中山大學
SUN YAT-SEN UNIVERSITY

实验报告

SUN YAT-SEN UNIVERSITY

院（系）数据科学与计算机学院 学 号 ----------

专 业 软件工程(移动信息工程) 实验人 ----------

实验题目：PageRank 算法在 hadoop 和 Spark 中的实现 2018 年 1 月 7 日

一、实验环境

按如下顺序搭建环境：

1. 3 台 64 位 ubuntu14.04 虚拟机，host 命名分别为 master，slave1，slave2；并且虚拟机之间要可以互相 ping 同且能够通过 ssh 无密码互相登录。

2.安装 java 开发环境 JDK：jdk-8u60-linux-x64.tar.gz

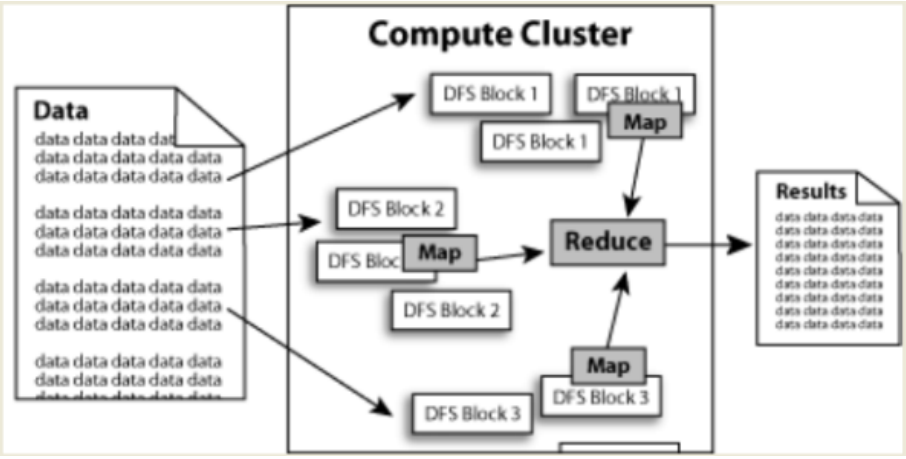
3.安装 hadoop：hadoop-2.6.0.tar.gz

4.安装 spark：spark-1.6.0-bin-hadoop2.6.tgz(选择 spark on yarn 模式安装)

二、Hadoop 和 Spark 的编程模型原理

1. hadoop 的 MapReduce 编程模型原理

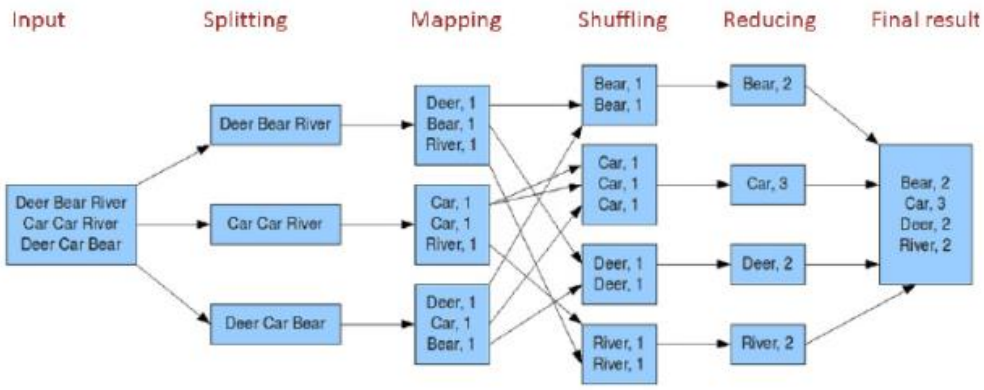
Hadoop 框架最核心设计就是 HDFS 和 MapReduce。其中 HDFS 是数据存储系统，提供了海量的数据存储，MapReduce 则是一个对数据进行计算处理的编程模型，整个 Hadoop 处理过程如下：



MapReduce 的处理过程分为两个步骤：map 和 reduce。每个阶段的输入输出都是 key-value 的形式，key 和 value 的类型可以自行指定。一个常用的方法是对于输入，将行作为一个 key，将行的内容作为 value，那么每一次，输入会按行切开成一个一个 split，每一个 input split 会创建一个 task

调用 Map 计算，在 Map 中我们写好的处理函数中，会对每一个 input split 作相应处理，最后将结果以<key， value>形式输出，具体什么是 key 什么是 value 就看你下一步希望对什么东西进行处理。之后 hadoop 会按照 key 值将 map 的输出整理后作为 Reduce 的输入，在这里 Reduce 的输入实际上是<key， List<value> >， 即将一个 key 的所有 value 合并成一个数组，这个数组整体为一个 value，然后传入到 reduce 中，那么在 reduce 中我们就可以一次性对一个 key 的所有 value 进行处理，最后将结果还以 key-value 的形式输出，作为整个 job 的输出，保存在 HDFS 上。

一个实例如下图，是统计每个单词出现了多少次，map 阶段对每行每遇到一个单词就将输出一个<单词，1>的对，在 reduce 阶段将同一个单词为 key 的 key-value 对的 value 值相加（因为这些 value 值都是 1），然后就可以得到 key 这个单词出现了多少次，最后输出。



2. Spark 编程原理

Spark 是类 Hadoop MapReduce 的通用并行框架，他拥有 Hadoop MapReduce 所具有的优点，也就是说 Spark 也可以进行 MapReduce 作业，而这也是我们在本次实验中使用的。不同于 Hadoop 的 MapReduce，Spark 没有文件管理功能，它需要依赖 Hadoop 分布式文件系统 HDFS，但是 Hadoop 在任务中会将数据先输出到文件系统下一步又从文件系统中读取，而 Spark 是在内存中处理一切数据，

这使得 Spark 的运算速度比 Hadoop 要快得多。

RDD 是 Spark 特有的数据模型，称为弹性分布式数据集，有容错机制，可以被缓存，支持并行操作。一个 RDD 代表一个分区里的数据集，Spark 的所有运算都是在 RDD 上进行的。在这里我们不需要深入理解 RDD 的模型原理，我们只需要知道 RDD 本质上就是一个数组。

RDD 有两种操作算子：

1. transformation（转换），当 RDD 执行转换操作转换成新的 RDD 时或者将外部数据转换成 RDD 时，实际操作并没有立刻执行，只是记住了逻辑操作。

2. Action（执行）：触发 Spark 作业的执行，真正触发转换算子的操作。

常用的 RDD 操作的 API 如下：

操作类型	函数名	作用
转换操作	Map()	参数是函数，函数应用于 RDD 每一个元素，返回值是新的 RDD
	flatMap()	参数是函数，函数应用于 RDD 每一个元素，将元素数据进行拆分，变成迭代器，返回值是新的 RDD
	mapValues()	同基本转换操作中的 map，只不过 mapValues 是针对[K,V]中的 V 值进行 map 操作。
	filter()	参数是函数，函数会过滤掉不符合条件的元素，返回值是新的 RDD
	distinct()	没有参数，将 RDD 里的元素进行去重操作
	union()	参数是 RDD，生成包含两个 RDD 所有元素的新 RDD
	intersection()	参数是 RDD，求出两个 RDD 的共同元素
	subtract()	参数是 RDD，将原 RDD 里和参数 RDD 里相同的元素去掉
	cartesian()	参数是 RDD，求两个 RDD 的笛

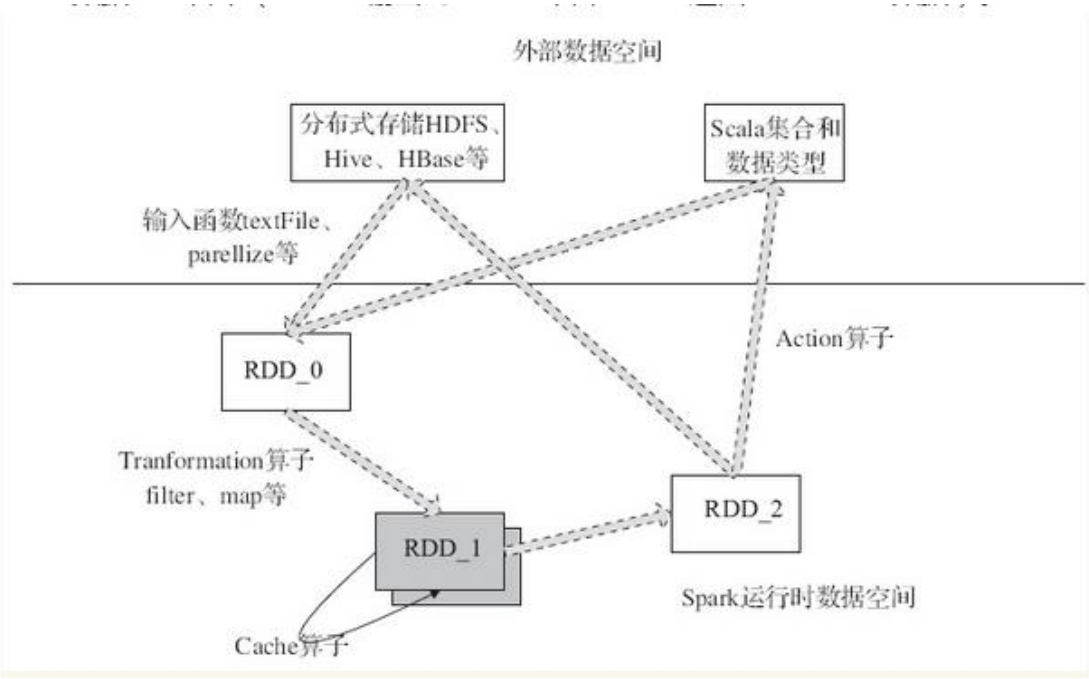
		卡儿积
行动操作	collect()	返回 RDD 所有元素
	count()	RDD 里元素个数
	countByValue()	各元素在 RDD 中出现次数
	reduce()	reduce 将 RDD 中元素前两个传给输入函数，产生一个新的 return 值，新产生的 return 值与 RDD 中下一个元素（第三个元素）组成两个元素，再被传给输入函数，直到最后只有一个值为止。
	reduceByKey()	reduceByKey 就是对元素为 KV 对的 RDD 中 Key 相同的元素的 Value 进行的 reduce 操作，因此，Key 相同的多个元素的值被 reduce 为一个值，然后与原 RDD 中的 Key 组成一个新的 KV 对。
	foreach(func)	对 RDD 每个元素都是使用特定函数

Spark 计算流程：

1. 输入: 在 Spark 程序运行中，数据从外部数据空间（例如，HDFS、Scala 集合或数据）输入到 Spark，数据就进入了 Spark 运行时数据空间，会转化为 Spark 中的数据块，通过 BlockManager 进行管理。

2. 运行: 在 Spark 数据输入形成 RDD 后, 便可以通过变换算子 fliter 等, 对数据操作并将 RDD 转化为新的 RDD，通过行动（Action）算子，触发 Spark 提交作业。如果数据需要复用，可以通过 Cache 算子，将数据缓存到内存。

3. 输出: 程序运行结束数据会输出 Spark 运行时空间, 存储到分布式存储中（如 saveAsTextFile 输出到 HDFS）或 Scala 数据或集合中（collect 输出到 Scala 集合，count 返回 Scala Int 型数据）



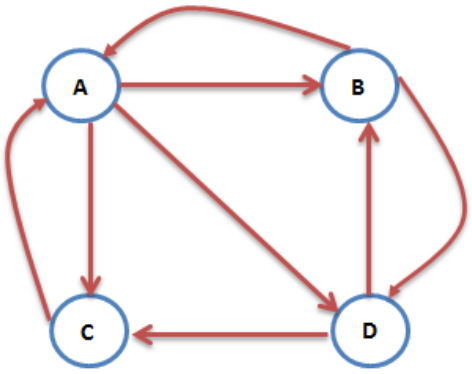
三、PageRank 算法介绍

1. PageRank 简介

PageRank 是 Google 专有的算法，用于衡量特定网页相对于搜索引擎索引中的其他网页而言的重要程度它由 Larry Page 和 Sergey Brin 在 20 世纪 90 年代后期发明。PageRank 实现了将链接价值概念作为排名因素。PageRank 将对页面的链接看成投票，指示了重要性。简单来说，PageRank 算法计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank 就是估计这个悠闲的上网者分布在各个网页上的概率。

2. PageRank 基本模型

我们将每个网页抽象成有向图中的一个结点，如果网页 A 可以通过链接跳转到网页 B，那么我们就在 A 和 B 之间建一条有向边 A->B，下面是一个简单的例子：



在上面这个例子中，A 可以跳转到 B、C、D，那么一个悠闲的上网者就会以 1/3 的概率跳转到这 3 个页面的某一个。同理一个网页有 K 个可跳转网页，那么他会以 1/K 的概率跳转到这 K 个页面中的一个。假设一共有 n 个网页，我们可以将所有网页间的转移概率写成一个 n*n 转移矩阵 M，其中如果网页 j 有 k 个出链，那么对每一个出链指向的网页 i，有 M[i][j]=1/k，而其他网页的 M[i][j]=0。那么上图例子中的转移矩阵如下：

$$\mathrm{M} = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

初始时，假设上网者在每一个网页的概率都是相等的，那么初始的 PageRank 值（为方便，以下简称 PR 值）就是 1/n，将其表示成一个 1*n 的列向量 PR，上面列子的 PR 如下：

$$PR_0 = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

新的 PR 值由下面的矩阵乘法计算得到：

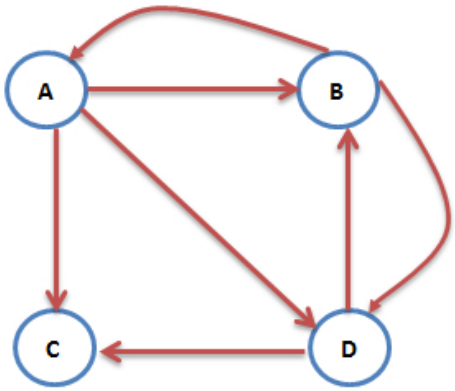
$$PR_{i+1} = M * PR_i$$

重复上式计算知道 PR 收敛为止，例子中最终 PR 收敛成 PR=[3/9, 2/9, 2/9, 2/9]'

3. 终止点问题

上述上网者的行为是一个马尔科夫的过程的实例，要满足收敛性，需要具备一个条件：图是强连通的，即从任意网页可以到达其他任意网页。

实际上，互联网的网页不满足强连通性质，因为有一些网页不指向任何网页，这样到最后上网者到达这样的网页之后就没办法继续走，导致前面累计的概率会被清 0。一个例子如下：



对应的转移矩阵：

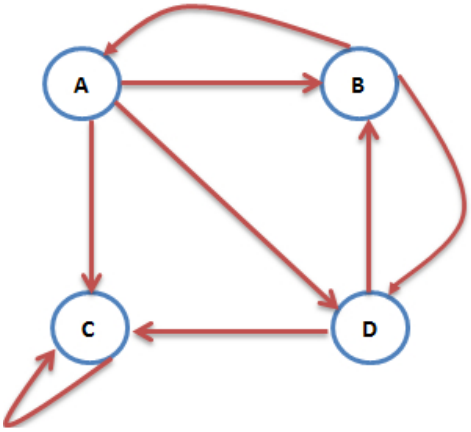
$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

PR 的迭代变化过程，最终会到 0：

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 3/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 5/48 \\ 7/48 \\ 7/48 \\ 7/48 \end{bmatrix} \begin{bmatrix} 21/288 \\ 31/288 \\ 31/288 \\ 31/288 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

4. 陷阱问题

还有一个问题是陷阱问题，即有些网页不存在指向其他网页的链接，但存在指向自己的链接，比如下面这个例子：



上网者到达 C 页面后，就陷入了陷阱，再也不能从 C 出来，这会导致最终概率分布值全部转移到 C，使得其他网页的概率分布值为 0.上述问题的转移矩阵为：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

不断迭代 PR 的变化如下：

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 3/24 \\ 5/24 \\ 11/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 5/48 \\ 7/48 \\ 29/48 \\ 7/48 \end{bmatrix} \begin{bmatrix} 21/288 \\ 31/288 \\ 205/288 \\ 31/288 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

4. PageRank 优化模型

在实际上网过程中，上网者不止会通过网页中的链接去往其他网而言，他也可能有一定概率直接在地址栏里输入新网址去往其他网页，这就给解决终止点问题和陷阱问题提供了方法。很显然，在地址栏输入某个网址的概率是 1/n。我们这里再设置一个阻尼系数α，这个系数的作用在于我们假设上网者查看当前网页的概率为α，那么他通过地址栏输入新网址去其他页面的概率就是（1-α）。所以原来的 PR 值的计算公式就转换成如下式子：

$$PR_{i+1} = \alpha * M * PR_i + (1-\alpha) * P2$$

其中$P2 = \begin{bmatrix} 1/n \\ 1/n \\ \dots \\ 1/n \\ 1/n \end{bmatrix}$

至此，一个完善的 PageRank 计算模型就做好了，不会再陷入终止点问题和陷阱问题。值得注意的是，

Google 公司将阻尼系数设为 0.85，而我们在本次实验中也采用这个值。

四、PageRank 在 hadoop Map-Reduce 上的实现设计

1. 基本思想与问题提出

根据前人的经验，上面 PageRank 的计算过程，一般迭代 20~30 次就基本收敛了。那么这里我们设计的基本想法是：

1. 设置一个最大迭代次数，这里我们设为了 20

2. 将每一次迭代过程作为一次完整的 Map-Reduce 过程，也就是说下一次迭代 Map 过程的输入是上一次迭代的 Reduce 的输出。

基本想法成型之后，那么问题也就随之而来，主要有下面几个问题：

1. 如何设计输入数据格式，使得其既能满足输入的格式要求，又能满足输出的格式要求，即输入输出的数据格式是一样的，这是一个最重要的难点。

2. 明确 Map 过程和 Reduce 过程分别需要处理什么。

2. 数据格式处理

为了解决输入数据格式问题，我们首先思考计算过程需要什么。很明显，需要两样东西，一个是转移矩阵，一个是当前 PR 值向量。但是，由于网页数量可以很多，我们不可能用一个二维矩阵来存储转移矩阵。注意到矩阵是系数的，我们可以用一个稀疏矩阵的形式来表示：将每一个网页和其能跳转到

的网页作为一行，那么原理中的例子就如下：

```
A:B,C,D
B:A,D
C:A
D:B,C
```

第一行 A：B,C,D 就表示从 A 可以跳转去 B,C,D。后面以此类推。然后转移概率可以根据有多少个课跳转网站算出来。这样我们就解决了转移矩阵的输入问题。那么 PR 初始向量呢，虽然大家都知道是 1/n，n 是网站数量，但是统计网站数量需要额外一个 MapReduce 来实现，比较麻烦，这里直接将 n 当成命令行参数输入，然后输入文件中先手动加上初始 PR 概率，如下：

```
A 0.25:B,C,D
B 0.25:A,D
C 0.25:A
D 0.25:B,C
```

第一行 A 0.25：B，C，D 就表示，网页 A 当前 PR 值为 0.25，可跳转页面为 B,C,D，后面每行以此类推。这个格式同时也解决了我们 Reduce 的输出格式问题。Reduce 也按这个格式输出即可，这样只有 PR 值是变的，其他都不变，不如重新思考如何读入上一次迭代的结果。数据格式的难题到此解决。

3. Map 过程

在 Map 过程我们要做的就是将转移概率和当前 PR 概率提取出来，并且按照键值对输出。比如第一行 A 0.25：B，C，D，对于 B，它可以获得从 A 转移到它的概率 1/3，在后面对 B 的 PR 值的计算中它还需要用到 A 的当前 PR 值，所以还需要获得 A 的当前 PR 值 0.25，因为 PR'[B] = …+ M[B][A]*PR[A]+…。

那么我们设计键值对如下(关键技术点)：

<"subPage", "PR: nowPage PRvalueForNowPage">

<"subPage", "M: nowPage MvalueForNowPageToSubPage">

前者意义就是从 subPage 会使用到 nowPage 的当前 PR 值。 后者意义就是从 nowPage 转移到 subPage 的转移概率是 MvalueForNowPageToSubPage。对于例子中的 B, 第一行输出的键值对就是：

<"B","M: A 0.333"> （我称为 B 的 M 键值对）

<"B","PR: A 0.25"> (我称为 B 的 PR 键值对)

其他的以此类推。当然仅有这两者还不够，因为最后 reduce 的输出中还需要包含每个网页可以跳转到哪几个网页，所以在 map 中我们需要将这一项原封不动的作为输出输出给 reduce。比如对于第一行就可以设计如下键值对：

<"A", "nextPage: B,C,D"> （我称为 A 的 nextPage 键值对）

总结上面的得到 Map 的计算过程如下：

1. 将输入（输入文本的每一行）按照冒号切开得到：
   A. 当前页及其 PR 值
   B. 当前页可跳转的页
2. 输出 A 的 nextPage 键值对
3. 将可跳转页按逗号切开，对每个可跳转页，输出该跳转页的 M 键值对以及 PR 键值对，其中 M 键值对中的转移概率的值可以通过可跳转页的数量得到，

注释代码如下：

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```java
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class PRMapper extends Mapper<LongWritable, Text, Text, Text>{
    //mapper的作用是从输入文件中获取转移矩阵的转移概率以及当前PR值
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
            /*
                当前页i PRi: 可跳转页1, 可跳转页2, ...
            */
            //将输入按: 切开, line[0] = 当前页 Vn  line[1] = 可跳转页1, 可跳转页2, ...
            String[] line = value.toString().split(":");
            String nextPage = line[1];
            int N = nextPage.split(",").length; //当前页有N个可跳转页
            String nowPage = line[0].split("\\s+")[0];
            String PR = line[0].split("\\s+")[1];
            context.write(new Text(nowPage), new Text("nextPage:"+nextPage));//当前页可跳转的页
            for(String subpage:nextPage.split(",")){
                //转移矩阵中从nowPage跳转到一个subpage的概率
                context.write(new Text(subpage), new Text("M:"+nowPage+" "+String.valueOf(1.0/N)));
                //当前subpage用到的nowPage的PR值
                context.write(new Text(subpage), new Text("PR:"+nowPage+" "+PR));
            }
    }
}
```

3. Reduce 过程

在 reduce 过程中，对于同一个键值，以 A 为例，它会收到以下键值对：

<”A”, “nextPage: B,C,D”>

<”A”, “PR: B 0.25”>

<”A”, “PR: C 0.25”>

<”A”, “M: B 0.5”>

<”A”, “M: C 1”>

我们在 reduce 的工作就是将上面这些键值对的 value 按照计算公式整合并输出。对每个键值对判断它是 nextPage 键值对、M 键值对、PR 键值对的哪一种， 对于 nextPage 键值对先保存该值等到最终输出再处理、对于 PR 键值对，和 M 键值对我们先将它们的 value 分别存储一个 HashMap 中，比如<”A”,

“PR: B 0.25”>， 那么 value 按<B, 0.25>的形式入一个名为 PR 的 HashMap 中，<”A”, “M: B 0.5”>则将 value 按<B, 0.25>的形式存入一个名为 M 的 HashMap 中，这样处理完之后，PR 这个 HashMap 就包含了当前 KEY （在这里就是 A）计算新 PR 值需要用到的所有父网页(即能跳转到它的网页)的 PR 值。而 M 这个 HashMap 就包含所用父网页到它的转移概率(关键点)。接下来我们只需要将两个 HashMap 中相同键值的 value 按公式计算在合并就可以了（这一步实际上是 PageRank 的基本模型计算: PR[i] = sigma（M[i][j]*PR[j])）。

即有:

$$\text{sum} = \sum_{key} M[key] * PR[key]$$

最后计算优化后的 PageRank 模型即可:

$$\text{newPR} = \text{damping} * \text{sum} + (1 - \text{damping})/\text{num}$$

其中 damping 是阻尼系数，num 是网站数量。

注释代码如下：

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.util.Map;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class PRReducer extends Reducer<Text, Text, Text, Text> {
    //对于一个页面KEY，计算其新的PR值
    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        Configuration conf = context.getConfiguration();
        double damping = Double.parseDouble(conf.get("damping"));//阻尼系数
        int num = Integer.parseInt(conf.get("num"));//网站数量

        Map<String, Double> PR = new HashMap<String,Double>();
        Map<String, Double> M = new HashMap<String,Double>();
```

```
        String nextPage = "";
        for (Text val : values) {
            String[] line = val.toString().split(":");
            if(line[0].compareTo("nextPage") == 0){
                //取出当前页可跳转的页面方便后面按照mapper输入文件格式输出
                nextPage = line[1];
            }
            else {
                String[] tmp = line[1].split(" ");
                double p = Double.parseDouble(tmp[1]);
                if(line[0].compareTo("PR") == 0){ //取出tmp的PR值存入PR 这个HashMap中
                    PR.put(tmp[0], p);
                }
                else if(line[0].compareTo("M") == 0){//取出从tmp转移到当前页的转移概率存到M这个HashMap中
                    M.put(tmp[0], p);
                }
            }
        }
        double sum=0;
        for(Map.Entry<String,Double> entry: PR.entrySet()){
            sum += M.get(entry.getKey()) * entry.getValue(); //计算PageRank基础模型 sigma(M[key]*PR[key])
        }
        double newPR = damping*sum + (1-damping)/num; // 计算优化后的PageRank模型

        System.out.println("    ");
        System.out.println(String.valueOf(key) + "'s New PageRank is " + String.valueOf(newPR));
        /*
            按照MAPPER中的输入格式输出
            当前页 当前页的PR值: 可跳转页1, 可跳转页2, ....
        */
        context.write(key, new Text(String.valueOf(newPR)+":"+nextPage));
    }
}
```

4. 主函数

主函数主要功能就是设置 hadoop 的任务。但是这个函数中有两点是非常重要的：

1. 循环迭代，每次迭代都是一次新的 hadoop 任务，每次要将上一次迭代的输出路径作为下一次的输入路径。这一步很关键，是解决如何在 hadoop 实现循环迭代问题的关键点，也是我们这次实验的一个难点。

2. 在前面的 reduce 过程中，需要用到网站的数量，重新用一个新的 mapreduce 过程实现网站数量统计非常麻烦，我们这里采用直接通过命令行参数传入这个网站数量。所以在主函数中还需要取出这个数量，并通过 hadoop 的 Configuration 类自定义变量来存这个值，同时还有阻尼系数值。这样

在后面 reduce 过程中可以同样通过 hadoop 的 Configuration 类取出这两个值（关键）。

注释代码如下：

```
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;


public class PageRank {

    public static void main(String[] args) throws Exception {
        double damping=0.85;
        ArrayList<String> ouputList=new ArrayList<String>();
        Configuration conf1 = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf1, args).getRemainingArgs();
            if (otherArgs.length != 3) {
                System.err.println("Usage: PageRank <num> <in> <out>");
                System.exit(2);
            }
        if (check(otherArgs[0]) == false){
            System.err.println("Please use interger as num");
            System.exit(2);
        }
        String num = otherArgs[0]; //从命令行参数中取出网页数量
        String input = otherArgs[1]; //从命令行参数中取出输入路径
        String OriOutput = otherArgs[2]; //从命令行参数中取出原始输入路径
        String output = otherArgs[2] + "1";

        for(int i = 1; i <= 20; i++){
            Configuration conf = new Configuration();
            conf.set("num", num);  //通过configuration 设置自定义类变量 num 和 damping
            conf.set("damping", String.valueOf(damping));
            //设置hadoop任务
            Job job = Job.getInstance(conf, "PageRank");
            job.setJarByClass(PageRank.class);
            job.setMapperClass(PRMapper.class);
            job.setReducerClass(PRReducer.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            FileInputFormat.setInputPaths(job, new Path(input));
            FileOutputFormat.setOutputPath(job, new Path(output));
            //一次迭代结束之后将这次的输出路径作为下一次的输入路径
            input = output;
            ouputList.add(output);
            output = OriOutput + String.valueOf(i+1);

            System.out.println("the "+i+"th step is finished");
```

```
            job.waitForCompletion(true);
        }

        for(int i=0;i<ouputList.size()-1;i++){ //删除多余的输出文件
            Configuration conf=new Configuration();
            Path path=new Path(ouputList.get(i));
            FileSystem fs=path.getFileSystem(conf);
            fs.delete(path,true);
        }

    }

    public static boolean check(String str) {
        for (int i = str.length();--i>=0;){
            if (!Character.isDigit(str.charAt(i))){
                return false;
            }
        }
        return true;
    }
}
```

四、PageRank 在 Spark 上的实现

由于在 Spark 上我们同样是采用 MapReduce 的形式，所以过程与 hadoop 上是一样的。代码采用 python 书写，比较简洁。具体如下：

1. 通过 Spark 的 textFile 函数获取输入数据。在 python 中会将每一行作为一个元素存到 list 中这里的 list 其实是一个 RDD。

```
inputData = sc.textFile(inputfile)
```

2. 使用 map 函数将每一个输入转换成键值对（page， links），即 page 是当前页，links 是能跳转的页。如（"A"， "B,C,D"）

```
link = inputData.map(lambda x: (x.split(":")[0], x.split(":")[1])) # links
```

3. 获取网页数量并通过 mapValues 函数初始化 PR 值为 1/n，并生成键值对（page， PR）到 RDD 中。如（A， 0.25）

```
n = inputData.count()  # get number of web pages
PageRank = link.mapValues(lambda x: 1.0/n)  # initial PR value
```

4.对于每一次迭代：

A. 将（page， links）和（page， PR）合并成新键值对（page， （links， PR））。然后通过 spark 的 flatMap 函数计算，计算过程是将 links 中的子页面分割开来，则可以计算某个子页面的基础 PageRank 模型求和前的一项：即对于每个子页面 i，计算所有 M[i][j]*PR[j]：

```
PageRank = link.join(PageRank).flatMap(calc)  # M[i][j]*PR[j]
```

FlatMap 用到的 calc 函数：

```
def calc(x):
    # x = (page, (links, rank))
    links = x[1][0]
    PageRank = x[1][1]
    links = links.split(",")
    n = len(links)
    result = []
    for x in links:
        result.append((x, PageRank*1.0/n))  # M[i][j]*PR[j]
    return result
```

B. 对于每个子页面 i，求和相关的值，即计算 sigma（M[i][j]*PR[j]），这一步就是完成 PageRank 基础模型的计算。

```
PageRank = PageRank.reduceByKey(lambda x, y: x+y) # sum = sigma(M[i][j]*PR[j])
```

C. 最后通过阻尼系数和网页总数计算优化后的模型

```
PageRank = PageRank.mapValues(lambda x: 0.85*x+0.15/n) # damping*sum+(1-damping)/n
```

完整代码：

```
from  pyspark import SparkContext, SparkConf


def calc(x):
    # x = (page, (links, rank))
    links = x[1][0]
    PageRank = x[1][1]
    links = links.split(",")
    n = len(links)
    result = []
    for x in links:
        result.append((x, PageRank*1.0/n))  # M[i][j]*PR[j]
    return result
```

```
if __name__ == "__main__":

    sc = SparkContext("local", "PR")
    inputfile = "/sparkinput/b.txt"
    inputData = sc.textFile(inputfile)
    link = inputData.map(lambda x: (x.split(":")[0], x.split(":")[1])) # links
    n = inputData.count()  # get number of web pages
    PageRank = link.mapValues(lambda x: 1.0/n)  # initial PR value

    for i in range(20):
        PageRank = link.join(PageRank).flatMap(calc)  # M[i][j]*PR[j]
        PageRank = PageRank.reduceByKey(lambda x, y: x+y) # sum = sigma(M[i][j]*PR[j])
        PageRank = PageRank.mapValues(lambda x: 0.85*x+0.15/n) # damping*sum+(1-damping)/n
    print PageRank.collect()
```

五、实验结果

为方便在报告中展示，我们先直接采用原理中的小数据集

1. hadoop 上的结果。

运行截图：

```
root@master:~# hadoop jar PageRank.jar PageRank 4 /PageRank_input /PageRank_output/output
the 1th step is finished
18/01/07 17:59:53 INFO client.RMProxy: Connecting to ResourceManager at master/192.168.1.212:8032
18/01/07 17:59:54 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with To
olRunner to remedy this.
18/01/07 17:59:55 INFO input.FileInputFormat: Total input paths to process : 1
18/01/07 17:59:56 INFO mapreduce.JobSubmitter: number of splits:1
18/01/07 17:59:56 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1515318970163_0001
18/01/07 17:59:57 INFO impl.YarnClientImpl: Submitted application application_1515318970163_0001
18/01/07 17:59:57 INFO mapreduce.Job: The url to track the job: http://master:8088/proxy/application_1515318970163_0001/
18/01/07 17:59:57 INFO mapreduce.Job: Running job: job_1515318970163_0001
18/01/07 18:00:10 INFO mapreduce.Job: Job job_1515318970163_0001 running in uber mode : false
18/01/07 18:00:10 INFO mapreduce.Job:  map 0% reduce 0%
18/01/07 18:00:22 INFO mapreduce.Job:  map 100% reduce 0%
18/01/07 18:00:29 INFO mapreduce.Job:  map 100% reduce 100%
18/01/07 18:00:29 INFO mapreduce.Job: Job job_1515318970163_0001 completed successfully
18/01/07 18:00:29 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=327
                FILE: Number of bytes written=211495
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=152
                HDFS: Number of bytes written=104
                HDFS: Number of read operations=6
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=8612
                Total time spent by all reduces in occupied slots (ms)=4016
                Total time spent by all map tasks (ms)=8612
                Total time spent by all reduce tasks (ms)=4016
                Total vcore-seconds taken by all map tasks=8612
                Total vcore-seconds taken by all reduce tasks=4016
                Total megabyte-seconds taken by all map tasks=8818688
                Total megabyte-seconds taken by all reduce tasks=4112384
        Map-Reduce Framework
                Map input records=4
                Map output records=20
                Map output bytes=281
                Map output materialized bytes=327
                Input split bytes=108
                Combine input records=0
                Combine output records=0
                Reduce input groups=4
                Reduce shuffle bytes=327
                Reduce input records=20
                Reduce output records=4
                Spilled Records=40
```

查看输出结果：

```
root@master:~# hdfs dfs -cat /PageRank_output/output20/*
A       0.32456140075268647:B,C,D
B       0.22514619974910452:A,D
C       0.22514619974910452:A
D       0.22514619974910452:B,C
root@master:~#
```

上如红框中的值就是最终的 PR 值。

2. Spark 上的结果。

运行截图，命令中我将输出结果重定向输出到 result.txt 中，这样之后就不用再一大堆信息中去找。

```
root@master:~# pyspark PageRank.py > result.txt
WARNING: Running python applications through 'pyspark' is deprecated as of Spark 1.0.
Use ./bin/spark-submit <python file>
18/01/07 18:12:08 INFO spark.SparkContext: Running Spark version 1.6.3
18/01/07 18:12:08 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
18/01/07 18:12:09 INFO spark.SecurityManager: Changing view acls to: root
18/01/07 18:12:09 INFO spark.SecurityManager: Changing modify acls to: root
18/01/07 18:12:09 INFO spark.SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(root); users with modif
y permissions: Set(root)
18/01/07 18:12:09 INFO util.Utils: Successfully started service 'sparkDriver' on port 38437.
18/01/07 18:12:10 INFO slf4j.Slf4jLogger: Slf4jLogger started
18/01/07 18:12:10 INFO Remoting: Starting remoting
18/01/07 18:12:10 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriverActorSystem@192.168.1.212:45229]
18/01/07 18:12:10 INFO util.Utils: Successfully started service 'sparkDriverActorSystem' on port 45229.
18/01/07 18:12:10 INFO spark.SparkEnv: Registering MapOutputTracker
18/01/07 18:12:10 INFO spark.SparkEnv: Registering BlockManagerMaster
18/01/07 18:12:10 INFO storage.DiskBlockManager: Created local directory at /usr/local/spark/blockmgr-2f147929-179c-4717-a20e-e0bf6c554b05
18/01/07 18:12:10 INFO storage.MemoryStore: MemoryStore started with capacity 853.1 MB
18/01/07 18:12:10 INFO spark.SparkEnv: Registering OutputCommitCoordinator
18/01/07 18:12:10 INFO server.Server: jetty-8.y.z-SNAPSHOT
18/01/07 18:12:10 INFO server.AbstractConnector: Started SelectChannelConnector@0.0.0.0:4040
18/01/07 18:12:10 INFO util.Utils: Successfully started service 'SparkUI' on port 4040.
18/01/07 18:12:10 INFO ui.SparkUI: Started SparkUI at http://192.168.1.212:4040
18/01/07 18:12:11 INFO util.Utils: Copying /root/PageRank.py to /usr/local/spark/spark-d9488a23-9eb6-4a42-9eec-cf16abcc401f/userFiles-0eba311e-befe-4e4c-8438-1963
f8e08ac7/PageRank.py
18/01/07 18:12:11 INFO spark.SparkContext: Added file file:/root/PageRank.py at file:/root/PageRank.py with timestamp 1515319931005
18/01/07 18:12:11 INFO executor.Executor: Starting executor ID driver on host localhost
18/01/07 18:12:11 INFO util.Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 35106.
18/01/07 18:12:11 INFO netty.NettyBlockTransferService: Server created on 35106
18/01/07 18:12:11 INFO storage.BlockManagerMaster: Trying to register BlockManager
18/01/07 18:12:11 INFO storage.BlockManagerMasterEndpoint: Registering block manager localhost:35106 with 853.1 MB RAM, BlockManagerId(driver, localhost, 35106)
18/01/07 18:12:11 INFO storage.BlockManagerMaster: Registered BlockManager
18/01/07 18:12:12 INFO scheduler.EventLoggingListener: Logging events to hdfs://master:9000/historyserverforSpark/local-1515319931089
18/01/07 18:12:12 INFO storage.MemoryStore: Block broadcast_0 stored as values in memory (estimated size 229.4 KB, free 852.9 MB)
18/01/07 18:12:12 INFO storage.MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated size 19.6 KB, free 852.9 MB)
18/01/07 18:12:12 INFO storage.BlockManagerInfo: Added broadcast_0_piece0 in memory on localhost:35106 (size: 19.6 KB, free: 853.1 MB)
18/01/07 18:12:12 INFO spark.SparkContext: Created broadcast 0 from textFile at NativeMethodAccessorImpl.java:-2
18/01/07 18:12:12 INFO mapred.FileInputFormat: Total input paths to process : 1
18/01/07 18:12:13 INFO spark.SparkContext: Starting job: count at /root/PageRank.py:22
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Got job 0 (count at /root/PageRank.py:22) with 1 output partitions
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Final stage: ResultStage 0 (count at /root/PageRank.py:22)
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Parents of final stage: List()
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Missing parents: List()
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Submitting ResultStage 0 (PythonRDD[2] at count at /root/PageRank.py:22), which has no missing parents
18/01/07 18:12:13 INFO storage.MemoryStore: Block broadcast_1 stored as values in memory (estimated size 5.7 KB, free 852.9 MB)
18/01/07 18:12:13 INFO storage.MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 3.5 KB, free 852.9 MB)
18/01/07 18:12:13 INFO storage.BlockManagerInfo: Added broadcast_1_piece0 in memory on localhost:35106 (size: 3.5 KB, free: 853.1 MB)
18/01/07 18:12:13 INFO spark.SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scala:1006
18/01/07 18:12:13 INFO scheduler.DAGScheduler: Submitting 1 missing tasks from ResultStage 0 (PythonRDD[2] at count at /root/PageRank.py:22)
18/01/07 18:12:13 INFO scheduler.TaskSchedulerImpl: Adding task set 0.0 with 1 tasks
18/01/07 18:12:13 INFO scheduler.TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, partition 0,ANY, 2169 bytes)
18/01/07 18:12:13 INFO executor.Executor: Running task 0.0 in stage 0.0 (TID 0)
18/01/07 18:12:13 INFO executor.Executor: Fetching file:/root/PageRank.py with timestamp 1515319931005
```

查看结果：

```
root@master:~# cat result.txt
[(u'A', 0.32456140075268647), (u'C', 0.22514619974910452), (u'B', 0.22514619974910452), (u'D', 0.22514619974910452)]
```

可以看到这个结果跟 hadoop 的运行结果是一致的。