

**课 程 实 验 报 告**

**课程名称： 大数据分析**

**专业班级： 大数据2101班**

**学 号： U202115652**

**姓 名： 李嘉鹏**

**指导教师： 王蔚**

**报告日期： 2023年5月11日**

**计算机科学与技术学院**

**目录**

[实验一 wordCount算法及其实现 1](#_Toc135394162)

[**1.1实验目的** 1](#_Toc135394163)

[**1.2 实验内容** 1](#_Toc135394164)

[**1.3 实验过程** 1](#_Toc135394165)

[1.3.1 编程思路 1](#_Toc135394166)

[1.3.2 函数模块定义 3](#_Toc135394167)

[1.3.3 数据结构与存储形式 4](#_Toc135394168)

[1.3.4 遇到的问题及解决方式 4](#_Toc135394169)

[1.3.5 实验测试与结果分析 4](#_Toc135394170)

[**1.4 实验总结** 8](#_Toc135394171)

[**1.5 参考文献** 8](#_Toc135394172)

# 实验一 wordCount算法及其实现

## **1.1实验目的**

1、理解map-reduce算法思想与流程；

2、应用map-reduce思想解决wordCount问题；

3、（可选）掌握并应用combine与shuffle过程。

## **1.2 实验内容**

提供9个预处理过的源文件（source01-09）模拟9个分布式节点，每个源文件中包含一百万个由英文、数字和字符（不包括逗号）构成的单词，单词由逗号与换行符分割。

要求应用map-reduce思想，模拟9个map节点与3个reduce节点实现wordCount功能，输出对应的map文件和最终的reduce结果文件。由于源文件较大，要求使用多线程来模拟分布式节点。

学有余力的同学可以在map-reduce的基础上添加combine与shuffle过程，并可以计算线程运行时间来考察这些过程对算法整体的影响。

提示：实现shuffle过程时应保证每个reduce节点的工作量尽量相当，来减少整体运行时间。

## **1.3 实验过程**

### 1.3.1 编程思路

整体上，本次实验主要分为四个模块：map、combine、shuffle、reduce。

对于第一个模块map，编写了mapper函数实现了对源文件source01~09的键值对的分解，通过循环方式依次启动九个map进程。我利用time.perf\_counter()函数记录第一个进程开始的时间，并在九个进程结束时再分布记录一次系统时间，两个时间相减就能得到运行到某一个进程时所耗费的总时间。最后，依据字典序将map结果输出。map的核心代码如下。

|  |
| --- |
| **with** write **as** f:  **for** words **in** lines:  **for** word **in** words:  f.write(**"{},{}\n"**.format(word, 1)) *#每个词出现一次为一个键值对* |

Python的多线程功能十分强大，可以高效处理大量任务，因此本次实验中，我全部采用了多线程的方式。创建进程的代码如下。

|  |
| --- |
| map\_threads=[] **for** i **in** range(9):  *#循环9次，依次启动9个mapper进程* t=threading.Thread(target=mapper(**f"source0{**i+1**}"**,**f"map{**i+1**}"**))  map\_threads.append(t)  t.start() start = time.perf\_counter()count=1 *#循环计数器* **for** t **in** map\_threads:  t.join()  print(**f"map\_t{**count**}: %s s"** %(time.perf\_counter() - start)) *# 计算多线程运行时间* count+=1 |

对于第二个模块combine，编写了combiner函数，首先创建一个空字典用以记录源文件中的单词和相应出现的次数，接下来对map得到的文件进行遍历。同样地，我通过循环依次进行九个combine进程，并分别记录消耗的时间。combine的核心代码如下。

|  |
| --- |
| **for** line **in** read:  line = line.strip()  word, count = line.split(**','**, 1)  **try**:  count = int(count)  **except** ValueError:  **continue  if** word **in** count\_dict.keys(): *#已经查找到这个词，count+1* count\_dict[word] = count\_dict[word] + count  **else**: *#第一次找到这个词，count=1* count\_dict[word] = count |

对于第三个模块shuffle，编写了shuffle函数，利用哈希函数对上一步得到的combine结果进行划分。具体而言，我调用了库函数hash(word)得到每一个单词的hash值，由于最终有3个reduce节点，我在shuffle阶段需要将全部单词分配到三个“桶”中。因此我将hash值对3取余，使最终的单词分布趋于平衡（即每个reduce节点的负载近似一致）。shuffle的核心代码如下。

|  |
| --- |
| **for** line **in** file:  line = line.strip()  word, count = line.split(**','**, 1)  **if** (word[0] >= **'a' and** word[0]<=**'z' and** hash(word)%3==0) **or** (word[0] >= **'A' and** word[0]<=**'Z' and** hash(word)%3==0): *#利用哈希函数对word进行shuffle* write1.write(**"{},{}\n"**.format(word, count))  **elif** (word[0] >= **'a' and** word[0]<=**'z' and** hash(word)%3==1) **or** (word[0] >= **'A' and** word[0]<=**'Z' and** hash(word)%3==0):  write2.write(**"{},{}\n"**.format(word, count))  **else**:  write3.write(**"{},{}\n"**.format(word, count))  *#没有直接使用特定首字母mod3的方式（如下），避免了负载不均衡性  # if (word[0] >= 'a' and word[0]<='z' and ord(word[0])%3==0) or (word[0] >= 'A' and word[0]<='Z' and ord(word[0])%3==0)  # elif (word[0] >= 'a' and word[0]<='z' and ord(word[0])%3==1) or (word[0] >= 'A' and word[0]<='Z' and ord(word[0])%3==1)* |

最终，对于模块reduce，本质上和combine函数一致，但仅需要启动三个进程。编写完上述四个模块后，依次运行map、combine、shuffle、reduce，即可得到运行结果文件。

算法的整体流程图如图1-1所示。

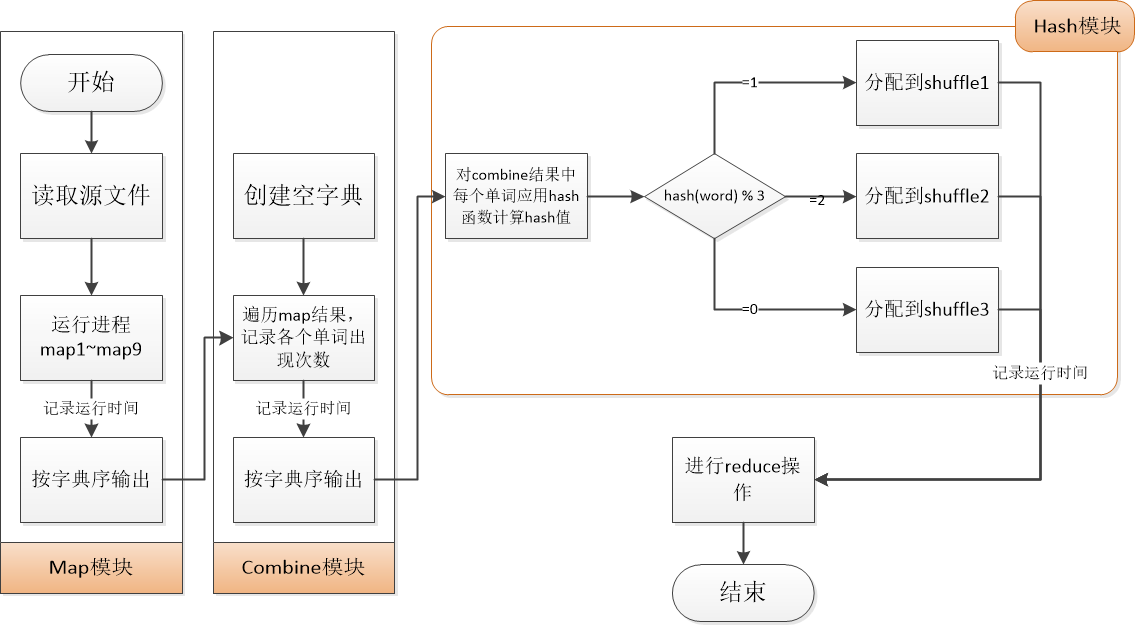


图1-1：算法流程图

### 1.3.2 函数模块定义

**（1）mapper**

①函数参数：readfile, writefile

②函数功能：完成map任务，逐个统计出现的单词。

**（2）combiner**

①函数参数：inputfile, outputfile

②函数功能：完成combine任务，并按照单词的字典序排序输出。

**（3）shuffle**

①函数参数：readfile

②函数功能：完成shuffle任务，依据哈希值将单词分到三个桶里。

**（4）reduce**

①函数参数：readfile, writefile

②函数功能：对shuffle后的结果进行处理。

### 1.3.3 数据结构与存储形式

**（1）map\_threads、combine\_threads、shuffle\_threads**

①类型：列表

②作用：储存map、combine、shuffle三个模块的多线程。

**（2）count\_dict**

①类型：字典

②作用：存放每个单词及其出现的次数。

### 1.3.4 遇到的问题及解决方式

本次实验最大的问题就是在写循环开始多个进程时不太会处理。在网上查阅资料发现，Python的进程也可以被视为数组元素，由此可采用xxx\_threads=[]来创建一个空的进程列表，每次循环中通过下面两条语句在进程列表中添加新的进程即可。

t=threading.Thread(target=mapper(f"source0{i+1}",f"xxx{i+1}"))

xxx\_threads.append(t)

这样避免了复制多句重复语句并修改细节（例如循环次数）的繁琐，在进程数量变化时也能很方便地进行调节。

### 1.3.5 实验测试与结果分析

依次运行map、combine、shuffle、reduce，得到运行结果文件map1~9、combine1~9、shuffle1~3、reduce1~3，如图1-2所示。

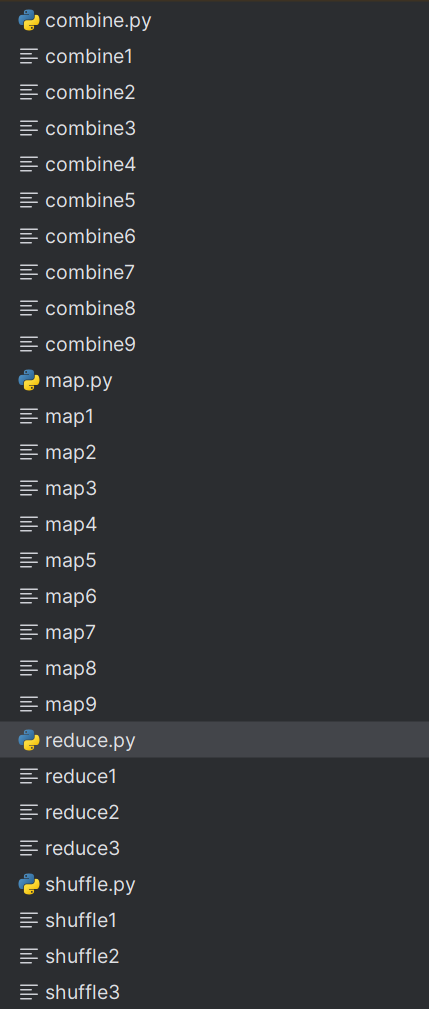
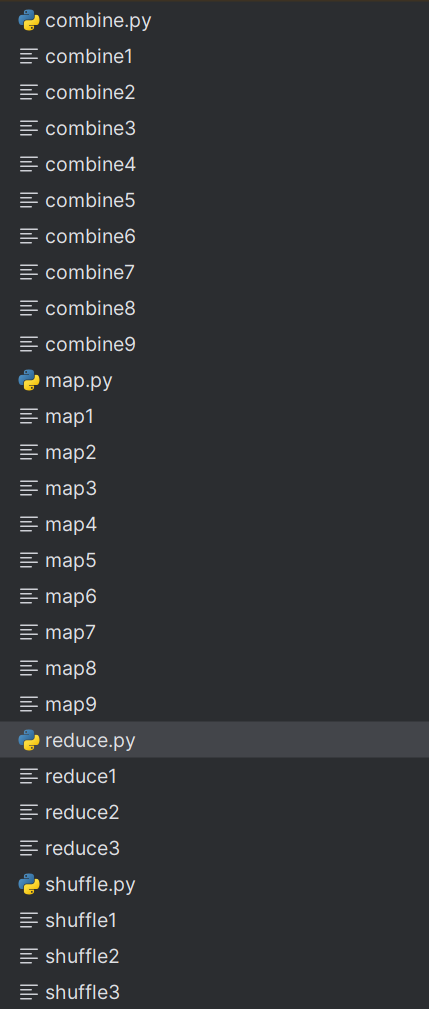
 

图1-2：结果文件列表

对于四个模块map、combine、shuffle、reduce，分别统计了相应的运行时间，如图1-3~图1-6所示。对其分别进行逐差即可得到每个进程的运行时间。

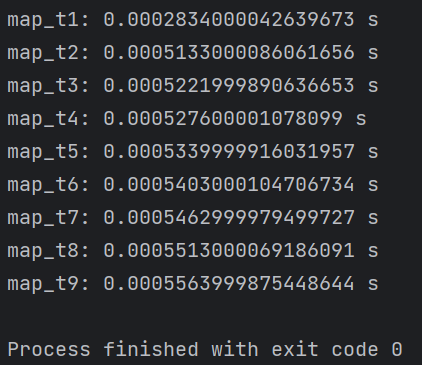
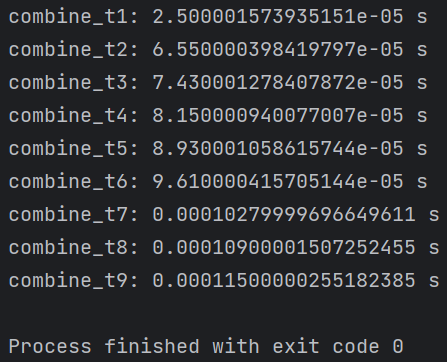
 

图1-3：map模块九个进程花费时间 图1-4：combine模块九个进程花费时间

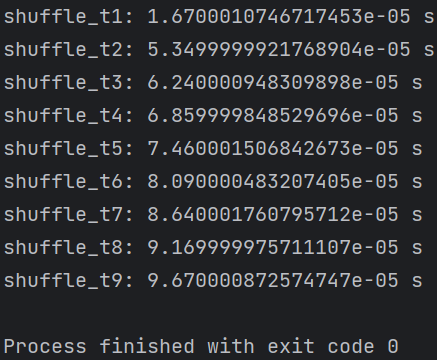


图1-5：shuffle模块九个进程花费时间

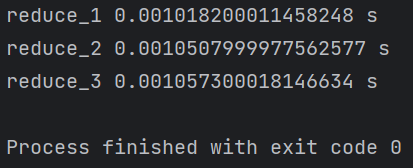


图1-6：reduce模块三个进程花费时间

从中可以看出，combine和shuffle两个模块运行速度很快，不会对算法的整体实现耗时产生影响。

接下来，对结果文件进行分析，在map1~9中，可以看到源文件的文本已经被按照“单词 空格 单词”的格式拆分开，并将每个单词的每一次出现进行了记录，如图1-7所示。

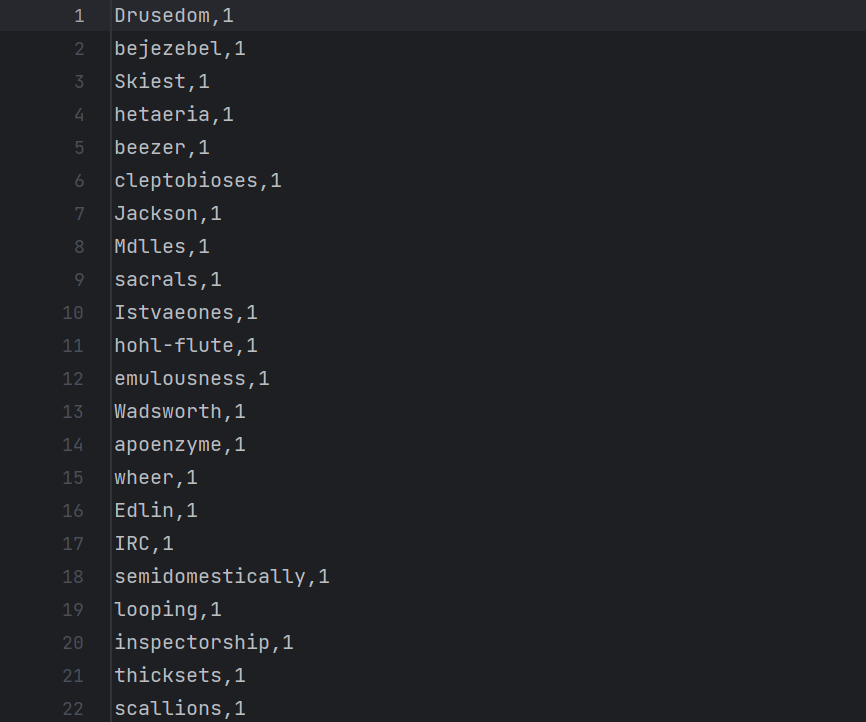
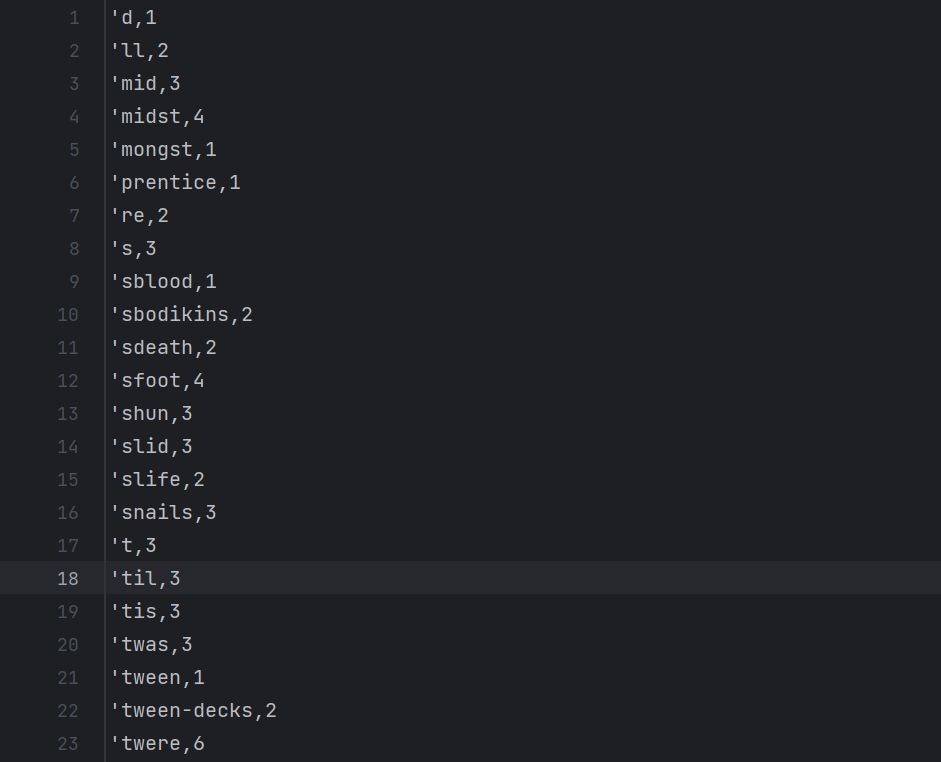
 

图1-7：map结果文件 图1-8：combine结果文件

在combine1~9中，可以看到已经将上述map文件进行了汇总，按照单词-出现次数的格式输出到文件中，并按字母序排序，如上图1-8所示。

在shuffle1~3中，根据前面用hash值模3的思路，可以在三个文件中找到相似的单词（即不是简单地只包含某几个首字母，或是只出现大写或小写单词等），且三个文件的单词数量相近，如图1-9、图1-10所示。可见，这个方法在很大程度上避免了工作负载不均衡的问题。

最后，reduce文件将三个shuffle结果文件进行合并，如图1-11所示。

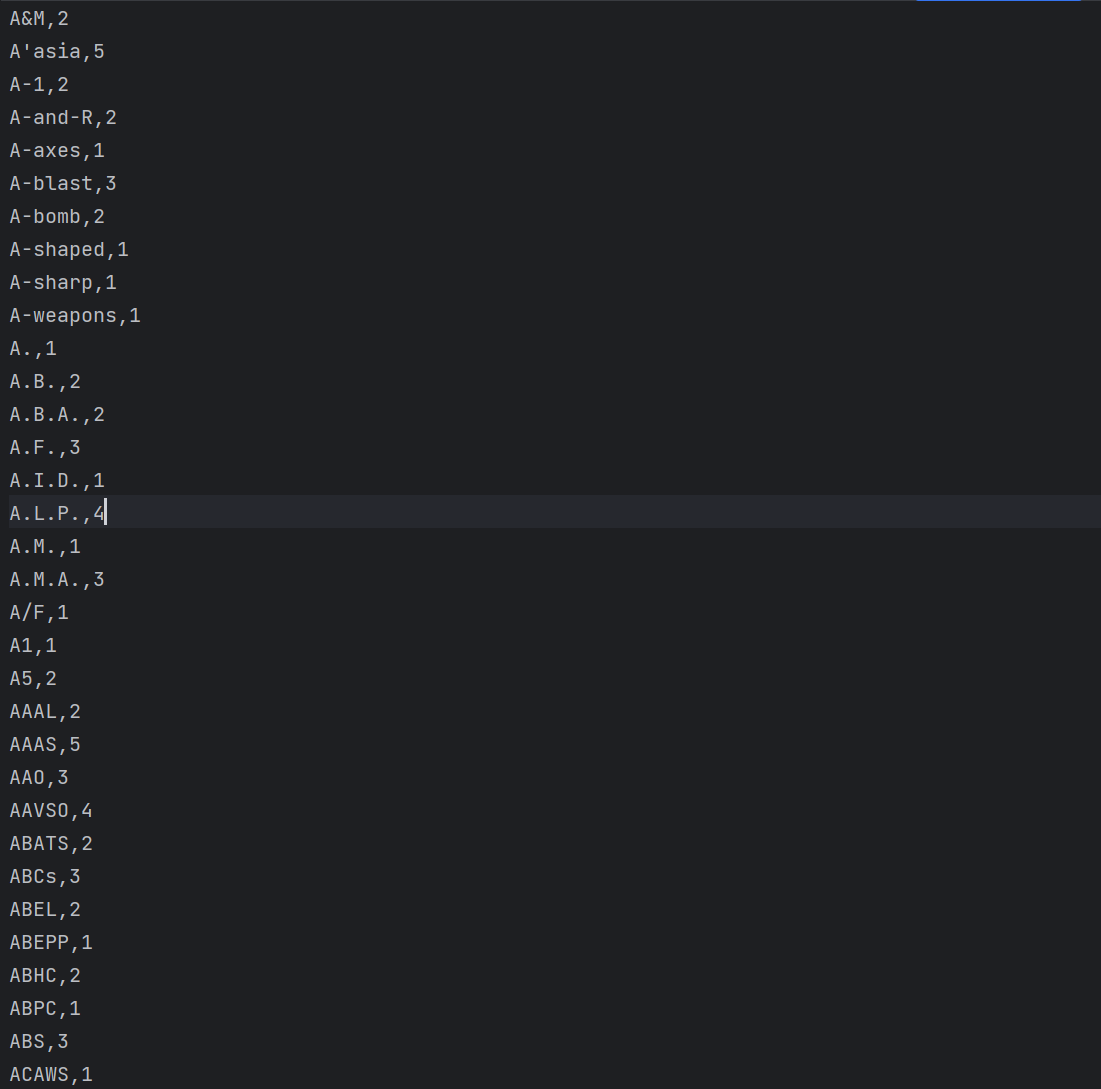


图1-9：shuffle结果文件

图1-10：shuffle1、shuffle2、shuffle3的行数

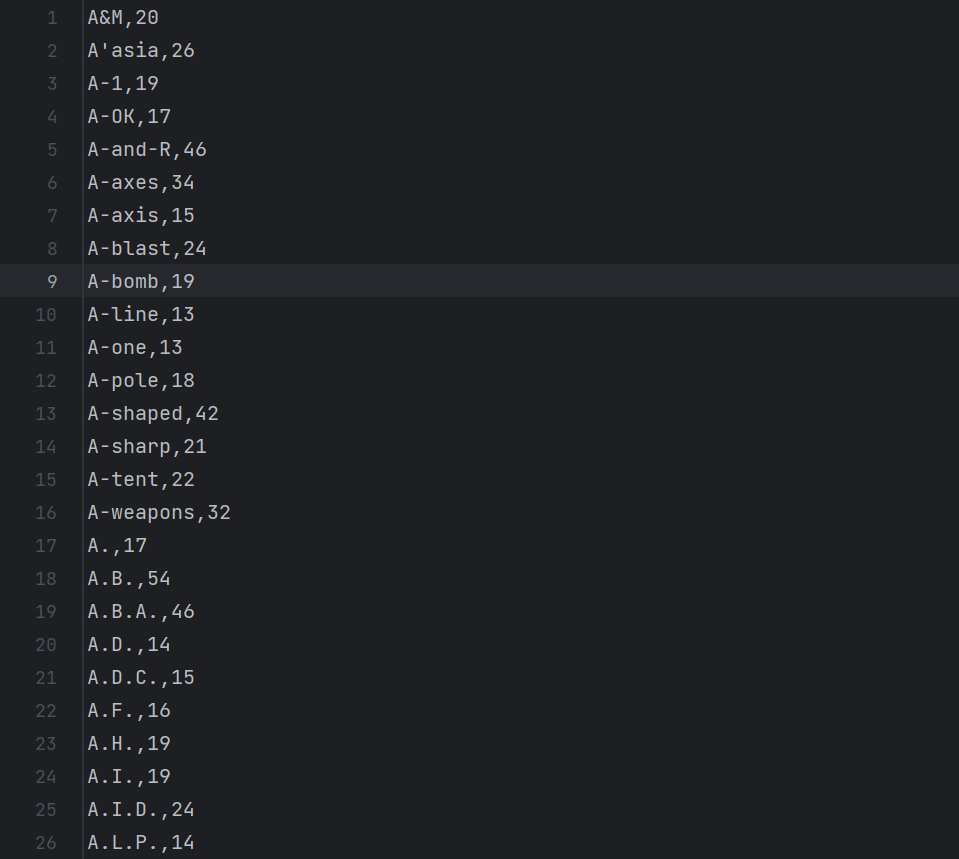


图1-11：reduce结果文件

## **1.4 实验总结**

本次实验是对大数据分析课上MapReduce理论的实现，让我通过具体的数据集深入理解了map、combine、shuffle、reduce这四个模块的实现方法以及它们各自需要注意的细节。同时，之前的很多实验都是基于C++完成的，这次使用Python语言实现让我感受到了Python语言的灵活性以及其读入、写入文件的高效性。

MapReduce是一种面向大规模数据集的并行计算算法。通过对本次实验中九个大文件的操作，我进一步理解了map、combine、reduce的基本思想，并通过尝试得到了一种表现较好的shuffle方法（利用哈希函数hash(word)完成）。

## **1.5 参考文献**

[1] MapReduce详解 <https://blog.csdn.net/woaini886353/article/details/124687084>

[2] Map函数和Reduce函数 <https://zhuanlan.zhihu.com/p/172532077>

[3] MapReduce编程实例：单词计数 <https://zhuanlan.zhihu.com/p/138993195>