Partitioning Mutate, Example 2

John Mount, Win-Vector LLC

2017-11-24

`Sparklyr`, with its `dplyr` translations allows `R`, to perform the heavy lifting that has traditionally been the exclusive domain of proprietary systems such as `SAS`. In general, `dplyr` is good at handling intermediate variables in the mutate function so users don't need to think about it. However, some of that breaks down when the processing is done on the `Apache Spark` side. Win-Vector LLC developed the `seplyr` package to use with consulting clients to mitigate some of these situations.[1] In this article we will demonstrate we `seplyr` functions: `if_else_device()` and `partition_mutate_qt()`.

[1] And we distribute the package as open-source to give back to the `R` community.

This is a follow-on example building on our "Partitioning Mutate" article, showing a larger block sequence based on swaps.[2] For more motivation and context please see the first article.

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

Please consider the following example data (on a remote `Spark` cluster).

```
class(d)
```

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

```
d %.>%
  # avoid https://github.com/tidyverse/dplyr/issues/3216
  dplyr::collect(.) %.>%
  knitr::kable(.)
```

rowNum	a_1	a_2	b_1	b_2	c_1	c_2	d_1	d_2	e_1	e_2
1	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
2	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
3	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
4	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
5	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

We find in non-trivial projects it is often necessary to simulate block-`if(){}else{}` structures in `dplyr` pipelines.

For our example: suppose we wish to assign columns in a complementary to treatment and control design[3]

[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.

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

[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.

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

```
## [1] '0.5.2'
```

```
plan <- if_else_device(
  testexpr =
    "rand()>=0.5",
     thenexprs = c(
       "a_1" := "'treatment'",
       "a_2" := "'control'"),
     elseexprs =  c(
       "a_1" := "'control'",
       "a_2" := "'treatment'"))  %.>%
  partition_mutate_se(.)
```

We are using the indent notation to indicate the code-blocks we are simulating with row-wise `if(){}else{}` blocks.[5] The `if_else_device` is also using quoted expressions (or value-oriented standard notation).[6]

In the end we can examine and execute the mutate plan:

[5] For more on this concept, please see: the `if_else_device` reference.

[6] One can over-worry about this, but in the end all a non-standard evaluation scheme saves you is a few quote marks (at the cost of transparency, and a lot of down-stream headaches).

Our advice is to compose the expressions using your smart `R`-code editor of choice and then throw on the additional quote marks after you have the statements as you want them.

```
print(plan)
```

```
## $group00001
## ifebtest_m0bj42h52urx
##         "rand()>=0.5"
##
## $group00002
##                                                 a_1
## "ifelse( ifebtest_m0bj42h52urx, 'treatment', a_1)"
##                                                 a_2
##   "ifelse( ifebtest_m0bj42h52urx, 'control', a_2)"
##
## $group00003
##                                                      a_1
##   "ifelse( !( ifebtest_m0bj42h52urx ), 'control', a_1)"
##                                                      a_2
## "ifelse( !( ifebtest_m0bj42h52urx ), 'treatment', a_2)"
```

```
d %.>%
  mutate_seb(., plan) %.>%
  select_se(., grepdf('^ifebtest_.*', ., invert=TRUE)) %.>%
  dplyr::collect(.) %.>%
  knitr::kable(.)
```

rowNum	a_1	a_2	b_1	b_2	c_1	c_2	d_1	d_2	e_1	e_2
1	control	treatment	NA	NA	NA	NA	NA	NA	NA	NA
2	control	treatment	NA	NA	NA	NA	NA	NA	NA	NA

rowNum	a_1	a_2	b_1	b_2	c_1	c_2	d_1	d_2	e_1	e_2
3	control	treatment	NA	NA	NA	NA	NA	NA	NA	NA
4	treatment	control	NA	NA	NA	NA	NA	NA	NA	NA
5	control	treatment	NA	NA	NA	NA	NA	NA	NA	NA

Our larger goal was to perform this same operation on each of the 5 letter groups.

We do this easily as follows:[7]

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

```
plan <- lapply(c('a', 'b', 'c', 'd', 'e'),
               function(gi) {
                 if_else_device(
                   "rand()>=0.5",
                   thenexprs = c(
                     paste0(gi, "_1") := "'treatment'",
                     paste0(gi, "_2") := "'control'"),
                   elseexprs =  c(
                     paste0(gi, "_1") := "'control'",
                     paste0(gi, "_2") := "'treatment'"))
               }) %.>%
  unlist(.) %.>%
  partition_mutate_se(.)

d %.>%
  mutate_seb(., plan) %.>%
  select_se(., grepdf('^ifebtest_.*', ., invert=TRUE)) %.>%
  dplyr::collect(.) %.>%
  knitr::kable(.)
```

rowNum	a_1	a_2	b_1	b_2	c_1	c_2	d_1	d_2	e_1
1	treatment	control	control	treatment	control	treatment	treatment	control	control
2	treatment	control	control	treatment	treatment	control	control	treatment	treatment
3	treatment	control	control	treatment	control	treatment	control	treatment	treatment
4	control	treatment	treatment	control	treatment	control	treatment	control	treatment
5	treatment	control	control	treatment	control	treatment	control	treatment	treatment

Please keep in mind: we are using a very simple and regular sequence only for purposes of illustration. The intent is to show the types of issues one runs into when standing-up non-trivial applications in `Sparklyr`.

The purpose of `seplyr::partition_mutate_qt()` is to re-arrange statements and break them into blocks of non-dependent statements (no statement in a block depends on any other in the same block, and all value dependencies are respected by the block order). `seplyr::partition_mutate_qt()` if further defined to do this in a performant manner.[8]

[8] That is to pick a small number of blocks, in our case the plan consisted of 3 blocks. The simple method of introducing a block boundary at each first use of derived value (without statement re-ordering) would create a very much larger set of blocks (which cause problems of their own). In particular the impression code and comments of upcoming `dplyr` fix appear to indicate an undesirable large number of blocks solution.

Without such partition planning the current version of `dplyr` (0.7.4) the results of `dplyr::mutate()` do not seem to be well-defined when values are created and re-used in the same `dplyr::mutate()` block. This is not a currently documented limitation, but it is present:

```
ex <- dplyr::mutate(d,
                    condition_tmp = rand()>=0.5,
                    a_1 = ifelse( condition_tmp,
                                  'treatment',
                                  a_1),
                    a_2 = ifelse( condition_tmp,
                                  'control',
                                  a_2),
                    a_1 = ifelse( !( condition_tmp ),
                                  'control',
                                  a_1),
                    a_2 = ifelse( !( condition_tmp ),
                                  'treatment',
                                  a_2))

knitr::kable(dplyr::collect(dplyr::select(ex, a_1, a_2)))
```

a_1	a_2
control	treatment
NA	control
NA	control
NA	control
NA	control

Notice above the many `NA` columns, which are errors.[9]

[9] Note: no mere re-ordering of the statements would give this result.

```
dplyr::show_query(ex)
```

```
## <SQL>
## SELECT `rowNum`, `a_1`, `b_1`, `b_2`, `c_1`, `c_2`, `d_1`, `d_2`, `e_1`, `e_2`, `condition_tmp`, CAS
## FROM (SELECT `rowNum`, `b_1`, `b_2`, `c_1`, `c_2`, `d_1`, `d_2`, `e_1`, `e_2`, `condition_tmp`, CASE
## FROM (SELECT `rowNum`, `a_1`, `a_2`, `b_1`, `b_2`, `c_1`, `c_2`, `d_1`, `d_2`, `e_1`, `e_2`, RAND()
## FROM `d`) `gqogyxwwtg`) `rkrxzpmkhg`
```

Looking at the query we see that one of the conditional statements is missing (notice only 3 case statements, not 4):[10]

[10] Likely the `dplyr` `SQL` generator does not perform a correct live-value analysis and therefor gets fooled into thinking a statement can safely be eliminated (when it can not). `seplyr::partition_mutate_qt()` performs a correct live value calculation and make sure `dplyr::mutate()` is only seeing trivial blocks (blocks where no value depends on any calculation in the same block).

Conclusion

`seplyr::if_else_device()` and `seplyr::partition_mutate_qt()` type capability is essential for executing non-trivial code at scale in `Sparklyr`. For more on the `if_else_device` we suggest reading up on the function reference example, and for a review on the `partition_mutate` variations we suggest the "Partitioning Mutate" article.

Links

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* "non standard evaluation" interfaces (such as `dplyr`) *much* easier.
- **cdata**: supplies pivot/un-pivot functionality at big data scale.
- **rquery**: (in development) big data scale relational data operators.
- **seplyr**: supplies improved interfaces for many data manipulation tasks.
- **replyr**: supplies tools and patches for using `dplyr` on big data.

Partitioning mutate articles:

- **Partitioning Mutate**: basic example.
- **Partitioning Mutate, Example 2**: `ifelse` example.
- **Partitioning Mutate, Example 3** `rquery` example.

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