Partitioning Mutate

John Mount, Win-Vector LLC

2017-11-19

When using `R` to work with a big-data data service such as `Apache Spark` using `sparklyr` the following considerations are critical.

- You must cache and partition at points.[1]
- You must try to limit the set of columns you are working on (so that you are working on small cache-able projections of your large data).[2]
- You must try to limit the number of sequential steps you specify as they are *actualy implemented by nesting of queries*.[3]

The point is: you can't always expect code that is not adapted to the environment run well.

Let's set up a working example.[4]

[1] However, you must limit how often you do this and free unneeded caches.

[2] The query optimizer may not be able to skip over producing columns that you are not actually using, but are in fact specified in intermediate queries.

[3] The nesting gets expensive and eventually fails. A classic example of a leaky abstraction. We have simple examples of too many sequenced `mutates()` exhausting `Sparklyr`.

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

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

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

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

```
## [1] '0.5.1'
```

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

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

```
sc <-
  sparklyr::spark_connect(version = '2.2.0',
                          master = "local")
d <- dplyr::starwars %.>%
  select_se(., qc(name,
                  height, mass,
                  hair_color,
                  eye_color,
                  birth_year)) %.>%
  dplyr::copy_to(sc, ., name = 'starwars')
```

```
class(d)
```

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

```
d %.>%
  head(.) %.>%
  dplyr::collect(.) %.>%
  knitr::kable(.)
```

name	height	mass	hair_color	eye_color	birth_year
Luke Skywalker	172	77	blond	blue	19.0
C-3PO	167	75	NA	yellow	112.0
R2-D2	96	32	NA	red	33.0
Darth Vader	202	136	none	yellow	41.9
Leia Organa	150	49	brown	brown	19.0
Owen Lars	178	120	brown, grey	blue	52.0

The issue is: generalizations of the following pipeline can be very expensive to realize (due to the nesting of queries).

```
d %.>%
  dplyr::mutate(., a := 1) %.>%
  dplyr::mutate(., b := 2) %.>%
  dplyr::mutate(., c := 3) %.>%
  dplyr::show_query(.)
```

```
## <SQL>
## SELECT `name`, `height`, `mass`, `hair_color`, `eye_color`, `birth_year`, `a`, `b`, 3.0 AS `c`
## FROM (SELECT `name`, `height`, `mass`, `hair_color`, `eye_color`, `birth_year`, `a`, 2.0 AS `b`
## FROM (SELECT `name`, `height`, `mass`, `hair_color`, `eye_color`, `birth_year`, 1.0 AS `a`
## FROM `starwars`) `ynkftgutsg`) `gwllklbvir`
```

The seemingly equivalent pipeline can be much more performant:

```
d %.>%
  dplyr::mutate(.,
                a := 1,
                b := 2,
                c := 3) %.>%
  dplyr::show_query(.)
```

```
## <SQL>
## SELECT `name`, `height`, `mass`, `hair_color`, `eye_color`, `birth_year`, 1.0 AS `a`, 2.0 AS `b`, 3.0
## FROM `starwars`
```

However: it is hard to give the advice "put everything into one mutate" as the exact availability and semantics of derived columns has never really been specified in `dplyr`[5]

The additional confounding issue is code like the following currently throws:

```
dplyr::mutate(d,
   a := 1,
   b := a,
   c := b)
```

[5] It is more often a bit if "it works in memory, and it may or may not work against big data sources." `sparklyr` issue 1015, `dplyr` issue 2481, and `dplyr` issue 3095.

```
# Error: org.apache.spark.sql.AnalysisException: cannot resolve '`b`'
```

It appears there is a `dplyr` fix in the works.[6]

If the included descriptive comment:

[6] `dplyr` commit "Improve subquery splitting in mutate"

```
 # For each expression, check if it uses any newly created variables.
 # If so, nest the mutate()
```

correctly describes the calculation sequence (possibly nest once per expression), then the mutate would introduce a new stage at each first use of a derived column.

That would mean a sequence such as the following would in fact be broken into a sequence of mutates, with a new mutate introduced at least after each condition.[7]

[7] This code is simulating a sequence of blocks of conditional column assignments. Such code is quite common in production `Spark` projects, especially those involving the translation of legacy imperative code such as `SAS`. The issue is: we don't have a control structure that chooses which column to assign to, until we introduce `seplyr::if_else_device()`.

That is the following would get translated from this:

```
d %.>%
  dplyr::mutate(.,
                condition1 := height>=150,
                  mass := ifelse(condition1,
                                 mass + 10,
                                 mass),
                  hair_color := ifelse(condition1,
                                       'brown',
                                       hair_color),
                condition2 := birth_year<50,
                  eye_color := ifelse(condition2,
                                      'blue',
                                      eye_color),
                  name := ifelse(condition2,
                                 tolower(name),
                                 name))
```

To something like this:

```
d %.>%
  dplyr::mutate(.,
                condition1 := height>=150)  %.>%
     dplyr::mutate(.,
                   mass := ifelse(condition1,
                                  mass + 10,
                                  mass),
                   hair_color := ifelse(condition1,
                                        'brown',
                                        hair_color),
```

```
                condition2 := birth_year<50)  %.>%
    dplyr::mutate(.,
                  eye_color := ifelse(condition2,
                                      'blue',
                                      eye_color),
                  name := ifelse(condition2,
                                 tolower(name),
                                 name))
```

Now it might be the case it takes 3 or more levels of dependence to trigger the issue, but the issue remains:

> The `mutate` gets broken into a number of sub-`mutate`s proportional to the number of derived columns used later, and not proportional to the (usually much smaller) dependency depth of re-uses.

This can be a problem. We have routinely seen blocks where there are 50 or more such variables re-used. This is when the dependence depth is only 2 or 3 (meaning the expressions could be re-ordered efficiently).

The thing we are missing is: all of the condition calculations could be done together in one step (as they do not depend on each other) and then all the statements that depend on their consequences can also be executed in another large step.

`seplyr::partition_mutate_qt()` supplies exactly the needed partitioning service.[8]

[8] We could try to re-order the statements by hand- but then we would break up all of the simulated code blocks and produce hard to read and maintain code. It is better to keep the code in a meaningful arrangement and have a procedure to re-optimize the code to minimize nesting.

```
plan <- partition_mutate_qt(
  condition1 := height>=150,
     mass := ifelse(condition1,
                    mass + 10, mass),
     hair_color := ifelse(condition1,
                          'brown', hair_color),
  condition2 := birth_year<50,
     eye_color := ifelse(condition2,
                         'blue', eye_color),
     name := ifelse(condition2,
                    tolower(name), name))
print(plan)
```

```
## $group00001
##        condition1        condition2
##   "height >= 150" "birth_year < 50"
##
## $group00002
##                                    mass
```

```
##       "ifelse(condition1, mass + 10, mass)"
##                                   hair_color
## "ifelse(condition1, \"brown\", hair_color)"
##                                    eye_color
##   "ifelse(condition2, \"blue\", eye_color)"
##                                        name
##   "ifelse(condition2, tolower(name), name)"

res <- mutate_seb(d, plan)

res  %.>%
  dplyr::show_query(.)

## <SQL>
## SELECT `height`, `birth_year`, `condition1`, `condition2`, CASE WHEN (`condition1`) THEN (`mass` + 1
## FROM (SELECT `name`, `height`, `mass`, `hair_color`, `eye_color`, `birth_year`, `height` >= 150.0 AS
## FROM `starwars`) `mgmdvatwkf`

res %.>%
  head(.) %.>%
  # collect to avoid https://github.com/rstudio/sparklyr/issues/1134
  dplyr::collect(.)  %.>%
  knitr::kable(.)
```

height	birth_year	condition1	condition2	mass	hair_color	eye_color	name
172	19.0	TRUE	TRUE	87	brown	blue	luke skywalker
167	112.0	TRUE	FALSE	85	brown	yellow	C-3PO
96	33.0	FALSE	TRUE	32	NA	blue	r2-d2
202	41.9	TRUE	TRUE	146	brown	blue	darth vader
150	19.0	TRUE	TRUE	59	brown	blue	leia organa
178	52.0	TRUE	FALSE	130	brown	blue	Owen Lars

The idea is: no matter how many statements are present `seplyr::partition_mutate_qt()` breaks the `mutate()` statement into a sequence of length proportional only the the value dependency depth (in this case: 2), and *not* proportional to the number of introduced values (which can be as long as the number of conditions introduced).

The above situation is admittedly ugly, but not something you can wish away if you want to support actual production use-cases.[9]

For an example bringing out more of these issues please see here.

[9] And if you want to support porting working code from other systems, meaning a complete re-design is not on the table.

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.
- **seplyr**: supplies improved interfaces for many data manipulation tasks.
- **replyr**: supplies tools and patches for using `dplyr` on big data.

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