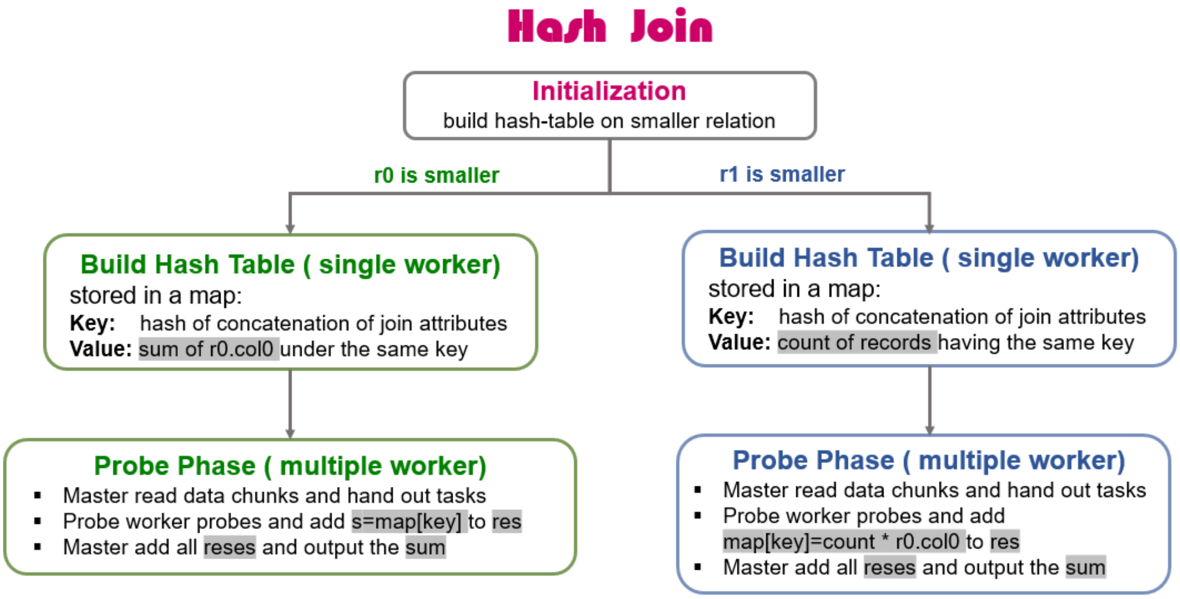
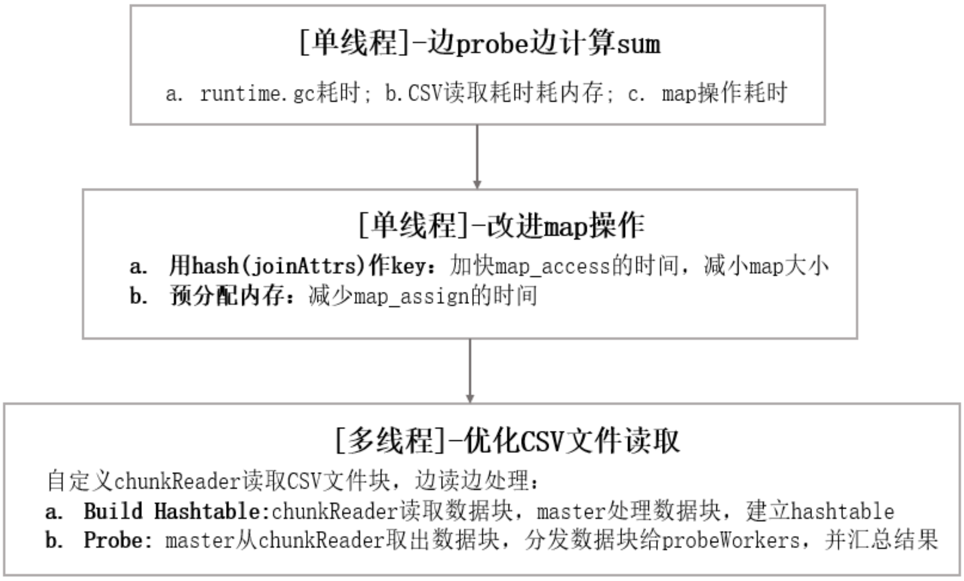
1. Ideas

考虑到MergeJoin要求排序，是全有序的，而HashJoin是先分类，缩小范围后再join。所以在Join Attribute无序的情况下，MergeJoin是要比HashJoin慢的。看了一下`t`文件夹下要排序的文件，第一列是依次递增的，之后的列是无序的。不知道之后正式测试的文件会是什么样，这里还是只考虑了HashJoin。基本思路如下图所示。


Hash Join
Initialization
build hash-table on smaller relation
r0 is smaller
r1 is smaller
Build Hash Table (single worker)
stored in a map:
Key: hash of concatenation of join attributes
Value: sum of r0.col0 under the same key
Build Hash Table (single worker)
stored in a map:
Key: hash of concatenation of join attributes
Value: count of records having the same key
Probe Phase (multiple worker)
Master read data chunks and hand out tasks
Probe worker probes and add s=map[key] to res
Master add all reses and output the sum
Probe Phase (multiple worker)
Master read data chunks and hand out tasks
Probe worker probes and add map[key]=count * r0.col0 to res
Master add all reses and output the sum

2. PProf

基本思路如下图所示。


[单线程]-边probe边计算sum
a. runtime.gc耗时; b.CSV读取耗时耗内存; c. map操作耗时
[单线程]-改进map操作
a. 用hash(joinAttrs)作key: 加快map_access的时间，减小map大小
b. 预分配内存: 减少map_assign的时间
[多线程]-优化CSV文件读取
自定义chunkReader读取CSV文件块，边读边处理:
a. Build Hashtable:chunkReader读取数据块，master处理数据块，建立hashtable
b. Probe: master从chunkReader取出数据块，分发数据块给probeWorkers，并汇总结果

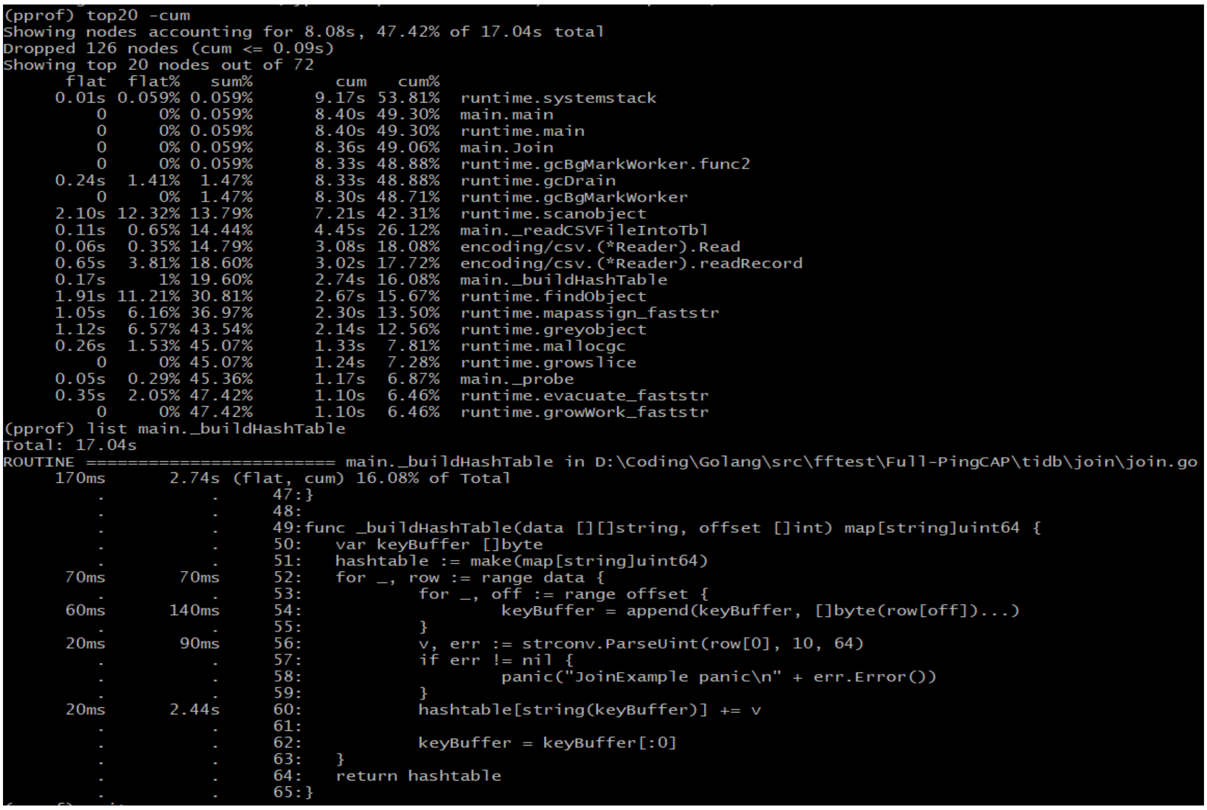
> Phase 1: [单线程]--边probe边计算sum

这一步相对于Example的优化主要体现在：

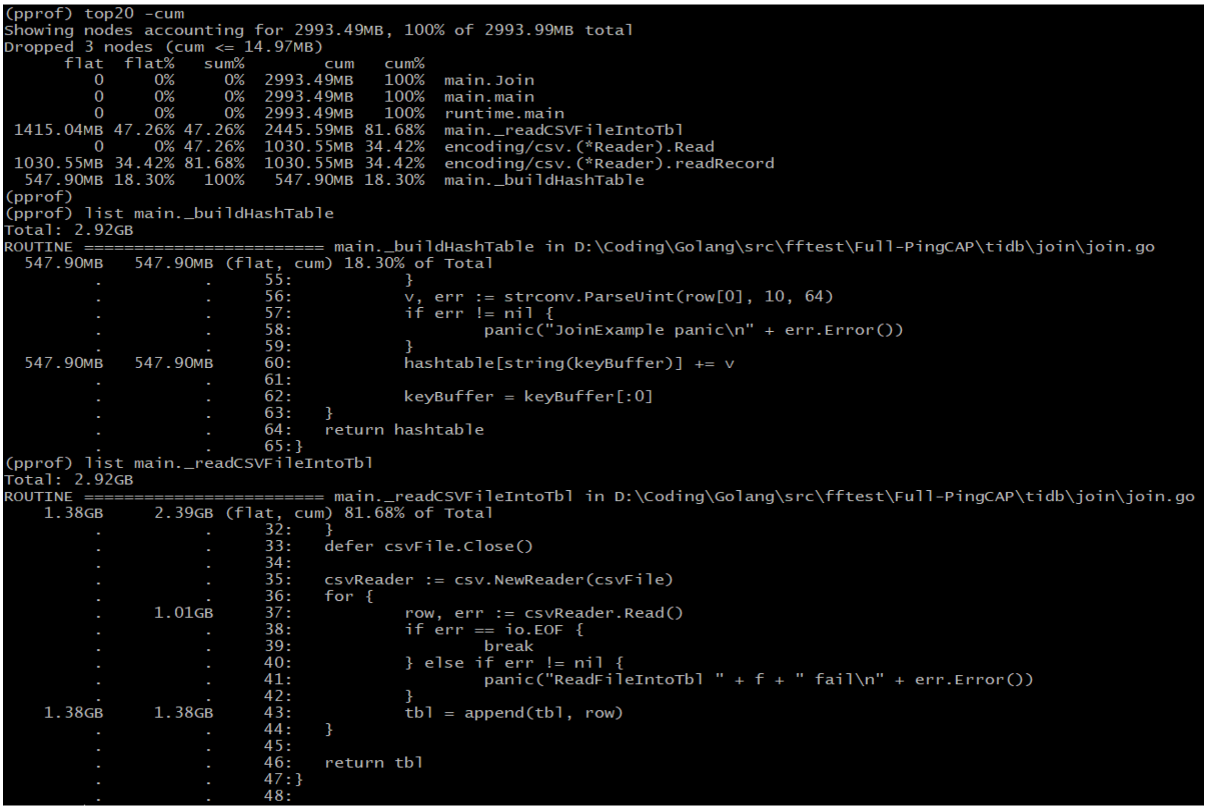
1. 减少了map的大小。原来在相同key下的map需要存相应row number的slice，现在只需要存这些row的col0的和。

2. 减少了probe后再用row number去提取相应col0求和的开销。

但是从pprof的结果来看，提升的空间还有很大： a. GC耗时太长；b. CSV读取操作耗时耗内存严重； c. map_assign和map_access耗时太长，map扩容重映射的情况也很严重。pprof CPU和pprof mem的结果分别如下图所示。



```
(pprof) top20 -cum
Showing nodes accounting for 8.08s, 47.42% of 17.04s total
Dropped 126 nodes (cum <= 0.09s)
Showing top 20 nodes out of 72
      flat  flat%   sum%        cum   cum%
     0.01s 0.059% 0.059%      9.17s 53.81%  runtime.systemstack
         0     0% 0.059%      8.40s 49.30%  main.main
         0     0% 0.059%      8.40s 49.30%  runtime.main
         0     0% 0.059%      8.36s 49.06%  main.Join
         0     0% 0.059%      8.33s 48.88%  runtime.gcBgMarkWorker.func2
     0.24s  1.41%  1.47%      8.33s 48.88%  runtime.gcDrain
         0     0%  1.47%      8.30s 48.71%  runtime.gcBgMarkWorker
     2.10s 12.32% 13.79%      7.21s 42.31%  runtime.scanobject
     0.11s  0.65% 14.44%      4.45s 26.12%  main._readCSVFileIntoTbl
     0.06s  0.35% 14.79%      3.08s 18.08%  encoding/csv.(*Reader).Read
     0.65s  3.81% 18.60%      3.02s 17.72%  encoding/csv.(*Reader).readRecord
     0.17s     1% 19.60%      2.74s 16.08%  main._buildHashTable
     1.91s 11.21% 30.81%      2.67s 15.67%  runtime.findObject
     1.05s  6.16% 36.97%      2.30s 13.50%  runtime.mapassign_faststr
     1.12s  6.57% 43.54%      2.14s 12.56%  runtime.greyobject
     0.26s  1.53% 45.07%      1.33s  7.81%  runtime.mallocgc
         0     0% 45.07%      1.24s  7.28%  runtime.growslice
     0.05s  0.29% 45.36%      1.17s  6.87%  main._probe
     0.35s  2.05% 47.42%      1.10s  6.46%  runtime.evacuate_faststr
         0     0% 47.42%      1.10s  6.46%  runtime.growWork_faststr
(pprof) list main._buildHashTable
Total: 17.04s
ROUTINE ======================== main._buildHashTable in D:\Coding\Golang\src\fftest\Full-PingCAP\tidb\join\join.go
     170ms      2.74s (flat, cum) 16.08% of Total
         .          .     47:}
         .          .     48:
         .          .     49:func _buildHashTable(data [][]string, offset []int) map[string]uint64 {
         .          .     50:   var keyBuffer []byte
         .          .     51:   hashtable := make(map[string]uint64)
      70ms       70ms     52:   for _, row := range data {
         .          .     53:           for _, off := range offset {
      60ms      140ms     54:                   keyBuffer = append(keyBuffer, []byte(row[off])...)
         .          .     55:           }
      20ms       90ms     56:           v, err := strconv.ParseUint(row[0], 10, 64)
         .          .     57:           if err != nil {
         .          .     58:                   panic("JoinExample panic\n" + err.Error())
         .          .     59:           }
      20ms      2.44s     60:           hashtable[string(keyBuffer)] += v
         .          .     61:
         .          .     62:           keyBuffer = keyBuffer[:0]
         .          .     63:   }
         .          .     64:   return hashtable
         .          .     65:}
```



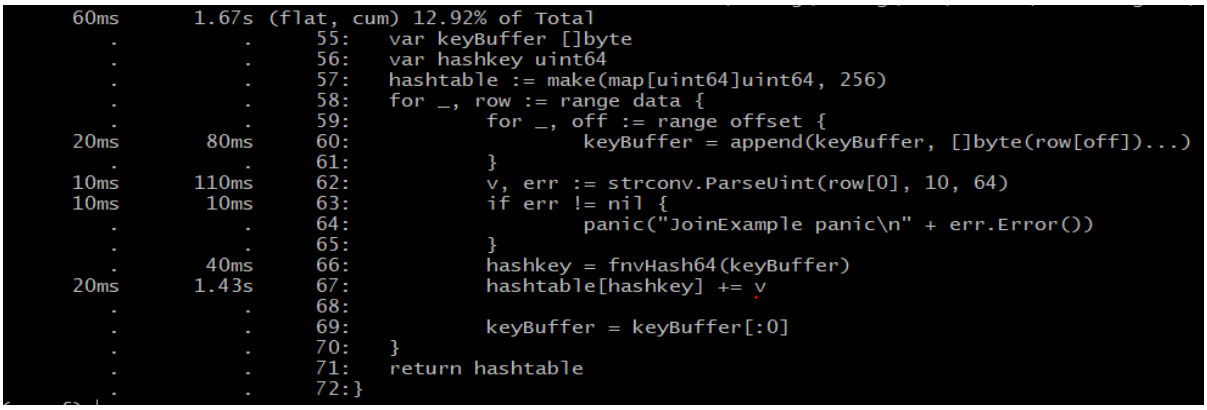
```
(pprof) top20 -cum
Showing nodes accounting for 2993.49MB, 100% of 2993.99MB total
Dropped 3 nodes (cum <= 14.97MB)
      flat  flat%   sum%        cum   cum%
         0     0%     0%  2993.49MB   100%  main.Join
         0     0%     0%  2993.49MB   100%  main.main
         0     0%     0%  2993.49MB   100%  runtime.main
 1415.04MB 47.26% 47.26%  2445.59MB 81.68%  main._readCSVFileIntoTbl
         0     0% 47.26%  1030.55MB 34.42%  encoding/csv.(*Reader).Read
 1030.55MB 34.42% 81.68%  1030.55MB 34.42%  encoding/csv.(*Reader).readRecord
  547.90MB 18.30%   100%   547.90MB 18.30%  main._buildHashTable
(pprof)
(pprof) list main._buildHashTable
Total: 2.92GB
ROUTINE ======================== main._buildHashTable in D:\Coding\Golang\src\fftest\Full-PingCAP\tidb\join\join.go
  547.90MB   547.90MB (flat, cum) 18.30% of Total
         .          .     55:           }
         .          .     56:           v, err := strconv.ParseUint(row[0], 10, 64)
         .          .     57:           if err != nil {
         .          .     58:                   panic("JoinExample panic\n" + err.Error())
         .          .     59:           }
  547.90MB   547.90MB     60:           hashtable[string(keyBuffer)] += v
         .          .     61:
         .          .     62:           keyBuffer = keyBuffer[:0]
         .          .     63:   }
         .          .     64:   return hashtable
         .          .     65:}
(pprof) list main._readCSVFileIntoTbl
Total: 2.92GB
ROUTINE ======================== main._readCSVFileIntoTbl in D:\Coding\Golang\src\fftest\Full-PingCAP\tidb\join\join.go
    1.38GB     2.39GB (flat, cum) 81.68% of Total
         .          .     32:   }
         .          .     33:   defer csvFile.Close()
         .          .     34:
         .          .     35:   csvReader := csv.NewReader(csvFile)
         .          .     36:   for {
         .     1.01GB     37:           row, err := csvReader.Read()
         .          .     38:           if err == io.EOF {
         .          .     39:                   break
         .          .     40:           } else if err != nil {
         .          .     41:                   panic("ReadFileIntoTbl " + f + " fail\n" + err.Error())
         .          .     42:           }
    1.38GB     1.38GB     43:           tbl = append(tbl, row)
         .          .     44:   }
         .          .     45:
         .          .     46:   return tbl
         .          .     47:}
         .          .     48:
```

> Phase 2: [单线程]--优化map操作

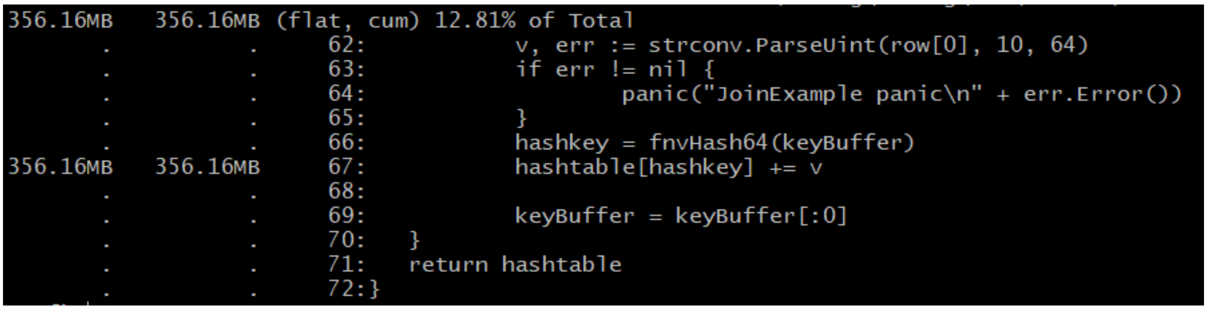
这一步相对于Example的优化主要体现在：

1. 用hash(joinAttrs) uint64类型, 替代joinAttrs string类型，作为map key以加速map_access，同时减小map的大小
2. 预分配map空间，以减少map扩容重映射的时间，减少map_assign的时间

pprof CPU和pprof mem的结果分别如下图所示，可以看到优化后的map耗时减少了1秒左右，内存占用也减少了200M左右。



```
      60ms      1.67s (flat, cum) 12.92% of Total
         .          .     55:    var keyBuffer []byte
         .          .     56:    var hashkey uint64
         .          .     57:    hashtable := make(map[uint64]uint64, 256)
         .          .     58:    for _, row := range data {
         .          .     59:            for _, off := range offset {
      20ms       80ms     60:                    keyBuffer = append(keyBuffer, []byte(row[off])...)
         .          .     61:            }
      10ms      110ms     62:            v, err := strconv.ParseUint(row[0], 10, 64)
      10ms       10ms     63:            if err != nil {
         .          .     64:                    panic("JoinExample panic\n" + err.Error())
         .          .     65:            }
         .       40ms     66:            hashkey = fnvHash64(keyBuffer)
      20ms      1.43s     67:            hashtable[hashkey] += v
         .          .     68:
         .          .     69:            keyBuffer = keyBuffer[:0]
         .          .     70:    }
         .          .     71:    return hashtable
         .          .     72:}
```



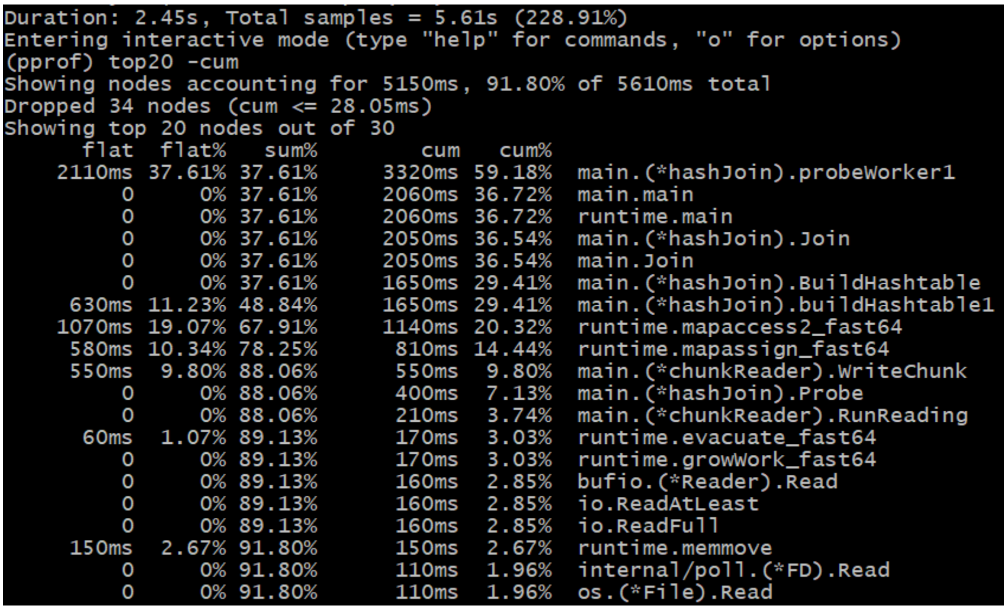
```
356.16MB   356.16MB (flat, cum) 12.81% of Total
       .          .     62:            v, err := strconv.ParseUint(row[0], 10, 64)
       .          .     63:            if err != nil {
       .          .     64:                    panic("JoinExample panic\n" + err.Error())
       .          .     65:            }
       .          .     66:            hashkey = fnvHash64(keyBuffer)
356.16MB   356.16MB     67:            hashtable[hashkey] += v
       .          .     68:
       .          .     69:            keyBuffer = keyBuffer[:0]
       .          .     70:    }
       .          .     71:    return hashtable
       .          .     72:}
```

Phase 3: [多线程]--优化CSV文件读取

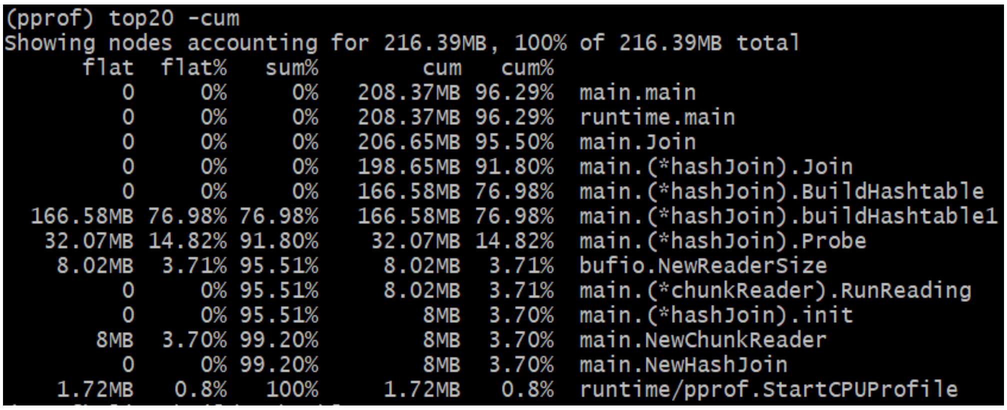
从之前的pprof结果可以看出：使用`csv`包一行一行地把数据读到一个slice中的方式，在时间上`csv/decoding`十分耗时，在空间上存放所有数据的slice开销巨大。于是决定放弃使用`csv`包，同时为了节省时间，采用读数据和处理数据同步进行的方式。程序中一共有三个角色：`chunkReader`、`master`、`worker`。其中`chunkReader`负责读取数据块，当chunkReader读完一个数据块后会通知master有新的数据块，`master`收到通知后，再去把新的数据块取过来，然后告诉chunkReader可以读下一个数据块了，并同时将新的数据块封装成task分发给worker处理。`worker`处理完自己的数据块后，将结果返还给master，由master汇总输出。处理思路大致如下：

1. `Build Hashtable`. chunkReader负责读数据块，master接收数据块并且处理数据块，建立hashtable。为了内存复用，master有一个dataChunk缓冲区，每次都将从chunkReader取下来的chunk放到这里
2. `Probe`. master负责从chunkReader那里取chunk，并将chunk分发给probeWorker。为了内存复用，master有一个[]dataChunk的缓冲区，在分发task给probeWorker时，会在task中指明该task对应的chunk在[]dataChunk中的index，当probeWorker处理完task后，会将这个index和处理结果一起返还给master，告诉master `dataChunk[index]`这个chunk可用了。如果chunkReader读数据比较快，而[]dataChunk中没有可用的chunk了，master才会增大[]dataChunk的空间。

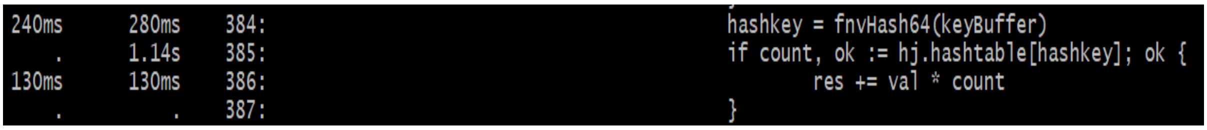
pprof CPU和pprof mem的结果分别如下图所示。可以看出，时间上相比于example优化了4倍左右，空间上优化了10倍左右。但最后map操作还是成为了系统的瓶颈。



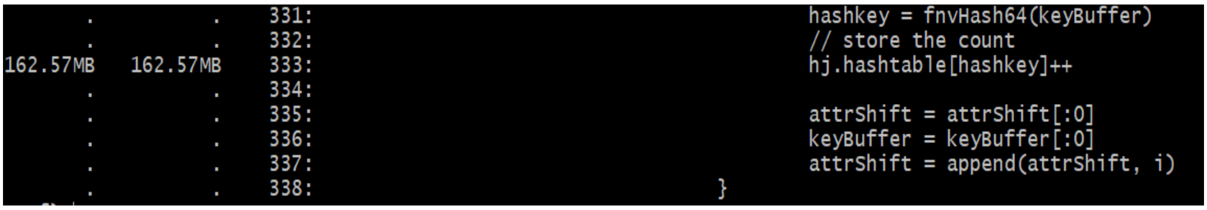
```
Duration: 2.45s, Total samples = 5.61s (228.91%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 5150ms, 91.80% of 5610ms total
Dropped 34 nodes (cum <= 28.05ms)
Showing top 20 nodes out of 30
      flat  flat%   sum%        cum   cum%
    2110ms 37.61% 37.61%     3320ms 59.18%  main.(*hashJoin).probeWorker1
         0     0% 37.61%     2060ms 36.72%  main.main
         0     0% 37.61%     2060ms 36.72%  runtime.main
         0     0% 37.61%     2050ms 36.54%  main.(*hashJoin).Join
         0     0% 37.61%     2050ms 36.54%  main.Join
         0     0% 37.61%     1650ms 29.41%  main.(*hashJoin).BuildHashtable
     630ms 11.23% 48.84%     1650ms 29.41%  main.(*hashJoin).buildHashtable1
    1070ms 19.07% 67.91%     1140ms 20.32%  runtime.mapaccess2_fast64
     580ms 10.34% 78.25%      810ms 14.44%  runtime.mapassign_fast64
     550ms  9.80% 88.06%      550ms  9.80%  main.(*chunkReader).WriteChunk
         0     0% 88.06%      400ms  7.13%  main.(*hashJoin).Probe
         0     0% 88.06%      210ms  3.74%  main.(*chunkReader).RunReading
      60ms  1.07% 89.13%      170ms  3.03%  runtime.evacuate_fast64
         0     0% 89.13%      170ms  3.03%  runtime.growWork_fast64
         0     0% 89.13%      160ms  2.85%  bufio.(*Reader).Read
         0     0% 89.13%      160ms  2.85%  io.ReadAtLeast
         0     0% 89.13%      160ms  2.85%  io.ReadFull
     150ms  2.67% 91.80%      150ms  2.67%  runtime.memmove
         0     0% 91.80%      110ms  1.96%  internal/poll.(*FD).Read
         0     0% 91.80%      110ms  1.96%  os.(*File).Read
```



```
(pprof) top20 -cum
Showing nodes accounting for 216.39MB, 100% of 216.39MB total
      flat  flat%   sum%        cum   cum%
         0     0%     0%   208.37MB 96.29%  main.main
         0     0%     0%   208.37MB 96.29%  runtime.main
         0     0%     0%   206.65MB 95.50%  main.Join
         0     0%     0%   198.65MB 91.80%  main.(*hashJoin).Join
         0     0%     0%   166.58MB 76.98%  main.(*hashJoin).BuildHashtable
  166.58MB 76.98% 76.98%   166.58MB 76.98%  main.(*hashJoin).buildHashtable1
   32.07MB 14.82% 91.80%    32.07MB 14.82%  main.(*hashJoin).Probe
    8.02MB  3.71% 95.51%     8.02MB  3.71%  bufio.NewReaderSize
         0     0% 95.51%     8.02MB  3.71%  main.(*chunkReader).RunReading
         0     0% 95.51%        8MB  3.70%  main.(*hashJoin).init
       8MB  3.70% 99.20%        8MB  3.70%  main.NewChunkReader
         0     0% 99.20%        8MB  3.70%  main.NewHashJoin
    1.72MB   0.8%   100%     1.72MB   0.8%  runtime/pprof.StartCPUProfile
```



```
240ms      280ms    384:                    hashkey = fnvHash64(keyBuffer)
    .      1.14s    385:                    if count, ok := hj.hashtable[hashkey]; ok {
130ms      130ms    386:                            res += val * count
    .          .    387:                    }
```



```
        .          .    331:                    hashkey = fnvHash64(keyBuffer)
        .          .    332:                    // store the count
162.57MB   162.57MB    333:                    hj.hashtable[hashkey]++
        .          .    334:
        .          .    335:                    attrShift = attrShift[:0]
        .          .    336:                    keyBuffer = keyBuffer[:0]
        .          .    337:                    attrShift = append(attrShift, i)
        .          .    338:            }
```