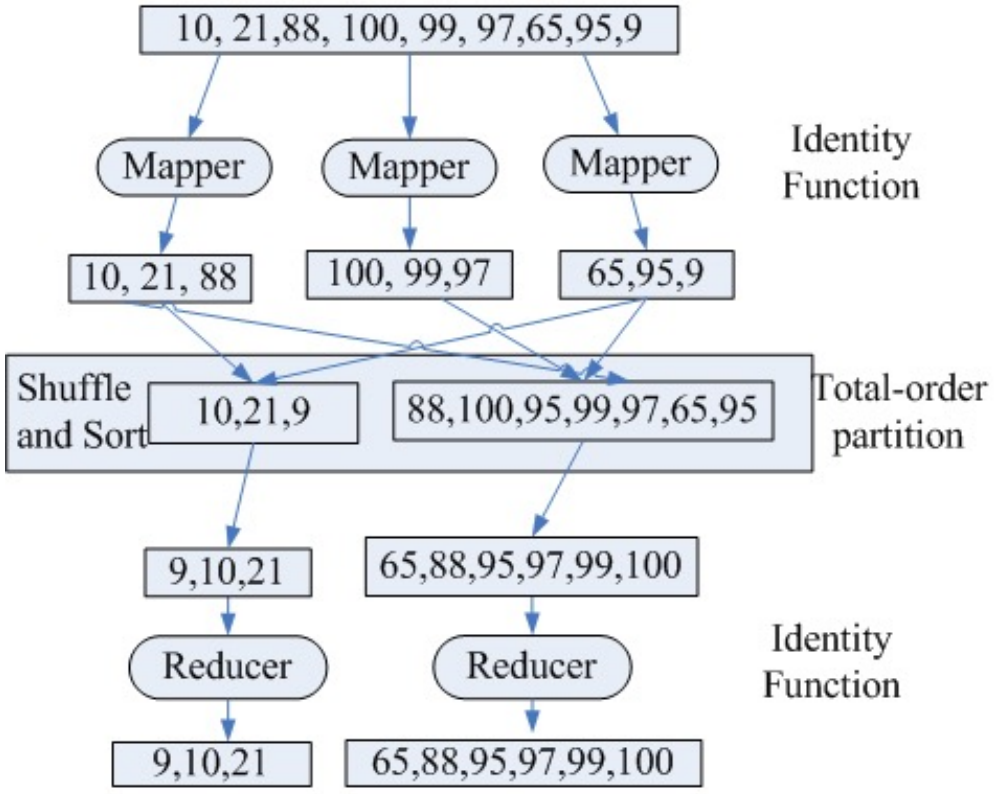


Figure: A simple MapReduce sorting example with 3 mappers, 2 reducers

Is there any problem here?

MapReduce排序算法

- Partitioner
  - 两个问题
    - (1) 如何避免在某些Reducer上聚集过多的数据而拖慢了整个程序
    - (2) 当有大量的key要分配到多个partition（也就是Reducer）时，如何高效地找到每个Key所属的partition
  - 对Partitioner的要求
    - 划分均匀
    - 查找快速
  - Thank God ☺
    - There exists a class, ***TotalOrderPartitioner*** in hadoop libs, which was originally used in TeraSort.

MapReduce排序算法

- TeraSort
  - In May 2008, running on a 910-node cluster, Hadoop sorted the 10 billion records (1 TB in size) in 209 seconds (3.48 minutes) to win the annual general purpose terabyte sort benchmark.
  - The cluster statistics were:
    - 910 nodes
    - 4 dual core Xeons @ 2.0ghz per a node
    - 4 SATA disks per a node
    - 8G RAM per a node
    - 1 gigabit Ethernet on each node
    - Red Hat Enterprise Linux Server Release 5.1 (kernel 2.6.18)
    - Sun Java JDK 1.6.0_05-b13
  - In May 2009, it was announced that a team at Yahoo! used Hadoop to sort one terabyte in 62 seconds.

MapReduce排序算法

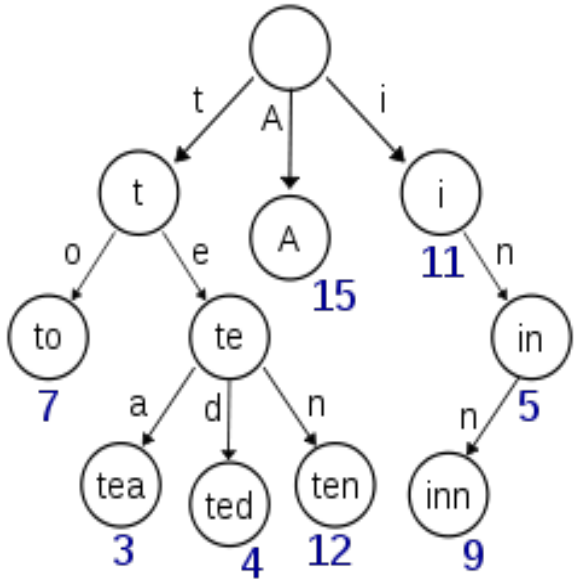
- TotalOrderPartitioner for TeraSort
  - TotalOrderPartitioner
    - 一个提供全序划分的*Partitioner*
    - 从Hadoop v0.19.0开始正式发布在库类中
  - 为满足两个要求所采用的策略
    - 通过采样获取数据的分布
    - 构建高效的划分模型

MapReduce排序算法

- TotalOrderPartitioner
  - 获取数据分布作均匀划分
    - Key 的分布未知
    - 预读一小部分数据采样(sample)
    - 对采样数据排序后均分，假设有N个reducer，则取得N-1个分割点
    - uses a sorted list of *N-1* sampled keys that define the key range for each reduce.
    - In particular, all keys such that ***sample[i-1]*** <= ***key*** < ***sample[i]*** are sent to reduce *i*. This guarantees that the output of reduce *i* are all less than the output of reduce *i+1*.
  - Example
    - 设reduce数目为3，采到9条记录：1,22,55,60,62,66,68,70,90
    - 取两个分割点60,68；划分区间为：[*,60), [60, 68), [68,*)

MapReduce排序算法

- TotalOrderPartitioner
  - 高效的划分模型
    - 若Key 的数据类型是BinaryComparable的，即两个对象的可以直接按字节比较大小（如Text），则以key构造Trie Tree；否则以二分查找来确定key的所属区间
    - Trie Tree，一种高效的适于查找的数据结构
    - The partitioner builds **a two level trie** that quickly indexes into the list of sample keys based on the first two bytes of the key. (ref: hadoop docs)
    - 两级的Trie可以最多对应大约256*256个reducer，通常是足够的



A trie for keys "A","to", "tea", "ted", "ten", "i", "in", and "inn".

源代码出处：
https://github.com/apache/hadoop/tree/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/terasort

MapReduce二级排序

- SecondarySort
  - Hadoop在将Mapper产生的数据输送给Reducer之前，会自动对它们进行排序
  - 那么，如果我们还希望按值排序，应该怎么做呢?
    - 二级排序。通过对key对象的格式进行小小的修改，二级排序可以在排序阶段将值的作用也施加进去。

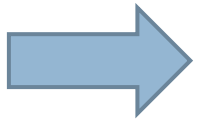
源代码出处：
https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/SecondarySort.java

MapReduce二级排序

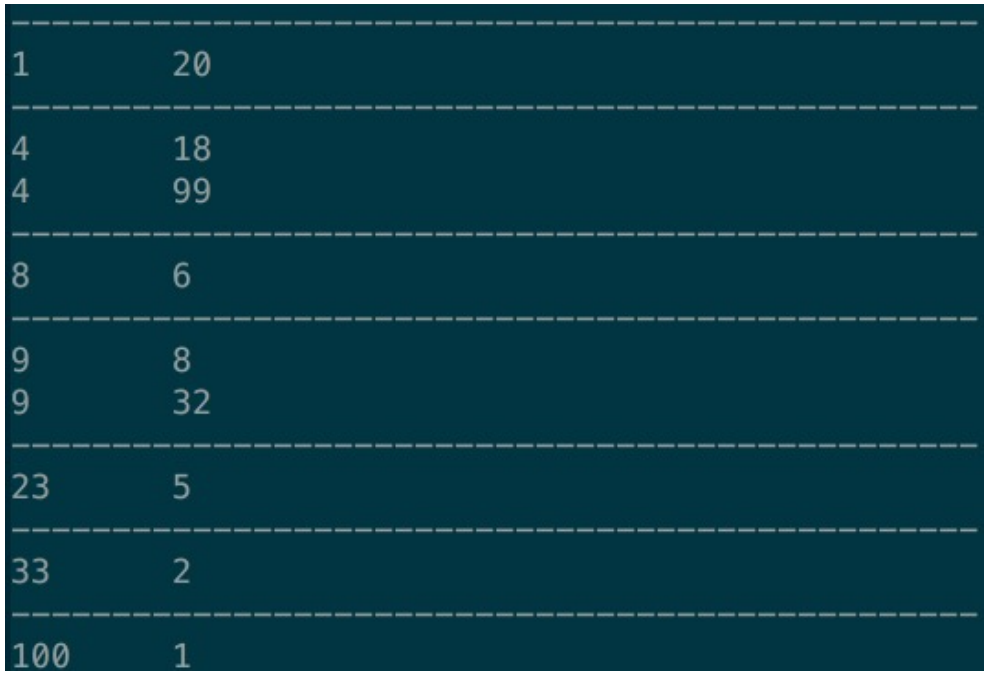
- SecondarySort
  - 方法1：Reducer将给定key的所有值都缓存起来，然后对它们再做一个Reducer内排序。但是，由于Reducer需要保存给定key的所有值，可能会导致出现内存耗尽的错误。
  - 方法2：将值的一部分或整个值加入原始key，生成一个合成key。
    - 生成组合key的过程很简单。我们需要先分析一下，在排序时需要把值的哪些部分考虑在内，然后，把它们加进key里去。随后，再修改key类的compareTo方法或是Comparator类，确保排序的时候使用这个组合而成的key。

MapReduce二级排序

- 例如对如下int pair进行排序：
  - 1 20
  - 9 8
  - 9 32
  - 33 2
  - 4 99
  - 4 18
  - 8 6
  - 100 1
  - 23 5



- 设计的(key value)对为((left, right),right)



MapReduce二级排序

- 主要工作：
  - 1. 自定义key
    - Define a pair of integers that are writable. They are serialized in a byte comparable format.
    - class IntPair implements WritableComparable<IntPair>
  - 2. 自定义Partitioner类
    - Partition based on the first part of the pair.
    - class FirstPartitioner extends Partitioner<IntPair, IntWritable>
  - 3. 自定义Key的比较类
    - A Comparator that compares serialized IntPair.
    - class Comparator extends WritableComparator

MapReduce二级排序

- 主要工作：
  - 4. 自定义分组比较类
    - Compare only the first part of the pair, so that reduce is called once for each value of the first part.
    - class FirstGroupingComparator extends WritableComparator
  - 5. 定义Mapper类
    - Read two integers from each line and generate a key, value pair as ((left, right), right).
  - 6. 定义Reducer类
    - A reducer class that just emits the sum of the input values.
- 实现： org.apache.hadoop.examples.SecondarySort

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