Efficient accumulation in R

July 27, 2015 John Mount Coding, Statistics, Tutorials R, R as it is

R has a number of very good packages for manipulating and aggregating data (dplyr, sqldf, ScaleR, data.table, and more), but when it comes to accumulating results the beginning R user is often at sea. The R execution model is a bit exotic so many R users are very uncertain which methods of accumulating results are efficient and which are inefficient.

Accumulating wheat (Photo: Cyron Ray Macey, some rights reserved)

In this latest "R as it is" (again in collaboration with our friends at Revolution Analytics) we will quickly become expert at efficiently accumulating results in R.

A number of applications (most notably simulation) require the incremental accumulation of results prior to processing. For our example, suppose we want to collect rows of data one by one into a data frame. Take the `mkRow` function below as a simple example source that yields a row of data each time we call it.

```
mkRow <- function(nCol) {
  x <- as.list(rnorm(nCol))
  # make row mixed types by changing first column to string
  x[[1]] <- ifelse(x[[1]]>0,'pos','neg')
  names(x) <- paste('x',seq_len(nCol),sep='.')
  x
}
```

The obvious "`for`-loop" solution is to collect or accumulate many rows into a data frame by repeated application of `rbind`. This looks like the following function.

```
mkFrameForLoop <- function(nRow,nCol) {
  d <- c()
  for(i in seq_len(nRow)) {
    ri <- mkRow(nCol)
    di <- data.frame(ri,
                     stringsAsFactors=FALSE)
    d <- rbind(d,di)
  }
  d
}
```

This would be the solution most familiar to many non-R programmers. The problem is: in R the above code is incredibly slow.

In R all common objects are usually immutable and can not change. So when you write an assignment like "`d <- rbind(d,di)`" you are *usually* not actually adding a row to an existing data frame, but constructing a new data frame that has an additional row. This new data frame replaces your old data frame in your current execution environment (R execution environments are mutable, to implement such changes). This means to accumulate or add `n` rows incrementally to a data frame, as in `mkFrameForLoop` we actually build `n` different data frames of sizes `1,2,...,n`. As we do work copying each row in each data frame (since in R data frame columns can potentially be shared, but not rows) we pay the cost of processing `n*(n+1)/2` rows of data. So: *no matter how expensive creating each row is,* for large enough `n` the time wasted re-allocating rows (again and again) during the repeated `rbind`s eventually dominates the calculation time. For large enough `n` you are wasting most of your time in the repeated `rbind` steps.

To repeat: it isn't just that accumulating rows one by one is "a bit less efficient than the right way for R". Accumulating rows one by one becomes arbitrarily slower than the right way (which should only need to manipulate `n` rows to collect `n` rows into a single data frame) as `n` gets large. Note: it isn't that beginning R programmers don't know what they are doing; it is that they are designing to the reasonable expectation that data frame is row-oriented and R objects are mutable. The fact is R data frames are column oriented and R structures are largely immutable (despite the syntax appearing to signal the opposite), so the optimal design is not what one might expect.

Given this how does anyone ever get real work done in R? The answers are:

- Experienced R programmers avoid the `for`-loop structure seen in `mkFrameForLoop`.
- In some specialized situations (where value visibility is sufficiently limited) R can avoid a number of the unnecessary user specified calculations by actual in-place mutation (which means R can in some cases change things when nobody is looking, so only *observable* object semantics are truly immutable).

The most elegant way to avoid the problem is to use R's `lapply` (or list apply) function as shown below:

```
mkFrameList <- function(nRow,nCol) {
  d <- lapply(seq_len(nRow),function(i) {
    ri <- mkRow(nCol)
    data.frame(ri,
               stringsAsFactors=FALSE)
  })
  do.call(rbind,d)
}
```

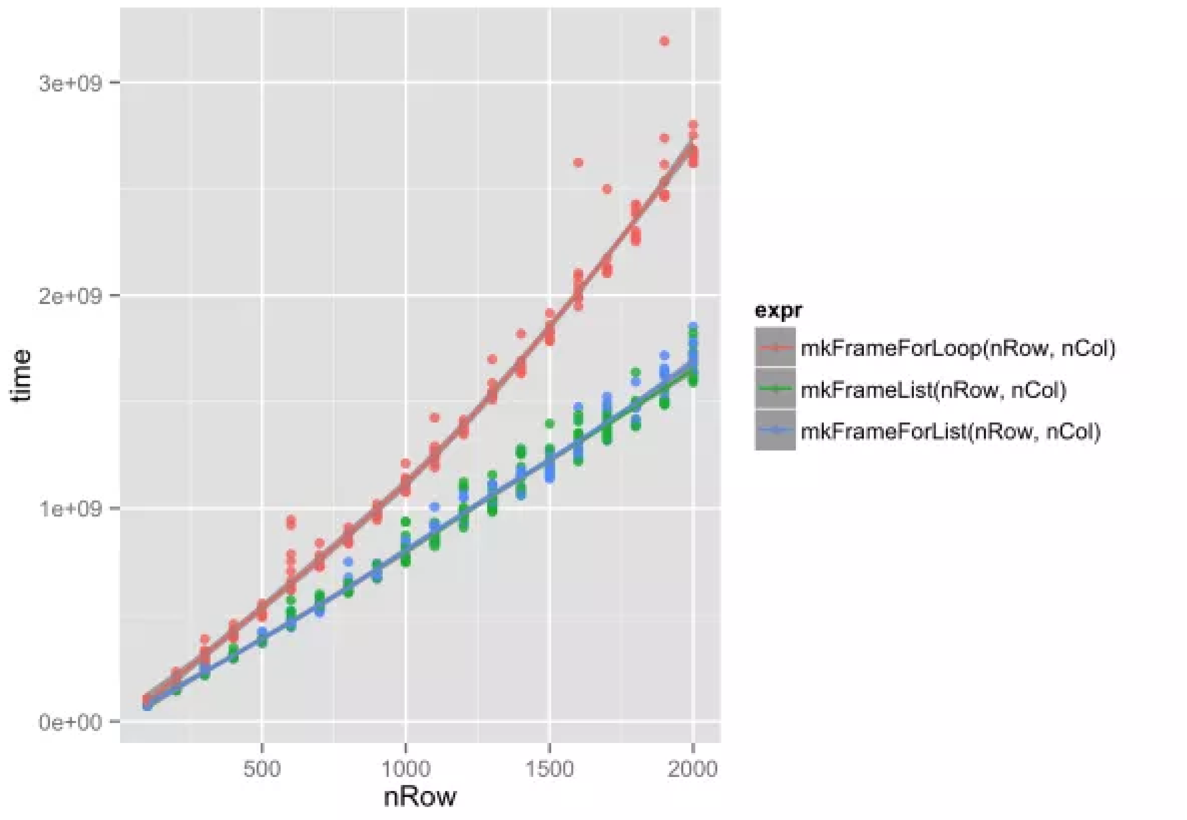
What we did is take the contents of the `for`-loop body, and wrap them in a function. This function is then passed to `lapply` which creates a list of rows. We then batch apply `rbind` to these rows using `do.call`. It isn't that the `for`-loop is slow (which many R users mistakingly believe), it is the *incremental* collection of results into a data frame is slow and that is one of the steps the `lapply` method is avoiding. While you can prefer `lapply` to `for`-loops always for stylistic reasons, it is important to understand when `lapply` is in fact quantitatively better than a `for`-loop (and to know when a `for`-loop is in fact acceptable). In fact a for-loop with a better binder such as `data.table::rbindlist` (assuming your code can work with a `data.table` which in some environments has different semantics) is among the fastest variations we have seen (as suggested by Arun Srinivasan in the comments below; another top contender are file based Split-Apply-Combine methods as suggested in comments by David Hood, ideas also seen in Map-

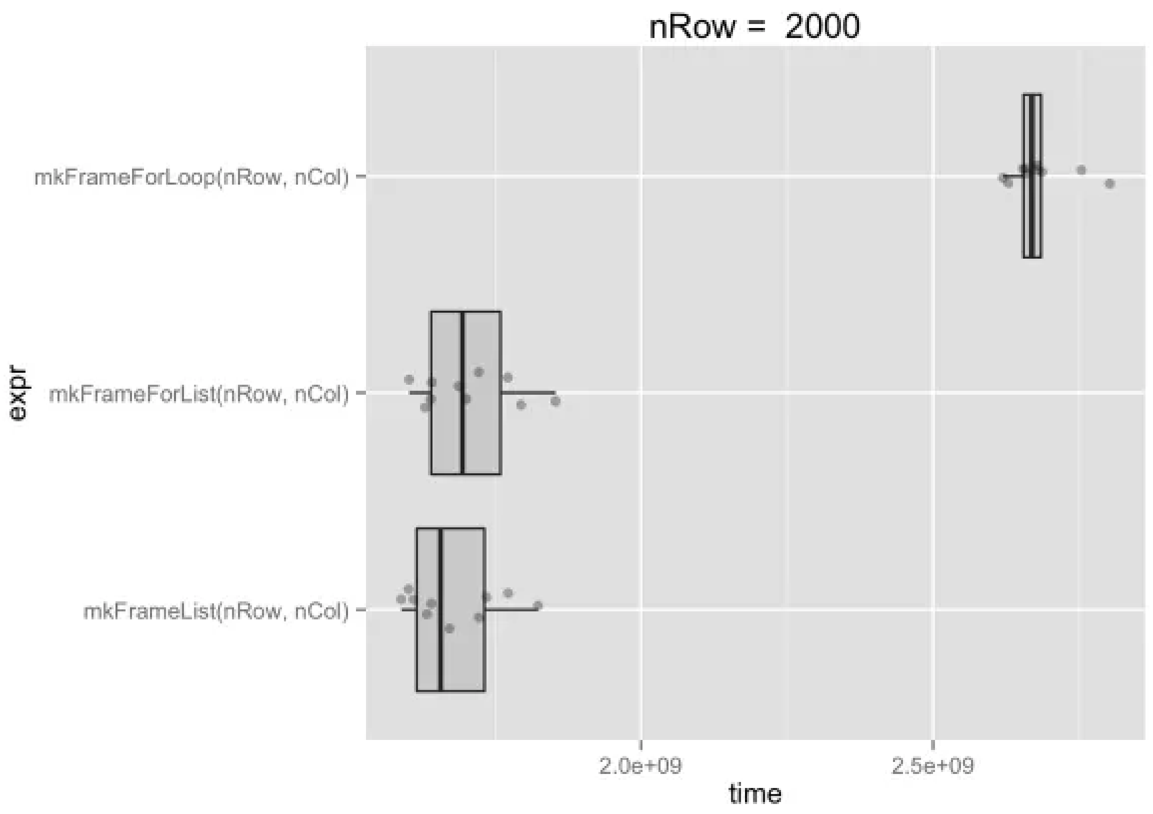
Reduce).

If you don't want to learn about `lapply` you can write fast code by collecting the rows in a list as below.

```
mkFrameForList <- function(nRow,nCol) {
  d <- as.list(seq_len(nRow))
  for(i in seq_len(nRow)) {
    ri <- mkRow(nCol)
    di <- data.frame(ri,
                     stringsAsFactors=FALSE)
    d[[i]] <- di
  }
  do.call(rbind,d)
}
```

The above code still uses a familiar `for`-loop notation and is in fact fast. Below is a comparison of the time (in MS) for each of the above algorithms to assemble data frames of various sizes. The quadratic cost of the first method is seen in the slight upward curvature of its smoothing line. Again, to make this method truly fast replace `do.call(rbind,d)` with `data.table::rbindlist(d)` (examples here).



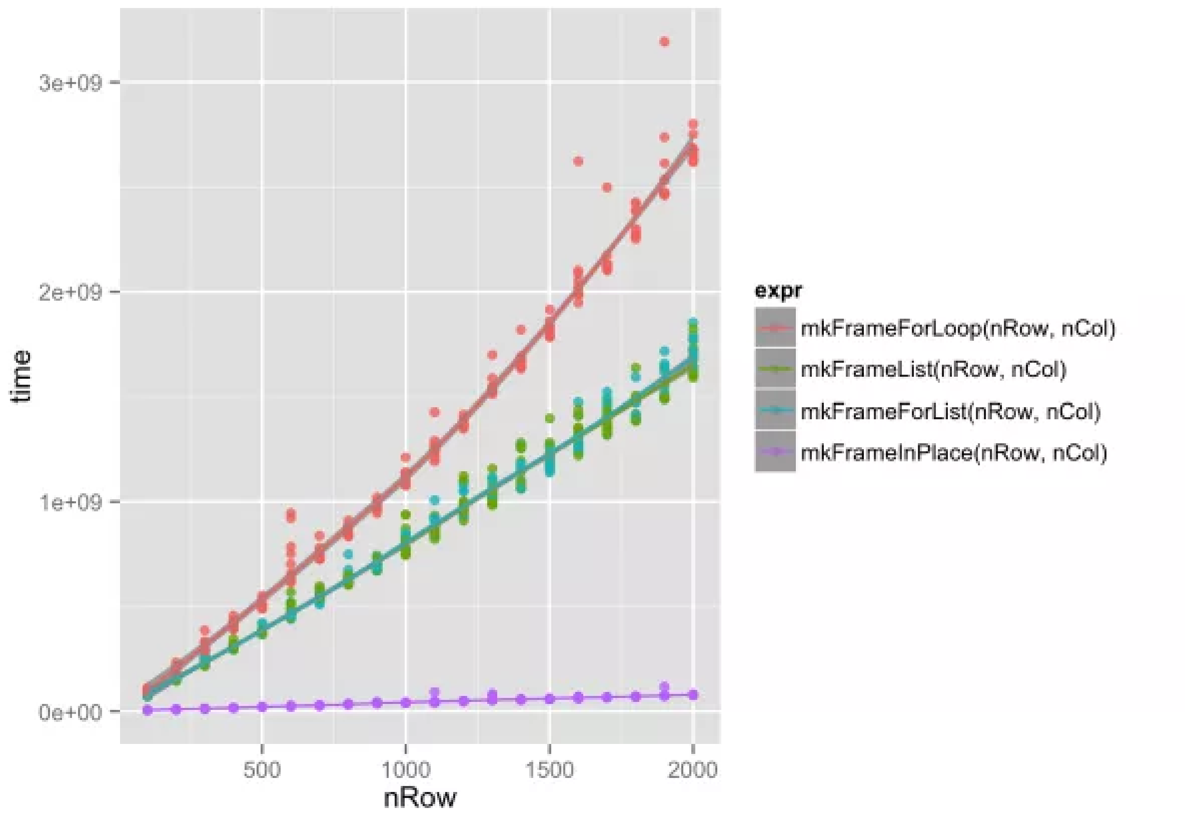


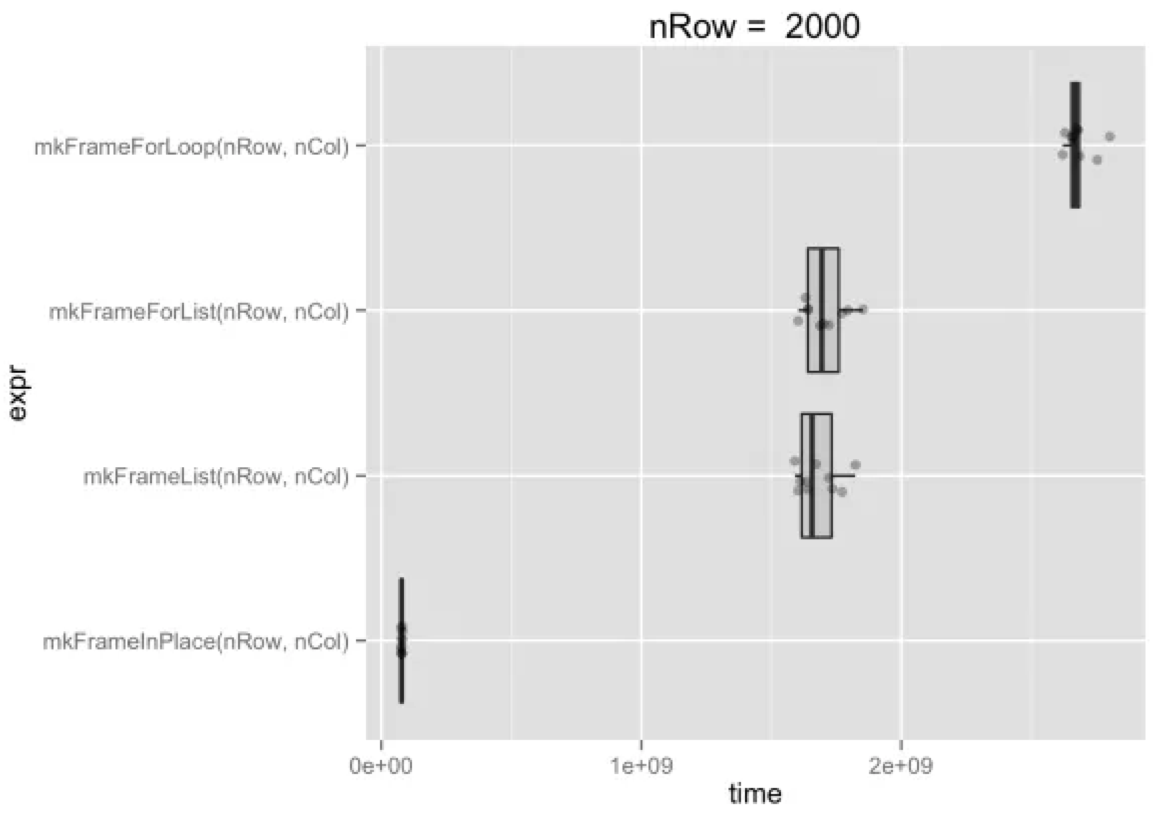
Execution time (MS) for collecting a number of rows (x-axis) for each of the three methods discussed. Slowest is the incremental for-loop accumulation.

The reason `mkFrameForList` is tolerable is in some situations R can avoid creating new objects and in fact manipulate data in place. In this case the list "`d`" is not in fact re-created each time we add an additional element, but in fact mutated or changed in place.

(edit) The common advice is we should prefer in-place edits. We tried that, but it wasn't until we (after getting feedback in our comments below) threw out the data frame class attribute that we got really fast code. The code and latest run are below (but definitely check out the comments following this article for the reasoning chain).

```
mkFrameInPlace <- function(nRow,nCol,classHack=TRUE) {
  r1 <- mkRow(nCol)
  d <- data.frame(r1,
                  stringsAsFactors=FALSE)
  if(nRow>1) {
    d <- d[rep.int(1,nRow),]
    if(classHack) {
      # lose data.frame class for a while
      # changes what S3 methods implement
      # assignment.
      d <- as.list(d)
    }
    for(i in seq.int(2,nRow,1)) {
      ri <- mkRow(nCol)
      for(j in seq_len(nCol)) {
        d[[j]][i] <- ri[[j]]
      }
    }
  }
  if(classHack) {
     d <- data.frame(d,stringsAsFactors=FALSE)
  }
  d
}
```





More timings.

Note that the in-place list of vectors method is faster than any of `lapply/do.call(rbind)`, `dplyr::bind_rows/replicate`, or `plyr::ldply`. This is despite having nested for-loops (one for rows, one for columns; though this is also why methods of this type can speed up even more if we use `compile:cmpfun`). At this point you should see: it isn't the for-loops that are the problem, it is any sort of incremental allocation, re-allocation, and checking.

At this point we are avoiding both the complexity waste (running an algorithm that takes time proportional to the square of the number of rows) and avoiding a lot of linear waste (re-allocation, type-checking, and name matching).

However, any in-place change (without which the above code would again be unacceptably slow) depends critically on the list value associated with "`d`" having very limited visibility. Even copying this value to another variable or passing it to another function can break the visibility heuristic and cause arbitrarily expensive object copying.

The fragility of the visibility heuristic is best illustrated with an even simpler example.

Consider the following code that returns a vector of the squares of the first `n` positive integers.

```
computeSquares <- function(n,messUpVisibility) {
  # pre-allocate v
  # (doesn't actually help!)
  v <- 1:n
  if(messUpVisibility) {
     vLast <- v
  }
  # print details of v
  .Internal(inspect(v))
  for(i in 1:n) {
    v[[i]] <- i^2
    if(messUpVisibility) {
      vLast <- v
    }
    # print details of v
    .Internal(inspect(v))
  }
  v
}
```

Now of course part of the grace of R is we never would have to write such a function. We could do this very fast using vector notation such as `seq_len(n)^2`. But let us work with this notional example.

Below is the result of running `computeSquares(5,FALSE)`. In particular look at the lines printed by the `.Internal(inspect(v))` statements and at the first field of these lines (which is the address of the value "`v`" refers to).

```
computeSquares(5,FALSE)

## @7fdf0f2b07b8 13 INTSXP g0c3 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
## @7fdf0e2ba740 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
## @7fdf0e2ba740 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,3,4,5
## @7fdf0e2ba740 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,4,5
## @7fdf0e2ba740 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,5
## @7fdf0e2ba740 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,25

## [1]  1  4  9 16 25
```

Notice that the address `v` refers to changes only once (when the value type changes from integer to real). After the one change the address remains constant (`@7fdf0e2ba740`) and the code runs fast as each pass of the for-loop alters a single position in the value referred to by `v` without any object copying.

Now look what happens if we re-run with `messUpVisibility`:

```
computeSquares(5,TRUE)

## @7fdf0ec410e0 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5
## @7fdf0d9718e0 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,2,3,4,5
## @7fdf0d971bb8 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,3,4,5
## @7fdf0d971c88 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,4,5
## @7fdf0d978608 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,16,5
## @7fdf0d9788e0 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,16,25

## [1]  1  4  9 16 25
```

Setting `messUpVisibility` causes the value referenced by “`v`” to also be referenced by a new variable named “`vLast`“. Evidently this small change is enough to break the visibility heuristic as we see the address of the value “`v`” refers to changes after each update. Meaning we have triggered a lot of expensive object copying. So we should consider the earlier `for`-loop code a bit fragile as small changes in object visibility can greatly change performance.

The thing to remember is: for the most part R objects are immutable. So code that appears to alter R objects is often actually simulating mutation by expensive copies. This is the concrete reason that functional transforms (like `lapply`) should be preferred to incremental or imperative appends.

R code for all examples in this article can be found here (this includes methods like pre-reserving space, the original vector experiments that originally indicated the object mutation effect), and a library wrapping the efficient incremental collector.

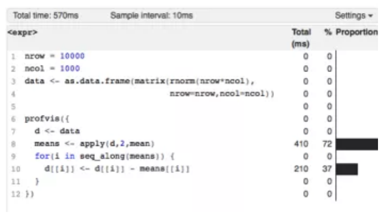
SHARE THIS:

5 7

RELATED

R style tip: prefer functions that return data frames
While following up on Nina Zumel's excellent Trimming the Fat from glm() Models in R I got to thinking about code style in R. And I realized: you can make

R annoyances
Readers returning to our blog will know that Win-Vector LLC is fairly "pro-R." You can take that to mean "in favor or R" or "professionally using R" (both
In "Coding"



```
nrow = 10000                                             0    0
ncol = 1000                                              0    0
data <- as.data.frame(matrix(rnorm(nrow*ncol),           0    0
                             nrow=nrow,ncol=ncol))       0    0
                                                         0    0
profvis({                                                0    0
  d <- data                                              0    0
  means <- apply(d,2,mean)                             410   72
  for(i in seq_along(means)) {                           0    0
    d[[i]] <- d[[i]] - means[[i]]                      210   37
  }                                                      0    0
})                                                       0    0
```

Shiny Developer Conference

In "Practical Data Science" In "Exciting Techniques"

July 27, 2015 John Mount Coding, Statistics, Tutorials R, R as it is

16 thoughts on “Efficient accumulation in R”

kasterma

July 28, 2015 at 5:56 am

My google-fu has not been enough to get more information on the notion of visibility you mention. I was fully unaware of the fact the R interpreter does this kind of analysis; do you have some references for finding out more?

jmount

July 28, 2015 at 6:20 am

Kasterma, I hadn’t found any such documentation on value visibility. But you can see it happening in the `computeSquares` code: the object value changes but the address remains constant. This is definitely in contrast to R’s usual observable immutable object behavior (which is also under documented, but here is one note on mutability/immutability: http://vita.had.co.nz/papers/mutatr.pdf).

Having to dig around a bit for what is really going on is a bit of why this series is named “R as it is.”

kasterma

July 28, 2015 at 6:26 am

I see it, and it is definitely a cool and interesting behaviour (that therefore I’d like to understand better).

jmount

July 28, 2015 at 7:20 am

Ah, gotcha. I have added a link to my original vector timing experiments that originally indicated the effect. I ran into it while preparing a corporate training course (in R and data science) for Salesforce. The timings are much more dramatic for vectors (including showing who much faster a fully vectorized method is than anything), but I felt a vector example is a bit artificial (that true result accumulation tends to be full rows not mere simple values).

(edit, good questions you got me wondering- I think I found it)

(some explanation: http://adv-r.had.co.nz/memory.html#modification)

Finding the source for assignment (and evidence it can re-use variables).

Following the note at http://stackoverflow.com/questions/19226816/how-can-i-view-the-source-code-for-a-function we visit http://svn.r-project.org/R/trunk/src/main/names.c and look for the string "[[<-". This is bound to a function called "do_subassign2". This brings us to http://svn.r-project.org/R/trunk/src/main/subassign.c .

And here is some code of the right flavor:

```
/* Ensure that the LHS is a local variable. */
   /* If it is not, then make a local copy. */

   if (MAYBE_SHARED(x))
   SETCAR(args, x = shallow_duplicate(x));
```

Git clone Winston Chang's copy of the repo for more search:

```
git clone https://github.com/wch/r-source.git
```

Look for MAYBE_SHARED:

```
cd r-source
git grep MAYBE_SHARED
```

Ah "src/include/Rinternals.h".

Also it looks like some sort of name count, ref-count, or local controlled duplication has been in the assign code since at least 1998 (maybe always).

David Hood

July 28, 2015 at 8:35 pm

The thing is, the memory allocation is the aggregating with rbind is in itself a huge bottleneck so you can effectively get an order of magnitude saving in time (and the larger the set of data the more significant the save) by going around the problem entirely. Consider this quick and dirty modification of your code:

```
mkFrameForLoop <- function(nRow,nCol) {
  d <- c()
  for(i in seq_len(nRow)) {
    ri <- mkRow(nCol)
    di <- data.frame(ri,
                     stringsAsFactors=FALSE)
    d <- rbind(d,di)
  }
  d
}

mkFrameList <- function(nRow,nCol) {
  d <- lapply(seq_len(nRow),function(i) {
    ri <- mkRow(nCol)
    data.frame(ri,
               stringsAsFactors=FALSE)
  })
  do.call(rbind,d)
}

mkRow2 <- function(nCol, fileloc) {
  x <- as.list(rnorm(nCol))
  # make row mixed types by changing first column to string
  x[[1]] <- ifelse(x[[1]]>0,'pos','neg')
  writeLines(paste(x, collapse=” ”), con = fileloc)
}

avoidAggregate <- function(nRow,nCol) {
  some.file <- tempfile()
  file.create(some.file)
  cf <- file(some.file, open=”a”)
  lapply(seq_len(nRow),function(i) {mkRow2(nCol, cf)})
  close(cf)
  numnumer <- rep(”numeric”, nCol - 1)
  read.table(some.file, nrows=nRow,
    colClasses= c(”character”, numnumer))
}

go <- Sys.time();a1 <- mkFrameForLoop(10000,10);print(Sys.time()-go)
go <- Sys.time();a2 <- mkFrameList(10000,10);print(Sys.time()-go)
go <- Sys.time();a3 <- avoidAggregate(10000,10);print(Sys.time()-go)
```

> *go <- Sys.time();a1 <- mkFrameForLoop(10000,10);print(Sys.time()-go)*
> *Time difference of 25.3197 secs*
> *go <- Sys.time();a2 <- mkFrameList(10000,10);print(Sys.time()-go)*
> *Time difference of 9.448417 secs*
> *go <- Sys.time();a3 <- avoidAggregate(10000,10);print(Sys.time()-go)*
> *Time difference of 0.6649339 secs*

---

jmount

July 28, 2015 at 9:18 pm

David,

Thanks for your comment I suspect in addition to allocation a lot of the lost time is in the name and type checking of all the rows. I was able to run your code and see similar timings for the two efficient methods (my first bad for-loop was taking too long, so I killed it). Your code was a real kick in the stomach.

Another way thing we could try is: pre-allocating into a character matrix like below (on a machine that seems to be no faster than the one you used). However in both cases (your code and my code) we have hard to maintain code (that I wouldn't want to teach a beginner) and we are damaging the floating point numbers by round-tripping them through a printed decimal representation (which isn't faithful to binary floating point).

There are also nice idiomatic ways to do this with `dplyr::bind_rows` and `plyr::ldply` which I am adding to the examples (though they don't get the speed of the file or matrix code).

```
mkRowC <- function(nCol) {
  xN <- rnorm(nCol)
  x <- as.character(xN)
  x[[1]] <- ifelse(xN[[1]]>0,'pos','neg')
  x
}

mkFrameMat <- function(nRow,nCol) {
  d <- matrix(data="",
     nrow=nRow,ncol=nCol)
  for(i in seq_len(nRow)) {
    ri <- mkRowC(nCol)
    d[i,] <- ri
  }
  d <- data.frame(d,
     stringsAsFactors=FALSE)
  if(nCol>1) {
    for(i in 2:nCol) {
      d[[i]] <- as.numeric(d[[i]])
    }
  }
  names(d) <- paste('x',
     seq_len(nCol),sep='.')
  d
}

go <- Sys.time()
aC <- mkFrameMat(10000,10)
print(Sys.time()-go)

## Time difference of 0.4311979 secs
```

David Hood

July 29, 2015 at 1:11 pm

In reality, the take-home of use file IO is that if you are in a situation of Split-Apply-Combine followed by write out to a file, the Combine is an unneeded step as it is implicit in the file. There are the sensible optimisations down thread for keeping it in R in working memory.

That said, I am fighting an urge to rejoin the optimisation race with the “stupid optimisation” of using a RAM disk for the file IO (since R would be holding the data in RAM anyway, you would have to have the RAM for it either way, if you didn’t have the RAM you would be writing out to disk so see above. But that is a theoretical kludge at this stage)

jmount

July 29, 2015 at 1:34 pm

David,

Sorry it became a race, as I think what is fastest is a side point (I was originally writing about how to get away from slowest).

Also I am not sure a ramdisk will help- likely your "file" is in virtual memory and performing at ram speeds already. Also I do endorse your point that if you are in a split-apply-combine situation file ops are okay (I've written on the technique myself http://www.win-vector.com/blog/2011/12/what-to-do-when-you-run-out-of-memory/). Mostly I was upset R is much slower than I would like until you jump through at least one more hoop.

John

David Hood

July 29, 2015 at 4:04 pm

I must admit, my understanding of virtual memory (once again, caveating not a solution for beginners) was that virtual memory maps to RAM then disk when it runs out, but using file I/O saves to disk, so using file I/O with a RAM disk (a pretend disk made of RAM) means that the file I/O disk speeds would get RAM speeds.

jmount

July 29, 2015 at 6:53 pm

I am sorry, I wasn't quite precise. What I meant was in some situations on some operating systems due to page cache (https://en.wikipedia.org/wiki/Page_cache) some IO operations in fact happen in RAM at RAM speeds (and then are pushed to disk later). I haven't looked into if/when R might benefit from such capabilities (such as do new files count, do writes count, would R need to ask for a memory mapped file, and so on).

Jim Hester (@the_belal)

July 29, 2015 at 5:48 am

An alternative approach than David's read.table() or John's matrix methods is to realize that data.frames are simply lists of vectors with equal lengths. If you know a priori the number and type of rows you can pre-allocate all of them first, then simply assign to each of them in turn.

An implementation of this (mkFrameVector) is at https://gist.github.com/jimhester/e725e1ad50a5a62f3dee#file-accumulation-r-L43-L57

That gist also contains David and John's methods, as well as a method using dplyr::bind_rows.

Interestingly the performance of the vector based method depends highly on whether the function has been byte compiled. Without byte compilation it is actually slower than the matrix method (with or without compilation). However when byte compiled the vector based method is by far the fastest.

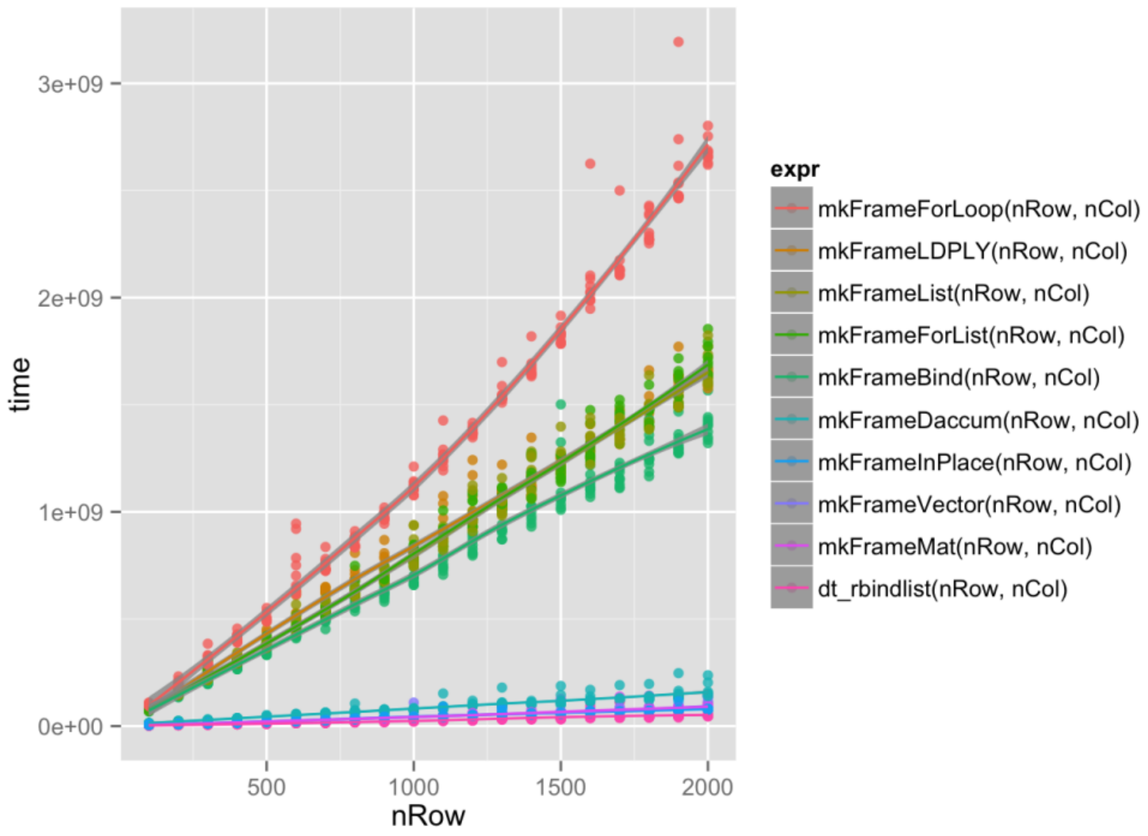
I actually think this vector based approach is a good way to teach this as it both de-mystifies how data.frames are implemented and gives you a good tool to iteratively build them by both rows or columns.

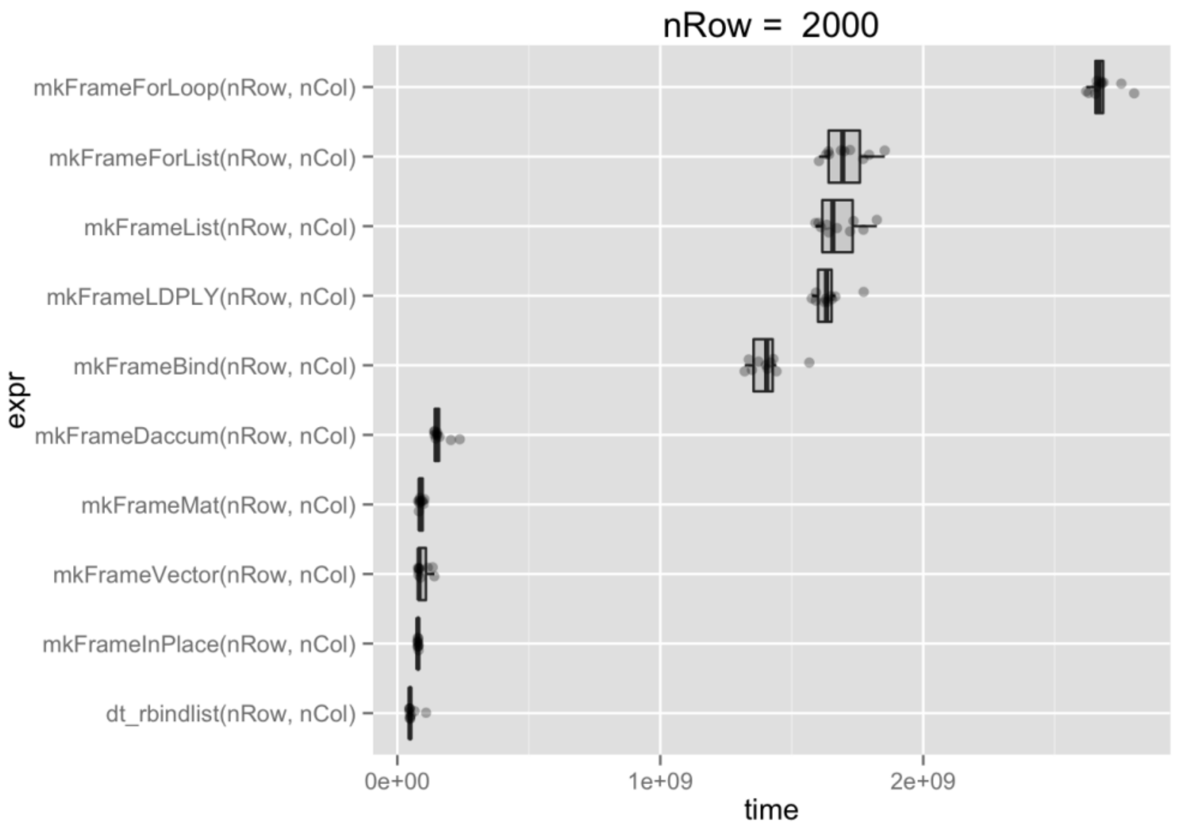
jmount

July 29, 2015 at 6:58 am

Thanks Jim,

I agree the fact that data frames are lists of columns (and not sequences of rows) is a very important point to know in R (and counter to some expectations). So more examples emphasizing it are good. I added a variation of your code to the examples http://winvector.github.io/Accumulation/Accum.html . Vector and matrix are now the winning methods. Likely compiled vector is the overall winner, and the sensitivity to compiling is probably do to the inner over-columns loop (as a lot of the other method attempt to batch that step). I also cleaned up the data-frame in-place code a bit, so it is now at least out of the pack (and ties for first if we drop the data.frame class label).





Arun Srinivasan (@arun_sriniv)

July 29, 2015 at 9:56 am

Hello,

I'd just accomplish the task as follows using `rbindlist()` from `data.table` (which is missing from the benchmarks so far) to bind the results:

```
dt_rbindlist <- function(nRow, nCol) {
set.seed(45L)
v = vector("list", nRow)
for (i in seq_len(nRow))
v[[i]] = mkRow(nCol)
rbindlist(v)
}
```

The `set.seed(45L)` is to ensure identical results. I compared this against `mkFrameInPlace()` (seems the fastest from plots) by adding `set.seed(45L)` to the first line.

I did not compare against `mkFrameMat()` as it results in all numeric columns being character type.

Here are the timings on 2000 rows and 20, 000 rows:

```
> nRow <- 2000L
> microbenchmark( mkFrameInPlace(nRow, nCol),
+ dt_rbindlist(nRow, nCol),
+ times = 10L)
Unit: milliseconds
expr min lq mean median uq max neval
mkFrameInPlace(nRow, nCol) 81.53944 85.27447 92.28244 92.03683 100.76136 101.90796
10
dt_rbindlist(nRow, nCol) 37.08913 40.04932 46.92061 44.64438 55.02693 58.25099 10

> nRow = 20000L
> microbenchmark( mkFrameInPlace(nRow, nCol),
+ dt_rbindlist(nRow, nCol),
+ times = 10L)
Unit: milliseconds
expr min lq mean median uq max neval
mkFrameInPlace(nRow, nCol) 867.6767 887.3262 930.5802 922.5305 973.3096 1014.222 10
dt_rbindlist(nRow, nCol) 460.3275 476.0989 542.7660 528.2651 559.0458 671.903 10
```

```
identical(mkFrameInPlace(nRow, nCol), setDF(dt_rbindlist(nRow, nCol)))
# [1] TRUE
```

jmount

July 29, 2015 at 3:22 pm

Thanks Arun,

I've added your function to the graphs. Looks like `data.table` is a very good choice (both in terms of speed and readability).

Arun Srinivasan (@arun_sriniv)

July 30, 2015 at 7:15 am

John, thanks. I think so too :-P.

jmount

August 10, 2015 at 1:46 pm

One minor quibble/edge-case though. `data.table` returns something that claims to be a `data.frame`, but the `[,columns]` selector does not work as it would for a `data.frame`. I am know this is a documented user facing feature, but it is a minor hazard for code that doesn't know if it has a `data.table` or a `data.frame` (and isn't in one of the special execution contexts). Example:

```
data.frame(x=7)[,'x']
## [1] 7
data.table::rbindlist(list(data.frame(x=7)))[,'x']
## [1] "x"  # what?
data.frame(data.table::rbindlist(list(data.frame(x=7))))[,'x']
## [1] 7
class(data.table::rbindlist(list(data.frame(x=7))))
## [1] "data.table" "data.frame"
```

Comments are closed.