

Accumulating results: small dataframes

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.

Define the time sequence for the benchmarking tests

We have two options, the long and short time sequences.

```
library('microbenchmark')
library('ggplot2')

set.seed(23525) # make run more repeatable

nCol <- 10 # number of columns for the test DF
# If we use seq.int(100, 2000, 100) we will test with dataframes of size:
# 100 200 300 400 500 600 700 800 900 1000
# 1100 1200 1300 1400 1500 1600 1700 1800 1900 2000
timeSeq <- seq.int(100, 2000, 100) # long sequence

# If we use seq.int(10, 200, 10) we will test with dataframes of size:
# 10 20 30 40 50 60 70 80 90 100
# 110 120 130 140 150 160 170 180 190 200
timeSeq <- seq.int(10, 200, 10) # short sequence
summary(timeSeq)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
10.0	57.5	105.0	105.0	152.5	200.0

Function to populate a row of data.

```
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
}
```

Function for benchmarking plots of all functions

```
# build function to plot timings
# devtools::install_github("WinVector/WVPlots")
# library('WVPlots')
plotTimings <- function(timings) {
  timings$expr <- reorder(timings$expr, -timings$time, FUN=max)
```

```

ggplot(data = timings, aes(x=nRow,y=time, color=expr)) +
  geom_point(alpha=0.8) + geom_smooth(alpha=0.8)

nmax <- max(timings$nRow)
tsub <- timings[timings$nRow==nmax,]
tsub$expr <- reorder(tsub$expr,tsub$time,FUN=median)
list(
  ggplot(data=timings,aes(x=nRow,y=time,color=expr)) +
    geom_point(alpha=0.8) + geom_smooth(alpha=0.8),
  ggplot(data=timings,aes(x=nRow,y=time,color=expr)) +
    geom_point(alpha=0.8) + geom_smooth(alpha=0.8) +
    scale_y_log10(),
  WVPlots::ScatterBoxPlot(tsub,'expr','time',
    title=paste('nRow = ',nmax)) +
    coord_flip()
)
}

```

1st test: mkFrameForLoop

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.

```

# The common wrong-way to accumulate the rows of data into a single data frame.
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
}

```

Timing showing the quadratic runtime.

```

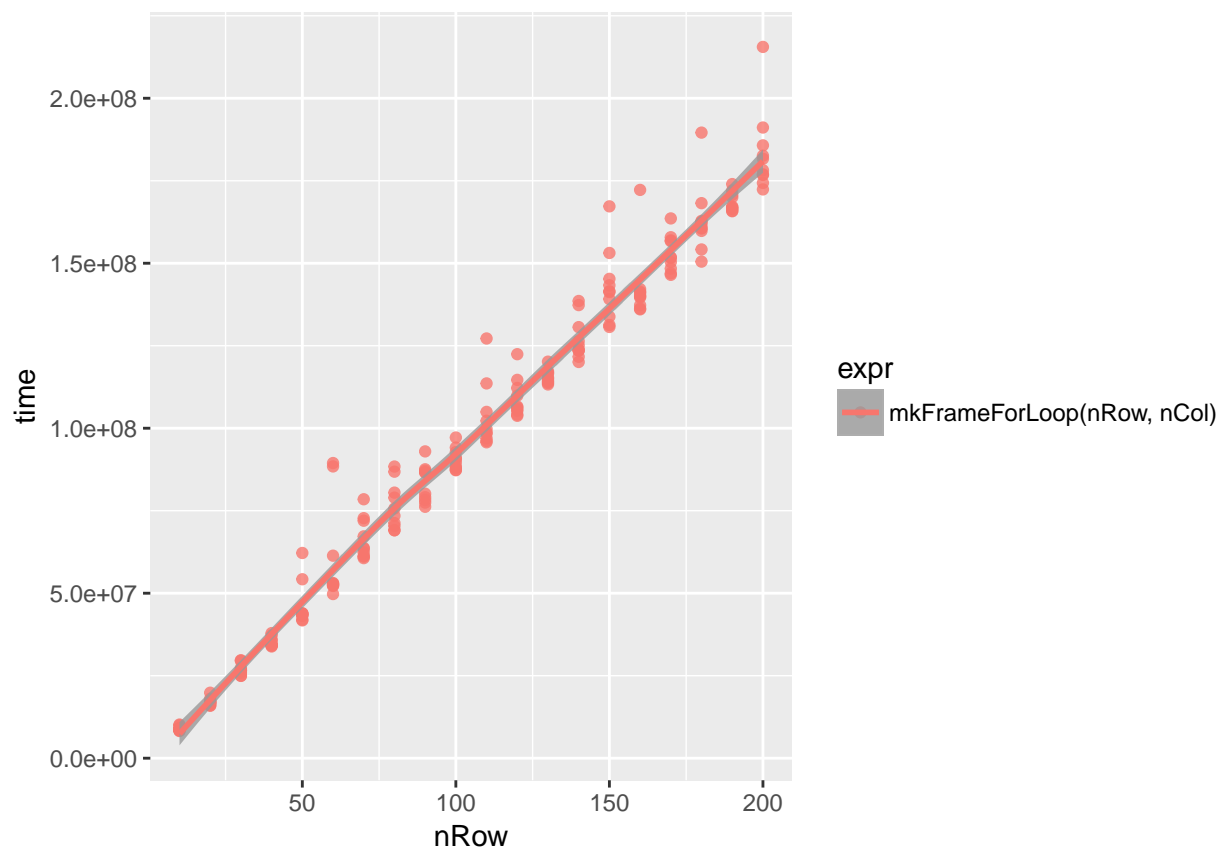
# this list will save the dataframes created during the test
timings <- vector("list", length(timeSeq)) # create a vector of lists

for(i in seq_len(length(timeSeq))) { # iterate through time sequence
  nRow <- timeSeq[[i]] # get number of rows for the current DF
  ti <- microbenchmark(
    mkFrameForLoop(nRow, nCol), # create DF
    times=10)
  ti <- data.frame(ti, stringsAsFactors = FALSE) # no factors
  ti$nRow <- nRow # document # rows
  ti$nCol <- nCol
  timings[[i]] <- ti # save DF in the list
}

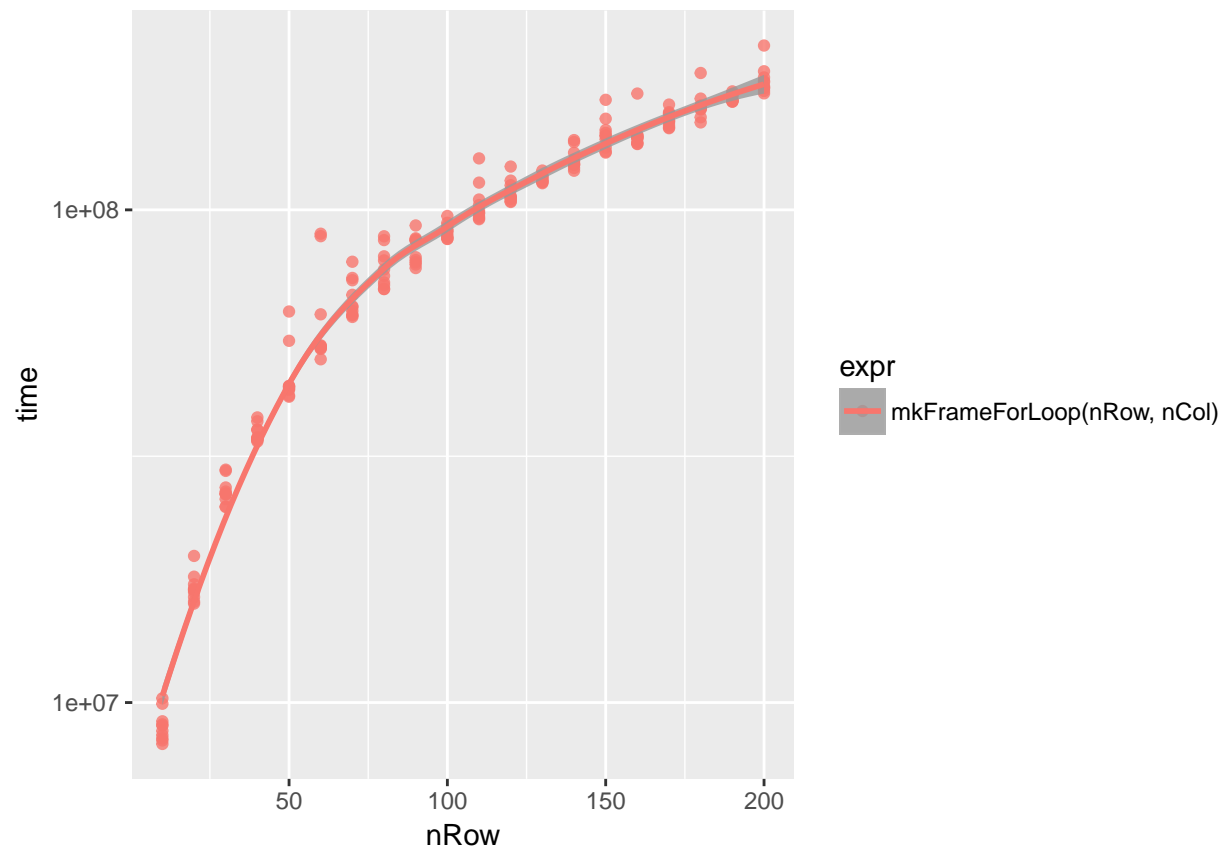
timings <- data.table::rbindlist(timings) # convert DF to DT
print(plotTimings(timings)) # plot the data

```

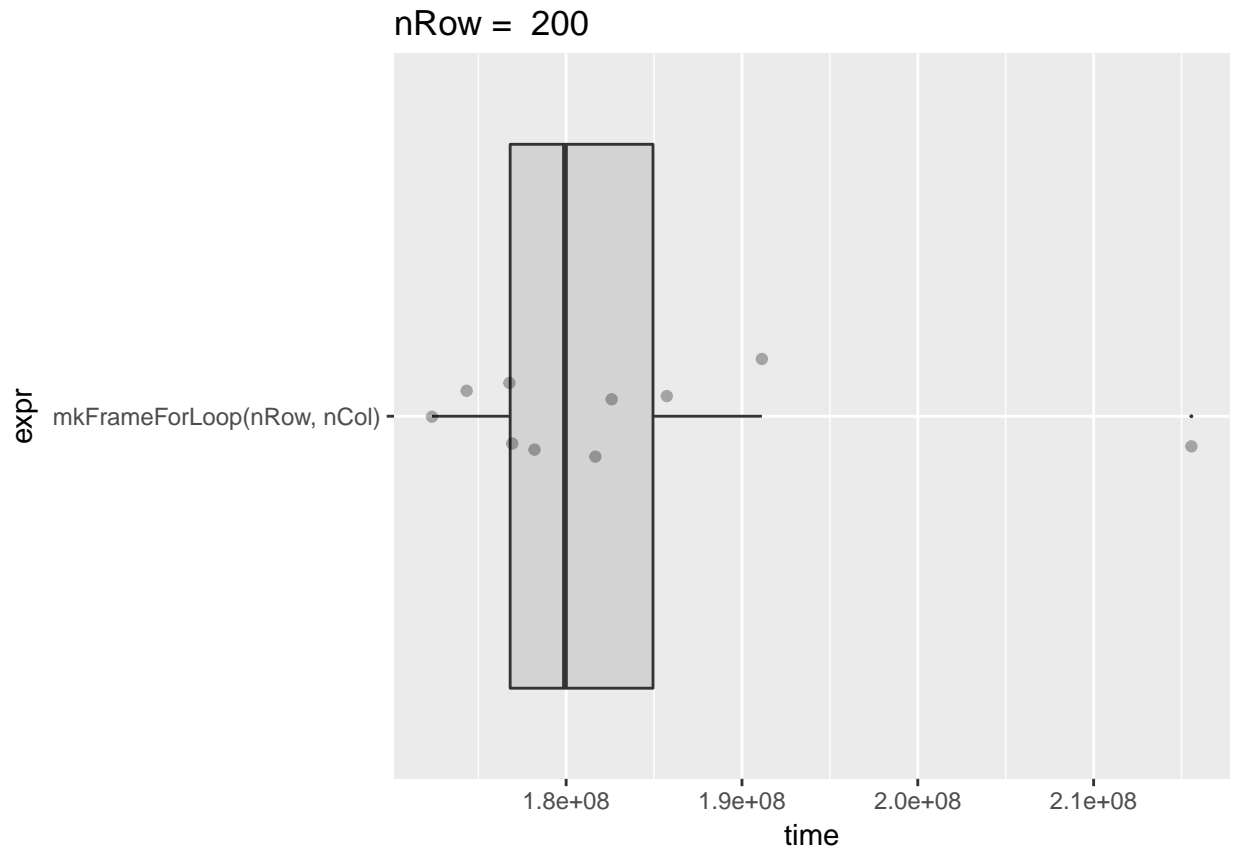
[[1]]



[[2]]



[[3]]



2nd test: Benchmark mkFrameForList, mkFrameList

A few roughly equivalent right ways to accumulate the rows.

```
# Simplest fix, collect the data in a list and
# grow the list. Exploits the fact that R can mutate
# common objects when object visibility is limited.
mkFrameForList <- function(nRow,nCol,verbose=FALSE) {
  d <- as.list(seq_len(nRow)) # pre-alloc destination
  for(i in seq_len(nRow)) {
    ri <- mkRow(nCol)
    di <- data.frame(ri,
                     stringsAsFactors=FALSE)
    d[[i]] <- di
    if(verbose) {
      print(pryr::address(d))
    }
  }
  do.call(rbind,d)
}

# Cleanest fix- wrap procedure in a function and
# use lapply.
mkFrameList <- function(nRow,nCol) {
  d <- lapply(seq_len(nRow),function(i) {
```

```

    ri <- mkRow(nCol)
    data.frame(ri,
               stringsAsFactors=FALSE)
  })
  do.call(rbind,d)
}

```

Confirm value getting altered in place (efficiency depends on interior columns also not changing address, which is also the case).

```
mkFrameForList(10, 5, TRUE)
```

```

[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"
[1] "0x18219c48"

  x.1      x.2      x.3      x.4      x.5
1 neg  0.2180051  0.65928193  0.50576371 -0.03234993
2 neg -1.2177309 -0.50864173 -0.08079341 -0.99977204
3 neg -2.6034505 -1.16420051  2.13052560  0.22547356
4 neg  0.5541223  0.41546126  0.24872508 -0.82441641
5 neg -0.8288038  0.15856544  1.88701077  0.50762296
6 pos -2.2443795  0.26623942  1.91731768  0.75888414
7 pos  1.1553502 -0.06070064  0.73363742 -1.28304563
8 neg -0.2118475  0.43565604  0.64812772  0.43103117
9 pos  1.5933721  0.82525165  0.43050060 -0.38508535
10 neg -0.1551818 -0.52340278  0.03164055  0.94051629

```

Get more timings and plots for two new functions.

```

timingsPrev <- timings # accumulate previous timings
timings <- vector("list", length(timeSeq))

for(i in seq_len(length(timeSeq))) {
  nRow <- timeSeq[[i]]
  ti <- microbenchmark(
    mkFrameForList(nRow,nCol),
    mkFrameList(nRow,nCol),
    times=10)
  ti <- data.frame(ti,
                   stringsAsFactors=FALSE)

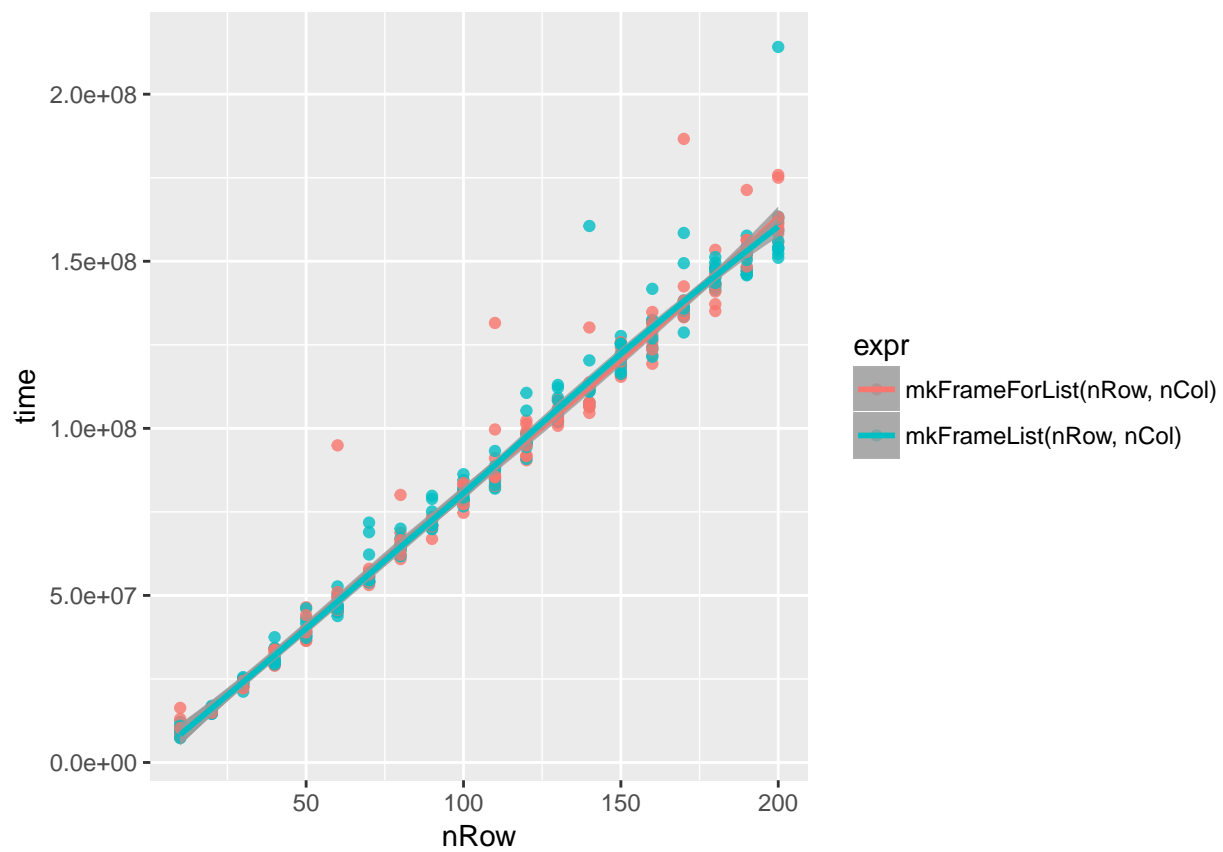
  ti$nRow <- nRow
  ti$nCol <- nCol
  timings[[i]] <- ti
}

timings <- data.table::rbindlist(timings)

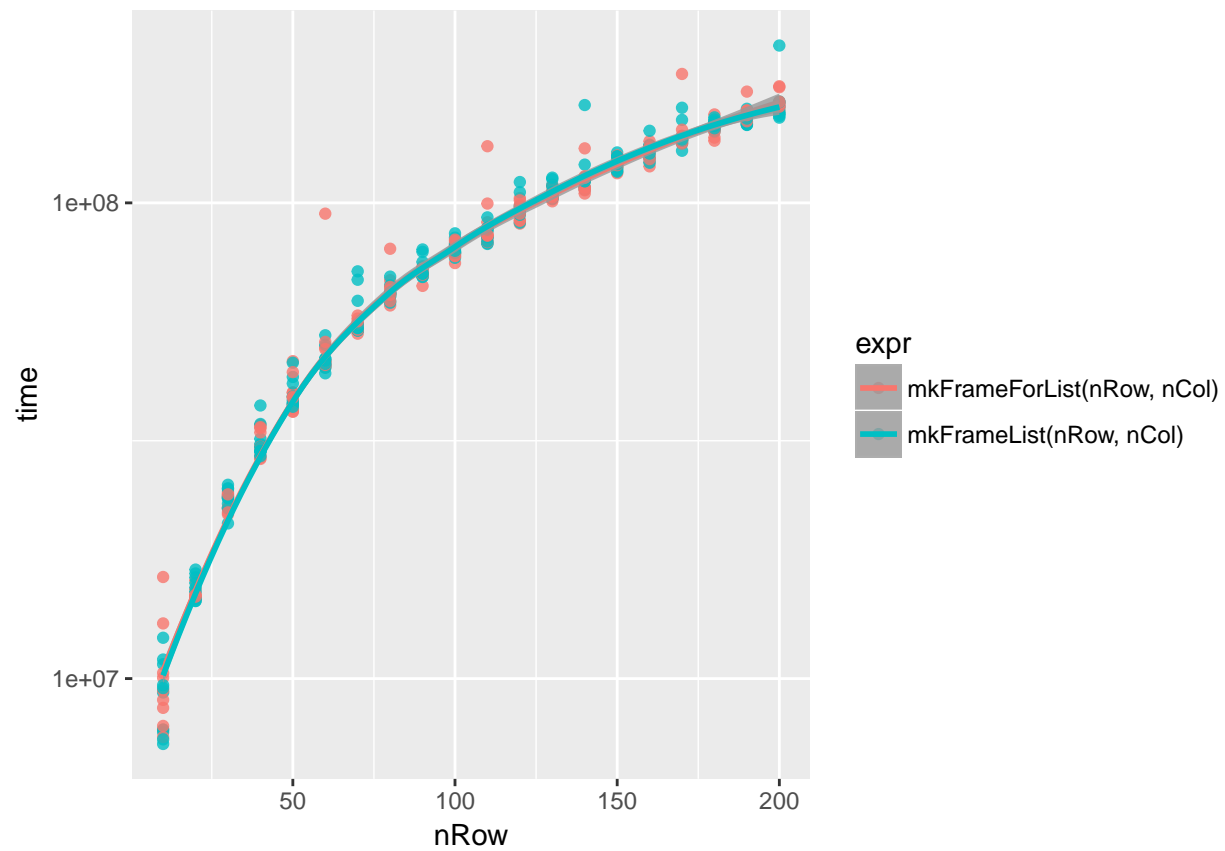
print(plotTimings(timings))

```

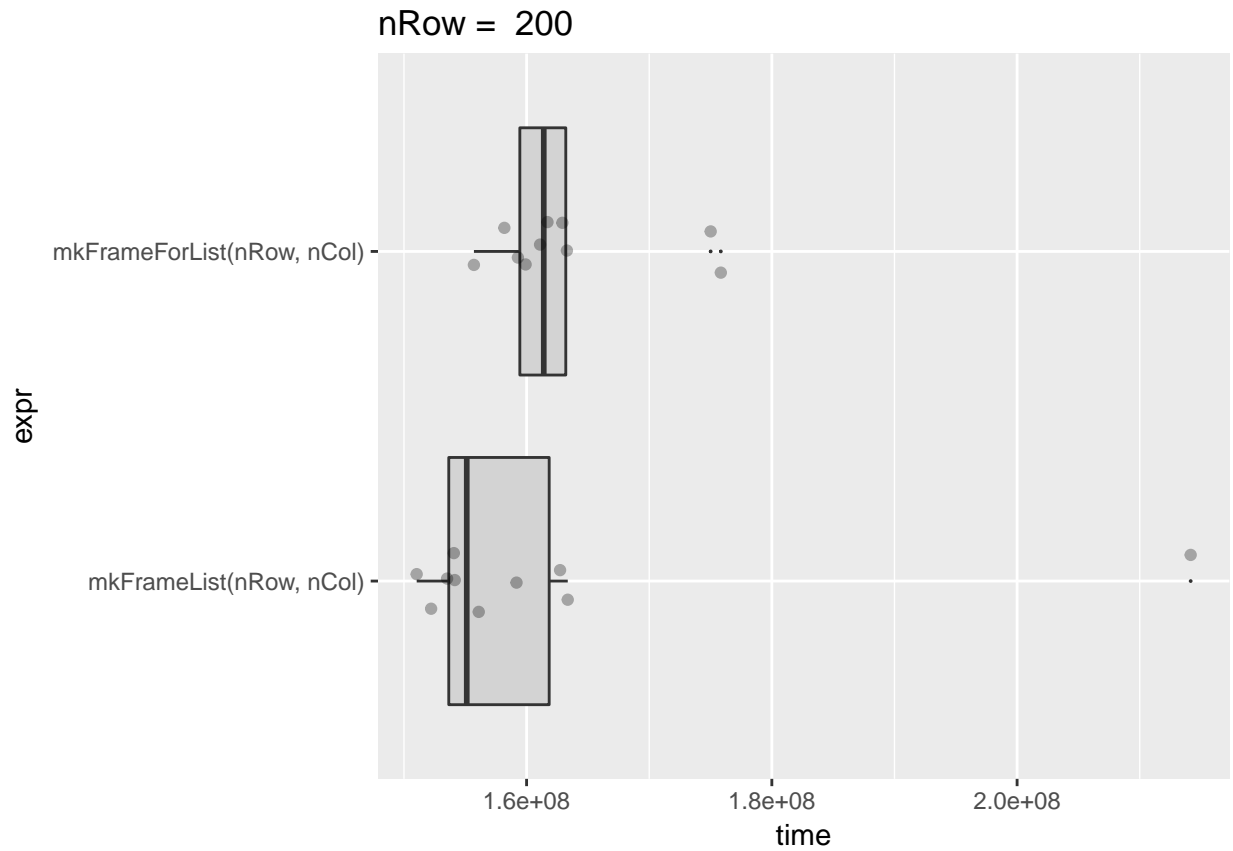
[[1]]



[[2]]



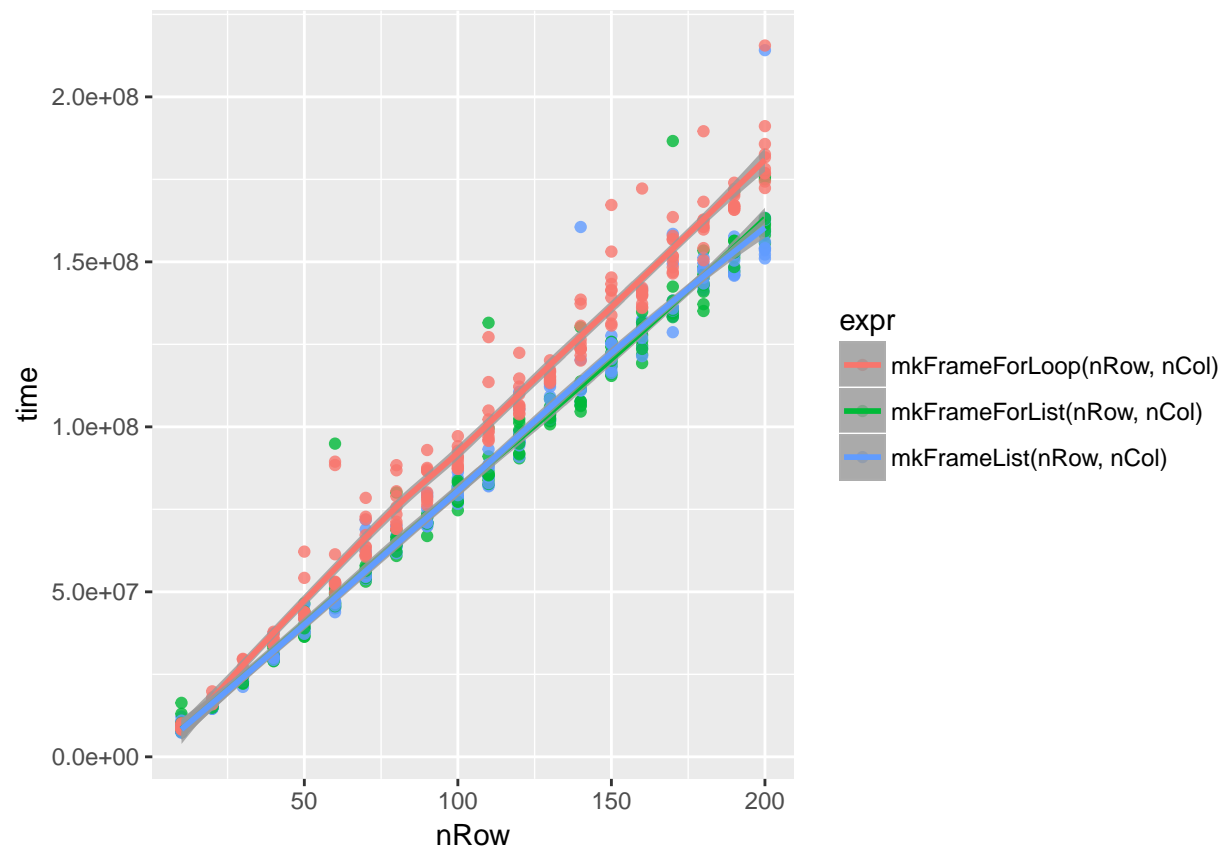
[[3]]



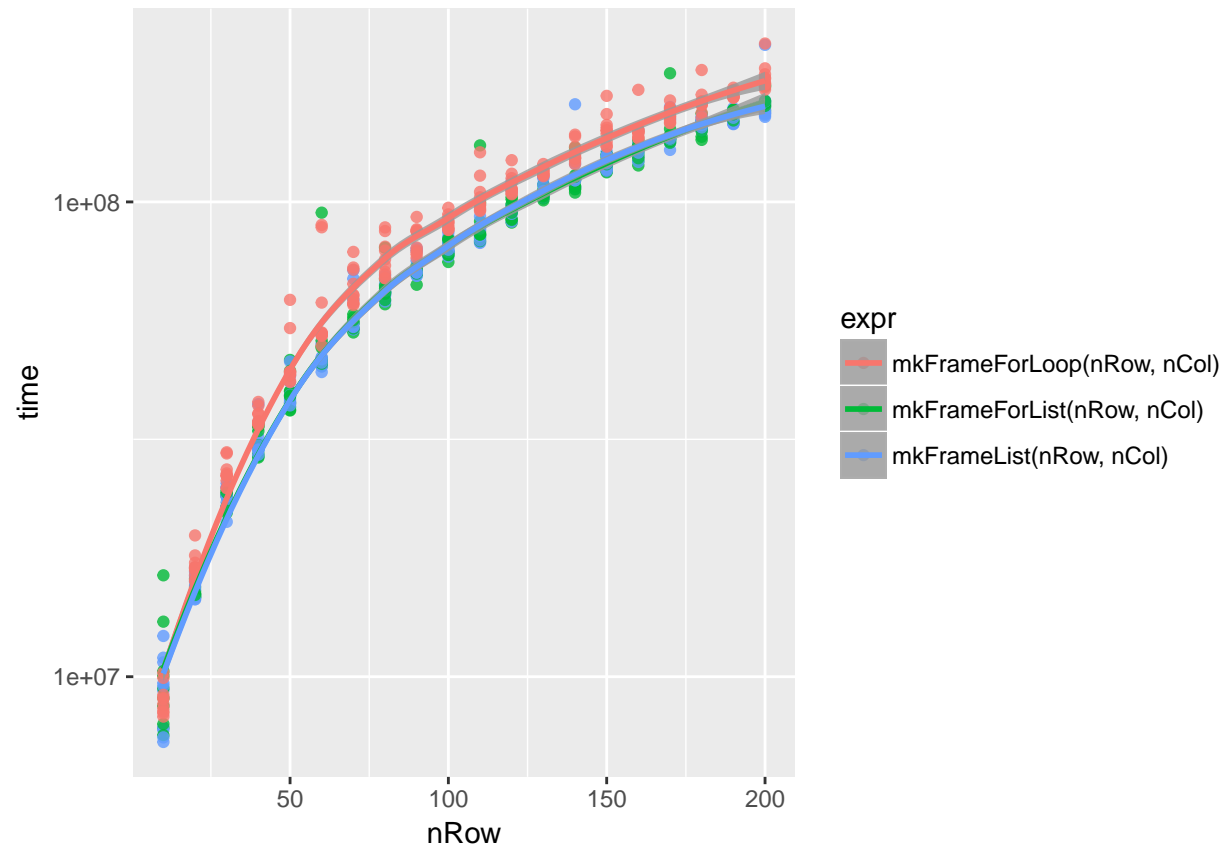
Report for previous and newer functions

```
timings <- rbind(timings, timingsPrev)
print(plotTimings(timings))

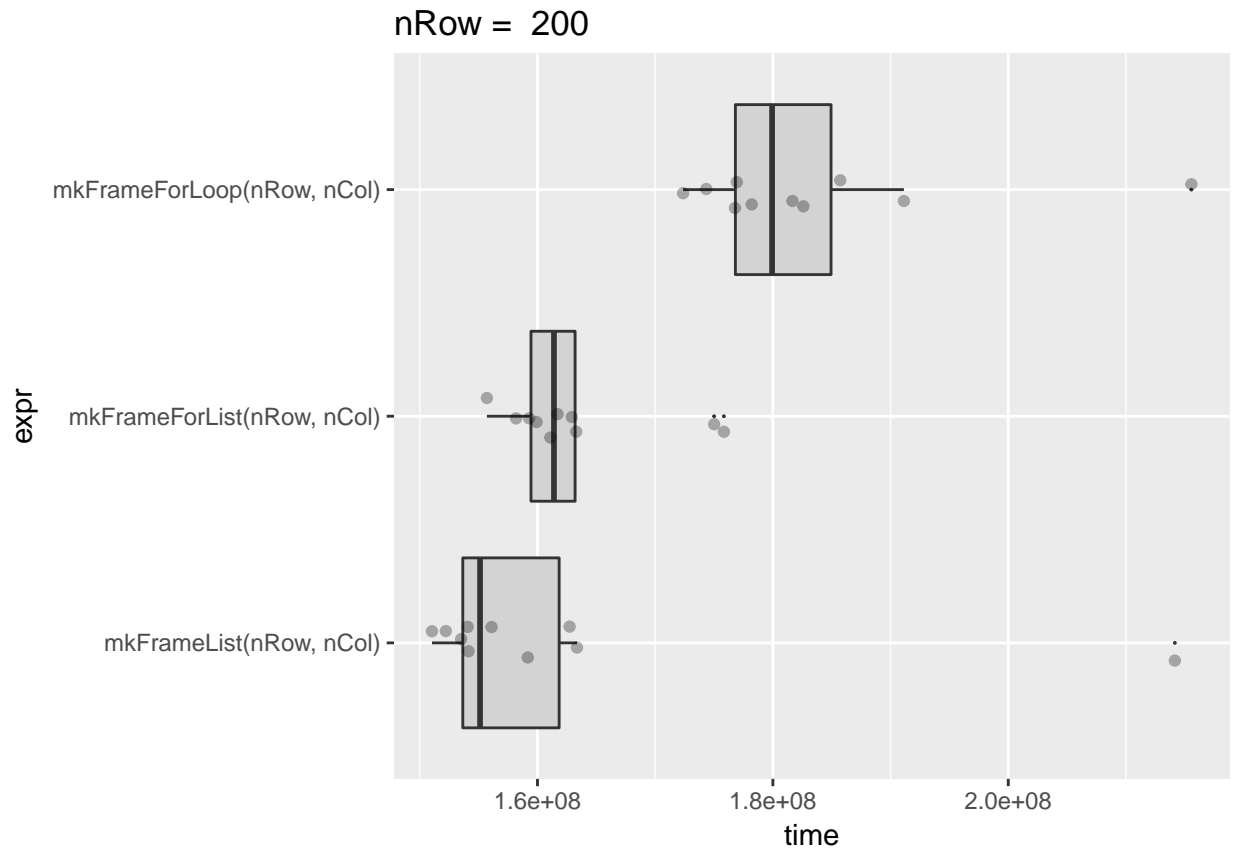
[[1]]
```



[[2]]

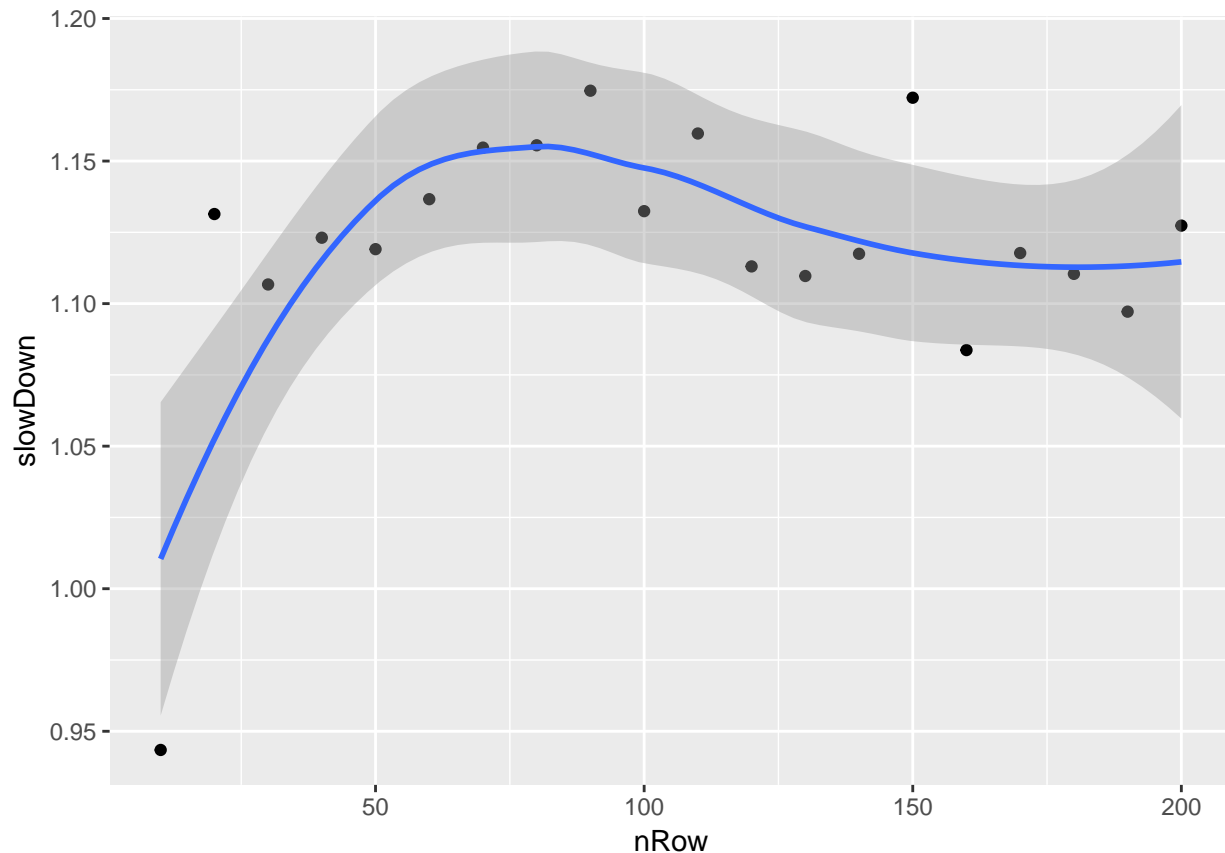


[[3]]



Show slowdown of incremental method versus others as a function of number of row.

```
timingsPrev <- timings
timings$isIncremental <- timings$expr=='mkFrameForLoop(nRow, nCol)'
agg <- aggregate(time~isIncremental+nRow,data=timings,FUN=median)
dInc <- agg[agg$isIncremental,]
dInc <- dInc[order(dInc$nRow),]
dRest <- agg[!agg$isIncremental,]
dRest <- dRest[order(dRest$nRow),]
dInc$slowDown <- dInc$time/dRest$time
ggplot(data=dInc,aes(x=nRow,y=slowDown)) +
  geom_point() + geom_smooth()
```



3rd test: mkFrameInPlace, mkFrameDataTableFor

```
# From Arun Srinivasan's comment
# http://www.win-vector.com/blog/2015/07/efficient-accumulation-in-r/comment-page-1/#comment-65994
# library('data.table')
mkFrameDataTableFor <- function(nRow, nCol) {
  v = vector("list", nRow)
  for (i in seq_len(nRow)) {
    v[[i]] = mkRow(nCol)
  }
  data.table::rbindlist(v)
}

mkFrameDataTableLapply <- function(nRow, nCol) {
  v <- lapply(seq_len(nRow), function(i) {
    mkRow(nCol)
  })
  data.table::rbindlist(v)
}

# Mucking with environments fix. Environments
# are mutable and tend to be faster than lists.
mkFrameEnvDataTable <- function(nRow, nCol) {
  e <- new.env(hash=TRUE, parent=emptyenv())
```

```

for(i in seq_len(nRow)) {
  ri <- mkRow(nCol)
  assign(as.character(i),ri,envir=e)
}
data.table::rbindlist(as.list(e))
}

# Another possibility, working in place.
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
}

```

Still more timings.

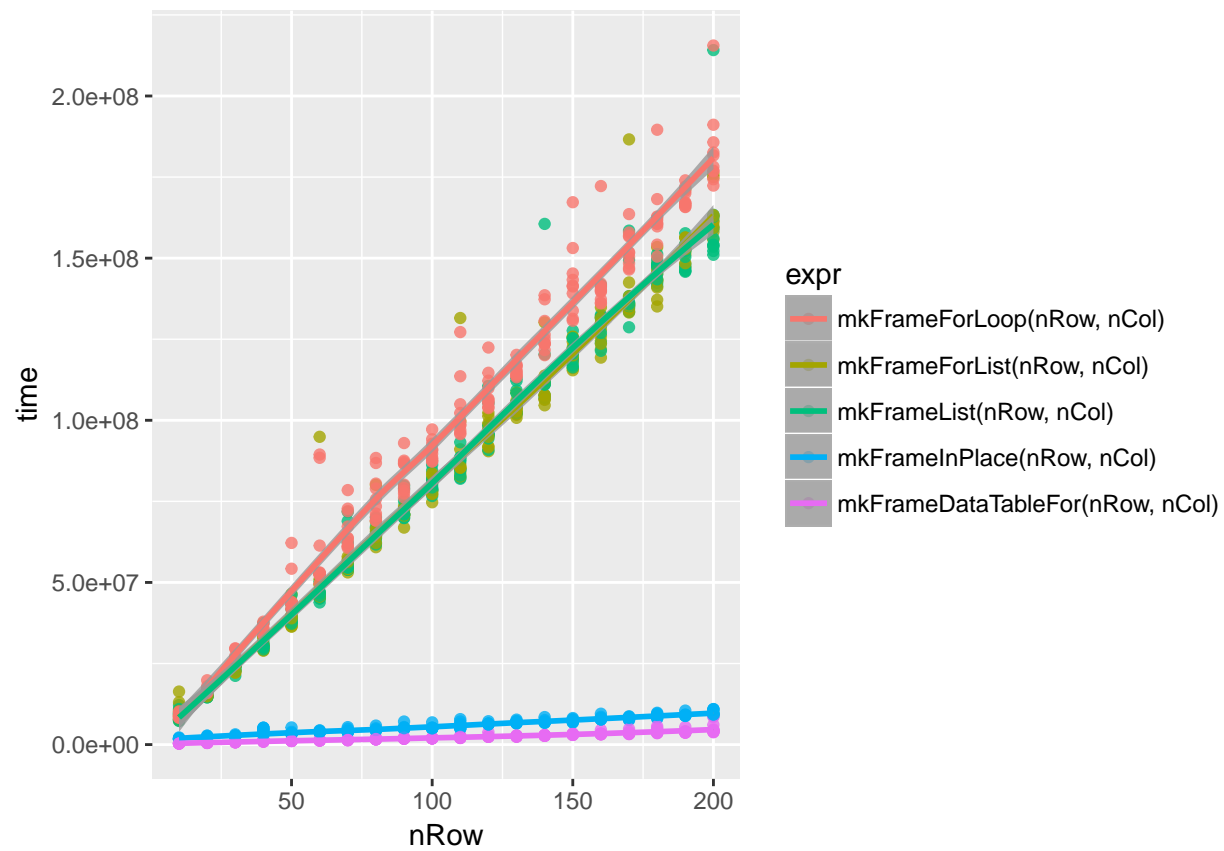
```

timings <- vector("list", length(timeSeq))
for(i in seq_len(length(timeSeq))) {
  nRow <- timeSeq[[i]]
  ti <- microbenchmark(
    mkFrameInPlace(nRow,nCol),
    mkFrameDataTableFor(nRow,nCol),
    times=10)
  ti <- data.frame(ti,
                  stringsAsFactors=FALSE)
  ti$nRow <- nRow
  ti$nCol <- nCol
  timings[[i]] <- ti
}
timings <- data.table::rbindlist(timings)
timings <- rbind(timings,timingsPrev)

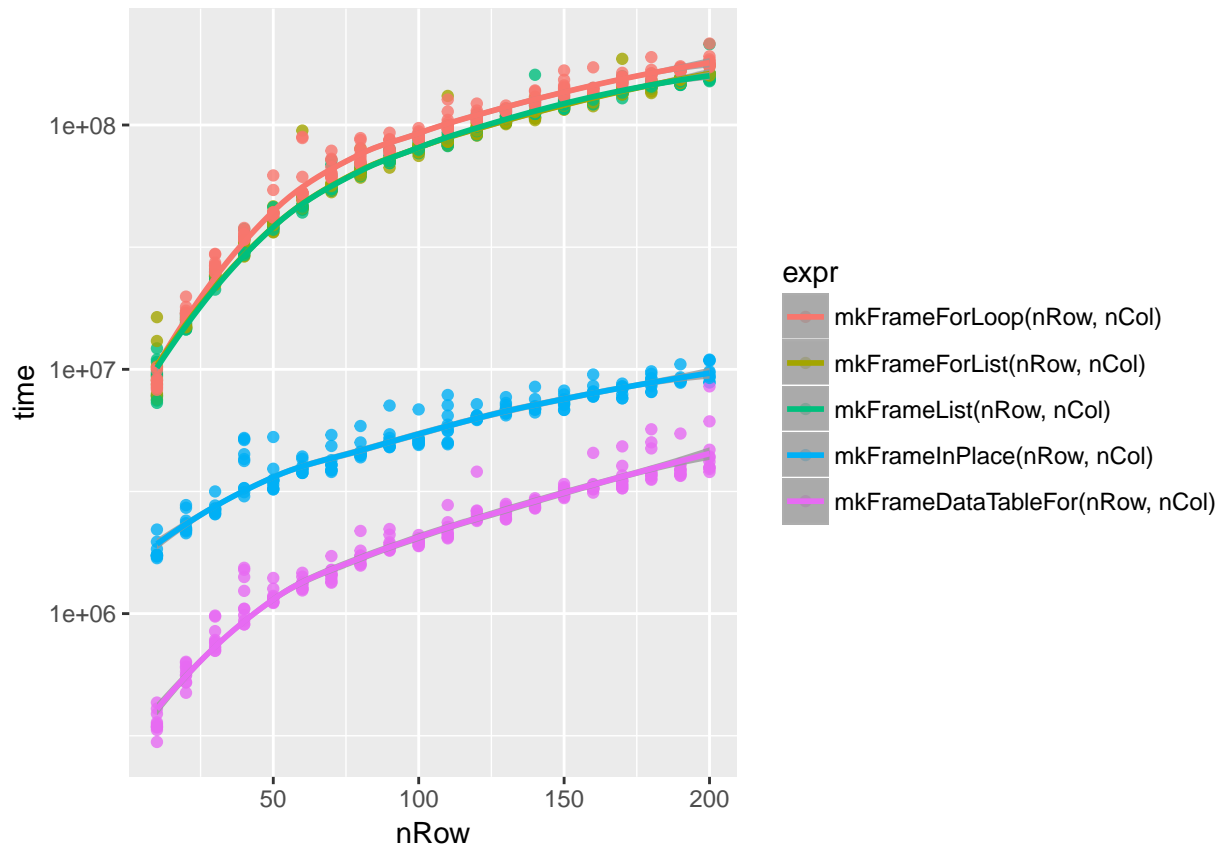
print(plotTimings(timings))

```

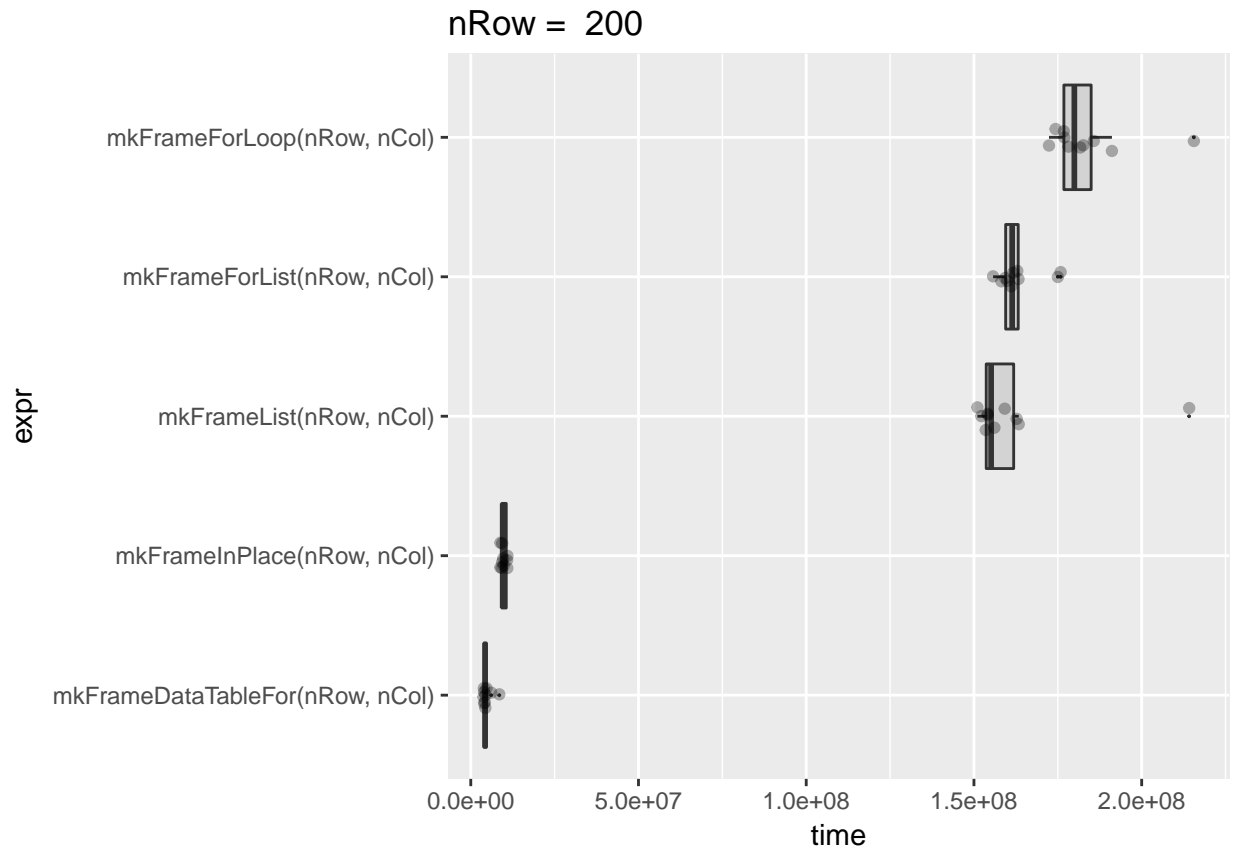
```
[[1]]
```



[[2]]



[[3]]



4th test: mkFrameDplyrBind, mkFramePlyrLdply, mkFrameMat, mkFrameVector, mkFrameDaccum

```
# library('dplyr')
# Fast method from https://twitter.com/drob/status/626146450842497028
mkFrameDplyrBind <- function(nRow,nCol) {
  dplyr::bind_rows(replicate(nRow,
    data.frame(mkRow(nCol),
      stringsAsFactors=FALSE),
    simplify=FALSE))
}

# library('plyr')
# Idiomatic plyr https://www.reddit.com/r/rstats/comments/3ex31f/efficient\_accumulation\_in\_r/ctjl9lq
mkFramePlyrLdply <- function(nRow,nCol) {
  plyr::ldply(seq_len(nRow),
    function(i) data.frame(mkRow(nCol),
      stringsAsFactors=FALSE))
}

# try to avoid all storage re-alloc, name checking, and type checking
mkFrameMat <- function(nRow,nCol) {
```

```

r1 <- mkRow(nCol)
isNum <- vapply(r1,is.numeric,logical(1))
numIndices <- seq_len(nCol)[isNum]
strIndices <- seq_len(nCol)[!isNum]
dN <- matrix(data="",
  nrow=nRow,ncol=length(numIndices))
dC <- matrix(data="",
  nrow=nRow,ncol=length(strIndices))
dN[1,] <- as.numeric(r1[numIndices])
dC[1,] <- as.character(r1[strIndices])
if(nRow>1) {
  for(i in seq.int(2,nRow,1)) {
    ri <- mkRow(nCol)
    dN[i,] <- as.numeric(ri[numIndices])
    dC[i,] <- as.character(ri[strIndices])
  }
}
dN <- data.frame(dN,stringsAsFactors=FALSE)
names(dN) <- names(r1)[numIndices]
dC <- data.frame(dC,stringsAsFactors=FALSE)
names(dC) <- names(r1)[strIndices]
cbind(dC,dN)
}

# allocate the vectors independently
# seems to be avoiding some overhead by not having data.frame class
# (close to current mkFrameInPlace, but faster)
# Adapted from: https://gist.github.com/jimhester/e725e1ad50a5a62f3dee#file-accumulation-r-L43-L57
mkFrameVector <- function(nRow,nCol) {
  r1 <- mkRow(nCol)
  res <- lapply(r1,function(cv) {
    if(is.numeric(cv)) {
      col = numeric(nRow)
    } else {
      col = character(nRow)
    }
    col[[1]] = cv
    col
  })
  if(nRow>1) {
    for(i in seq.int(2,nRow,1)) {
      ri <- mkRow(nCol)
      for(j in seq_len(nCol)) {
        res[[j]][[i]] <- ri[[j]]
      }
    }
  }
  data.frame(res,stringsAsFactors=FALSE)
}

# Efficient method that doesn't need to know how many rows are coming
# devtools::install_github("WinVector/daccum")

```

```

# library('daccum')
mkFrameDaccum <- function(nRow,nCol) {
  collector <- daccum::mkCollector()
  for(i in seq_len(nRow)) {
    ri <- mkRow(nCol)
    daccum::collectRows(collector,ri)
  }
  daccum::unwrapRows(collector)
}

```

Final comparison

```

timingsPrev <- timings
timings <- vector("list", length(timeSeq))

for(i in seq_len(length(timeSeq))) {
  nRow <- timeSeq[[i]]
  ti <- microbenchmark(
    mkFrameDplyrBind(nRow, nCol),
    mkFramePlyrLdply(nRow, nCol),
    mkFrameMat(nRow, nCol),
    mkFrameVector(nRow, nCol),
    mkFrameDaccum(nRow, nCol),
    mkFrameEnvDataTable(nRow, nCol),
    mkFrameDataTableLapply(nRow, nCol),
    times = 10)                                # benchmark 10 times

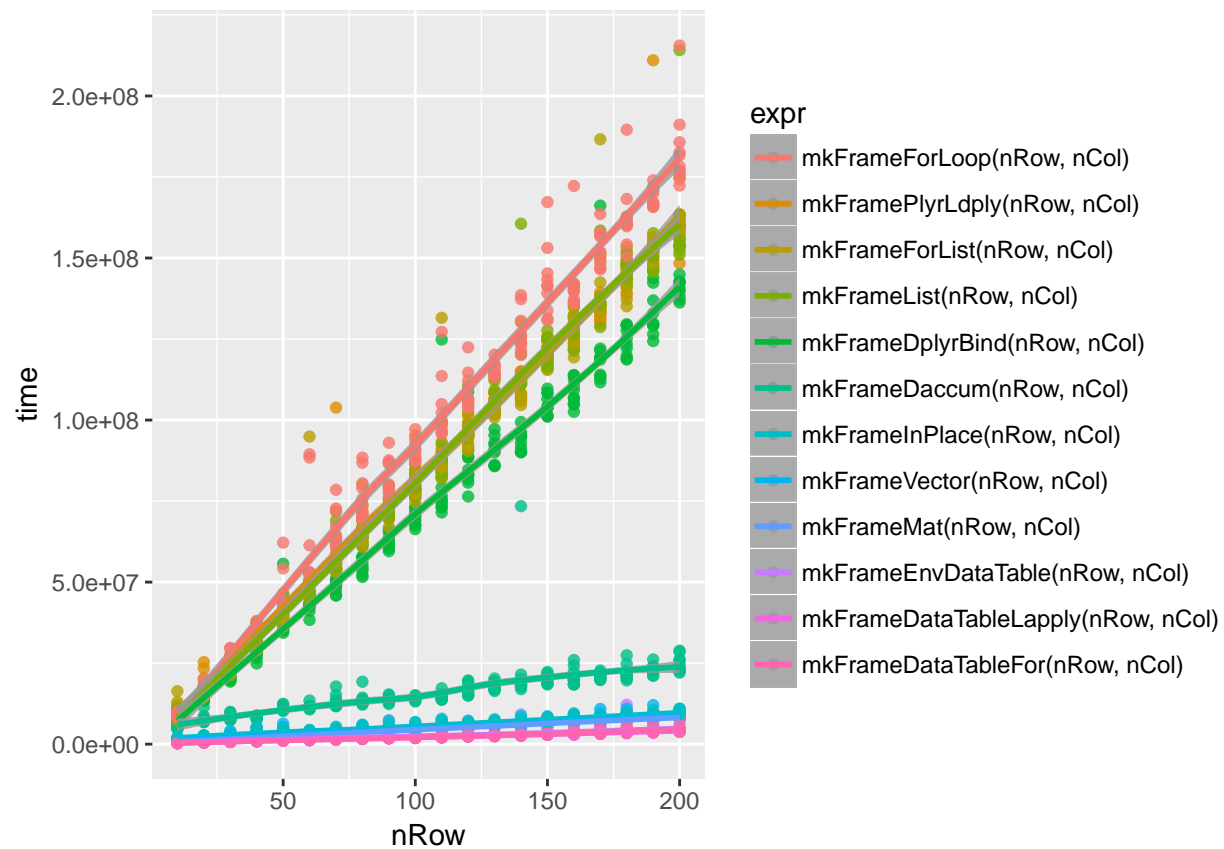
  ti <- data.frame(ti, stringsAsFactors=FALSE) # create DF with timings
  ti$nRow <- nRow                             # document # rows
  ti$nCol <- nCol                             # document # columns
  timings[[i]] <- ti                          # add DF to the list `timings`
}

timings <- data.table::rbindlist(timings, fill = TRUE)
timings <- rbind(timings, timingsPrev)

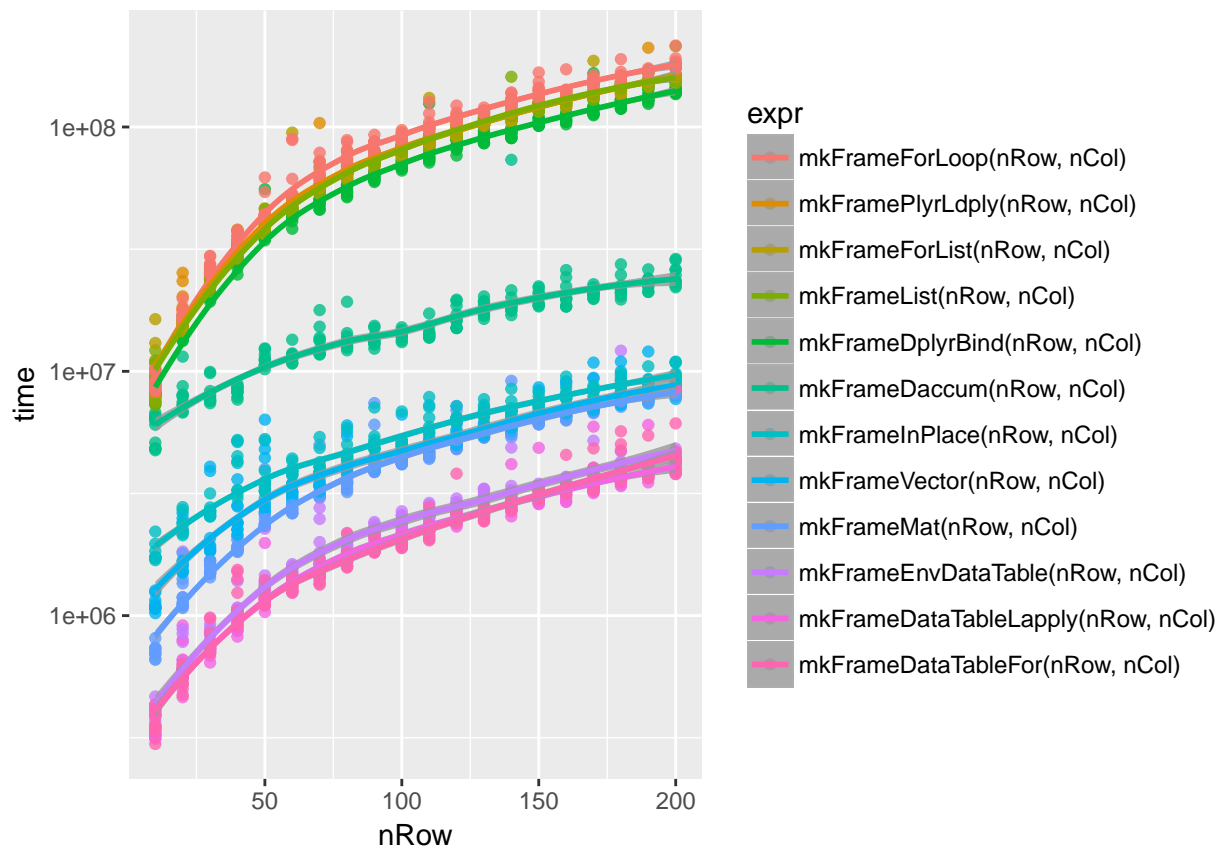
print(plotTimings(timings))

[[1]]

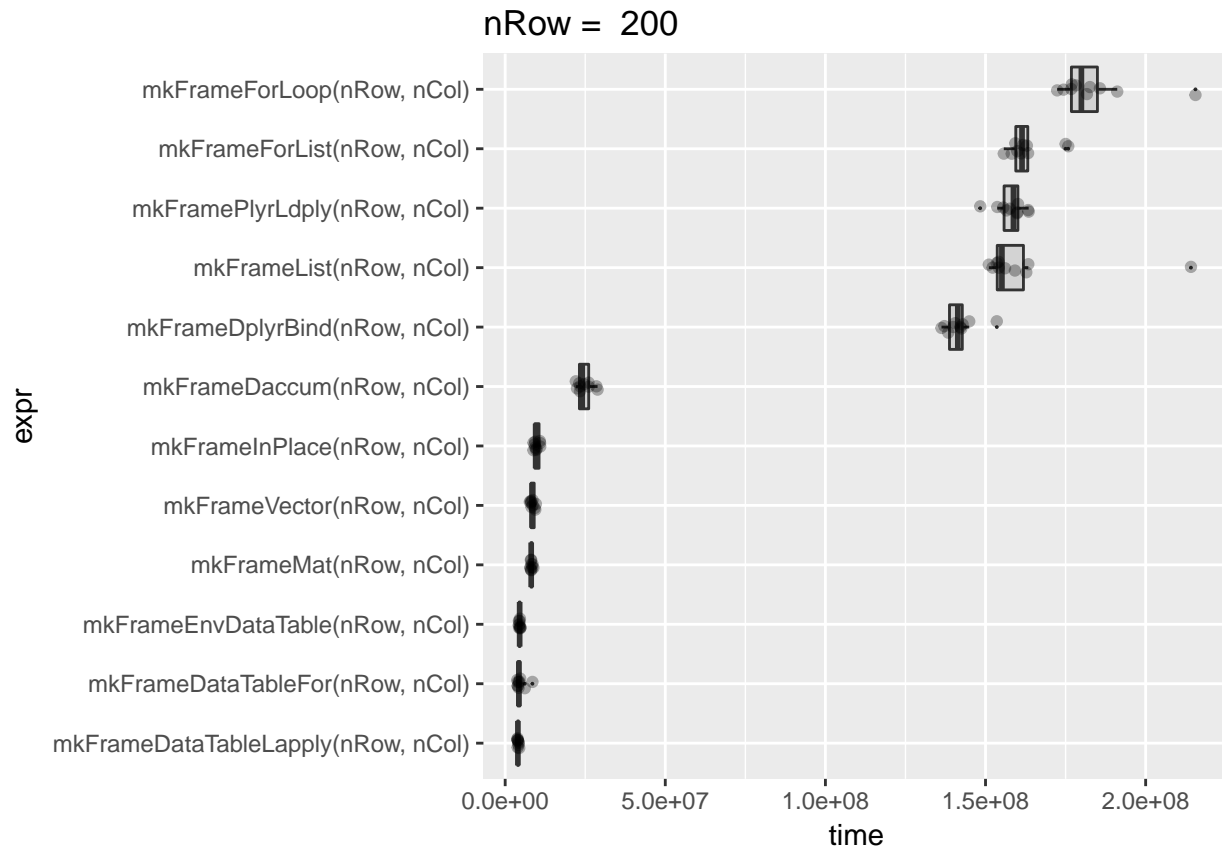
```



[[2]]



[[3]]



Miscellaneous

Show the in-place alteration of objects in a simpler setting.

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

```

    if(usePRYR) {
        print(pryr::address(v))
    }
}
v
}

```

Show how if the value associated with `v` is visible only to the variable name “`v`” that R will start performing in-place replacement (making calculation much cheaper).

```
computeSquares(5, FALSE)
```

```

@0x00000000180e4ed8 13 INTSXP g0c3 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
@0x0000000016fedf60 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
@0x0000000016fedf60 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,3,4,5
@0x0000000016fedf60 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,4,5
@0x0000000016fedf60 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,5
@0x0000000016fedf60 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,25

[1] 1 4 9 16 25

```

Show how if the value associated with `v` is visible to more than one variable that R will not performing in-place replacement (making calculation much more expensive).

```
computeSquares(5, TRUE)
```

```

@0x0000000018668250 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 1,2,3,4,5
@0x0000000017043c20 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,2,3,4,5
@0x0000000017043c88 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,3,4,5
@0x0000000017043948 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,4,5
@0x00000000170439b0 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,16,5
@0x00000000184fa640 14 REALSXP g0c4 [NAM(2)] (len=5, tl=0) 1,4,9,16,25

[1] 1 4 9 16 25

```

Check if we can use PRYR for addresses in this case.

```
computeSquares(5, FALSE, usePRYR=TRUE)
```

```

@0x0000000019598088 13 INTSXP g0c3 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
[1] "0x19598088"
@0x000000001807c7b0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,2,3,4,5
[1] "0x1807c7b0"
@0x000000001807c7b0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,3,4,5
[1] "0x1807c7b0"
@0x000000001807c7b0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,4,5
[1] "0x1807c7b0"
@0x000000001807c7b0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,5
[1] "0x1807c7b0"
@0x000000001807c7b0 14 REALSXP g0c4 [NAM(1)] (len=5, tl=0) 1,4,9,16,25
[1] "0x1807c7b0"

[1] 1 4 9 16 25

```