分布式算法

1. 分布式方案

1.1. 原本公式

$$PR(a) = [\beta(M + a^T(\frac{e}{n})) + (1-\beta)\frac{ee^T}{n}] * V$$

1.2. 分布式方案

1.2.1. 描述

公式当中的 $[\beta(M + a^T(\frac{e}{n})) + (1-\beta)\frac{ee^T}{n}]$ 是定值，记为 M^* ，则公式可以化简为 $PR = M^* \times V$ ，根据矩阵乘法的特性，只需要将矩阵按行进行拆分，每个节点只负责部分行的计算，最后将结果汇总即可。加入某个节点负责第 i 行的计算，那么只需要获取 M^* 第 i 行的数据和 V ，即可计算出 PR 的第 i 行的值，将所有节点计算的结果汇总就可以得到完成的 PR 。

1.2.2. 缺点

需要存储 M^* ，占用内存 $O(n^2)$ 极大，不适合大数据规模。

1.2.3. 优化内存

$M^* = [\beta(M + a^T(\frac{e}{n})) + (1-\beta)\frac{ee^T}{n}]$ ，其中 M 是稀疏矩阵，因为每个网页向外跳转的链接数不多。 矩阵 a 的数据规模也只有 $1 * n$ ，可以考虑通过邻接矩阵来保存 M ，将 M^* 的内存占用从 $O(n^2)$ 降低为 $O(m+n)$ 。

1.2.4. 优化计算

$$PR(a) = [\beta(M + a^T(\frac{e}{n})) + (1-\beta)\frac{ee^T}{n}] * V$$

$$PR(a) = \beta(M + a^T(\frac{e}{n})) * V + (1-\beta)\frac{ee^T}{n} * V$$

$$令：PR_2 = (M + a^T(\frac{e}{n})) * V$$

$$则：PR(a) = PR_2 + (1-\beta)\frac{ee^T}{n} * V$$

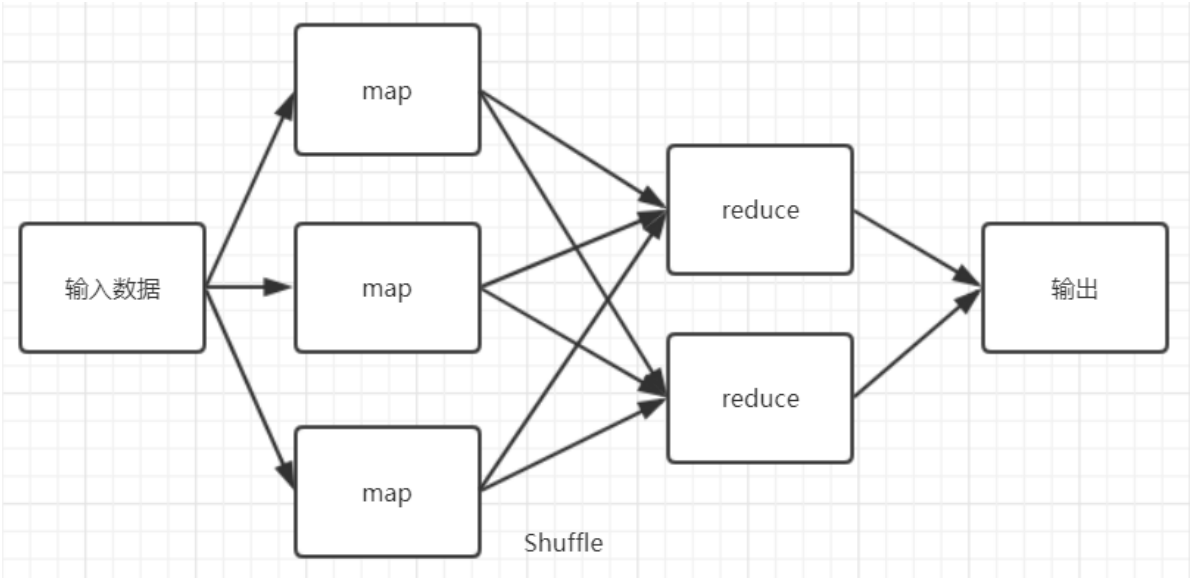
1.3. MapReduce

1.3.1. 介绍

Hadoop就是一个大数据开发所使用的分布式系统基础架构，由Apache软件基金会开发，主要用于海量数据的分布式处理。其核心是HDFS和MapReduce，二者分别用于大数据的存储和处理。HDFS即Hadoop分布式文件系统，可用于廉价计算机搭建的服务器集群，用于存储大量的数据，使得整个系统具备了高吞吐率、高容错性和高扩展性。MapReduce是面向大数据并行处理的计算模型，可以将大作业

拆分成小作业进行作业调度和容错管理，适用于数据的批量处理。MapReduce将复杂的分布式计算方式，高度抽象成了Map函数和Reduce函数，屏蔽了复杂的分布式系统底层实现，大大方便了分布式程序的开发。

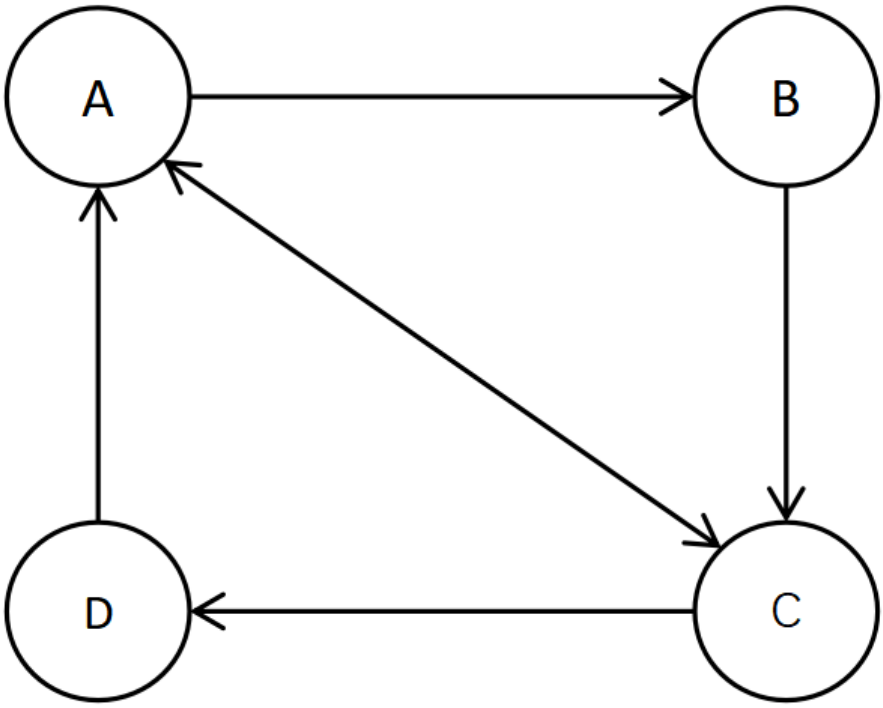
MapReduce的处理过程大致分为5个步骤，输入及数据分片，Map过程，Shuffle过程，Reduce过程，输出。



其中输入数据会被处理成形如<key1,value1>的若干分片，经过Map处理成形如<key2,value2>的分片。经过Shuffle过程的整合，会将相同的key2合并成为<key2,list(value2)>的分片移交给Reduce，最后Reduce会输出形如<key3,value3>的结果。

1.3.2. 整合

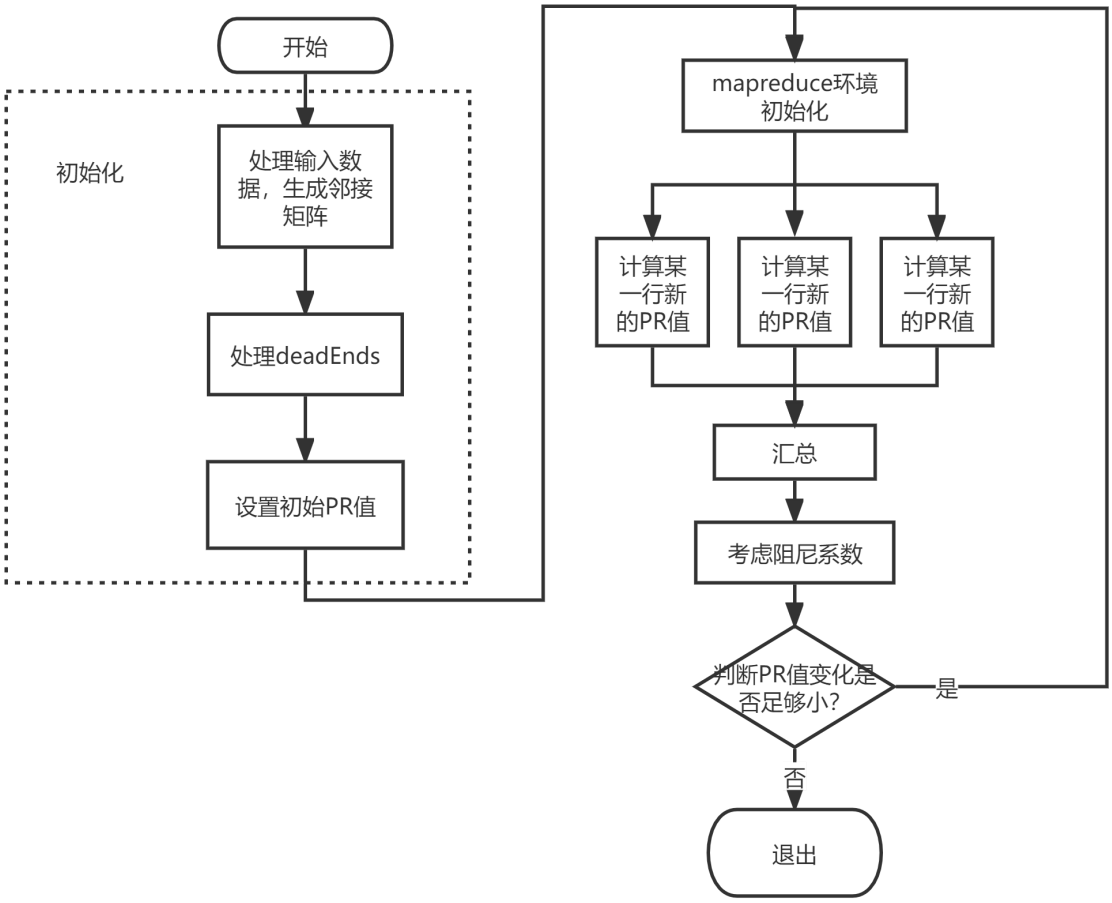
将矩阵的边 M 作为MapReduce的输入，这里采用了邻接矩阵的存储形式，每个Map单位单独负责一行的计算



过程名称	MapReduce
原始数据	反向邻接表,格式如下 终点编号 边1起点:边1权重 边2起点:边2权重 0 2:0.5 3:1 1 0:0.5 2 1:1 0:0.5 3 2:0.5
数据切分	0 2:0.5 3:1
map处理结果	假设V=[0.25,0.25,0.25,0.25]计算结果为0.25*0.5+0.25*1=0.375 输出<0,0.375>，<点编号，点的新权重>
suffer处理结果	由于每个map输出的键值都不同，所以结果同上
reduce处理结果	无需处理，将输入原样输出即可

2. 分布式实现

2.1. 流程图



2.2. 核心代码

邻接链表，采用了反向存储，其中存储了每条边的起点和权重

```
// 边的结构
public class Edge {
    int target;
    BigDecimal value;
    public Edge(int target){
        this.target = target;
    }
}

while((line = br.readLine()) != null){
    // 一行一行地处理...
    String[] num = line.split(" ");
    assert num.length == 2;
    int s = Integer.parseInt(num[0]);
    int t = Integer.parseInt(num[1]);
    // 边反向
    if(G[t] == null)G[t] = new LinkedList<>();
    G[t].add(new Edge(s));
    outDegree[s]++;
    hasOut[s] = true;
}
```

处理dead ends，对于没有出度的点，添加其对所有点的边，权重为 $\frac{1}{n}$ 。

```
// 获取deadends
String deadEndStr = context.getConfiguration().get("deadEnds");
String[] deadEnds = new String[0];
if(deadEndStr != null && deadEndStr.length() > 0)
    deadEnds = deadEndStr.split(" ");

BigDecimal[] row = new BigDecimal[n];
for(int i=0;i<n;i++){
    row[i] = BigDecimal.ZERO;
}
for(String deadEnd : deadEnds){
    row[Integer.parseInt(deadEnd)] = BigDecimal.ONE.divide(new BigDecimal(n));
}
```

考虑阻尼系数，经化简：$PR = \beta * PR_2 + (1-\beta)\frac{1}{n} * V$

```
// 考虑阻尼系数
for(int i=0;i<n;i++){
    newValues[i] = newValues[i].multiply(new BigDecimal(beta))
        .add(values[i].multiply(BigDecimal.ONE.divide(
            new BigDecimal(n)).multiply(new BigDecimal(1-beta))));
}
```

3. 分布式效果

运行效果截图

```
0.250000 0.250000 0.250000 0.250000  第0轮结果，差值0.500000
0.332812 0.110937 0.332812 0.110937  第1轮结果，差值0.221875
0.246143 0.147686 0.246143 0.147686  第2轮结果，差值0.098457
0.240297 0.109226 0.240297 0.109226  第3轮结果，差值0.043690
0.203570 0.106632 0.203570 0.106632  第4轮结果，差值0.019388
0.184970 0.090334 0.184970 0.090334  第5轮结果，差值0.008603
0.162252 0.082080 0.162252 0.082080  第6轮结果，差值0.003818
0.144845 0.071999 0.144845 0.071999  第7轮结果，差值0.001694
0.128174 0.064275 0.128174 0.064275  第8轮结果，差值0.000752
```