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;
```

```
22        # check the format of each column...skipped.
23        # check the value range of each column...skipped.
24        writer.writerow(row)
25      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。

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

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:
```

```
16      for i, j in zip(row, currentRow):
17        if i != j:
18          result = False
19          break
20      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。

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

import csv
import sys

if __name__ == '__main__':
  leftTableIn = sys.stdin
  rightTableIn = open(sys.argv[1], 'rb')
  outfile = sys.stdout

  leftReader = csv.reader(leftTableIn, delimiter=',',
    quotechar='"', quoting=csv.QUOTE_ALL, skipinitialspace=True)
  rightReader = csv.reader(rightTableIn, delimiter=',',
    quotechar='"', quoting=csv.QUOTE_ALL, skipinitialspace=True)
  writer = csv.writer(outfile, delimiter=',', quotechar='"',
    quoting=csv.QUOTE_ALL, skipinitialspace=True)

  try:
    r1 = leftReader.__next__()
    r2 = rightReader.__next__()
    while True:
      key1 = (r1[0], r1[2])
      key2 = (r1[0], r1[1])
      if key1 < key2:
        r1 = leftReader.__next__()
      elif key1 > key2:
        r2 = rightReader.__next__()
      else:
        row = []
        row.extend(r1)
        row.append(r2[2])
        writer.writerow(row)
  except StopIteration:
    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。

6.2 左连接

同前述内连接，但是步情况1.连接不上r_2的时候，将r_1补齐空列之后输出。

6.3 右连接

同前述左连接，但是将R_1、R_2的位置互换即可。

6.4 全外连接

可以有两种办法可选：

1. R_1、R_2先做一遍左连接，R_1、R_2再做一遍右连接，合并两个连接的结果，最后去掉重复值即可。注意要保持左连接和右连接的列顺序是相同的。
2. R_1、R_2先做左连接，再将结果与R_2做一个类似右连接的操作，区别于右连接的是：对于有匹配的行，直接丢弃；对于无匹配的行，填写空值后输出。

办法1.简单，但需要的步骤多，每个步骤都要用到文件；办法2.稍复杂，但步骤少可以只写一次文件，采用管道直接连接中间结果。

从性能上考虑，办法2. 较优。

7 过滤数据

筛选满足条件的数据是信息检索中必不可少的部分。将用户的Search Criteria转成过滤数据的Predicate。Search Criteria可能包含多个条件，为了简化问题，需要将每个条件做成一个Predicate，只做一项测试，多个条件需要多个Predicate，并且每个Predicate 之间可能是AND，也可能是OR，并且有优先级顺序要求。

7.1 谓词及表示方法

谓词是一个返回Boolean值的表达式，用于测试对象的性质是否为真，或对象之间的关系是否为真。

例如测试订单是否用银行卡支付；帐户余额是否大于100块；a同学的考试分数是否比b 同学的高，等等。

```
1 for row in input:
2   if predicate.test(row):
3     output.writerow(row)
```

如果只测试其中的某列或几列，也可以如此：

```
1 for row in input:
2   if predicate.test(tuple([row[0], row[2]])):
3     output.writerow(row)
```

经验不多的开发者在写过滤条件时，往往不注意将各谓词分解为基本的形式，并且与其它过滤条件，可能还与过滤数据无关的问题，合并一起解决。这样一来，大片的其它逻辑与谓词测试的内容混合在一起，修改起来要同时考虑多个问题。

这样在问题不太复杂的时候问题不大，而且可能性能比分离关注点来考虑更简单。在解决一个较大的问题的时候，往往复杂到做不下去。

例如，使用的“Predicate”并不返回Boolean值，而是直接返回满足测试条件的数据，并且做了处理，列数，字段的含义都可能已经变了，因为恰好下一步也需要这个数据，这是常见的问题之一。例如，

```
1 for row in input:
2   result = predicate.test(row):
```

```
3   if result != None:
4     output.writerow(result)
```

上面的代码中，`result == row`未必能成立。光看这段代码，天知道`predicate.test(row)`里面做了什么。

做Predicate的时候要把握以下原则，这些原则一般情况下应当尊守：

1. Predicate只测试数据是否满足所指定的条件，满足反回True，不满足返回False。不能更改输入的数据。
2. 不做与指定的测试无关的任何其它工作，例如，提取数据，计算平均值之类。如果所需要做的其它工作与测试结果有关，应交给Predicate的使用者来完成；如果无关，则更应分开来做，既不能在Predicate里做，又不能在Predicate的使用者里面做。
3. Predicate不能有状态。其返回值仅取决于输入的条件，包括数据和测试条件。重复测试同一条数据，得到的结果应相同。如果使用lambda或闭包，其中捕获的参数不应发生变化。

下面对这3条举例说明。

原则1. 可能出于下一步的数据处理方便的目的，在当前Predicate中对输入数据作调整，容易实现，又比较方便。这样做，从名义上，Predicate只做测试，而不作修改，而事实上却做修改，容易误导代码的读者；而且关注点不分离，引起复杂度上升，例如，如果下阶段增加了一个Predicate需要没有修改的数据，这样就有矛盾了：如果去掉原Predicate对数据的更改，则后面的处理不能正常运行，也要改。

原则2. 例如，有人做了一个比较大小的Predicate，用于扫描csv文件是否为升序，通过比较第n条和第$n+1$条数据的大小。因此，可以顺便用来统计csv文件的行数。假设数据满足升序条件，对每一条数据都做了扫描。

如果csv文件总共有N条数据，Predicate将会被调用$N-1$次。反过来，如果Predicate 被调用了$N-1$次，那么csv文件的行数应是N行。

假设文件既不满足升序，又不满足降序，那么在中间某一位置k，Predicate返回False，测试终止，这时候Predicate统计的行数只能是前面有$k+1$条数据满足条件，这个与行数不符。

另有一排序功能需要用到比较的Predicate，于是该Predicate直接被用于排序，但排序会用到多少次比较与数据的条数的关系不是这样的，于是这个行数统计毫无意义。

原则3. 例如比较大小。如果$a>b$，第一次测试`greaterPred.test(a, b) == True` 应能成立，而且无论做多少次测试，只要a,b保持原值，`greaterPred.test(a, b) == True`必须都成立。

如果使用闭包，捕获的参数可能是引用一个闭包外的对象。由于这个对象不是常量，且有并发访问，这个对象的状态可能会被修改，引起闭包的状态也随之变更。这样一来，闭包的Test Criteria就变了，使用同样的数据做测试，返回值可能会与改变前不同。

谓词分两种类型的：Unary和Binary。前者只需要输入一条数据，后者需要两条。其它类型的可以通过这两类组合运算，或者将数据整合得到。

```
1 def LE(a, b):
2   return a <= b
3
4 pred = lambda x : LE(x, 10)
5
6 for row in inputFile:
7   if pred(row):
8     writer.writerow(row)
```

测试一条数据中的某一属性是否为真，可以用Unary。某些情况看似需要两个数据输入的，也可以用Unary，但是使用lambda或闭包。例如，测试一个数是否大于某个常数，需要比较两个数，

但是常数可以使用lambda和一个LE()函数来捕获，这样，得到的lambda对象只需要输入一个数。而对于检查顺序是否升序这种情况，要比较的两个对象没有一个常量，因此直接用二元的LE()函数。

```
def isAscending(l):
  n = None
  for i in l:
    if n is not None:
      if not LE(n, i):
        return False
    n = i
  return True
```

以上代码有几个问题：假定每一行数据都不能是空行；有一个状态量n，使得程序的分支数比较多。如果这样写，分支总数没有变，但是每个子程序的分支数少了：

```
class LEPreviousPred():
  def __init__(self):
    self.prev = None
  def test(x):
    old = self.prev
    self.prev = x
    if self.prev is not None:
      return LE(old, x)
    return False

def isAscending(l):
  pred = LEPreviousPred()
  lePrevious = lambda x : pred.test(x)
  for i in l:
    if not lePrevious(i):
      return False
    n = i
  return True
```

就这个例子而言，除了isAscending()函数简化了一丁点，这个做法总代码行数还多了，不划算。但是这个代码演示了将Predicate的测试逻辑与调用者分离的一种办法。

在有较多Predicate一起参与的时候，程序逻辑势必复杂，这样一来，问题分成3份：

1. 单个基本Predicate本身的实现。这个与实际要做的过滤程序程序的复杂度没有关系。
2. 使用Predicate的过滤程序。这个程序只有一个单条件的if测试。
3. 多个基本的Predicate的组合，成为整个过滤子程序的Predicate，也就是2.中单条件if语句所要测试的终极条件。

而且此代码与前面原则3.不符。类似还有与时钟相关的测试，例如检测某事件发生的时刻与当前时间的差距是否在某一范围内。由于时钟一直在走，必须每次都要取当前时间，而其它的测试不必，这样在形式上不能统一，需要增加处理的复杂度。如果将取时钟的动作放在Predicate内，则只需要将数据行送到Predicate作测试即可，成为一元Predicate。

这类测试的Predicate可以做成满足或不满足原则3.形式。如果能大幅度简化程序或提高性能，并且不降底可读性的情况下可以考虑不遵守，但是如果没有换取足够的好处，不应违反原则。

对于Predicate的接口形式，只要使用起来方便即可，使用lambda、普通函数、类实例都可以。如果过滤数据的时候用到了很多个Predicate，每个Predicate最好按类型统一接口，这样形式上保持一致，可以用一致的方法处理；这些Predicate最好组合成一个Predicate，能一次做完全部的测试，以便调用者在使用时，不必关注Predicate之间的关系细节，只需关注它满足或不满足条件。

同时使用多个Predicate可以减少扫描文件的次数，但是，是否要分几次扫描文件，要视情况而定。以下情况，如果用到Predicate，最好分步骤，一次只做一件事；而且，不能与纯粹的条件过滤一起做。一般条件过滤只测试一行数据，且不改变这行数据。

1. 去重复值。需要一次处理两行。
2. 聚集。需要按Group处理，Group内都是多行。
3. 根据输入列，计算其他列的值。输出的行与输入行不一样。

主要问题是一起做会大幅度增加代码的复杂度，使得可读性、可维护性降底。

7.2 谓词的连接关系

前面已经提到Predicate的表示及如何使用。实际的数据过滤程序中，很少用到单一Predicate。最基本的组合关系就是AND和OR加上优先级。

组合关系，本身就是另一重Predicate。比如在n个Predicate $P_1, \cdots, P_n$中，至少有m个为True，则Predicate P为True。因此，除测试$P_i \in P_1, \cdots, P_n$以外，还要测试为True的P_i的个数是否不小于m。此类Predicate层出不穷，需要随业务需求而定，不能一一列举，我们只列最基本的形式。

7.2.1 AND

必须每个Predicate返回True才算满足条件。直接使用编程语言的“与”运算符的版本不举例。下面是支持多个Predicate的示例代码，注意如果`predicates` 是空列表的时候，所有数据都匹配，要考查这是否为预期的行为。

```
1 for row in inputFile:
2   match = True
3   for p in predicates:
4     if not p(row):
5       match = False
6       break
7   if match:
8     writer.writerow(row)
```

但是如果我们的Predicate并不都是用AND来连接的，那么这办法处理AND就不太好。如果我们不加处理地直接使用Predicate，又要处理好它们的关系，我们需要复杂的逻辑结构来利用这些Predicate，而且越复杂，越难实现，越难修改。

最好的办法是不影响过滤程序的结构，只改Predicate及其关系。我们将上面的代码改成下面的形式：

```
1 class AND():
2   def __init__(self, predicates):
3     self.predicates = predicates
4
5   def test(row):
6     for p in self.predicates:
7       if not p(row):
```

```
8           return False
9       return True
10
11 andAll = AND(predList)
12 andAllPred = lambda x : andAll.test(x)
13
14 for row in inputFile:
15   if andAllPred(row):
16     writer.writerow(row)
```

主程序就仅剩第14 ~ 16行这三行。不管我们的条件如何，程序的结构应是如此。

是直接使用编程语言的“与”运算符，还是使用一个Predicate列表，要视复杂程度而定。如果条件不多，应直接使用编程语言的逻辑运算符，或部分使用，避免过度工程，吃力不讨好；如果逻辑复杂，则全部使用本小节的办法。分寸的拿捏要看实现的难度和程序的可读性，在两者之间取得平衡。

7.2.2 OR

类似AND连接，OR的直接实现代码如下：

```
1 for row in inputFile:
2   match = False
3   for p in predicates:
4     if p(row):
5       match = True
6       break
7   if match:
8     writer.writerow(row)
```

同样，如果要解耦Predicate和过滤程序的关系，代码如下：

```
1 class OR():
2   def __init__(self, predicates):
3     self.predicates = predicates
4
5   def test(row):
6     for p in self.predicates:
7       if p(row):
8         return True
9     return False
10
11 orAll = OR(predList)
12 orAllPred = lambda x : orAll.test(x)
13
14 for row in inputFile:
15   if orAllPred(row):
16     writer.writerow(row)
```

7.2.3 优先级

从前面的利用AND和OR组合Predicate的例子，我们知道组合后的结果可以表达为一个Predicate。同理，优先级高的Predicate先运算，可以将它们做成一个Predicate再参与运算。如果要求当前的表达式的值，参与当前表达式运算的Predicate或表达式必须先求值，于是，优先级就实现了。例如，我们要实现$(A \wedge B) \vee (C \wedge D)$，可以用下面的代码：

```
andAB =  AND([A, B])
andCD =  AND([C, D])
andABPred = lambda x : andAB.test(x)
andCDPred = lambda x : andCD.test(x)

orAll = OR([andABPred, andCDPred])
orAllPred = lambda x : orAll.test(x)

for row in inputFile:
  if orAllPred(row):
    writer.writerow(row)
```

8 聚集

关于骤集函数的定义，参考https://en.wikipedia.org/wiki/Aggregate_function。

要理解聚集，必须先搞清楚分组。聚集一般是按分组的，例如，找出每个班中数学考试分数最高的，就是按班级分组，找出每个班的最高分，有多少个班就有多少个最高分；除非全班旷考，或者存在无人的班级—有这可能性？如果不指定分组条件，将是全部学生中最高的，只有最多一个数值。

8.1 group by

在SQL中的`order by`子句一般出现在带有聚集运算的`select`语句尾部。一般写SQL的时候，如果没有使用聚集函数，又用到了`group by`子句，只有选择的列与`group by`的列数相同的时候是没有问题的，其它选择会报错。

使用SQL语句`group by`的结果未必是同时`order by`那些`group by`所指定的那些列的，只保证每一个group内部的行都相邻。如果不指定`order by`，RDBMS会使用最有效率的方法`group by`。

然而RDBMS高效`group by`的方法，我们在python代码中用起来不容易，不现实。我们为了让同一group的行都在一起，我们先将数据按`group by`列`order by`，即先排序，用前面第3节提到的Linux的`sort`。

如果没有用聚集函数，排完序之后，要除掉重复行。使用第5节的方法去除重复行。

如果使用了聚集，则在使用聚集函数的同时，必须检查每行是否是同一group。如果正在处理的行不属于当前group则应计算当前group的聚集并输出，再用当前行重新开一个group作为当前group。由于做了group检测，所以不必再做去重这一步。

```
class IsInGroup():
  def __init__(self, colnumbers):
    self.colnumbers = colnumbers
    self.curRow = None

  def test(row):
```

```
    prevRow = self.curRow
    self.curRow = row

    if prevRow is None:
      return False

    for colno in colnumbers:
      if not (prevRow[colno] == row[colno]):
        return False
    return True

groupByColumnNumbers = [0, 2]
isInGroup = IsInGroup(groupByColumnNumbers)
isInGroupTest = lambda x : isInGroup.test(x)

for row in inputFile:
  if isInGroupTest(row):
    ...
```

8.2 having

SQL语句的where子句不能用于聚集函数，只能用于表或视图的列。SQL可以用having来对聚集列做过滤。

我们在处理csv文件的时候，没有必须做此区别的限制。我们可以在输出聚集的结果之前，对结果做过滤，也可以单独设一过滤环节，用管道连接到聚集的输出。

过滤数据用第 节所述的方法。

8.3 max()

按前面的group by和having的方法，实现max()函数的功能。示例代码如下：

```
def getGroupByCols(row, colNumbers):
  groupCols = []
  for colno in colNumbers:
    groupCols.append(row[colno])
  return groupCols

col4max = 1
max = None
for row in inputFile:
  if isInGroupTest(row):
    if row[col4max] > max:
      max = row[col4max]
  else:
    outrow = getGroupByCols(row, groupByColumnNumbers)
    outrow.append(max)
    max = None
    writer.writerow(outrow)
```

可以将上面的代码重构为通用的max()子程序。变化点在group by的列号表和要max()的列号。作为练习。

本文中余下的其它聚集函数可以用类似的办法实现。

8.4 min()

8.5 avg()

8.6 count()

8.7 sum()

9 补缺失值

10 转置