

**数据仓库课程作业**

|  |
| --- |
| 数据存储练习 |

指导教师*:* **朱宏明**

刘文朔 (1851008)

谭梓煊 (1853434)

Data Warehouse

Tongji University

School of Software Engineering

# 分布式集群实践

搭建Hadoop环境（小型分布式集群或伪分布式集群），在Hadoop环境中编写词频计算程序，计算作业1中下载的用户评论文件中不同的单词数量和每个单词出现的频次。（请考虑什么是不同的单词，例如单复数等均为同一个单词，不同时态也为同一个单词）

## 基本思路

分析此次作业需求，我们得到了如下思路：

1. 首先对原始评论文件进行预处理：去掉不属于单词的非打印字符、去掉语段中的标点符号等。这一步使用Python和C编写脚本实现。
2. 搭建Hadoop集群，编写WordCount程序，对预处理过的程序进行词频统计，得到一张词频表。这一步使用Java语言实现。
3. 对词频表中的词条进行进一步的处理：包括合并同一单词的单复数、不同时态；将误判为一个的多个单词拆分、去除部分非单词等。这一步使用Python语言，结合一些自然语言处理的库完成。

## 实现步骤

### 数据清洗

原始评论数据集的格式见作业一报告。

1. 该文件使用UTF-8编码不能正常读入，因此使用iso-8859-1，依次从文件中读出单个字符进行判断。我们的目标是保留所有的单词并尽量去除所有不是单词的片段；
2. 在本次作业中，我们仅需要使用评论的内容部分，因此需要对原始数据进行筛选，仅保留summary和text两部分内容；
3. 原始数据中包含了“review”, “user”等前缀标识符，这些标识符仅供标记内容类型，实际意义不大，因此在这一步去除；
4. 观察文件后发现，该文件中影响划分的乱码主要有：非字符、html标签、&nbsp等html字符转义；
5. 原始数据集文本中包含若干<br />标签，根据分析可能是原始数据采集过程中出现了一些问题。我们在这一步对<br />标签进行了过滤；
6. 原始文本中包含众多不可打印的非法字符，如果不在此步进行过滤，将会直接影响Hadoop词频统计的结果。因此我们对非法字符进行了过滤。
7. 因此过滤时：遇到<，判断是否为html标签，为则一起去除；遇到&时留下，之后使用Python脚本过滤。其余的非字符的char则单独去除，但’符号例外，因为英语中存在It’s这类简写的表达方式。（以上的去除意为改成空格）
8. 本程序使用C语言写成，运行速度较快。
9. 运行结束后，文本中除了会保存纯粹的英文单词外，还会保存含数字的字段。

行筛选脚本（Python）：

1. count = 0
2. lastKeep = 0
3. with **open**("movies.txt", encoding="iso-8859-1") as f,
4. **open**("movies\_text.txt", "w", encoding="iso-8859-1") as fo:
5. for (index, line) in **enumerate**(f):
6. line = line.replace("<br />", "").strip()
7. if line == "": continue
8. elif line.startswith("review/summary:"):
9. value = line[16:].strip()
10. fo.write(value + "\n")
11. count += 1
12. lastKeep = 1
13. elif line.startswith("review/text:"):
14. value = line[13:].strip()
15. fo.write(value + "\n")
16. count += 1
17. lastKeep = 1
18. elif line.startswith(("product/", "review/")):
19. lastKeep = 0
20. else:
21. if lastKeep == 1:
22. fo.write(line + "\n")
23. count += 1
24. if index % 100000 == 0:
25. **print**(index)
26. **print**("Fin.")
27. **print**("Number of lines:", count)

字符过滤脚本（C）：

由于源文件体积较大，使用Python处理的速度缓慢，所以这里使用C语言实现。

1. #define **\_FILE\_OFFSET\_BITS** 64
2. #include <stdio.h>
3. #include <ctype.h>
4. #include <time.h>
5. #define **CHUNK\_SIZE** 1024
6. int **main**()
7. {
8. **FILE** \*fp = **fopen**("movies\_text.txt", "rb");
9. **FILE** \*fp1 = **fopen**("movies\_text\_out.txt", "w");
10. long long t = **time**(**NULL**);
11. long long sum = 0;
12. char buf[**CHUNK\_SIZE**], \*p;
13. while (!**feof**(fp)) {
14. sum += **fread**(buf, 1, **CHUNK\_SIZE**, fp);
15. p = buf;
16. for (int i = 0; i < **CHUNK\_SIZE**; i++) {
17. if (**ispunct**(buf[i]) && buf[i] != '\'' && buf[i] != '&' || !**isprint**(buf[i]) && buf[i] != '\n') {
18. if (\*(p - 1) != ' ') \*p++ = ' ';
19. } else {
20. \*p++ = buf[i];
21. }
22. }
23. **fwrite**(buf, 1, p - buf, fp1);
24. if (sum % (1024 \* 1024 \* 50) == 0) { **printf**("%lld bytes\n", sum); }
25. }
26. **printf**("%lld bytes in %llds\n", sum, **time**(**NULL**) - t);
27. **fclose**(fp);
28. **fclose**(fp1);
29. return 0;
30. }

预处理后的文件大概有7G左右。

### Hadoop词频统计

#### Hadoop集群搭建

因为Hadoop集群的部署设置较为复杂，我们使用docker实现hadoop的自动化部署。

我们使用了big-data-europe提供的[hadoop-docker](https://github.com/big-data-europe/docker-hadoop)镜像，在本地机器上运行的Ubuntu系统下搭建了具有5个节点的伪分布式集群。5个节点的具体职能如下图所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 容器ID | Name Node | Data Node | Resource Manager | Node Manager | History  Server |
| 容器1 | √ |  |  |  |  |
| 容器2 |  | √ |  |  |  |
| 容器3 |  |  | √ |  |  |
| 容器4 |  |  |  | √ |  |
| 容器5 |  |  |  |  | √ |

Tab. 1.1 Hadoop集群部署情况

#### 词频统计程序设计

词频统计（WordCount）程序的设计参考了Hadoop提供的example代码。首先定义Map类和Reduce类，定义map和reduce的操作。然后在main函数中创建一个Job，设置Job的输入输出和Map、Reduce类，等待Job完成即可。

Mapper操作：

1. public static class **Map** extends **Mapper**<LongWritable, Text, Text, IntWritable> {
2. private final static IntWritable one = new **IntWritable**(1);
3. private Text word = new **Text**();
4. public void **map**(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
5. String line = value.**toString**();
6. StringTokenizer tokenizer = new **StringTokenizer**(line);
7. while (tokenizer.**hasMoreTokens**()) {
8. word.**set**(tokenizer.**nextToken**());
9. context.**write**(word, one);
10. }
11. }
12. }

Reducer操作：

1. public static class **Reduce** extends **Reducer**<Text, IntWritable, Text, IntWritable> {
2. public void **reduce**(Text key, Iterable<IntWritable> values, Context context)
3. throws IOException, InterruptedException {
4. int sum = 0;
5. for (IntWritable val : values) {
6. sum += val.**get**();
7. }
8. context.**write**(key, new **IntWritable**(sum));
9. }
10. }

main函数：

1. public static void **main**(String[] args) throws Exception {
2. Configuration conf = new **Configuration**();
3. Job job = new **Job**(conf, "wordcount");
4. job.**setOutputKeyClass**(Text.class);
5. job.**setOutputValueClass**(IntWritable.class);
6. job.**setMapperClass**(Map.class);
7. job.**setReducerClass**(Reduce.class);
8. job.**setInputFormatClass**(TextInputFormat.class);
9. job.**setOutputFormatClass**(TextOutputFormat.class);
10. FileInputFormat.**addInputPath**(job, new **Path**(args[0]));
11. FileOutputFormat.**setOutputPath**(job, new **Path**(args[1]));
12. job.**waitForCompletion**(true);
13. }

处理完成之后将得到一张“单词-数量”表，其大小在10~20M左右。

### 后处理

后处理部分使用Python库：pattern。此库可以分析英文词语的变形，然后输出原型。比如”is”和”are”，会全部还原成”be”，一起统计。

该库使用一个句子作为输入，然后构建出一个分析树，含有句子内单词的各项信息。事实上这个库应该是用一个完整的句子作为输入才能得到一个比较好的结果，但如此分析源文件的话就太慢了，因此只能退而求其次，采取先统计后合并的方式，把统计好的单词作为输入，然后得到结果。

此部分还有一个任务，就是去除之前C语言不方便去除的&quot。至于It’s这类简写，pattern库可以识别并处理成两个单词。

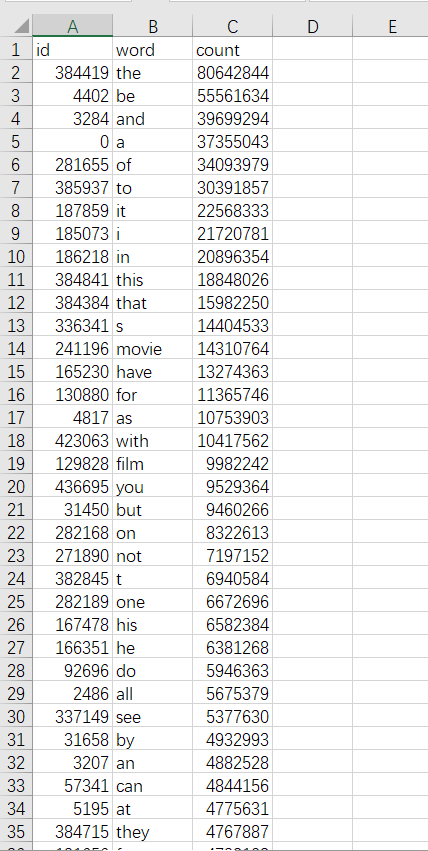
后处理部分

1. from pattern.text.en import \*
2. import pandas as pd
3. *# 获取单词*
4. word\_dict = {}
5. result = {}
6. with **open**('movies\_out.txt', 'r') as f:
7. str0 = f.readline()
8. loop = 0
9. while str0:
10. loop += 1
11. text = str0.strip().replace("&quot", "").split()
12. num = int(text[-1])
13. del text[-1]
14. for word in text:
15. tree = parsetree(word, lemmata=True)
16. for i in tree:
17. for lemma in i.lemma:
18. if lemma.isalpha():
19. word\_dict.setdefault(lemma, 0)
20. word\_dict[lemma] += num
21. str0 = f.readline()

## 最终结果

我们最终分析得到749,347个不同的单词及对应的出现频次。最终结果的部分截图如下：

经过审视最终结果，我们发现一个有趣的特点：出现频次较低的词往往是拼写错误的单词，或者是重复单个字母构成的语气词。使用次数5000次以下的基本全部都是不常见的词；1000次以下的基本都是拼错的词或无意义语气词。



# 图数据库实践

搭建Neo4j图数据库，导入作业1中下载的用户评论文件，并计算共同评论某几部电影（例如哈利波特系列7部电影等，查询者可以自行定义）最多的用户集合是什么？

## 基本思路

编写ETL脚本，从用户评论文件中提取出用户节点，电影节点和用户、电影间的评论关系，将其导入Neo4j图数据库中，设计查询语句完成任务。

## 实现步骤

### 编写ETL脚本

本次练习使用的用户评论数据来自于SNAP发布的Web data: Amazon movie reviews数据集。该数据集包含了跨度超过10年, 总数超过8百万条的亚马逊电影评论. 每条评论包括商品信息、用户信息、评分和纯文本格式的评论内容。

首先分析用户评论文件的数据格式：

product/productId: B00006HAXW

review/userId: A1RSDE90N6RSZF

review/profileName: Joseph M. Kotow

review/helpfulness: 9/9

review/score: 5.0

review/time: 1042502400

review/summary: Pittsburgh - Home of the OLDIES

review/text: I have all of the doo wop DVD's and this one is as good or better than the

1st ones. Remember once these performers are gone, we'll never get to see them again.

Rhino did an excellent job and if you like or love doo wop and Rock n Roll you'll LOVE

this DVD !!

可以发现，每一行有效数据均以固定的前缀开头，前缀和数据内容之间以分号分隔，每八行数据为一组，每组评论数据以“product/productId”开头，因此可以以此为一组评论开始的标志。

部分数据，如profileName，text等，内部可能还有换行符，导致出现非固定前缀开头的行，在ETL脚本中，对于这种情况，我们选择将这行的数据添加到上一行末尾。

根据SNAP发布数据集时附带的说明，最终解析出来的评论数目、用户数目和产品数量一致。说明ETL脚本的功能没有问题。

|  |  |
| --- | --- |
| Number of reviews | 7,911,684 |
| Number of users | 889,176 |
| Number of products | 253,059 |
| Users with > 50 reviews | 16,341 |
| Median no. of words per review | 101 |
| Timespan | Aug 1997 - Oct 2012 |

Tab. 2.1 数据集统计信息

ETL脚本运行后将得到三个csv格式的文件，分别存储了用户节点、商品节点和评论关系的数据。后续将把这些数据导入Neo4j中进行查询。

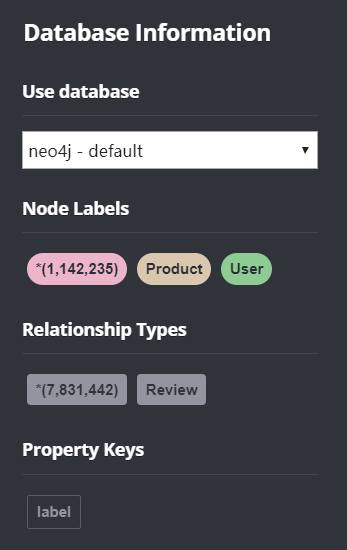
### 导入节点和关系

启动Neo4j Desktop，新建一个数据库，将上一步得到的三个csv格式的文件复制到%AppData%/../Local/Neo4j/Relate/Data/dbmss/dbms-xxxxxxxxxxxxxxx/import/目录下。

启动Neo4j Browser界面，然后依次执行下面的SQL命令：

1. 导入商品节点数据：
2. :auto USING PERIODIC COMMIT
3. LOAD CSV WITH HEADERS FROM "file:///product.csv" AS line
4. CREATE (p:Product{label:line.productId})
5. 导入用户节点数据：
6. :auto USING PERIODIC COMMIT
7. LOAD CSV WITH HEADERS FROM "file:///user.csv" AS line
8. CREATE (p:User{label:line.userId})
9. 在商品节点和用户节点上分别建立索引：
10. CREATE INDEX ON :Product(label)
11. CREATE INDEX ON :User(label)
12. 导入评论关系数据（约1h）：
13. :auto USING PERIODIC COMMIT
14. LOAD CSV WITH HEADERS FROM "file:///review.csv" AS line
15. MATCH (from:User{label:line.userId}), (to:Product{label:line.productId})
16. MERGE (from)-[:Review]->(to)

数据导入完成之后，在左侧的Database Information部分可以看到一共导入了1,142,235个节点和7,831,442条边。



### 设计查询语句

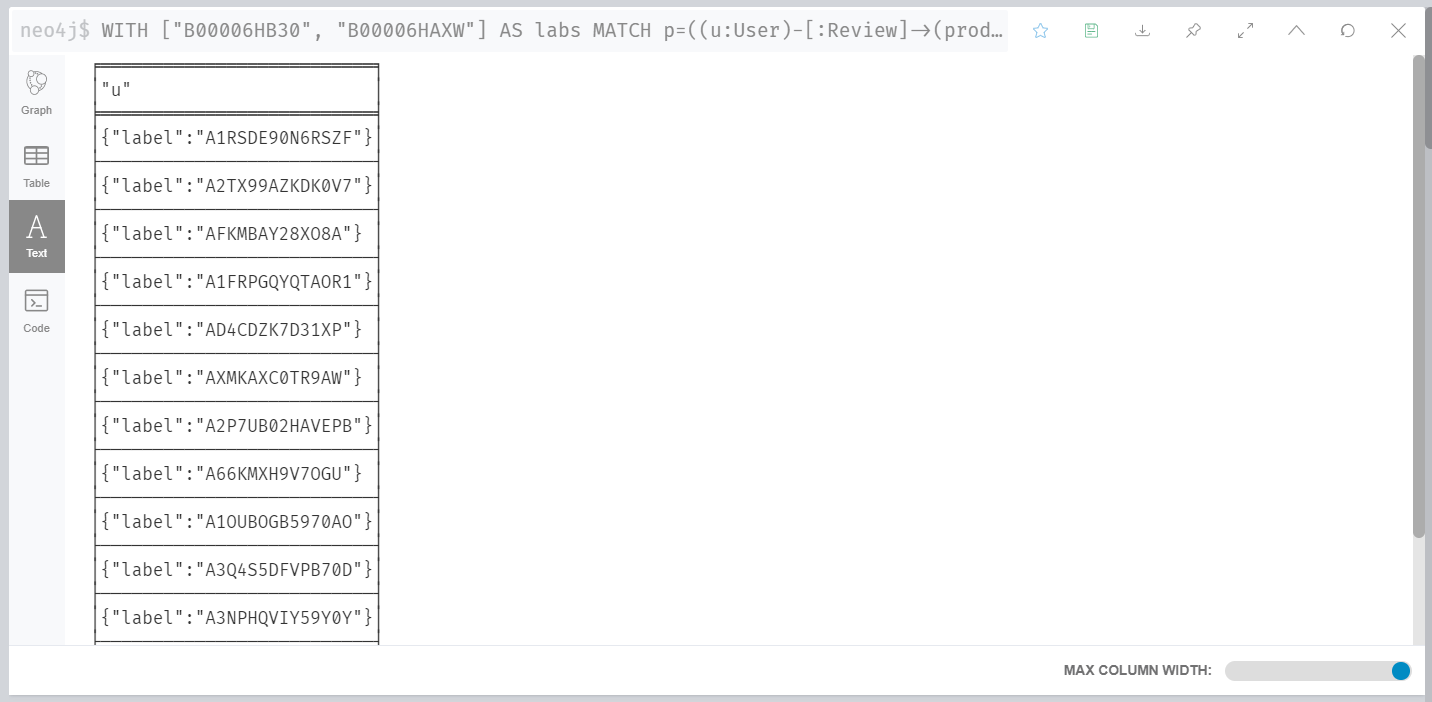
根据要求，设计查询Cypher语句如下：

1. 查询节点
2. WITH ["B00006HB30", "B00006HAXW", "B000F4RHUC"] AS labs
3. MATCH p=((u:User)-[:Review]->(prod:Product))
4. WHERE ALL (
5. lab IN labs
6. WHERE EXISTS ( (u)-[:Review]->(:Product{label:lab}) )
7. )
8. AND prod.label IN labs
9. RETURN DISTINCT u
10. 查看可视化结果
11. WITH ["B00006HB30", "B00006HAXW", "B000F4RHUC"] AS labs
12. MATCH p=((u:User)-[:Review]->(prod:Product))
13. WHERE ALL (
14. lab IN labs
15. WHERE EXISTS ( (u)-[:Review]->(:Product{label:lab}) )
16. )
17. AND prod.label IN labs
18. RETURN DISTINCT p

## 最终结果

例：查询同时评论了"B00006HB30", "B00006HAXW"三部电影的用户群体

1. 查询用户列表



1. 查看可视化结果

