HW6 Report

Task 1: Word Count

实现方法

首先对于行数据使用 `re.split` 进行切分，分割符为非字母数字字符，使用 `flatMap` 函数将数据展平，然后使用 `map` 函数将数据转换为 `(word, 1)` 的形式，之后使用 `reduceByKey` 函数对相同的key进行求和，最后使用 `sortBy` 函数对结果进行排序。
在排序后，使用 `take` 函数取出前10个结果，然后将结果打印出来。

程序运行

```
$ python3 word_count.py /hw6_data/pg100.txt
```

实验结果

如下图：

```
2024210897@intro00:~/BigDataSystem_HW6$ python3 word_count.py /hw6_data/pg100.txt
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
('', 172754)
('the', 25743)
('I', 23849)
('and', 20184)
('to', 17424)
('of', 17301)
('a', 13974)
('you', 12901)
('my', 11411)
('in', 11384)
Total program time: 5.56 seconds
```

可以看到，结果与预期一致，运行时间为 `5.56s` 。

Task 2: PageRank

实现方法

首先对每行使用map函数，借助自定义的 parse_neighbors 将行字符串切割转换为 (src, dst) 的形式，然后使用 distinct 对相同的边进行去重，并使用 groupByKey 函数依据相同的 src 进行分组，形成【（src，（dest， dest2，……））……】的形式；为便于以后反复调用，将获得的 links 图缓存到内存中。
然后使用 mapValues 初始化每个节点的 rank 为 1.0 ，使用 for 循环迭代 numIterations 次，对每个节点的 rank 进行更新，更新方法为：对于每个节点，将其 rank 分给其目的节点，然后将每个节点的 rank 进行求和，最后结合阻尼因子 d 进行更新。具体计算方式为：首先将 links 图与 ranks 进行 join 操作，得到【（src，（（dest， dest2，……）， rank））……】的形式；之后使用自定义的 compute_contribs 函数计算每个节点对其目的节点的贡献值，并用 flatMap 展开外层列表，使 RDD 变成以 dest 为 key 、贡献值为 value 的形式，从而可以依据相同的 dest 进行分类；最后使用 reduceByKey 函数对相同的 src 进行求和，得到【（src， rank）……】的形式，并结合阻尼因子 d=0.8 用 mapValue 进行更新。
最后，以 rank 为 key 从大到小进行排序（相当于以 -rank 为 key 调用 takeOrdered ），取出前5个结果，并打印出来。

程序运行

```
$ python3 pagerank.py /hw6_data/small.txt
$ python3 pagerank.py /hw6_data/full.txt
```

实验结果

small.txt运行结果如下：

```
2024210897@intro00:~/BigDataSystem_HW6$ python3 page_rank.py /hw6_data/small.txt
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Top 5 nodes by PageRank:
Node ID: 71, PageRank: 0.0190909
Node ID: 32, PageRank: 0.0165023
Node ID: 31, PageRank: 0.0160401
Node ID: 33, PageRank: 0.0152616
Node ID: 81, PageRank: 0.0152199
Total program time: 15.25 seconds
```

运行用时 15.25s ， 评分前五的结果为：

```
Node ID: 71, PageRank: 0.0190909
Node ID: 32, PageRank: 0.0165023
Node ID: 31, PageRank: 0.0160401
Node ID: 33, PageRank: 0.0152616
Node ID: 81, PageRank: 0.0152199
```

full.txt运行结果如下：

```
2024210897@intro00:~/BigDataSystem_HW6$ python3 page_rank.py /hw6_data/full.txt
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Top 5 nodes by PageRank:
Node ID: 368, PageRank: 0.0020289
Node ID: 526, PageRank: 0.0019387
Node ID: 696, PageRank: 0.0019227
Node ID: 701, PageRank: 0.0018952
Node ID: 897, PageRank: 0.0018451
Total program time: 15.98 seconds
```

运行用时 15.98s ， 评分前五的结果为：

```
Node ID: 368, PageRank: 0.0020289
Node ID: 526, PageRank: 0.0019387
Node ID: 696, PageRank: 0.0019227
Node ID: 701, PageRank: 0.0018952
Node ID: 897, PageRank: 0.0018451
```

一些收获

实验发现，如果在初始化 ranks 时，不使用 mapValues 而是直接使用 map ， 则会导致出现 StackOverflowError ， 猜测原因是 map 函数会将 links 图的 key 一并引入后续分析链中，导致内存溢出。同时，将迭代轮数 numIterations 设置为 80 时，也可能（但不是一定会）出现 StackOverflowError ， 还没想明白为什么会出现这样的随机性。

目前想到的解决方法包括：1. 调大内存；2. 使用 checkpoint 存储中间结果，从而中断分析链条，减少内存占用。

同时发现，程序处理 full.txt 与 small.txt 的运行时间并没有太大的差别。猜测是数据量增大时，服务器调用的资源也会增多，并行的进程、线程数变多，从而导致运行时间并没有太大的差别。