**使用MapReduce解决矩阵乘法的问题**

想在海量数据中淘到金子，强大的挖掘工具是必不可少的，而诸如回归、聚类、主成分分析、决策树等数据挖掘算法常常涉及大规模矩阵运算。这其中，大矩阵乘法具有较大的时间消耗，是算法的瓶颈。所以将矩阵乘法移植到分布式系统中进行运算，可谓是基本需求，所以本文暂且从最基础开始，简单介绍使用MapReduce实现矩阵乘法的方式。

简单回顾一下矩阵乘法：

IMG_256

矩阵乘法要求左矩阵的列数与右矩阵的行数相等，m×n的矩阵A，与n×p的矩阵B相乘，结果为m×p的矩阵C。

为了方便描述,先进行假设：

* 矩阵A的行数为m，列数为n，aij为矩阵A第i行j列的元素。
* 矩阵B的行数为n，列数为p，bij为矩阵B第i行j列的元素。

# 分析

因为分布式计算的特点，需要找到相互独立的计算过程，以便能够在不同的节点上进行计算而不会彼此影响。根据矩阵乘法的公式，C中各个元素的计算都是相互独立的，即各个cij在计算过程中彼此不影响。这样的话，在Map阶段可以把计算所需要的元素都集中到同一个key中，然后，在Reduce阶段就可以从中解析出各个元素来计算cij。

另外，以a11为例，它将会在c11、c12……c1p的计算中使用。也就是说，在Map阶段，当我们从HDFS取出一行记录时，如果该记录是A的元素，则需要存储成p个<key, value>对，并且这p个key互不相同；如果该记录是B的元素，则需要存储成m个<key, value>对，同样的，m个key也应互不相同；但同时，用于存放计算cij的ai1、ai2……ain和b1j、b2j……bnj的<key, value>对的key应该都是相同的，这样才能被传递到同一个Reduce中。

# 设计

普遍有一个共识是：数据结构+算法=程序，所以在编写代码之前需要先理清数据存储结构和处理数据的算法。

## 算法

### map阶段

在map阶段，需要做的是进行数据准备。把来自矩阵A的元素aij，标识成p条<key, value>的形式，key="i,k",（其中k=1,2,...,p），value="a:j,aij"；把来自矩阵B的元素bij，标识成m条<key, value>形式，key="k,j"（其中k=1,2,...,m），value="b:i,bij"。

经过处理，用于计算cij需要的a、b就转变为有相同key（"i,j"）的数据对，通过value中"a:"、"b:"能区分元素是来自矩阵A还是矩阵B，以及具体的位置（在矩阵A的第几列，在矩阵B的第几行）。

### shuffle阶段

这个阶段是Hadoop自动完成的阶段，具有相同key的value被分到同一个Iterable中，形成<key,Iterable(value)>对，再传递给reduce。

### reduce阶段

通过map数据预处理和shuffle数据分组两个阶段，reduce阶段只需要知道两件事就行：

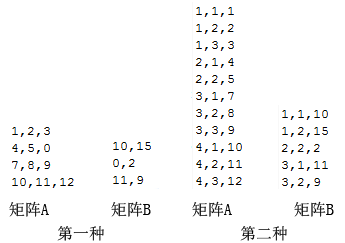
* <key,Iterable(value)>对经过计算得到的是矩阵C的哪个元素？因为map阶段对数据的处理，key（i,j）中的数据对，就是其在矩阵C中的位置，第i行j列。
* Iterable中的每个value来自于矩阵A和矩阵B的哪个位置？这个也在map阶段进行了标记，对于value（x:y,z），只需要找到y相同的来自不同矩阵（即x分别为a和b）的两个元素，取z相乘，然后加和即可。

## 数据结构

计算过程已经设计清楚了，就需要对数据结构进行设计。大体有两种设计方案：

第一种：使用最原始的表示方式，相同行内不同列数据通过","分割，不同行通过换行分割；

第二种：通过行列表示法，即文件中的每行数据有三个元素通过分隔符分割，第一个元素表示行，第二个元素表示列，第三个元素表示数据。这种方式对于可以不列出为0的元素，即可以减少稀疏矩阵的数据量。



在上图中，第一种方式存储的数据量小于第二种，但这只是因为例子中的数据设计成这样。在现实中，使用分布式计算矩阵乘法的环境中，大部分矩阵是稀疏矩阵，且数据量极大，在这种情况下，第二种数据结构的优势就显现了出来。**而且**，因为使用分布式计算，如果数据大于64M，在map阶段将不能够逐行处理，将不能确定数据来自于哪一行。**不过**，由于现实中对于大矩阵的乘法，考虑到存储空间和内存的情况，需要特殊的处理方式，有一种是将矩阵进行行列转换然后计算，这个时候第一种还是挺实用的。

# 编写代码

## 第一种数据结构

代码为：

import java.io.IOException;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class MatrixMultiply {

public static class MatrixMapper extends

Mapper<LongWritable, Text, Text, Text> {

private String flag = null;// 数据集名称

private int rowNum = 4;// 矩阵A的行数

private int colNum = 2;// 矩阵B的列数

private int rowIndexA = 1; // 矩阵A，当前在第几行

private int rowIndexB = 1; // 矩阵B，当前在第几行

@Override

protected void setup(Context context) throws IOException,

InterruptedException {

flag = ((FileSplit) context.getInputSplit()).getPath().getName();// 获取文件名称

}

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] tokens = value.toString().split(",");

if ("ma".equals(flag)) {

for (int i = 1; i <= colNum; i++) {

Text k = new Text(rowIndexA + "," + i);

for (int j = 0; j < tokens.length; j++) {

Text v = new Text("a," + (j + 1) + "," + tokens[j]);

context.write(k, v);

}

}

rowIndexA++;// 每执行一次map方法，矩阵向下移动一行

} else if ("mb".equals(flag)) {

for (int i = 1; i <= rowNum; i++) {

for (int j = 0; j < tokens.length; j++) {

Text k = new Text(i + "," + (j + 1));

Text v = new Text("b," + rowIndexB + "," + tokens[j]);

context.write(k, v);

}

}

rowIndexB++;// 每执行一次map方法，矩阵向下移动一行

}

}

}

public static class MatrixReducer extends

Reducer<Text, Text, Text, IntWritable> {

@Override

protected void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {

Map<String, String> mapA = new HashMap<String, String>();

Map<String, String> mapB = new HashMap<String, String>();

for (Text value : values) {

String[] val = value.toString().split(",");

if ("a".equals(val[0])) {

mapA.put(val[1], val[2]);

} else if ("b".equals(val[0])) {

mapB.put(val[1], val[2]);

}

}

int result = 0;

Iterator<String> mKeys = mapA.keySet().iterator();

while (mKeys.hasNext()) {

String mkey = mKeys.next();

if (mapB.get(mkey) == null) {// 因为mkey取的是mapA的key集合，所以只需要判断mapB是否存在即可。

continue;

}

result += Integer.parseInt(mapA.get(mkey))

\* Integer.parseInt(mapB.get(mkey));

}

context.write(key, new IntWritable(result));

}

}

public static void main(String[] args) throws IOException,

ClassNotFoundException, InterruptedException {

String input1 = "hdfs://192.168.1.128:9000/user/lxh/matrix/ma";

String input2 = "hdfs://192.168.1.128:9000/user/lxh/matrix/mb";

String output = "hdfs://192.168.1.128:9000/user/lxh/matrix/out";

Configuration conf = new Configuration();

conf.addResource("classpath:/hadoop/core-site.xml");

conf.addResource("classpath:/hadoop/hdfs-site.xml");

conf.addResource("classpath:/hadoop/mapred-site.xml");

conf.addResource("classpath:/hadoop/yarn-site.xml");

Job job = Job.getInstance(conf, "MatrixMultiply");

job.setJarByClass(MatrixMultiply.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(MatrixMapper.class);

// job.setReducerClass(MatrixReducer.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, new Path(input1), new Path(input2));// 加载2个输入数据集

Path outputPath = new Path(output);

outputPath.getFileSystem(conf).delete(outputPath, true);

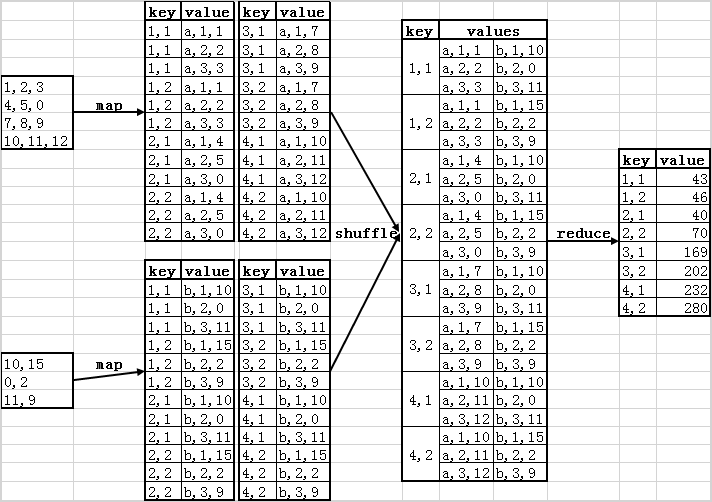
FileOutputFormat.setOutputPath(job, outputPath);

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

}

}

绘图演示效果：



## 第二种数据结构

代码为：

import java.io.IOException;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

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.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class SparseMatrixMultiply {

public static class SMMapper extends Mapper<LongWritable, Text, Text, Text> {

private String flag = null;

private int m = 4;// 矩阵A的行数

private int p = 2;// 矩阵B的列数

@Override

protected void setup(Context context) throws IOException,

InterruptedException {

FileSplit split = (FileSplit) context.getInputSplit();

flag = split.getPath().getName();

}

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] val = value.toString().split(",");

if ("t1".equals(flag)) {

for (int i = 1; i <= p; i++) {

context.write(new Text(val[0] + "," + i), new Text("a,"

+ val[1] + "," + val[2]));

}

} else if ("t2".equals(flag)) {

for (int i = 1; i <= m; i++) {

context.write(new Text(i + "," + val[1]), new Text("b,"

+ val[0] + "," + val[2]));

}

}

}

}

public static class SMReducer extends

Reducer<Text, Text, Text, IntWritable> {

@Override

protected void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {

Map<String, String> mapA = new HashMap<String, String>();

Map<String, String> mapB = new HashMap<String, String>();

for (Text value : values) {

String[] val = value.toString().split(",");

if ("a".equals(val[0])) {

mapA.put(val[1], val[2]);

} else if ("b".equals(val[0])) {

mapB.put(val[1], val[2]);

}

}

int result = 0;

// 可能在mapA中存在在mapB中不存在的key，或相反情况

// 因为，数据定义的时候使用的是稀疏矩阵的定义

// 所以，这种只存在于一个map中的key，说明其对应元素为0，不影响结果

Iterator<String> mKeys = mapA.keySet().iterator();

while (mKeys.hasNext()) {

String mkey = mKeys.next();

if (mapB.get(mkey) == null) {// 因为mkey取的是mapA的key集合，所以只需要判断mapB是否存在即可。

continue;

}

result += Integer.parseInt(mapA.get(mkey))

\* Integer.parseInt(mapB.get(mkey));

}

context.write(key, new IntWritable(result));

}

}

public static void main(String[] args) throws IOException,

ClassNotFoundException, InterruptedException {

String input1 = "hdfs://192.168.1.128:9000/user/lxh/matrix/t1";

String input2 = "hdfs://192.168.1.128:9000/user/lxh/matrix/t2";

String output = "hdfs://192.168.1.128:9000/user/lxh/matrix/out";

Configuration conf = new Configuration();

conf.addResource("classpath:/hadoop/core-site.xml");

conf.addResource("classpath:/hadoop/hdfs-site.xml");

conf.addResource("classpath:/hadoop/mapred-site.xml");

conf.addResource("classpath:/hadoop/yarn-site.xml");

Job job = Job.getInstance(conf, "SparseMatrixMultiply");

job.setJarByClass(SparseMatrixMultiply.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

job.setMapperClass(SMMapper.class);

job.setReducerClass(SMReducer.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, new Path(input1), new Path(input2));// 加载2个输入数据集

Path outputPath = new Path(output);

outputPath.getFileSystem(conf).delete(outputPath, true);

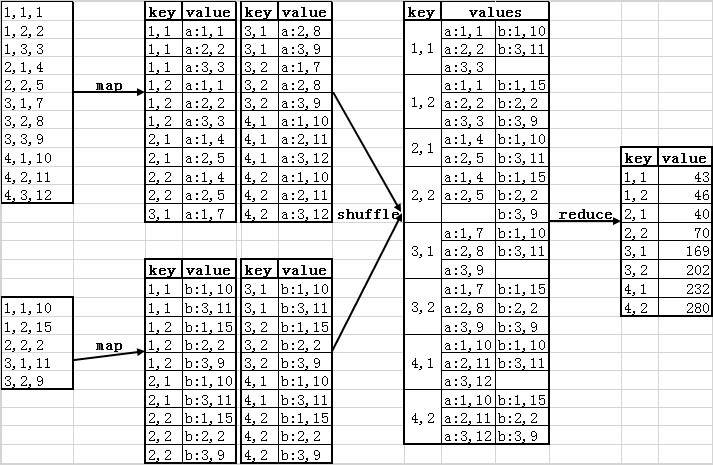
FileOutputFormat.setOutputPath(job, outputPath);

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

}

}

绘图演示效果：



## 代码分析

比较两种代码，可以很清楚的看出，两种实现只是在map阶段有些区别，reduce阶段基本相同。对于其中关于行i、列j定义不是从0计数（虽然我倾向于从0开始计数，不用写等号，简单），是为了更直观的观察数据处理过程是否符合设计。

在第一种实现中，需要记录当前是读取的哪一行数据，所以，这种仅适用于不需要分块的小文件中进行的矩阵乘法运算。第二种实现中，每行数据记录了所在行所在列，不会有这方面的限制。

在第二种实现中，遍历两个HashMap时，取mapA的key作为循环标准，是因为在一般情况下，mapA和mapB的key是相同的（如第一种实现），因为使用稀疏矩阵，两个不相同的key说明是0，可以舍弃不参与计算，所以只使用mapA的key，并判断mapB是否存在该key对应的值。

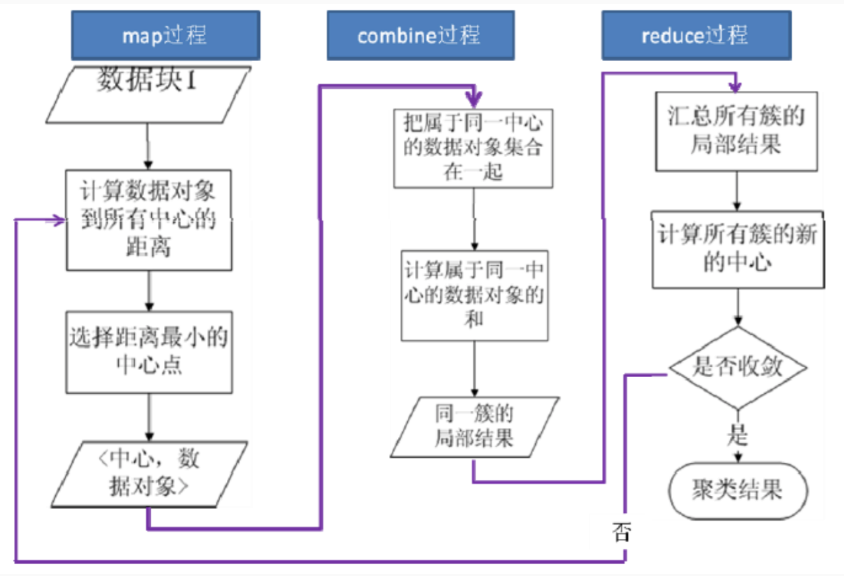
两种实现的reduce阶段，计算最后结果时，都是直接使用内存存储数据、计算结果，所以当数据量很大的时候（通常都会很大，否则不会用分布式处理），极易造成内存溢出，所以，对于大矩阵的运算，还需要其他的转换方式，比如**行列相乘运算、分块矩阵运算、基于最小粒度相乘的算法**等方式。另外，因为这两份代码都是demo，所以代码中缺少过滤错误数据的部分。

#### ****MapReduce实现 K-means 算法****

对一个对象求最近簇心并不受对另外一个对象求最近簇心的影响，所以这一步是可以并行的。由于每次迭代要用到的簇心是上一次迭代的结果，所以些迭代是串行的关系。

k-means 聚类算法进行 MapReduce 的基本思路:对串行算法中每 1 次迭代启 动对应的 1 次 MapReduce 计算过程,完成数据记录到聚类中心的距离计算以及新 的聚类中心的计算。

基于 MapReduce 的 k-means 并行算法主要有两部分,第一部分用于数据处理, 第二部分是 k-means 并行算法的实现部分。基于 MapReduce 的 k-means 算法的流程图如下图所示：



通过不同次的实验结果发现,初始的聚类中心对最后的聚类结果存在影响,  
迭代次数对最后的聚类结果存在影响,并且孤立点即聚类中不相似的文档对最后 的聚类结果也存在着影响。

第二个需要改进的地方是目前比较相似度的文档向量矩阵维数可能很大,所 以效率不是很高,可以取每个文档 TFIDF 值最高的前 20 个词构建文档向量矩阵, 这样或许可以提高效率。

从运行时间、磁盘利用率也看出了MR不适合做迭代类型的任务，中间结果都要落地到HDFS上，再读出进行下一次job的计算。迭代式任务还是需要使用spark这类计算模型提高效率。