目录

1 实验介绍

1.1 实验内容

1.1.1 实验八

1. 使用 Hadoop 的示例程序计算圆周率 π, 调整 maps 和 samples 参数, 观察运行时间与计算结果精度变化.

2. 计算圆周率 π, 使其有效精度达到小数点后 5 位.

1.1.2 实验九

1. 编写 mapper.py 和 reducer.py, 计算一篇英文文章中不同字母开头的单词的平均长度.

2. 编写 mapper.py 和 reducer.py, 实现 PageRank 算法. 每一次 map 和 reduce 操作完成 PageRank 的一次迭代计算.

1.2 实验环境

操作系统:Ubun 18.04 LTS Python 版本:3.7 Hadoop 版本:2.2.0

1.3 所需技能

实验八主要在于 Hadoop 环境的准备, 在正确完成了环境配置之后, 只需对 Hadoop 的基本工作方式有所了解即可.

实验九进一步深入 Hadoop, 需要掌握其文件系统的操作以及 map-reduce 工作原理. 同时, 还需要了解基本的 PageRank 算法原理.

表 1: 使用 Hadoop 计算圆周率

Number of Maps	Number of Samples	Time(s)	$\acute{\pi}$
2	10	20.214	3.80000000000000000000
5	10	22.430	3.28000000000000000000
10	10	27.392	3.20000000000000000000
2	100	17.596	3.12000000000000000000
10	100	40.041	3.14800000000000000000
100	1000000	310.256	3.14159256000000000000

2 实验过程

2.1 估算圆周率

Hadoop 所提供的示例程序中, 使用 Quasi-Monte Carlo 方法计算圆周率. 这种方法在 $[0,1] \times [0,1]$ 的正方形区域中随机产生许多点, 然后通过计算这些点与原点的距离, 判断其是否在单位圆中. 在大量统计的基础上, 便可估算出圆周率的值.

对应 Hadoop 计算圆周率程序运行的参数, 在 map 操作中, 进行 Number of Maps 次模拟实验, 每次实验中随机产生 Number of Samples 个点, 最后在 reduce 操作中, 汇总计算所有点. 那么, 根据直觉,Maps 数和 Samples 数的乘积越大, 总点数越多, 最后的估算结果精度应当越高. 至于运行时间, 当 Samples 数较小时, 每次 map 操作的开销主要在于 map 本身, 而非其中的模拟计算, 故运行时间应当主要取决于 Maps 数.

使用不同的参数运行该程序, 每组参数重复三次, 结果取平均值, 见表1. 观察结果, 基本与上述分析相符. 同时在表1的末行, 给出了一种使得 $\acute{\pi}$ 的估算精度达到小数点后 5 位的方案.

2.2 统计平均单词长度

2.2.1 Map

在 mapper.py 中, 通过 `sys.stdin` 读取由 Hadoop 传递而来的行文本, 对其进行处理:

首先, 去除文本中除了 26 个英文字母外的其他字符, 如标点符号及数字, 将其用空格替代以便进行单词划分. 然后, 对于划分出的每个单词, 计算其长度, 并输出 map 的结果 (字母转为小写形式), 形如: `<letter> <length>` . 代码如下:

```
#!/usr/bin/env python3
import sys
from string import ascii_letters

valid_chars = ascii_letters
for line in sys.stdin:
    stripped_line = "".join([x if x in valid_chars else " " for x in line])
    words_list = stripped_line.split()
    for word in words_list:
```

```
        print("{letter}\t{length}".format(
            letter=word[0].lower(), length=len(word)))
```

2.2.2 Reduce

在 reducer.py 中, 接受 Hadoop 通过 `sys.stdin` 传递而来的 Map 后的临时结果, 对其进行合并处理. 对于统计单词平均长度, 只需将记录同一字母开头的单词个数以及总长度, 最后两者相除, 将结果通过 `print` 函数输出到 `sys.stdin` ,Hadoop 会将其收集并存入结果文件.

```
#!/usr/bin/env python3
import sys

current_letter = None
current_count = None
current_sum = None

for line in sys.stdin:
    letter, length = line.split()
    length = int(length)
    if letter == current_letter:
        current_sum += length
        current_count += 1
    else:
        if current_letter != None:
            print("{letter}\t{average}".format(
                letter=current_letter, average=current_sum/current_count))
        current_letter, current_count, current_sum = letter, 1, length
if current_letter:
    print("{letter}\t{average}".format(
        letter=current_letter, average=current_sum/current_count))
```

注意在以上的代码中, 利用了 Hadopp 会将 Map 结果按关键词排序再传递给 Reduce 的特点, 即保证了在遇到一个新的开头字母时, 之前的字母一定已经处理完成.

2.2.3 运行结果

以 pg5000.txt 作为输入, 运行得到结果如下:

```
a   3.3844186012320447
b   4.571903651903652
c   6.565201857806359
d   5.653363797918253
e   6.0105919003115265
```

```
f   5.084103179364127
g   5.517767833640431
h   3.902831155521394
i   3.141337386018237
j   4.685469475187433
k   5.132145052243393
l   5.144624981135872
m   4.992715998655261
n   4.329627695292099
o   2.82933814830957
p   6.584160116823433
q   6.113682777399592
r   6.426794468427599
s   5.148828229301456
t   3.636275394148561
u   5.010820244328098
v   5.809677419354839
w   4.46050039135246
x   3.3380281690140845
y   3.664545888341021
z   4.636690647482014
```

2.3 实现 PageRank 算法

2.3.1 基本原理

根据 PageRank 的公式, 一个页面的 PageRank 值由两部分组成: 直接被随机选中的概率, 顺着链接被访问到的概率:

$$PR(u) = \frac{1-d}{N} + d\sum_{v\in B_u}\frac{PR(v)}{N_v}$$

式中 $d \in (0,1)$ 为 damping factor,N 为总网页数,B_u 为所有链接指向 u 的网页的集合,N_v 为网页 v 所包含的链接总数.

可以证明,PageRank 是收敛的. 那么, 可以避免直接求其解析解, 而是经过多次迭代, 逐渐接近其数值解. 而每一次的迭代, 可通过一次 Map-Reduce 循环来实现.

2.3.2 Map

在 mapper.py 中, 通过 `sys.stdin` 读取输入文件的行文本. 但需要格外注意的是,Hadoop 可能会将输入的文件分割为多个部分, 分别交由不同的 mapper 同时处理, 也就是说, 在 mapper.py 中, 必须假定只能获取到输入文件的一部分, 否则运行结果会出现错误.

因此, 需要在 mapper.py 中设定好具体的网页总数 N, 而不能在运行时对输入进行统计. 此外, 为了方便在 Reduce 时得到网页所指向的其他网页, 还需要将网页间的链接关系作为 Map 结果输出. 如下:

```python
#!/usr/bin/env python3
import sys

damping_factor = 0.85
page_num = 4   # or some other number

for line in sys.stdin:
    line = line.split()
    try:
        page_id, initial_prob = int(line[0]), float(line[1])
    except:
        continue
    print("{page} plus {plus}".format(page=page_id,
    ↪  plus=(1-damping_factor)/page_num))
    if len(line) <= 2:  # for pages that link to no other pages
        prob_per_page = initial_prob/(page_num-1) * damping_factor
        for page in range(1, page_num+1):
            if page != page_id:
                print("{page} plus {plus}".format(page=page,
                ↪  plus=prob_per_page))
    else:
        links_list = line[2:]
        print("{page} links".format(page=page_id), *links_list)
        # print all the links of the current page
        prob_per_link = initial_prob/(len(links_list)) * damping_factor
        for link in links_list:
            print("{page} plus {plus}".format(page=link, plus=prob_per_link))
```

对以上程序段稍做说明: 对于读入的一行文本, 譬如`1 0.25 2 3 4`, 首先从中提取出 1 和 0.25, 分别为当前网页编号和初始概率, 根据 PageRank 的公式, 首先输出`1 plus 0.0375`. 然后判断该网页是否有指向其他网页的链接, 若无, 则认为其链接了所有网页, 故将剩余概率等分给所有其他网页; 若有, 则先输出其链向的网页`1 links 2 3 4`, 再将其概率等分给这些网页. 故对于该行文本, 所有输出为:

```
1 plus 0.037500000000000006
1 links 2 3 4
2 plus 0.07083333333333333
3 plus 0.07083333333333333
4 plus 0.07083333333333333
```

2.3.3 Reduce

由于在 Map 操作中, 已经充分为 Reduce 提供了便利, 故 reducer.py 只需要将 Map 的输出合并, 算出网页对应的总概率以及链接关系:

```python
#!/usr/bin/env python3
import sys

current_page = None
current_prob = None
current_links = None

for line in sys.stdin:
    line = line.split()
    page, operation_type = int(line[0]), line[1]
    if page != current_page:
        if current_page:
            print(current_page, current_prob, *current_links)
        current_page, current_prob, current_links = page, 0, None
    if operation_type == "links":
        current_links = line[2:]  # note here we don't convert page_id to int
    elif operation_type == "plus":
        current_prob += float(line[2])
if current_page:
    print(current_page, current_prob, *current_links)
```

2.3.4 Bash 脚本

为了得到 PageRank 的收敛值, 需要多次迭代, 即多次 Map-Reduce 循环. 至于迭代的终止条件, 最佳的做法是判断迭代后变化量是否足够小, 而在这次实验中, 我采取了简易的做法, 设定一定的迭代次数后终止.

编写的 pagerank.sh 如下:

```bash
#/bin/bash

hadoop_command='hadoop jar
↪  /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar
↪  -files mapper.py,reducer.py -mapper mapper.py -reducer reducer.py'
mv='hadoop fs -mv '
rm='hadoop fs -rm -r'
cp2local='hadoop fs -copyToLocal '
input='tempinput'
```

```
for (( i = 1; i < $1+1; i++ )); do
    echo "Page Rank Iteration $i"
    output="pagerank_tempoutput_$i"
    eval "$hadoop_command -input $input -output $output -jobconf
    ↪  mapred.job.name=\"Page Rank Iteration $i\""
    input=$output
    eval "$rm $input/_SUCCESS"
done

mkdir ~/pagerank_result
eval "$cp2local $output/* ~/pagerank_result"
```

2.3.5 运行结果

输入为:

```
1 0.25 2 3 4
2 0.25 3 4
3 0.25 4
4 0.25 2
```

设置迭代次数为 3, 运行:

```
./pagerank.sh 3
```

则每次迭代后, 结果如下:

```
1 0.037500000000000006 2 3 4
2 0.3208333333333333 3 4
3 0.21458333333333335 4
4 0.42708333333333337 2

1 0.037500000000000006 2 3 4
2 0.4111458333333334 3 4
3 0.18447916666666664 4
4 0.36687499999999995 2

1 0.037500000000000006 2 3 4
2 0.35996875 3 4
3 0.22286197916666667 4
4 0.3796692708333333 2
```

3 总结与分析

编写在 Hadoop 中运行的程序不同于编写普通的程序.

这一点我一开始并没有充分意识到, 因而在实验过程中遇到了一些挫折, 调试许久, 通过反复对比输入输出以及输出中间过程量, 最终才发现 BUG 的根源在于我没有完全理解 Hadoop 的 MapRuduce 工作流程. 譬如, 一开始在 PageRank 的 mapper.py 中, 我通过统计读入的行数来确定总网页数 N, 并以此输出网页直接被随机选中的概率 $(1-d)/N$, 但这种做法在 Hadoop 中是行不通的, 因为 Hadoop 会将输入文件分割成多个片段, 交由不同的 mapper 并行处理. 所以, 必须手动指定 N 的值, 以保证结果的正确.

总而言之, 通过这两次实验, 我对 Hadoop 有了基本的认识, 收获颇丰.