Partitioning Mutate, Example 2

John Mount, Win-Vector LLC

2017-11-20

This is a follow-on example of the use of `seplyr::partition_mutate_qt()` showing a larger block sequence based on swaps.[1] For motivation and context please see the first article.

```
packageVersion("dplyr")
```

```
## [1] '0.7.4'
```

```
library("seplyr")
```

```
## Loading required package: wrapr
```

```
packageVersion("seplyr")
```

```
## [1] '0.1.7'
```

```
sc <-
  sparklyr::spark_connect(version = '2.2.0',
                          master = "local")
dL <- data.frame(rowNum = 1:5)
d <- dplyr::copy_to(sc, dL,
                    name = 'd',
                    overwrite = TRUE,
                    temporary = TRUE)

class(d)
```

```
## [1] "tbl_spark" "tbl_sql"   "tbl_lazy"
## [4] "tbl"
```

```
dplyr::glimpse(d)
```

```
## Observations: 5
## Variables: 1
## $ rowNum <int> 1, 2, 3, 4, 5
```

It is often necessary to simulate block commands with `ifelse()` style functionality. For example if we want to assign complimentary pairs of users into treatment and control for many groups we might use code such as the following.[2]

Suppose we wish to assign columns in a complementary to treatment and control design[3] And further suppose we want to keep the random variables driving our decisions around for diagnosis and debugging.

To write such a procedure in pure `dplyr` we might simulate block with code such as the following[4]

[1] The source code for this article can be found here.

[2] A better overall design would be to use `cdata::moveValuesToRowsN()`, then perform a single bulk operation on rows, and then pivot/transpose back with `cdata::moveValuesToColumnsN()`. But let's see how we simply work with a problem at hand.

[3] Abraham Wald designed some sequential analysis procedures in this way as Nina Zumel remarked. Another string example is conditionals where you are trying to vary on a per-row basis which column is assigned to, instead of varying what value is assigned from.

[4] Only showing work on the `a` group right now. We are assuming we want to perform this task on all the grouped letter columns.

```
nrow <- nrow(dL)
dL %.>%
 dplyr::mutate(.,
  rand_a := runif(nrow),
   choice_a := rand_a>=0.5,
    a_1 := ifelse(choice_a,
                  'treatment',
                  'contol'),
    a_2 := ifelse(choice_a,
                  'control',
                  'treatment')
  ) %.>%
  dplyr::glimpse(.)
```

```
## Observations: 5
## Variables: 5
## $ rowNum   <int> 1, 2, 3, 4, 5
## $ rand_a   <dbl> 0.71855184, 0.28867647,...
## $ choice_a <lgl> TRUE, FALSE, FALSE, TRU...
## $ a_1      <chr> "treatment", "contol", ...
## $ a_2      <chr> "control", "treatment",...
```

Above we are using the indent notation to indicate the code-blocks we are simulating with the `ifelse()` notation.[5]

With big data in `Spark` we could try something like the following:

```
d %.>%
 dplyr::mutate(.,
  rand_a := rand(),
   choice_a := rand_a>=0.5,
    a_1 := ifelse(choice_a,
                  'treatment',
                  'contol'),
    a_2 := ifelse(choice_a,
                  'control',
                  'treatment')
  )
# Error: org.apache.spark.sql.AnalysisException: cannot resolve '`choice_a`' ...
```

This currently fails due to the chain of dependence between `rand_a`, `choice_a` and `a_1. However we want to write the transform in as few`mutate()‘ statements as practical because: sequencing mutates is implemented through nesting queries (which eventually fail).

```
d %.>%
 dplyr::mutate(., rand_a := rand()) %.>%
```

[5] What we are working around is the lack of an operator that allows us to select per-row where assignments go, which would complement `ifelse()`'s ability to select per-row where values come from.

```
  dplyr::mutate(., choice_a := rand_a>=0.5) %.>%
   dplyr::mutate(., a_1 := ifelse(choice_a,
                                  'treatment',
                                  'contol')) %.>%
   dplyr::mutate(., a_2 := ifelse(choice_a,
                                  'control',
                                  'treatment')) %.>%
  dplyr::show_query(.)
```

```
## <SQL>
## SELECT `rowNum`, `rand_a`, `choice_a`, `a_1`, CASE WHEN (`choice_a`) THEN ("control") ELSE ("treatmen
## FROM (SELECT `rowNum`, `rand_a`, `choice_a`, CASE WHEN (`choice_a`) THEN ("treatment") ELSE ("contol
## FROM (SELECT `rowNum`, `rand_a`, `rand_a` >= 0.5 AS `choice_a`
## FROM (SELECT `rowNum`, RAND() AS `rand_a`
## FROM `d`) `hukmvqguot`) `hucuebsnxp`) `ewfhjmfjfp`
```

`seplyr::partition_mutate_qt()` is designed to fix this in a performant manner.[6]

Let's try this query again:

```
plan <-
 partition_mutate_qt(
  rand_a := rand(),
  choice_a := rand_a>=0.5,
   a_1 := ifelse(choice_a,
                 'treatment',
                 'contol'),
   a_2 := ifelse(choice_a,
                 'control',
                 'treatment')
  )
print(plan)
```

```
## $group00001
##   rand_a
## "rand()"
##
## $group00002
##        choice_a
## "rand_a >= 0.5"
##
## $group00003
##                                             a_1
##  "ifelse(choice_a, \"treatment\", \"contol\")"
##                                             a_2
## "ifelse(choice_a, \"control\", \"treatment\")"
```

[6] And as we discussed before we have reason to believe the upcoming `dplyr` fix will be simple in-order `mutate()` splitting, which can not be performant on `Sparklyr` due to sequential statement nesting, again please see our earlier note.

```
res <- d
for(stepi in plan) {
  res <- mutate_se(res, stepi, splitTerms = FALSE)
}
dplyr::glimpse(res)
```

```
## Observations: 5
## Variables: 5
## $ rowNum   <int> 1, 2, 3, 4, 5
## $ rand_a   <dbl> 0.94408630, 0.37318891,...
## $ choice_a <lgl> TRUE, FALSE, FALSE, FAL...
## $ a_1      <chr> "treatment", "contol", ...
## $ a_2      <chr> "control", "treatment",...
```

That worked! The point of this note is: this will also work with a much longer sequence.[7]

```
plan <-
 partition_mutate_qt(
  rand_a := rand(),
   choice_a := rand_a>=0.5,
    a_1 := ifelse(choice_a,
                  'treatment',
                  'contol'),
    a_2 := ifelse(choice_a,
                  'control',
                  'treatment'),
  rand_b := rand(),
   choice_b := rand_b>=0.5,
    b_1 := ifelse(choice_b,
                  'treatment',
                  'contol'),
    b_2 := ifelse(choice_b,
                  'control',
                  'treatment'),
  rand_c := rand(),
   choice_c := rand_c>=0.5,
    c_1 := ifelse(choice_c,
                  'treatment',
                  'contol'),
    c_2 := ifelse(choice_c,
                  'control',
                  'treatment'),
  rand_d := rand(),
   choice_d := rand_d>=0.5,
```

[7] Please keep in mind: we are using a very simple and regular sequence only for purposes of illustration. There are better ways to perform this particular vary regular assignment. That is not going to be the case with non-trivial `Sparklyr` applications, in particular those that are ports of large existing systems.

```
    d_1 := ifelse(choice_d,
                  'treatment',
                  'contol'),
    d_2 := ifelse(choice_d,
                  'control',
                  'treatment'),
  rand_e := rand(),
   choice_e := rand_e>=0.5,
    e_1 := ifelse(choice_e,
                  'treatment',
                  'contol'),
    e_2 := ifelse(choice_e,
                  'control',
                  'treatment')
  )
print(plan)
```

```
## $group00001
##   rand_a   rand_b   rand_c   rand_d   rand_e
## "rand()" "rand()" "rand()" "rand()" "rand()"
##
## $group00002
##        choice_a        choice_b
## "rand_a >= 0.5" "rand_b >= 0.5"
##        choice_c        choice_d
## "rand_c >= 0.5" "rand_d >= 0.5"
##        choice_e
## "rand_e >= 0.5"
##
## $group00003
##                                             a_1
##  "ifelse(choice_a, \"treatment\", \"contol\")"
##                                             a_2
## "ifelse(choice_a, \"control\", \"treatment\")"
##                                             b_1
##  "ifelse(choice_b, \"treatment\", \"contol\")"
##                                             b_2
## "ifelse(choice_b, \"control\", \"treatment\")"
##                                             c_1
##  "ifelse(choice_c, \"treatment\", \"contol\")"
##                                             c_2
## "ifelse(choice_c, \"control\", \"treatment\")"
##                                             d_1
##  "ifelse(choice_d, \"treatment\", \"contol\")"
##                                            d_2
## "ifelse(choice_d, \"control\", \"treatment\")"
##                                            e_1
##  "ifelse(choice_e, \"treatment\", \"contol\")"
##                                            e_2
## "ifelse(choice_e, \"control\", \"treatment\")"
```

```
res <- d
for(stepi in plan) {
  res <- mutate_se(res,
                   stepi,
                   splitTerms = FALSE)
}
dplyr::glimpse(res)
```

```
## Observations: 5
## Variables: 21
## $ rowNum   <int> 1, 2, 3, 4, 5
## $ rand_a   <dbl> 0.4446904, 0.5263769, 0...
## $ rand_b   <dbl> 0.05909384, 0.95070253,...
## $ rand_c   <dbl> 0.9656228, 0.6200699, 0...
## $ rand_d   <dbl> 0.0009386058, 0.4048268...
## $ rand_e   <dbl> 0.2256684, 0.8840970, 0...
## $ choice_a <lgl> FALSE, TRUE, FALSE, TRU...
## $ choice_b <lgl> FALSE, TRUE, FALSE, FAL...
## $ choice_c <lgl> TRUE, TRUE, TRUE, FALSE...
## $ choice_d <lgl> FALSE, FALSE, FALSE, TR...
## $ choice_e <lgl> FALSE, TRUE, FALSE, FAL...
## $ a_1      <chr> "contol", "treatment", ...
## $ a_2      <chr> "treatment", "control",...
## $ b_1      <chr> "contol", "treatment", ...
## $ b_2      <chr> "treatment", "control",...
## $ c_1      <chr> "treatment", "treatment...
## $ c_2      <chr> "control", "control", "...
## $ d_1      <chr> "contol", "contol", "co...
## $ d_2      <chr> "treatment", "treatment...
## $ e_1      <chr> "contol", "treatment", ...
## $ e_2      <chr> "treatment", "control",...
```

```
dplyr::show_query(res)
```

```
## <SQL>
## SELECT `rowNum`, `rand_a`, `rand_b`, `rand_c`, `rand_d`, `rand_e`, `choice_a`, `choice_b`, `choice_c
## FROM (SELECT `rowNum`, `rand_a`, `rand_b`, `rand_c`, `rand_d`, `rand_e`, `rand_a` >= 0.5 AS `choice_
## FROM (SELECT `rowNum`, RAND() AS `rand_a`, RAND() AS `rand_b`, RAND() AS `rand_c`, RAND() AS `rand_d
## FROM `d`) `bhelpmciyx`) `lepxyqhyzm`
```

Notice the above still only broke the query into three blocks *independent of the number of blocks we are trying to simulate in the mutate*. Further notice that in turn the depth of derived `SQL` query nesting was only the number of blocks (again 3).

The number of blocks is the dependency depth of the system of assignments, which can be *much* smaller than the number of new-values used (the number of blocks a non-reordering split may use, probably already around 10 blocks even in this example; and growing as the number of blocks grow).

`seplyr::partition_mutate_qt()` type capability is essential for executing non-trivial code at scale in `Sparklyr`.

Win-Vector LLC supplies a number of open-source `R` packages for working effectively with big data. These include:

- **wrapr**: supplies code re-writing tools that make coding *over* `dplyr` much easier.
- **cdata**: supplies pivot/un-pivot functionality at big data scale.
- **seplyr**: supplies improved interfaces for many data manipulation tasks.
- **replyr**: supplies tools and patches for using `dplyr` on big data.

And issues such as the above are often discussed on the Win-Vector blog.