Stat 243: Problem Set 1

Eugene Yedvabny

09/22/2014

Question 1

The following code is very straighforward and god-awfully slow. For whichever reason, R's line-by-line reading of bz2 file is an oder of magnitude slower than Python's or compiled C++'s. The code reads in the file line-by-line and copies the line into the appropriate output if it matches the filtering column. The output filenames are taken from the values of the subsetting column.

I tried to do error sanity checking to maintain robustness, but I mostly stuck to the defined interface from the Piazza description. E.g. subsetting and filtering columns will be provided as column indices, not names, and the zipped csv will be on disk. There is very little memory requirement, as the code reads one line at a time. But if we assume file I/O is the bottleneck, this line-by-line reading carries a steep price. I've tried chunked reads (100000 lines at a time), but they don't speed up things considerably while adding extra processing logic.

In order to avoid too much I/O with constant file writing, I open a new output file for every *unique* key within the subsetting column. To my surprise the R lists act like hash-tables, so arbitrary strings can be used as persisten list keys. After all reading and writing is complete, I close all connections at once using the built-in closeAllConnections() function.

In conclusion, file reads in R are frustratingly slow.

```
cat ZipChomp.R
```

```
#!/usr/bin/env RScript

# An RScript to subset and filter a bz-ipped CSV
# Inputs: CSV name, column index for strat, column index for filter, filter key
# Outputs: bz-ipped files for each element of the subsetting column
#
# CC Eugene Yedvabny, 2014

# Capture the provided arguments
args <- commandArgs(TRUE)

# Sanity check that we have enough arguments
if(length(args) < 4){
    stop("Please provide the bz2 file name, stratifying column, subsetting column and desired subsetting
}

# Assign the variables
input.name <- args[1]
strat.col <- as.numeric(args[2])
sort.col <- as.numeric(args[3])
sort.value <- args[4]

# Checking that file exists
# Unfortunately no checks for correct bz type
if (!file.exists(input.name)){
    stop("Specified file doesn't exist")
}

# Open the input file for reading
input.handle <- bzfile(args[1], "r")

# Count the number of fields
input.header <- readLines(input.handle, n=1)
input.cols <- strsplit(input.header,",")[[1]]
if(strat.col < 0 || strat.col > length(input.cols) || sort.col < 0 || sort.col > length(input.cols)){
    close(input.handle)
    stop("Please use column indices within bounds of the input file")
}

# Create a list of output file handles
# it will be keyed by unique values of strat.col
output.files <- list()

# Read the rest of the file line by line appending to right section
input.length <- 0
while(length(input.line <- readLines(input.handle,1))) {

    input.fields <- strsplit(input.line,",")[[1]]

    # Only stratify rows which match the sorting value
    if(input.fields[sort.col] == sort.value){

        # Open the output file if one is not open yet
        # Otherwise create a new file handle
        output.key <- input.fields[strat.col]
        output.file <- output.files[[output.key]]
        if(is.null(output.file)) {
            output.file.name <- paste(output.key,".csv.bz2",sep="")
            output.file <- bzfile(output.file.name,'w')
            output.files[[output.key]] <- output.file
        }

        # Write the filtered line into the now-open file
        write(input.line,output.file)
    }

    input.length <- input.length + 1
}

# Indicate that we're done processing
write(sprintf("Done: processed %d lines (excluding the header)",input.length),stdout())

# Close all the files
closeAllConnections()
```

Question 2

During the evaluation of the first code chunk, `myFuns` is assigned a list of three anonymous *argument-less* functions that will return the value of `i` when executed. The functions are all defined in the scope of the global environment, so since they do not take `i` as an argument, they will search for the value of `i` in their parent (e.g. global) environment. Since `i` was used as the index variable for setting myFuns, after execution it is valued at **3**.

The first round of execution uses `j` as the index variable. `i` is still **3** in the global environment, so every call to an element of myFuns pulls in that value, results in **3** printed three times.

The second evaluation now uses `i` as the index value. Since `i` was already part of the global environment and the execution is *taking place in* the global enviroment, the value of `i` is reset at every iteration and then pulled into the executing element of `myFuns`. This results in the observed sequence of **1**, **2**, **3**.

In the case of the function generator, the list that becomes `myFuns` is defined in the environment of `funGenerator`. The environment is transient but since the generated list is assigned to `myFuns` the environment persists for the lifetime of the list. `i` was used as the indexing variable in the scope of the generating environment and is set to **3** at the end of the inner for-loop. When elements of `myFuns` are evaluated, the anonymous function finds `i` set to **3** in its nearest parent environment, hence the invariant return value.

Question 3

The code is executing four nested function, resulting in a five-frame call stack (if we include 0th global frame): global -> `sapply` -> `anon function` -> `ls` -> `sys.frame`. Since `ls` is the third function in the sequence, it can't actually access `sys.frame`'s stack frame, hence the indexing from 0. The `sapply` function executes the anonymous function on each element of the 0:3 vector, passing the element as the argument `x`. `ls` prints out the variables within an environment designated by *envir* argument, which in turn is set by `sys.frame`. As x varies, ls prints out the variables of the next stack frame (up to it's own, **3**).

In that context, the first entry of the sapply result is a list of all variables in *frame 0*, i.e. the *global environment*. The next entry lists the members of *frame 1*: the environment of `sapply`. This is followed by members of *frame 2* - the environment of the anonymous function - and elements of *frame 3* - the environment of `ls` itself.

When executed from within RStudio or just the R shell, the stack and elements within each frame are very sparse. *frame 0* is empty; *frame 1* has the implicit parameters of `sapply` (`simplify` and `USE.NAMES`) and the provided parameters `X` (capitalized, so I am assuming that's a placeholder for how R refers to the first argument) and `FUN` (the anonymous function); *frame 2* has just `FUN` and `X` (passed in from the calling environment); *frame 3* has `x` set to **3**, which is the last value of the original vector.

Now, for me the interesting discovery within this question was the difference between the global environment of RStudio and the global environment of knitr. In contrast to the empty global env of R/RStudio, the knitr environment seems to manually import a lot of functions into the calling stack. Somethign I should definitely be aware of in case of any name-space conflicts.