R users have been enjoying the benefits of SQL query generators for quite some time, most notably using the dbplyr package. I would like to talk about some features of our own rquery query generator, concentrating on derived result re-use.

**Introduction**

SQL represents value use by nesting. To use a query result within another query one writes the query to be used inside the query using the values. R query generators can also represent value use by nesting, but they also tend to expose a sequential pipe notation where value use is represented by sequencing.

We will demonstrate this with an example.

library("rquery")

raw\_connection <- DBI::dbConnect(RSQLite::SQLite(),

":memory:")

db\_rquery <- rquery\_db\_info(

connection = raw\_connection,

is\_dbi = TRUE,

connection\_options = rq\_connection\_tests(raw\_connection))

d <- data.frame(x = -3:3)

d0 <- rq\_copy\_to(db\_rquery, "d", d)

print(d0)

## [1] "table(`d`; x)"

ops <- d0 %.>%

extend(., y = x^2 -1) %.>%

select\_rows(., x > y)

cat(format(ops))

## table(`d`;

## x) %.>%

## extend(.,

## y := x ^ 2 - 1) %.>%

## select\_rows(.,

## x > y)

Notice how in the rquery operation we write operations one after another (using the dot-arrow pipe notation), and the operations are similarly printed.

Whereas the generated SQL is written in a nested format.

cat(to\_sql(ops, db\_rquery))

## SELECT \* FROM (

## SELECT

## `x`,

## POWER ( `x` , 2 ) - 1 AS `y`

## FROM (

## SELECT

## `x`

## FROM

## `d`

## ) tsql\_38280614414821583912\_0000000000

## ) tsql\_38280614414821583912\_0000000001

## WHERE `x` > `y`

Notice the database handle is kept separate from the operators. Furthermore execution is not entangled with operator definition, but is a separate step (performed through materialze(), execute() or even sending the query to the database via a pipe).

ops %.>%

db\_rquery %.>%

knitr::kable(.)

| **x** | **y** |
| --- | --- |
| 0 | -1 |
| 1 | 0 |

**Re-using derived values**

The piped operator notation runs into trouble if we re-use derived values. SQL can re-use tables and views, but can not conveniently re-use queries. So if use query results more than once the notation expands our [directed acyclic graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph) specification into a possibly much larger tree.

The following is an artificial example to demonstrate the issue.

d1 <- natural\_join(d0, d0,

by = "x", jointype = "LEFT")

d2 <- natural\_join(d1, d1,

by = "x", jointype = "LEFT")

d3 <- natural\_join(d2, d2,

by = "x", jointype = "LEFT")

cat(format(d3))

## table(`d`;

## x) %.>%

## natural\_join(.,

## table(`d`;

## x),

## j= LEFT, by= x) %.>%

## natural\_join(.,

## table(`d`;

## x) %.>%

## natural\_join(.,

## table(`d`;

## x),

## j= LEFT, by= x),

## j= LEFT, by= x) %.>%

## natural\_join(.,

## table(`d`;

## x) %.>%

## natural\_join(.,

## table(`d`;

## x),

## j= LEFT, by= x) %.>%

## natural\_join(.,

## table(`d`;

## x) %.>%

## natural\_join(.,

## table(`d`;

## x),

## j= LEFT, by= x),

## j= LEFT, by= x),

## j= LEFT, by= x)

Notice the depth 3 expression exploded into tree with 7 joins.

rquery‘s query diagrammer can help spot and diagnose these issues.

d3 %.>%

op\_diagram(., merge\_tables = TRUE) %.>%

DiagrammeR::grViz(.) %.>%

DiagrammeRsvg::export\_svg(.) %.>%

write(., file="query\_growth\_diagram.svg")

## Warning in op\_diagram(., merge\_tables = TRUE): possible repeated calculation:

## natural\_join(.1, .2, j= LEFT, by= x)

The gold nodes are possibly repeated calculations, and the warning also notes the issue.

The above example may seem unnatural- but there are a number of places where it is natural to re-use intermediate results. One such circumstance is comparing values in groups without the use of window functions

The query explosion issue is not unique to rquery, dplyr has the same issue.

library("dplyr")

##

## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':

##

## filter, lag

## The following objects are masked from 'package:base':

##

## intersect, setdiff, setequal, union

d0\_dplyr <- tbl(raw\_connection, "d")

d1\_dplyr <- left\_join(d0\_dplyr, d0\_dplyr,

by = "x")

d2\_dplyr <- left\_join(d1\_dplyr, d1\_dplyr,

by = "x")

d3\_dplyr <- left\_join(d2\_dplyr, d2\_dplyr,

by = "x")

dbplyr::remote\_query(d3\_dplyr)

## SELECT `TBL\_LEFT`.`x` AS `x`

## FROM (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM `d` AS `TBL\_LEFT`

## LEFT JOIN `d` AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_LEFT`

## LEFT JOIN (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM `d` AS `TBL\_LEFT`

## LEFT JOIN `d` AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_LEFT`

## LEFT JOIN (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM `d` AS `TBL\_LEFT`

## LEFT JOIN `d` AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_LEFT`

## LEFT JOIN (SELECT `TBL\_LEFT`.`x` AS `x`

## FROM `d` AS `TBL\_LEFT`

## LEFT JOIN `d` AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

## ) AS `TBL\_RIGHT`

## ON (`TBL\_LEFT`.`x` = `TBL\_RIGHT`.`x`)

One could hope the query optimizer will detect and eliminate the common sub-expressions, but that is not always going to be the case. In fact sometimes the very size of a query turns off the query optimizer in systems such as Spark. It is better to organize your calculation to not emit so many common sub-expressions in the first place. And we also show a variation of the example using PostreSQL where landing intermediate results speeds up both dplyr and rquery (dplyr being faster in this particular example; which is not always the case and may be the case, and here may be due to rquery‘s natural\_join() coalesce semantics).

**Basic solutions**

dplyr can easily overcome this limitation with its compute() node, which lands or materializes intermediate results for re-use.

d1\_dplyr <- compute(left\_join(d0\_dplyr, d0\_dplyr,

by = "x"))

d2\_dplyr <- compute(left\_join(d1\_dplyr, d1\_dplyr,

by = "x"))

d3\_dplyr <- compute(left\_join(d2\_dplyr, d2\_dplyr,

by = "x"))

dbplyr::remote\_query(d3\_dplyr)

## SELECT \*

## FROM `ofsgqnxcku`

Notice the dplyr::compute() results are actually tables (not general queries).

rquery can also fix the issue by landing intermediate results, though the table lifetime tracking is intentionally more explicit (and the separation between code and results is also deliberately much more explicit).

tmps <- wrapr::mk\_tmp\_name\_source()

d1\_mat <- materialize(

db\_rquery,

natural\_join(d0, d0,

by = "x", jointype = "LEFT"),

table\_name = tmps(),

temporary = TRUE, overwrite = TRUE)

d2\_mat <- materialize(

db\_rquery,

natural\_join(d1\_mat, d1\_mat,

by = "x", jointype = "LEFT"),

table\_name = tmps(),

temporary = TRUE, overwrite = TRUE)

d3\_mat <- materialize(

db\_rquery,

natural\_join(d2\_mat, d2\_mat,

by = "x", jointype = "LEFT"),

table\_name = tmps(),

temporary = TRUE, overwrite = TRUE)

cat(format(d3\_mat))

## table(`tmpnam\_82654876444053153969\_0000000002`;

## x)

**More advanced solutions**

With a more advanced “relop\_list” notation we can both build the efficient query plan, but also the diagram certifying the lack of redundant stages.

relop\_list <- rquery::make\_relop\_list(tmps)

d1\_ops <- natural\_join(d0, d0,

by = "x", jointype = "LEFT") %.>%

relop\_list

d2\_ops <- natural\_join(d1\_ops, d1\_ops,

by = "x", jointype = "LEFT") %.>%

relop\_list

d3\_ops <- natural\_join(d2\_ops, d2\_ops,

by = "x", jointype = "LEFT") %.>%

relop\_list

relop\_list %.>%

get\_relop\_list\_stages(.) %.>%

op\_diagram(., merge\_tables = TRUE) %.>%

DiagrammeR::grViz(.) %.>%

DiagrammeRsvg::export\_svg(.) %.>%

write(., file="query\_growth\_diagram2.svg")

The relop\_list collector is introducing and managing intermediate tables. It is simple to materialized inspect the results.

print(relop\_list)

## $tmpnam\_82654876444053153969\_0000000003

## [1] "table(`d`; x) %.>% natural\_join(., table(`d`; x), j= LEFT, by= x)"

##

## $tmpnam\_82654876444053153969\_0000000004

## [1] "table(tmpnam\_82654876444053153969\_0000000003; x) %.>% natural\_join(., table(tmpnam\_82654876444053153969\_0000000003; x), j= LEFT, by= x)"

##

## $tmpnam\_82654876444053153969\_0000000005

## [1] "table(tmpnam\_82654876444053153969\_0000000004; x) %.>% natural\_join(., table(tmpnam\_82654876444053153969\_0000000004; x), j= LEFT, by= x)"

result <- relop\_list %.>% db\_rquery

print(result)

## [1] "table(`tmpnam\_82654876444053153969\_0000000005`; x)"

DBI::dbReadTable(raw\_connection, result$table\_name) %.>%

knitr::kable(.)

| **x** |
| --- |
| -3 |
| -2 |
| -1 |
| 0 |
| 1 |
| 2 |
| 3 |

**Conclusion**

rquery is an advanced SQL query generator for R that includes a number of tools for managing complex queries. If your queries are taking substantial development time or substantial run time you should definitely consider trying the rquery system. For convenience there is also data.table based implementation of the rquery grammar called rqdatatable (which tends to be *much* faster that dplyr).

# clean up tmps

intermediates <- tmps(dumpList = TRUE)

for(ti in intermediates) {

rquery::rq\_remove\_table(db\_rquery, ti)

}

DBI::dbDisconnect(raw\_connection)