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

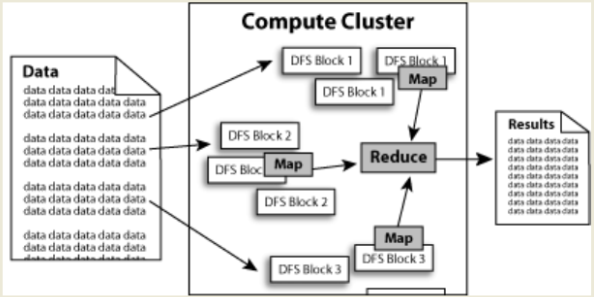
3.安装hadoop：hadoop-2.6.0.tar.gz

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

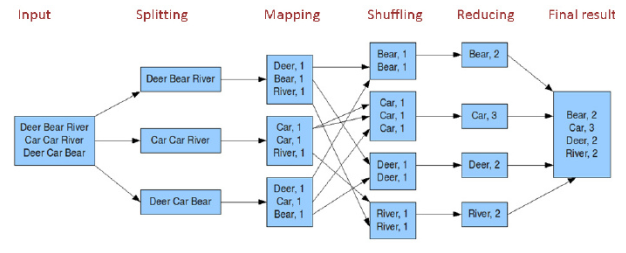
1. hadoop的MapReduce编程模型原理

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：

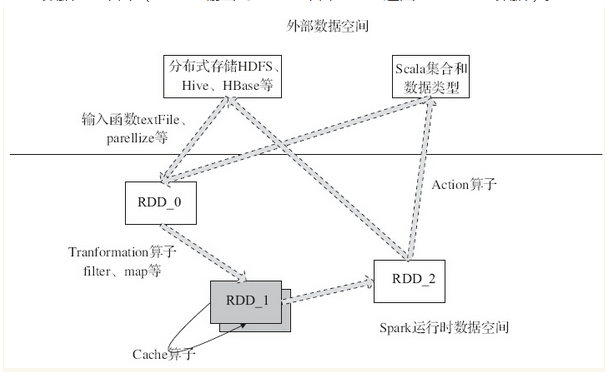
|  |  |  |
| --- | --- | --- |
| 操作类型 | 函数名 | 作用 |
| 转换操作 | Map() | 参数是函数，函数应用于RDD每一个元素，返回值是新的RDD |
| flatMap() | 参数是函数，函数应用于RDD每一个元素，将元素数据进行拆分，变成迭代器，返回值是新的RDD |
| mapValues() | 同基本转换操作中的map，只不过mapValues是针对[K,V]中的V值进行map操作。 |
| filter() | 参数是函数，函数会过滤掉不符合条件的元素，返回值是新的RDD |
| distinct() | 没有参数，将RDD里的元素进行去重操作 |
| 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）



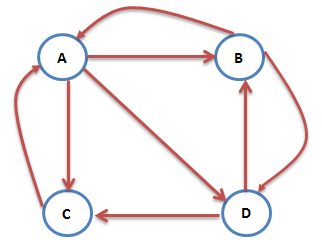
三、PageRank 算法介绍

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：

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：

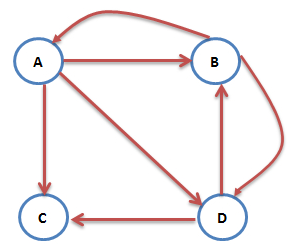
新的PR值由下面的矩阵乘法计算得到：

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

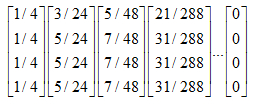
上述上网者的行为是一个马尔科夫的过程的实例，要满足收敛性，需要具备一个条件: 图是强连通的，即从任意网页可以到达其他任意网页。

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：



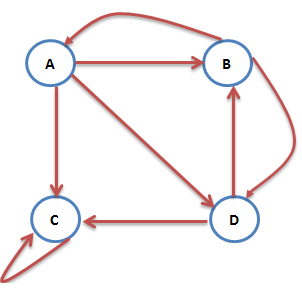
对应的转移矩阵：

PR的迭代变化过程，最终会到0：



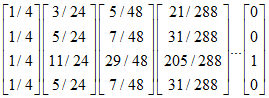
4. 陷阱问题

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：

不断迭代PR的变化如下：



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：

（1-α）

其中

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

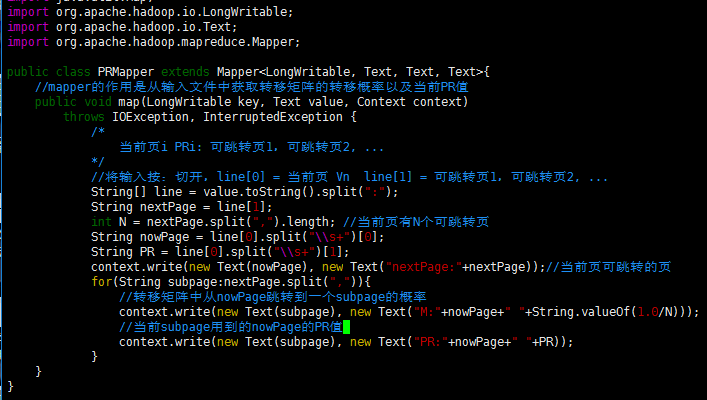
B. 当前页可跳转的页

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：





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”>

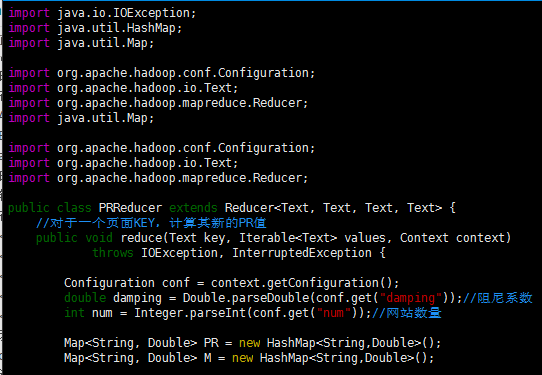
我们在reduce的工作就是将上面这些键值对的value按照计算公式整合并输出。对每个键值对判断它是nextPage键值对、M键值对、PR键值对的哪一种， 对于nextPage键值对先保存该值等到最终输出再处理、对于PR键值对，和M键值对我们先将它们的value分别存储一个HashMap中，比如<”A”, “PR: B 0.25”>， 那么value按<B, 0.25>的形式入一个名为PR的HashMap中，<”A”, “M: B 0.5”>则将value按<B, 0.25>的形式存入一个名为M的HashMap中，这样处理完之后，PR这个HashMap就包含了当前KEY （在这里就是A）计算新PR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:

最后计算优化后的PageRank模型即可:

其中damping是阻尼系数，num是网站数量。

注释代码如下：





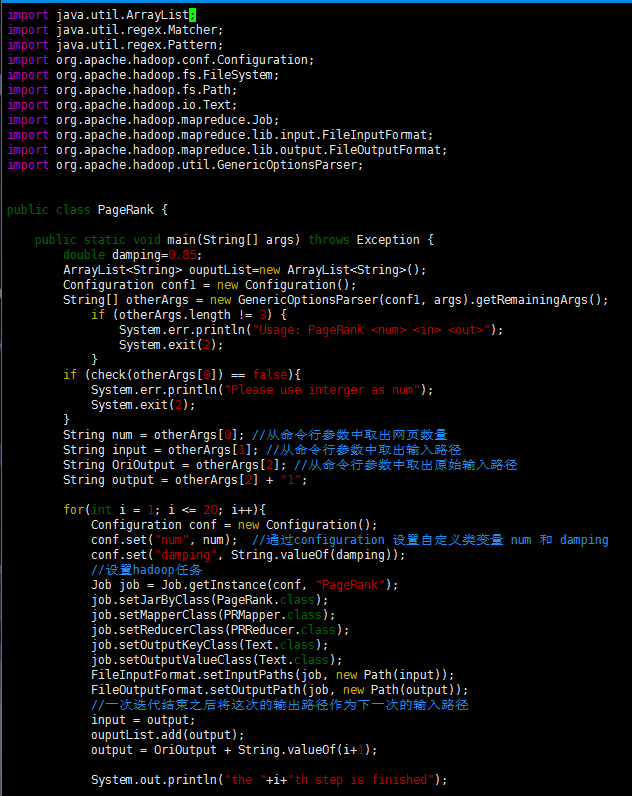
4. 主函数

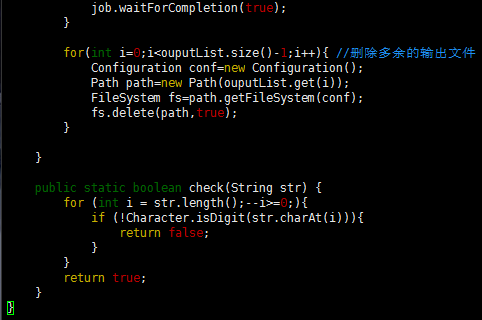
主函数主要功能就是设置hadoop的任务。但是这个函数中有两点是非常重要的：

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：





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。



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””）



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）

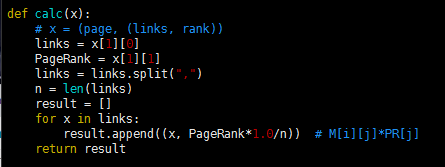


4.对于每一次迭代：

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]：



FlatMap用到的calc函数：



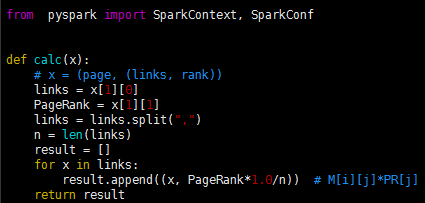
B. 对于每个子页面i，求和相关的值，即计算sigma（M[i][j]\*PR[j]），这一步就是完成PageRank基础模型的计算。

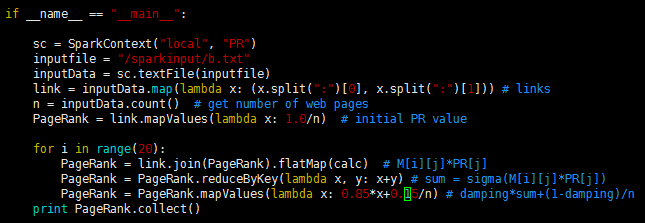


C. 最后通过阻尼系数和网页总数计算优化后的模型



完整代码：



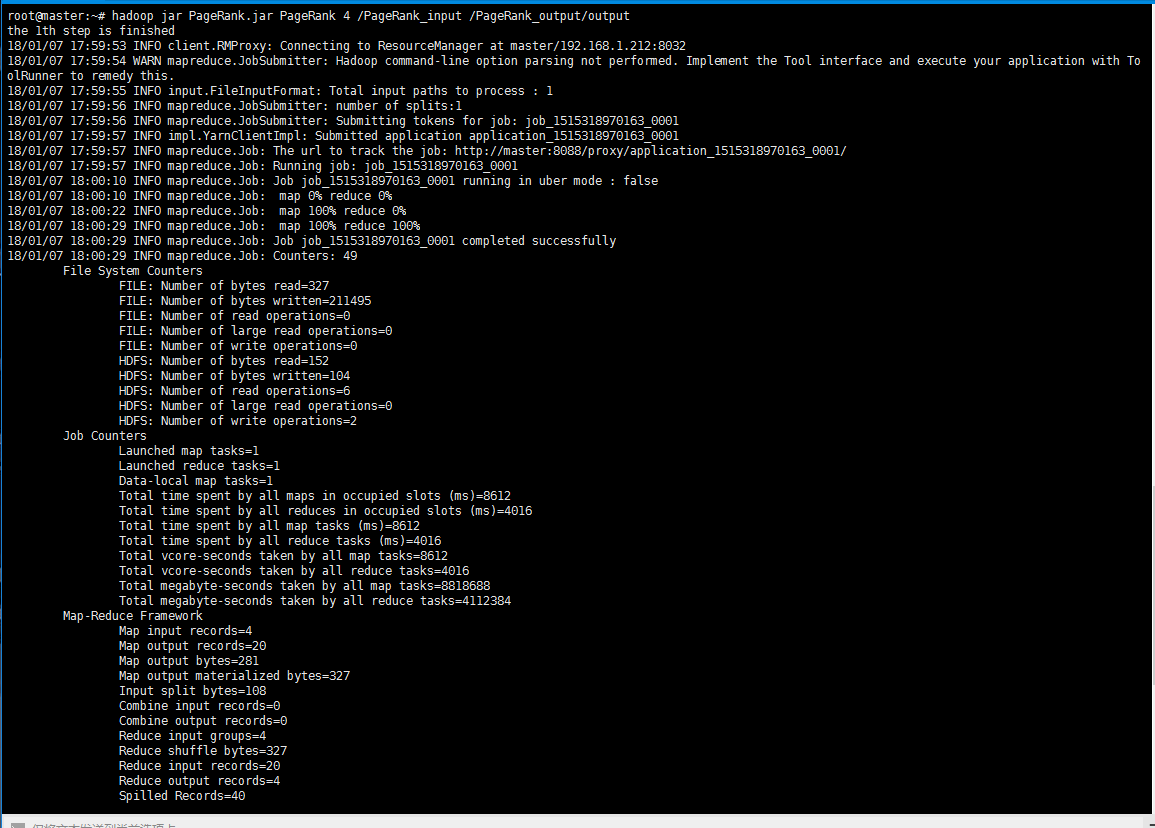


五、实验结果

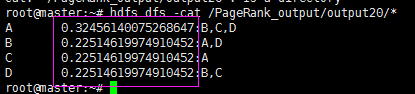
为方便在报告中展示，我们先直接采用原理中的小数据集

1. hadoop 上的结果。

运行截图：



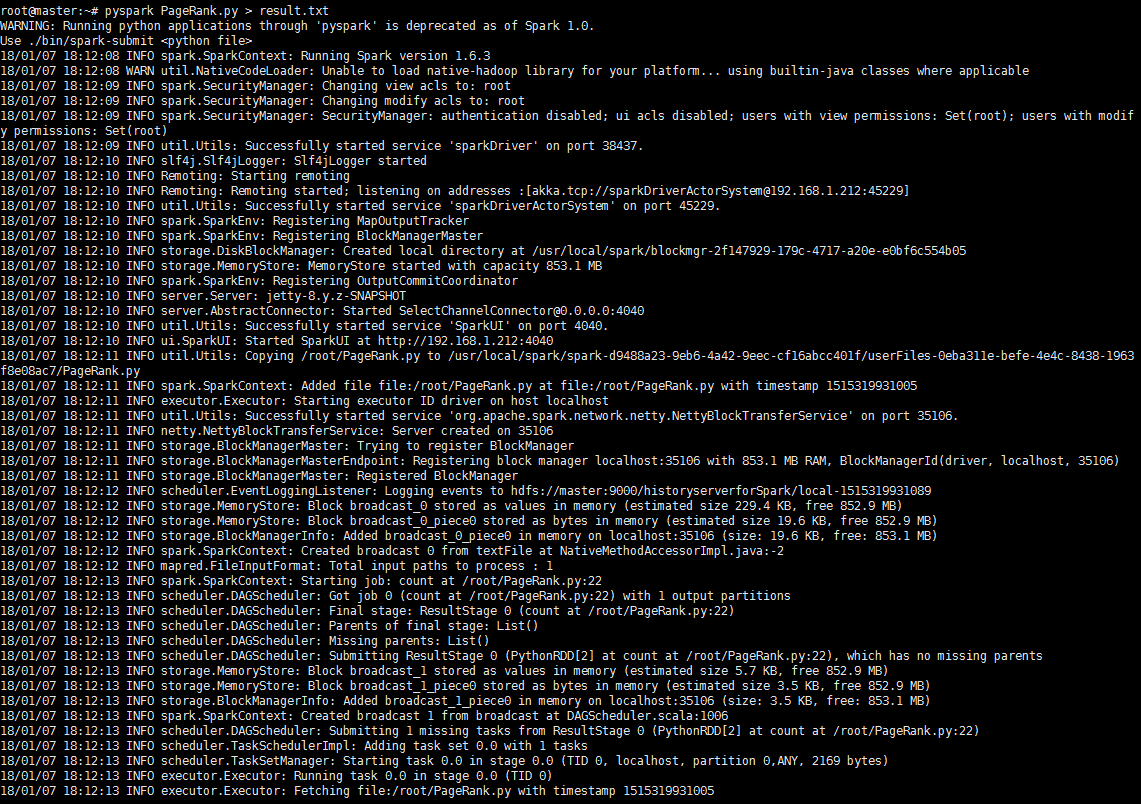
查看输出结果：



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。



查看结果：



可以看到这个结果跟hadoop的运行结果是一致的。

对比可知Spark的代码不仅更快，因为它不需要从磁盘中反复存取数据，而且实现起来更加简洁。