Package 'rqdatatable'

August 21, 2023

Type Package

Title 'rquery' for 'data.table'

Version 1.3.3

Date 2023-08-19

Maintainer John Mount <jmount@win-vector.com>

Description Implements the 'rquery' piped Codd-style query algebra using 'data.table'. This allows for a high-speed in memory implementation of Codd-style data manipulation tools.

URL https://github.com/WinVector/rqdatatable/,
https://winvector.github.io/rqdatatable/

BugReports https://github.com/WinVector/rqdatatable/issues

License GPL-2 | GPL-3

Encoding UTF-8

ByteCompile true

VignetteBuilder knitr

Depends R (>= 3.4.0), wrapr (>= 2.0.9), rquery (>= 1.4.9)

Imports data.table (>= 1.12.2)

RoxygenNote 7.2.3

Suggests knitr, rmarkdown, DBI, RSQLite, parallel, tinytest

NeedsCompilation no

Author John Mount [aut, cre],
Win-Vector LLC [cph]

Repository CRAN

Date/Publication 2023-08-21 08:00:02 UTC

R topics documented:

`rqdatatable-package` `rqdatatable`*: Relational Query Generator for Data Manipulation Implemented by data.table*

Description

Implements the `rquery` piped query algebra using `data.table`. This allows for a high-speed in memory implementation of Codd-style data manipulation tools.

Author(s)

Maintainer: John Mount `<jmount@win-vector.com>`

Other contributors:

- Win-Vector LLC [copyright holder]

See Also

Useful links:

- https://github.com/WinVector/rqdatatable/
- https://winvector.github.io/rqdatatable/
- Report bugs at https://github.com/WinVector/rqdatatable/issues

---

`ex_data_table` *Execute an* `rquery` *pipeline with* `data.table` *sources.*

Description

`data.tables` are looked for by name in the `tables` argument and in the execution environment. Main external execution interface.

Usage

```
ex_data_table(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Details

- `ex_data_table_step.relop_drop_columns`: implement drop columns
- `ex_data_table_step.relop_extend`: implement extend/assign operator
- `ex_data_table_step.relop_natural_join`: implement natural join
- `ex_data_table_step.relop_non_sql`: direct function (non-sql) operator (not implemented for `data.table`)
- `ex_data_table_step.relop_null_replace`: implement NA/NULL replacement

- ex_data_table_step.relop_orderby: implement row ordering
- ex_data_table_step.relop_project: implement row ordering
- ex_data_table_step.relop_rename_columns: implement column renaming
- ex_data_table_step.relop_select_columns: implement select columns
- ex_data_table_step.relop_select_rows: implement select rows
- ex_data_table_step.relop_sql: direct sql operator (not implemented for data.table)
- ex_data_table_step.relop_table_source: implement data source
- ex_data_table_step.relop_theta_join: implement theta join (not implemented for data.table)
- ex_data_table_step.relop_unionall: implement row binding

Value

resulting data.table (intermediate tables can somtimes be mutated as is practice with data.table).

Examples

```
a <- data.table::data.table(x = c(1, 2) , y = c(20, 30), z = c(300, 400))
optree <- local_td(a) %.>%
   select_columns(., c("x", "y")) %.>%
   select_rows_nse(., x<2 & y<30)
cat(format(optree))
ex_data_table(optree)

# other ways to execute the pipeline include
data.frame(x = 0, y = 4, z = 400) %.>% optree
```

---

ex_data_table_parallel

Execute an rquery *pipeline with* data.table *in parallel.*

Description

Execute an rquery pipeline with data.table in parallel, partitioned by a given column. Note: usually the overhead of partitioning and distributing the work will by far overwhelm any parallel speedup. Also data.table itself already seems to exploit some thread-level parallelism (one often sees user time > elapsed time). Requires the parallel package. For a worked example with significant speedup please see https://github.com/WinVector/rqdatatable/blob/master/extras/Parallel_rqdatatable.md.

Usage

```
ex_data_table_parallel(
  optree,
  partition_column,
  cl = NULL,
  ...,
  tables = list(),
  source_limit = NULL,
  debug = FALSE,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
partition_column	character name of column to partition work by.
cl	a cluster object, created by package parallel or by package snow. If NULL, use the registered default cluster.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
debug	logical if TRUE use lapply instead of parallel::clusterApplyLB.
env	environment to look for values in.

Details

Care must be taken that the calculation partitioning is course enough to ensure a correct calculation. For example: anything one is joining on, aggregating over, or ranking over must be grouped so that all elements affecting a given result row are in the same level of the partition.

Value

resulting data.table (intermediate tables can sometimes be mutated as is practice with data.table).

---

ex_data_table_step.default

default non-impementation.

Description

Throw on error if this method is called, signalling that a specific `data.table` implemetation is needed for this method.

Usage

```
## Default S3 method:
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

---

ex_data_table_step.relop_drop_columns

Implement drop columns.

Description

data.table based implementation.

Usage

```
## S3 method for class 'relop_drop_columns'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_limit = NULL,
  source_usage = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
source_usage	list mapping source table names to vectors of columns used.
env	environment to work in.

Examples

```
dL <- data.frame(x = 1, y = 2, z = 3)
rquery_pipeline <- local_td(dL) %.>%
  drop_columns(., "y")
dL %.>% rquery_pipeline
```

---

ex_data_table_step.relop_extend

Implement extend/assign operator.

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_extend'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Details

Will re-order columns if there are ordering terms.

Examples

```
dL <- build_frame(
    "subjectID", "surveyCategory"     , "assessmentTotal", "one" |
    1          , "withdrawal behavior", 5                , 1     |
    1          , "positive re-framing", 2                , 1     |
    2          , "withdrawal behavior", 3                , 1     |
    2          , "positive re-framing", 4                , 1     )
rquery_pipeline <- local_td(dL) %.>%
  extend_nse(.,
             probability %:=%
               exp(assessmentTotal * 0.237)/
               sum(exp(assessmentTotal * 0.237)),
             count %:=% sum(one),
             rank %:=% rank(),
             orderby = c("assessmentTotal", "surveyCategory"),
             reverse = c("assessmentTotal"),
             partitionby = 'subjectID') %.>%
  orderby(., c("subjectID", "probability"))
dL %.>% rquery_pipeline
```

---

ex_data_table_step.relop_natural_join

Natural join.

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_natural_join'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
d1 <- build_frame(
    "key", "val", "val1" |
      "a"  , 1  ,  10    |
      "b"  , 2  ,  11    |
      "c"  , 3  ,  12    )
d2 <- build_frame(
    "key", "val", "val2" |
      "a"  , 5  ,  13    |
      "b"  , 6  ,  14    |
      "d"  , 7  ,  15    )

# key matching join
optree <- natural_join(local_td(d1), local_td(d2),
                       jointype = "FULL", by = 'key')
ex_data_table(optree)

# full cross-product join
# (usually with jointype = "FULL", but "LEFT" is more
# compatible with rquery field merge semantics).
optree2 <- natural_join(local_td(d1), local_td(d2),
                        jointype = "LEFT", by = NULL)
ex_data_table(optree2)
# notice ALL non-"by" fields take coalese to left table.
```

---

ex_data_table_step.relop_non_sql

Direct non-sql (function) node, not implemented for `data.table` *case.*

Description

Passes a single table to a function that takes a single data.frame as its argument, and returns a single data.frame.

Usage

```
## S3 method for class 'relop_non_sql'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

See Also

rq_df_funciton_node, rq_df_grouped_funciton_node

Examples

```
set.seed(3252)
d <- data.frame(a = rnorm(1000), b = rnorm(1000))

optree <- local_td(d) %.>%
  quantile_node(.)
d %.>% optree

p2 <- local_td(d) %.>%
  rsummary_node(.)
d %.>% p2

summary(d)
```

ex_data_table_step.relop_null_replace *Replace NAs.*

Description

data.table based implementation.

Usage

```
## S3 method for class 'relop_null_replace'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L ,  5  |
    NA ,  7  |
    NA , NA )
rquery_pipeline <- local_td(dL) %.>%
  null_replace(., c("x", "y"), 0, note_col = "nna")
dL %.>% rquery_pipeline
```

`ex_data_table_step.relop_orderby` *Reorder rows.*

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_orderby'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
    1L , "a" |
    3L , "c" )
rquery_pipeline <- local_td(dL) %.>%
  orderby(., "y")
dL %.>% rquery_pipeline
```

ex_data_table_step.relop_order_expr *Order rows by expression.*

Description

data.table based implementation.

Usage

```
## S3 method for class 'relop_order_expr'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
   -4L , "a" |
    3L , "c" )
rquery_pipeline <- local_td(dL) %.>%
  order_expr(., abs(x))
dL %.>% rquery_pipeline
```

`ex_data_table_step.relop_project` *Implement projection operator.*

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_project'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Examples

```
dL <- build_frame(
  "subjectID", "surveyCategory"     , "assessmentTotal" |
    1          , "withdrawal behavior", 5                |
    1          , "positive re-framing", 2                |
    2          , "withdrawal behavior", 3                |
    2          , "positive re-framing", 4                )
test_p <- local_td(dL) %.>%
  project(.,
          maxscore := max(assessmentTotal),
          count := n(),
          groupby = 'subjectID')
cat(format(test_p))
dL %.>% test_p
```

`ex_data_table_step.relop_rename_columns` *Rename columns.*

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_rename_columns'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
    1L , "a" |
    3L , "c" )
rquery_pipeline <- local_td(dL) %.>%
  rename_columns(., c("x" = "y", "y" = "x"))
dL %.>% rquery_pipeline
```

`ex_data_table_step.relop_select_columns` *Implement drop columns.*

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_select_columns'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Examples

```
dL <- data.frame(x = 1, y = 2, z = 3)
rquery_pipeline <- local_td(dL) %.>%
  select_columns(., "y")
dL %.>% rquery_pipeline
```

ex_data_table_step.relop_select_rows *Select rows by condition.*

Description

data.table based implementation.

Usage

```
## S3 method for class 'relop_select_rows'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
    1L , "a" |
    3L , "c" )
rquery_pipeline <- local_td(dL) %.>%
  select_rows_nse(., x <= 2)
dL %.>% rquery_pipeline
```

`ex_data_table_step.relop_set_indicator` *Implement set_indicatoroperator.*

Description

`data.table` based implementation.

Usage

```
## S3 method for class 'relop_set_indicator'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

`optree`	relop operations tree.
`...`	not used, force later arguments to bind by name.
`tables`	named list map from table names used in nodes to data.tables and data.frames.
`source_usage`	list mapping source table names to vectors of columns used.
`source_limit`	if not null limit all table sources to no more than this many rows (used for debugging).
`env`	environment to work in.

Examples

```
d <- data.frame(a = c("1", "2", "1", "3"),
                b = c("1", "1", "3", "2"),
                q = 1,
                stringsAsFactors = FALSE)
set <- c("1", "2")
op_tree <- local_td(d) %.>%
  set_indicator(., "one_two", "a", set) %.>%
  set_indicator(., "z", "a", c())
d %.>% op_tree
```

ex_data_table_step.relop_sql *Direct sql node.*

Description

Execute one step using the rquery.rquery_db_executor SQL supplier. Note: it is not a good practice to use SQL nodes in data.table intended pipelines (loss of class information and cost of data transfer). This implementation is only here for completeness.

Usage

```
## S3 method for class 'relop_sql'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
# WARNING: example tries to change rquery.rquery_db_executor option to RSQLite and back.
if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  # example database connection
  my_db <- DBI::dbConnect(RSQLite::SQLite(),
                          ":memory:")
  old_o <- options(list("rquery.rquery_db_executor" = list(db = my_db)))

  # example data
  d <- data.frame(v1 = c(1, 2, NA, 3),
                  v2 = c(NA, "b", NA, "c"),
                  v3 = c(NA, NA, 7, 8),
```

```
                  stringsAsFactors = FALSE)

  # example xform
  vars <- column_names(d)
  # build a NA/NULLs per-row counting expression.
  # names are "quoted" by wrapping them with as.name().
  # constants can be quoted by an additional list wrapping.
  expr <- lapply(vars,
                 function(vi) {
                   list("+ (CASE WHEN (",
                        as.name(vi),
                        "IS NULL ) THEN 1.0 ELSE 0.0 END)")
                 })
  expr <- unlist(expr, recursive = FALSE)
  expr <- c(list(0.0), expr)

  # instantiate the operator node
  op_tree <- local_td(d) %.>%
    sql_node(., "num_missing" %:=% list(expr))
  cat(format(op_tree))

  d %.>% op_tree

  options(old_o)
  DBI::dbDisconnect(my_db)
}
```

`ex_data_table_step.relop_table_source` *Build a data source description.*

Description

`data.table` based implementation. Looks for tables first in `tables` and then in `env`. Will accept any `data.frame` that can be converted to `data.table`.

Usage

```
## S3 method for class 'relop_table_source'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
    1L , "a" |
    3L , "c" )
rquery_pipeline <- local_td(dL)
dL %.>% rquery_pipeline
```

---

ex_data_table_step.relop_theta_join

Theta join (database implementation).

Description

Limited implementation. All terms must be of the form: "(table1.col CMP table2.col) (, (table1.col CMP table2.col))".

Usage

```
## S3 method for class 'relop_theta_join'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
d1 <- data.frame(AUC = 0.6, R2 = 0.2)
d2 <- data.frame(AUC2 = 0.4, R2 = 0.3)

optree <- theta_join_se(local_td(d1), local_td(d2), "AUC >= AUC2")

ex_data_table(optree, tables = list(d1 = d1, d2 = d2)) %.>%
  print(.)
```

---

ex_data_table_step.relop_unionall

Bind tables together by rows.

Description

data.table based implementation.

Usage

```
## S3 method for class 'relop_unionall'
ex_data_table_step(
  optree,
  ...,
  tables = list(),
  source_usage = NULL,
  source_limit = NULL,
  env = parent.frame()
)
```

Arguments

optree	relop operations tree.
...	not used, force later arguments to bind by name.
tables	named list map from table names used in nodes to data.tables and data.frames.
source_usage	list mapping source table names to vectors of columns used.
source_limit	if not null limit all table sources to no more than this many rows (used for debugging).
env	environment to work in.

Examples

```
dL <- build_frame(
    "x", "y" |
    2L , "b" |
    1L , "a" |
    3L , "c" )
rquery_pipeline <- unionall(list(local_td(dL), local_td(dL)))
dL %.>% rquery_pipeline
```

---

make_dt_lookup_by_column

Lookup by column function factory.

Description

Build data.table implementation of lookup_by_column. We do this here as rqdatatable is a data.table aware package (and rquery is not).

Usage

```
make_dt_lookup_by_column(pick, result)
```

Arguments

pick	character scalar, name of column to control value choices.
result	character scalar, name of column to place values in.

Value

f_dt() function.

Examples

```
df = data.frame(x = c(1, 2, 3, 4),
                y = c(5, 6, 7, 8),
                choice = c("x", "y", "x", "z"),
                stringsAsFactors = FALSE)
make_dt_lookup_by_column("choice", "derived")(df)

# # base-R implementation
# df %.>% lookup_by_column(., "choice", "derived")
# # # data.table implementation (requies rquery 1.1.0, or newer)
# # df %.>% lookup_by_column(., "choice", "derived",
# #                          f_dt_factory = rqdatatable::make_dt_lookup_by_column)
```

`rq_df_funciton_node` *Helper to build data.table capable non-sql nodes.*

Description

Helper to build data.table capable non-sql nodes.

Usage

```
rq_df_funciton_node(
  .,
  f,
  ...,
  f_db = NULL,
  columns_produced,
  display_form,
  orig_columns = FALSE
)
```

Arguments

`.`	or data.frame input.
`f`	function that takes a data.table to a data.frame (or data.table).
`...`	force later arguments to bind by name.
`f_db`	implementation signature: f_db(db, incoming_table_name, outgoing_table_name, nd, ...) (db being a database handle). NULL defaults to using f.
`columns_produced`	character columns produces by f.
`display_form`	display form for node.
`orig_columns`	orig_columns, if TRUE assume all input columns are present in derived table.

Value

relop non-sql node implementation.

See Also

ex_data_table_step.relop_non_sql, rq_df_grouped_funciton_node

Examples

```
# a node generator is something an expert can
# write and part-time R users can use.
grouped_regression_node <- function(., group_col = "group", xvar = "x", yvar = "y") {
  force(group_col)
  formula_str <- paste(yvar, "~", xvar)
  f <- function(df, nd = NULL) {
    dlist <- split(df, df[[group_col]])
    clist <- lapply(dlist,
                    function(di) {
                      mi <- lm(as.formula(formula_str), data = di)
                      ci <- as.data.frame(summary(mi)$coefficients)
                      ci$Variable <- rownames(ci)
                      rownames(ci) <- NULL
                      ci[[group_col]] <- di[[group_col]][[1]]
                      ci
                    })
    data.table::rbindlist(clist)
  }
  columns_produced =
     c("Variable", "Estimate", "Std. Error", "t value", "Pr(>|t|)", group_col)
  rq_df_funciton_node(
    ., f,
    columns_produced = columns_produced,
    display_form = paste0(yvar, "~", xvar, " grouped by ", group_col))
}

# work an example
set.seed(3265)
d <- data.frame(x = rnorm(1000),
                y = rnorm(1000),
                group = sample(letters[1:5], 1000, replace = TRUE),
                stringsAsFactors = FALSE)

rquery_pipeline <- local_td(d) %.>%
  grouped_regression_node(.)

cat(format(rquery_pipeline))

d %.>% rquery_pipeline
```

rq_df_grouped_funciton_node *Helper to build data.table capable non-sql nodes.*

Description

Helper to build data.table capable non-sql nodes.

Usage

```
rq_df_grouped_funciton_node(
  .,
  f,
  ...,
  f_db = NULL,
  columns_produced,
  group_col,
  display_form
)
```

Arguments

.	or data.frame input.
f	function that takes a data.table to a data.frame (or data.table).
...	force later arguments to bind by name.
f_db	implementation signature: f_db(db, incoming_table_name, outgoing_table_name) (db being a database handle). NULL defaults to using f.
columns_produced	character columns produces by f.
group_col	character, column to split by.
display_form	display form for node.

Value

relop non-sql node implementation.

See Also

ex_data_table_step.relop_non_sql, rq_df_funciton_node

Examples

```
# a node generator is something an expert can
# write and part-time R users can use.
grouped_regression_node <- function(., group_col = "group", xvar = "x", yvar = "y") {
  force(group_col)
  formula_str <- paste(yvar, "~", xvar)
  f <- function(di) {
    mi <- lm(as.formula(formula_str), data = di)
    ci <- as.data.frame(summary(mi)$coefficients)
    ci$Variable <- rownames(ci)
    rownames(ci) <- NULL
    colnames(ci) <- c("Estimate", "Std_Error", "t_value", "p_value", "Variable")
    ci
  }
  columns_produced =
    c("Estimate", "Std_Error", "t_value", "p_value", "Variable", group_col)
  rq_df_grouped_funciton_node(
    ., f,
    columns_produced = columns_produced,
    group_col = group_col,
    display_form = paste0(yvar, "~", xvar, " grouped by ", group_col))
}

# work an example
set.seed(3265)
d <- data.frame(x = rnorm(1000),
                y = rnorm(1000),
                group = sample(letters[1:5], 1000, replace = TRUE),
                stringsAsFactors = FALSE)

rquery_pipeline <- local_td(d) %.>%
  grouped_regression_node(.)

cat(format(rquery_pipeline))

d %.>% rquery_pipeline
```

---

set_rqdatatable_as_executor

Set rqdatatable package as default rquery executor

Description

Sets rqdatatable (and hence data.table) as the default executor for rquery).

Usage

```
set_rqdatatable_as_executor()
```

Index