Our goal has been to make rquery the best query generation system for R (and to make data\_algebra the best query generator for Python).

Lets see what rquery is good at, and what new features are making rquery better.

The idea is: the query is a first class citizen that we can use to design and optimize queries prior to translating them into a data transform action via data.table, SQL, Pandas, or other realizations.

For quite a while rquery has had query-narrowing. Columns that are not used in the calculation are eliminated early. Here is an example.

library(rquery)

ops <- mk\_td(

"d",

c("col1", "col2", "col3")) %.>%

extend(., sum23 := col2 + col3) %.>%

select\_columns(., 'sum23')

cat(format(ops))

## mk\_td("d", c(

## "col1",

## "col2",

## "col3")) %.>%

## extend(.,

## sum23 := col2 + col3) %.>%

## select\_columns(.,

## c('sum23'))

The above query (or operator [DAG](https://en.wikipedia.org/wiki/Directed_acyclic_graph)) represents working with a table that has columns col1, col2, col3. The example is specifying adding a new derived column named sum23 and then limiting down to only this new column. We’ve tried to use operator names that evoke [operator names used by Codd](https://en.wikipedia.org/wiki/Relational_algebra).

An important point is: the query is bound to a description of a data frame (or a schema), not bound to any one data frame. Thus we can re-use the query on new data.

The record-keeping in the query knows that only columns col2 and col2 are used.

columns\_used(ops)

## $d

## [1] "col2" "col3"

This allows “query narrowing” where the unused columns are not specified in intermediate queries. This is easiest to see if we convert the query to SQL.

ops %.>%

to\_sql(

.,

rquery::rquery\_default\_db\_info()) %.>%

cat(.)

## SELECT

## "sum23"

## FROM (

## SELECT

## "col2" + "col3" AS "sum23"

## FROM (

## SELECT

## "col2",

## "col3"

## FROM

## "d"

## ) tsql\_76973382323412881950\_0000000000

## ) tsql\_76973382323412881950\_0000000001

Notice col1 is never referred to. This can be handy when working with tables with hundreds of columns.

And, using rqdatatable we can use data.table as another data action implementation.

library(rqdatatable)

data.frame(col1 = 1, col2 = 2, col3 = 3) %.>%

ops %.>%

knitr::kable(.)

| **sum23** |
| --- |
| 5 |

rquery now also has query-shortening. Some dead-values can be eliminated during query construction, before any calculations are attempted.

ops <- mk\_td(

"example\_table",

c("col1", "col2", "col3")) %.>%

extend(., sum23 := col2 + col3) %.>%

extend(., x := 1) %.>%

extend(., x := 2) %.>%

extend(., x := 3) %.>%

extend(., x := 4) %.>%

extend(., x := 5) %.>%

select\_columns(., c('x', 'sum23'))

cat(format(ops))

## mk\_td("example\_table", c(

## "col1",

## "col2",

## "col3")) %.>%

## extend(.,

## sum23 := col2 + col3,

## x := 5) %.>%

## select\_columns(.,

## c('x', 'sum23'))

Obviously nobody would construct such a bad query, but it is nice that some of the “ick” is optimized automatically.

Both of the above optimizations are deliberately conservative. They are implemented to be correct (not give incorrect results), but are not necessarily super aggressive in eliminating all redundancy.

It is a bit long and technical. But both of these optimizations are easy due to the use of category theoretic ideas in the design of the rquery and data\_algebra packages

The short form is: the rquery/data\_algebra operators have an interpretation in a nice category over table [schemas](https://en.wikipedia.org/wiki/Database_schema). The schema objects give us pre-condition and post-condition record keeping which enforces correct query composition and query narrowing. The generality of arrow composition gives us the freedom to place optimizations in the composition step. This gives us more options then systems that are restricted to list-concatenation or function composition/abstraction as their notion of composition. It also lets us enforce and check conditions early.

rquery performs most of its checking during query construction. This can catch errors early and save a lot of development time.

ops\_bad <- mk\_td(

"example\_table",

c("col1", "col2", "col3")) %.>%

extend(., sum23 := col2\_MISSPELLED + col3)

## Error in check\_have\_cols(src\_columns, required\_cols, "rquery::extend"): rquery::extend unknown columns col2\_MISSPELLED

Notice an error was raised during query construction. We didn’t have to wait to supply data or translate to SQL.

Let’s take a look at the SQL translation of our final example query.

ops %.>%

to\_sql(

.,

rquery::rquery\_default\_db\_info()) %.>%

cat(.)

## SELECT

## "x",

## "sum23"

## FROM (

## SELECT

## "col2" + "col3" AS "sum23",

## 5 AS "x"

## FROM (

## SELECT

## "col2",

## "col3"

## FROM

## "example\_table"

## ) tsql\_28722584463189084716\_0000000000

## ) tsql\_28722584463189084716\_0000000001

There are some more things we would wish optimized away, such as both the inner and outer select. But the SQL is reasonably short, due to the intermediate stages that were optimized out of the original query. Later versions of the system will pick these up, and likely these are also easy for downstream SQL optimizers to eliminate.

An important point: optimizations performed during query construction are shared among all back-ends: data.table, SQL, and Pandas.

Please consider giving rquery a try.

**Appendix**

We often get asked “why bother with rquery, given dplyr was first.” I’d say: if you are happy with dplyr don’t worry about rquery. Though I would add: you really owe it to yourself to check out data.table, it is by far the best data manipulation system in R.

However, let’s take a look how dbplyr generates a similar SQL query.

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

packageVersion("dplyr")

## [1] '0.8.3'

library(dbplyr)

##

## Attaching package: 'dbplyr'

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

##

## ident, sql

packageVersion("dbplyr")

## [1] '1.4.2'

con <- DBI::dbConnect(RSQLite::SQLite(),

":memory:")

copy\_to(

con,

data.frame(col1 = 1, col2 = 2, col3 = 3),

name = 'd')

tbl(con, 'd') %>%

mutate(sum23 := col2 + col3) %>%

mutate(x := 1) %>%

mutate(x := 2) %>%

mutate(x := 3) %>%

mutate(x := 4) %>%

mutate(x := 5) %>%

select(x, sum23) %>%

show\_query()

##

## SELECT 5.0 AS `x`, `sum23`

## FROM (SELECT `col1`, `col2`, `col3`, `sum23`, 4.0 AS `x`

## FROM (SELECT `col1`, `col2`, `col3`, `sum23`, 3.0 AS `x`

## FROM (SELECT `col1`, `col2`, `col3`, `sum23`, 2.0 AS `x`

## FROM (SELECT `col1`, `col2`, `col3`, `sum23`, 1.0 AS `x`

## FROM (SELECT `col1`, `col2`, `col3`, `col2` + `col3` AS `sum23`

## FROM `d`)))))

The dplyr SQL query appears to have neither query narrowing nor query shortening. Again, a downstream SQL optimizer may be able to eliminate these steps (or it may not). However, it also would be desirable to have these sort of eliminations available when using data.table through dtplyr.

Also, dbplyr does not seem to catch errors until compute() or print() are called.

tbl(con, 'd') %>%

mutate(sum23 := col2\_MISSPELLED + col3) %>%

show\_query()

##

## SELECT `col1`, `col2`, `col3`, `col2\_MISSPELLED` + `col3` AS `sum23`

## FROM `d`

The above SQL refers to a non-existent column col2\_MISSPELLED. The query construction and SQL generation steps did not signal any error. Depending on how many queries and steps are before this, this could delay finding this mistake by quite a while (especially when using a high latency SQL engine such as Apache Spark).

DBI::dbDisconnect(con)