一个Linux上用python直接处理csv文件的方法

汪兴元

2015 年7 月21 日

摘要

我们常常需要处理体积很大的csv数据或日志文件。一般来讲，导入RDBMS来处理是常用的一种办法，但此办法也不完美，尤其是SQL这种描述型的语言。在表达算法的时候不如一般的程序设计语言的表达力强，某些应用场合虽然能够实现，但是难度太高，优化不易；批量处理大数据，通过在RDBMS上运行SQL效率并不高。

另一种办法就是使用Hadoop或Spark，或导入NoSQL中，再使用MapReduce。但是使用类似Hadoop之类的工具又可能太重，数据量不够运作一个Hadoop集群，还得承受其代价。

本文界绍了直接处理csv文件的一套办法，作为另一个数据处理的选项，示例代码用python，运行的操作系统为Linux，源程序及csv数据文件的编码采用utf8。在从原始数据到最终的目标结果，需要组合本文提到的各种算法，才能达到目的。通过管道-过滤器连接每一个算法，能够将整个处理算法分而治之，同时得到比较好的性能。

本文假设要处理的整个数据集无法一次装入机器的内存。

目录

1 校验数据

csv文件可能存在错误。在正常情况下，csv文件不应存在错误。但是写入csv的过程并非事务型的，不象RDBMS那样有很好的ACID保证，故磁盘耗尽，进程异常退出、挂起等因素，直接导致csv文件的格式并非预期。

检查数据没有统一的方法，要视数据自身的特点来做检查。一般是检查一些数据正确所需的必要条件，但必要并不一定充分。一般的检查方法有：

1. 检查列数。如果列数不等于预期值，可以确定此行数据错误。
2. 检查每列数据的格式，比如，数字，日期，或其它满足指定格式的文本串。可以偿试将其解析为对应的类型，看能否成功；或用正则表达式验证。
3. 校验字段的值。前两项都对的情况下，可以检查数据值的取值范围。比如健在的人的年龄不能是负数，也不能是数百以上；历史数据中的记录时间不可能超过当前的日历时钟。

```
#! /usr/bin/python
# coding=utf-8
# validate_history_ai.py

import csv
import sys

if __name__ == '__main__':
  infile = sys.stdin
  outfile = sys.stdout

  reader = csv.reader(infile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)
  writer = csv.writer(outfile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)

  for row in reader:
    try:
      # check if the number of columns is right.
      if len(row) != 12
        continue;
      # check the format of each column...skipped.
      # check the value range of each column...skipped.
      writer.writerow(row)
    except:
```

```
26          pass
27      finally:
28          pass
```

如果数据是已经通过有严格校验的系统中生成或导出的，原始数据本身无误，一般做以上三项检查可以查出我们能见到的全部错误。但是如果数据是人工填写或是有故障的机器生成的，这三项检查仍不能确保检查出全部错误。

对于错误的数据该如何处理，要视情况而定。有的业务场景，例如Web服务器的access log，价值不高；又如水温传感器输入的每分钟一次的历史数据，可以丢弃，之后使用插补法补缺；有的场景是不可以这样做的，如交易记录，需要重新导出此段数据或人工处理。样例代码中我们采用了丢弃的方法处理。

以上代码没有直接打开csv文件，而是从标准输入中读取文件，处理完毕之后再写到标准输出。这样的好处是便于通过管道过滤器连接多个处理进程，避免过高的耦合度。本文所有的处理程序都使用管道过滤器连接，不再赘述。

程序中对校验2.和3.未实现，可以练习下。对于机器导出的数据，做完1.可以去掉大部分错误。如果不打算实现2.和3.

可以将列数从命令行上读取，成为更通用的子程序。这个作为练习。

2 选择列

原始数据中有可能只有部分列是所要关心的，其它的与要解决的问题无关，可以丢弃。因此要选择列，类似SQL语句中的SELECT所要办的事情。样例程序如下所示，这里我们只对第1，3，4，5列感兴趣。

```
1  #! /usr/bin/python
2  # coding=utf-8
3  # select_history_ai.py
4
5  import csv
6  import sys
7
8  if __name__ == '__main__':
9    infile = sys.stdin
10   outfile = sys.stdout
11
12   reader = csv.reader(infile, delimiter=',', quotechar='"',
13     quoting=csv.QUOTE_ALL, skipinitialspace=True)
14   writer = csv.writer(outfile, delimiter=',', quotechar='"',
15     quoting=csv.QUOTE_ALL, skipinitialspace=True)
16
17   for row in reader:
18     outrow = []
19     outrow.append(row[0])
20     outrow.append(row[2])
21     outrow.append(row[3])
22     outrow.append(row[4])
23     writer.writerow(outrow)
```

可以将列下标从命令行输入，这样这个程序就成为一个通用的选择程序，而不是仅用于示例的场景。这个作为练习。

3 排序

我们对数据做去重、转置、分组、聚集，都需要先对数据排序。因此排序是非常重要的功能。不可或缺。

大数据的排序由于无法直接一次装入内存，故必须使用外部排序。如果能将数据切成能用内部排序的小块，排好序之后，再合并，那么我们就可以完成对大数据的排序。

我们不打算从头实现一个大数据排序工具。利用Linux的工具`sort`来做排序。先将csv文件拆分。可以在导出或生成csv的时候，每达到某一固定尺寸就rotate一下，当前文件改名，再创建一个新文件当作当前文件。

假设文件已经切成能用内存装下的多个小文件。注意，使用切割工具切文件，必须从整行数据的位置断开，否则，切口处的那条数据被切坏了。

对每一个文件，执行`sort`。下面的两段代码是在Linux的终端上输入的命令，“`$`”表示命令提示符。示例中有两个文件，这里只写了第一个文件的排序，另一个省略。

```
1 $ cat data/att-rec-utf8_part-1.txt \
2   | python python-src/select_att_rec.py \
3   | sort --field-separator=, --ignore-leading-blanks --stable \
4   --key=1.2n,2 --key=2,3 >part-1.csv
```

此命令先用`select_att_rec.py`选择所要的列，再对选择的结果排序，之后将结果重定向到文件`part-1.csv`。所有小文件都排序好之后，再对排序后的文件做merge:

```
1 $ sort --field-separator=, --ignore-leading-blanks --stable \
2   --key=1.2n,2 --key=2,3 -m part-*.csv > merge-sorted.csv
```

注意在做merge之前，必须先对每个文件排好序；merge时，使用的关键字、分隔符等参数必须与`sort`的时候所用的参数完全一致。

merge完之后，得到一个已经排好序的大文件。

4 转换ID字段

一般在集成两个系统的数据的时候，需要将两个系统的数据合并起来。但是两个系统的数据ID一般是相互独立分配的，因此，如果ID都是同一种类型，比如整数，几乎100% 会发生冲突，即不相同的两个对象，其ID相同。即使用字符串之类的做ID，也难以不发生冲突。因此必须要建立两个系统间的对象的对应关系。

例如，商品销售网站和库管理系统如果是两套独立的系统，分别由不同的团队开发，比如库存管理系统是外购的，商品销售网站是自建的，如果要将两套系统的数据整起来合做分析，又没有办法修改系统达到统一，就需要做ID转换。

下面的程序从标准输入读入历史数据，从命令行读取基础数据文件名，加载基础数据到字典中，利用基础数据中的$R(\mathrm{ID}_{src}, \mathrm{ID}_{dest})$关系，做了一个映射。逐行将原ID映射到目标ID的方法替换掉ID，实现ID转换的目的。

```
1 #! /usr/bin/python
2 # coding=utf-8
3 # transform_data_id.py
4
```

```
import csv
import sys
import traceback

dataObjectIds = {}

if __name__ == '__main__':
  infile = sys.stdin
  outfile = sys.stdout
  cfgfile = None
  if(len(sys.argv) > 1) :
    cfgfile = open(sys.argv[1], 'rb')

  cfgreader = csv.reader(cfgfile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)
  reader = csv.reader(infile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)
  writer = csv.writer(outfile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)

  for row in cfgreader:
    try:
      key = int(row[0]), int(row[1])
      value = int(row[2])
      dataObjectIds[key] = value
    except ValueError:
      pass
    finally:
      pass

  for row in reader:
    try:
      key = int(row[0]), int(row[2])
      dataId = dataObjectIds.get(key, None)
      converted = []
      if dataId != None:
        converted = [dataId, row[1]]
        converted.extend(row[3:])
        writer.writerow(converted)
    except:
      print >> sys.stderr, row, traceback.format_exc()
```

一般来讲，大多数系统中的某一基础数据对象的ID是可以一次全部装入内存的。比如超市的商品ID，虽然可能有数十万条，但是对于现代计算机的内存来讲并不算多。

对于历史记录，其身的ID，或Primary Key的数量大，其总体积有可能单台机器的内存装不下。不过一般情况下，需要做转换的必要性往往不大，只对其通过Foreign Key引用的基础数据对象的ID有转换的价值。如果实在需要转换，使用6.1提到的内连接方法来做大表的连接，再选择所

要的字段，包括新ID，不选原ID字段即可。

这个方法除了可以用于转换ID的场合，还可以用于更换名称。比如，我要将公司员工的行为的分析结果公布，但是不想让人知道员工的真实姓名或工号；或者谁购买了伟哥，将要商品名换掉。

样例程序中的ID列号、多少列都是hardcode进代码的。可以将关系搞成动态的，从命令行读入。设$C_{i_1},\cdots,C_{i_m}$是从要转换的数据中选出来作为原ID的列，$C_{j_1},\cdots,C_{j_n}$是要转成的目标ID列，$m,n\geqslant 1$。则转码可以视为从

$$\begin{aligned} transform &: R_{src}\to R_{dest} \\ \mathrm{ID}_{src}(C_{i_1},\cdots,C_{i_m}) &\mapsto \mathrm{ID}_{dest}(C_{j_1},\cdots,C_{j_n}) \end{aligned} \tag{1}$$

只需要在命令行上输入源ID的列号清单和目标ID的列号清单，在程序中，用迭代清单代替硬编码的取列语句即可。将hardcode转为动态，作为练习。

5 去重复值，整行distinct()

如果在内存中，用set来装数据，自动去重。但是我们的数据量大，不能装入内存，因此直接用set不行。如果对每个输入值，都去扫描一遍记录，如果共有n个记录，则对第1个记录，要扫描第$2,\cdots,n$共$n-1$个记录；对第2个记录，要扫描第$3,\cdots,n$共$n-2$个记录，对第$n-1$个记录要扫描第n个记录共1个记录。于是一共要扫描文件$n-1$次，转储到文件$n-1$次，其中最后一次是结果，中间$n-2$次是临时文件。这个性能相当差。共扫描记录的个次是

$$(n-1)+\cdots+1=\frac{1}{2}n(n-1)。 \tag{2}$$

csv文件每行的长度不固定，因此，定位记录的位置很难，往往扫描的记录数是n^2。

如果将数据先排序，则相同的两条记录会相邻，成为一组，于是，我们只需要比较相邻的两条数据是否相同，如果相同就丢弃一条，如果不同就输出这一条。示例代码如下，假设数据事先已经排好序。

```
#! /bin/env python
# coding=utf-8
# remove_duplicate.py

import csv
import sys

currentRow = None
def duplicate(row):
  global currentRow

  result = True
  if(currentRow == None):
    result = False
  else:
    for i, j in zip(row, currentRow):
      if i != j:
        result = False
        break
    currentRow = row
```

```
21    return result
22
23 if __name__ == '__main__':
24    infile = sys.stdin
25    outfile = sys.stdout
26    if(len(sys.argv) > 1) :
27      infile = open(sys.argv[1], 'rb')
28
29    reader = csv.reader(infile, delimiter=',', quotechar='"',
30      quoting=csv.QUOTE_ALL, skipinitialspace=True)
31    writer = csv.writer(outfile, delimiter=',', quotechar='"',
32      quoting=csv.QUOTE_ALL, skipinitialspace=True)
33
34    for row in reader:
35      if(not duplicate(row)):
36        writer.writerow(row)
```

6 连接

数据仅代表同一个表或关系的csv是不够的。两个表csv连接跟写SQL查询一样常见。如果是历史记录表跟基础数据表的连接，一般基础数据很小，历史数据大，所以可以采用 4所述的方法。即把历史表的Key作为dict的Key，基础表的Key作为dict的Value，逐行转换。

如果是内连接，则对于每行历史记录，必须找到对应的基础表中的记录才算连接上，否则要丢弃；而对于左连接，找不到对应的记录，则用空值填充。

对于右连接，和全外连接，由于历史数据很大，不能直接装入内存，这办法不适用。

接下来描述不将表装入内存的连接方法。

6.1 内连接

先将两个表的csv按连接的关键字排序，升序或降序都行，但必须都是升序或都是降序。

再逐行连接。不妨假设都按升序排好了序。设两个表分别为R_1，R_2，当前从这两张表取的数据行的行r_1，r_2。

1. 先从R_1中取第一行为r_1，取其连接的Key，设为K_1，取R_2中的第一行为r_2，取其Key，K_2，进行比较。如果$K_1 < K_2$，由于R_2为升序，r_2及R_2余下的行的Key均不小于K_2，因此，r_1与r_2及R_2余下任一行都连接不上。于是再取R_1中的下一行为r_1。
2. 如果$K_1 > K_2$，由于R_1升序，r_1及R_1余下的行的Key均不小于K_1，因此，r_2与r_1及R_1余下任一行都连接不上。于是再取R_2中的下一行为r_2。
3. 如果$K_1 = K_2$，则连接成功，输出连接后的行。再取R_2的下一行为r_2。重复以上步骤，直至所有的数据处理完毕。

此算法的python实现如下。

```
1 #! /bin/env python
2 # coding=utf-8
3 # remove_duplicate.py
4
5 import csv
```

```
 6 import sys
 7
 8 if __name__ == '__main__':
 9   leftTableIn = sys.stdin
10   rightTableIn = open(sys.argv[1], 'rb')
11   outfile = sys.stdout
12
13   leftReader = csv.reader(leftTableIn, delimiter=',',
14     quotechar='"', quoting=csv.QUOTE_ALL, skipinitialspace=True)
15   rightReader = csv.reader(rightTableIn, delimiter=',',
16     quotechar='"', quoting=csv.QUOTE_ALL, skipinitialspace=True)
17   writer = csv.writer(outfile, delimiter=',', quotechar='"',
18     quoting=csv.QUOTE_ALL, skipinitialspace=True)
19
20   try:
21     r1 = leftReader.__next__()
22     r2 = rightReader.__next__()
23     while True:
24       key1 = (r1[0], r1[2])
25       key2 = (r1[0], r1[1])
26       if key1 < key2:
27         r1 = leftReader.__next__()
28       elif key1 > key2:
29         r2 = rightReader.__next__()
30       else:
31         row = []
32         row.extend(r1)
33         row.append(r2[2])
34         writer.writerow(row)
35   except StopIteration:
36     pass;
```

此算法利用数据集的顺序，避免了重复扫描csv，只需one pass即可完成连接；缺点是必须先排序。

可以考虑将程序写成通用的，在命令行上指定连接谓词。这个可以作为练习。