Package ‘rquery’

August 20, 2023

Type Package
Title Relational Query Generator for Data Manipulation at Scale
Version 1.4.99
Date 2023-08-19
Maintainer John Mount <jmount@win-vector.com>
BugReports https://github.com/WinVector/rquery/issues
Description A piped query generator based on Edgar F. Codd’s relational algebra, and on production experience using ‘SQL’ and ‘dplyr’ at big data scale. The design represents an attempt to make ‘SQL’ more teachable by denoting composition by a sequential pipeline notation instead of nested queries or functions. The implementation delivers reliable high performance data processing on large data systems such as ‘Spark’, databases, and ‘data.table’. Package features include: data processing trees or pipelines as observable objects (able to report both columns produced and columns used), optimized ‘SQL’ generation as an explicit user visible table modeling step, plus explicit query reasoning and checking.
License GPL-2 | GPL-3
Encoding UTF-8
Depends R (>= 3.4.0), wrapr (>= 2.0.9)
Imports utils, stats, methods
Suggests DBI, RSQLite, rqdatatable (>= 1.3.2), igraph, knitr, rmarkdown, yaml, tinytest
RoxygenNote 7.2.3
ByteCompile true
VignetteBuilder knitr
NeedsCompilation no
Author John Mount [aut, cre], Win-Vector LLC [cph]
Repository CRAN
Date/Publication 2023-08-20 02:40:02 UTC

R topics documented:

- `rquery-package` .................................................. 4
- `actualize_join_plan` ........................................ 5
- `affine_transform` ............................................. 6
- `apply_right.relop` ........................................... 7
- `apply_right_S4,ANY.rquery_db_info-method` ............. 9
- `apply_right_S4,data.frame.relop_arrow-method` ......... 10
- `apply_right_S4,relop_arrow.relop_arrow-method` ....... 10
- `arrow` .................................................................. 11
- `assign_slice` ...................................................... 12
- `build_join_plan` ................................................ 13
- `columns_used` .................................................... 14
- `column_names` .................................................... 15
- `commencify` ......................................................... 15
- `complete_design` ................................................ 17
- `convert_yaml_to_pipeline` .................................... 18
- `count_null_cols` ................................................ 19
- `db_td` ................................................................. 20
- `describe_tables` ................................................ 22
- `drop_columns` ..................................................... 23
- `ex` .................................................................. 24
- `execute` ............................................................ 24
- `expand_grid` ....................................................... 26
- `extend` ............................................................... 27
- `extend_se` .......................................................... 28
- `format_node` ......................................................... 30
- `getDBOption` ...................................................... 30
- `graph_join_plan` ................................................ 31
- `if_else_block` ..................................................... 32
- `if_else_op` ........................................................ 34
- `inspect_join_plan` ............................................... 35
- `key_inspector_all_cols` ....................................... 37
- `key_inspector_postgresql` ..................................... 38
- `key_inspector_sqlite` .......................................... 38
- `local_td` ............................................................ 39
- `lookup_by_column` ............................................... 40
- `make_assignments` ............................................. 41
- `map_column_values` ............................................. 42
- `mark_null_cols` ................................................ 43
- `materialize` ........................................................ 44
- `materialize_node` .............................................. 46
- `mk_td` ............................................................... 47
- `natural_join` ...................................................... 48
R topics documented:

- non_sql_node
- normalize_clos
- null_replace
- op_diagram
- orderby
- order_expr
- order_expr_se
- order_rows
- pick_top_k
- pre_sql_sub_expr
- project
- project_se
- quantile_clos
- quantile_node
- quote_identifier
- quote_literal
- quote_string
- quote_table_name
- rename_columns
- row_counts
- rquery_apply_to_data_frame
- rquery_db_info
- rquery_default_db_info
- rq_colnames
- rq_coltypes
- rq_connection_advice
- rq_connection_name
- rq_connection_tests
- rq_copy_to
- rq_execute
- rq_get_query
- rq_head
- rq_nrow
- rq_remove_table
- rq_table_exists
- rstr
- rsummary
- rsummary_node
- select_columns
- select_rows
- select_rows_se
- setDBOpt
- setDBOption
- set_indicator
- sql_expr_set
- sql_node
- tables_used
- theta_join
rquery-package

theta_join_se .................................................. 94
topo_sort_tables .............................................. 95
to_sql ............................................................... 96
to_transport_representation ................................. 97
unionall ........................................................... 98
wrap ................................................................. 99

Index 100

rquery-package rquery: Relational Query Generator for Data Manipulation

Description

rquery supplies a piped query generator based on Edgar F. Codd’s relational algebra and operator names (plus experience using SQL and dplyr at big data scale). The design represents an attempt to make SQL more teachable by denoting composition a sequential pipeline notation instead of nested queries or functions. Package features include: data processing trees or pipelines as observable objects (able to report both columns produced and columns used), optimized SQL generation as an explicit user visible modeling step, and convenience methods for applying query trees to in-memory data.frames.

Details

Note: rquery is a "database first" design. This means choices are made that favor database implementation. These include: capturing the entire calculation prior to doing any work (and using recursive methods to inspect this object, which can limit the calculation depth to under 1000 steps at a time), preferring "tame column names" (which isn’t a bad idea in ‘R’ anyway as columns and variables are often seen as cousins), and not preserving row or column order (or supporting numeric column indexing). Also, rquery does have a fast in-memory implementation: rqdatatable (thanks to the data.table, so one can in fact use ‘rquery’ without a database.

Author(s)

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

Other contributors:

• Win-Vector LLC [copyright holder]

See Also

Useful links:

• https://github.com/WinVector/rquery/
• https://winvector.github.io/rquery/
• Report bugs at https://github.com/WinVector/rquery/issues
actualize_join_plan

Execute an ordered sequence of left joins.

Description

Please see vignette('DependencySorting', package = 'rquery') and vignette('joinController', package = 'rquery') for more details.

Usage

```r
actualize_join_plan(
  columnJoinPlan,
  ..., 
  jointype = "LEFT",
  add_ind_cols = FALSE,
  checkColClasses = FALSE
)
```

Arguments

- `columnJoinPlan`: columns to join, from `build_join_plan` (and likely altered by user). Note: no column names must intersect with names of the form `table_CLEANEDTABNAME_present`.
- `...`: force later arguments to bind by name.
- `jointype`: character, type of join to perform ("LEFT", "INNER", "RIGHT", ...).
- `add_ind_cols`: logical, if TRUE add indicators showing which tables supplied rows.
- `checkColClasses`: logical if true check for exact class name matches

Value

join optree

See Also

- `describe_tables`
- `build_join_plan`
- `inspect_join_plan`
- `graph_join_plan`

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "memory:"
  # example data
  DBI::dbWriteTable(my_db, 
    "meas1",
    data.frame(id = c(1, 2),
                weight = c(200, 120),
                height = c(60, 14)))
```
```r
DBI::dbWriteTable(my_db,
  "meas2",
  data.frame(pid= c(2,3),
             weight= c(105, 110),
             width= 1))

# get the initial description of table defs
tDesc <- describe_tables(my_db, qc(meas1, meas2))
# declare keys (and give them consistent names)
tDesc$keys[[1]] <- list(PatientID='id')
tDesc$keys[[2]] <- list(PatientID='pid')
# build the column join plan
columnJoinPlan <- build_join_plan(tDesc)
# decide we don't want the width column
columnJoinPlan$want[columnJoinPlan$resultColumn=="width"] <- FALSE
# double check our plan
if(!is.null(inspect_join_plan(tDesc, columnJoinPlan,
                            checkColClasses=TRUE))) {
  stop("bad join plan")
}

# actualize as left join op_tree
optree <- actualize_join_plan(columnJoinPlan,
                               checkColClasses=TRUE)
cat(format(optree))
print(execute(my_db, optree))

# if(requireNamespace("DiagrammeR", quietly = TRUE)) {
#   DiagrammeR::grViz(op_diagram(optree))
# }

DBI::dbDisconnect(my_db)
}
```

### affine_transform

**Implement an affine transformation**

**Description**

Implement an affine transformation

**Usage**

```r
affine_transform(source, linear_transform, offset, ..., env = parent.frame())
```

**Arguments**

- `source` relop source (or data.frame source)
- `linear_transform` matrix with row names taken from source column names (inputs), and column names are outputs.
- `offset` vector of offsets with names same as column names of `linear_transform`.  

apply_right.relop

... force later arguments to bind by name

env environment to look for values in.

Value

relop node

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- data.frame(AUC = 0.6, R2 = 0.2)
  source <- rq_copy_to(my_db, 'd',
    d,
    overwrite = TRUE,
    temporary = TRUE)
  linear_transform <- matrix(c(1, 1, 2, -1, 1, 0, 0, 0), nrow = 2)
  rownames(linear_transform) <- c("AUC", "R2")
  colnames(linear_transform) <- c("res1", "res2", "res3", "res4")
  offset <- c(5, 7, 1, 0)
  names(offset) <- colnames(linear_transform)

  optree <- affine_transform(source, linear_transform, offset)
  cat(format(optree))

  sql <- to_sql(optree, my_db)
  cat(sql)

  print(DBI::dbGetQuery(my_db, sql))
  print(as.matrix(d) %*% linear_transform + offset)

  DBI::dbDisconnect(my_db)
}
```

apply_right.relop  Execute pipeline treating pipe_left_arg as local data to be copied into database.

Description

Execute pipeline treating pipe_left_arg as local data to be copied into database.

Usage

```r
## S3 method for class 'relop'
apply_right(
```
pipe_left_arg,
pipe_right_arg,
pipe_environment,
left_arg_name,
pipe_string,
right_arg_name
)

Arguments

pipe_left_arg  left argument.
pipe_right_arg pipe_right_arg argument.
pipe_environment
    environment to evaluate in.
left_arg_name  name, if not NULL name of left argument.
pipe_string    character, name of pipe operator.
right_arg_name name, if not NULL name of right argument.

Value
data.frame

See Also

rquery_apply_to_data_frame

Examples

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

    # operations pipeline/tree
optree <- mk_td("d", "x") %>%
             extend(., y = x**x)

    # wrapr dot pipe apply_right dispatch
    # causes this statement to apply optree
    # to d.
data.frame(x = 1:3) %>%
                optree %>%
                print(.)

    # remote example
rq_copy_to(db, "d",
           data.frame(x = 7:8),
           db_environment(content = db))
}

apply_right_S4,ANY,rquery_db_info-method

```r
apply_right_S4(ANY,rquery_db_info-method)
```

Apply pipeline to a database.

**Description**

Apply pipeline to a database with relop

**Usage**

```r
## S4 method for signature 'ANY,rquery_db_info'
apply_right_S4(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name
)
```

**Arguments**

- `pipe_left_arg`  relop operation tree
- `pipe_right_arg` rquery_db_info
- `pipe_environment` environment to evaluate in.
- `left_arg_name` name, if not NULL name of left argument.
- `pipe_string` character, name of pipe operator.
- `right_arg_name` name, if not NULL name of right argument.

**Value**

result
apply_right_S4, data.frame, relop_arrow-method

* S4 dispatch method for apply_right.

Description

compose a data.frame and a relop_arrow class

Usage

## S4 method for signature 'data.frame, relop_arrow'
apply_right_S4(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name
)

Arguments

pipe_left_arg  left argument
pipe_right_arg pipe_right_arg argument
pipe_environment  environment to evaluate in
left_arg_name  name, if not NULL name of left argument.
pipe_string  character, name of pipe operator.
right_arg_name  name, if not NULL name of right argument.

Value

result

Description

compose two relop_arrow classes
## Data Arrow

A categorical arrow mapping a table to a table.

Usage

```markdown
arrow(pipeline, ..., free_table_key = NULL, strict = FALSE)
```

Arguments

- **pipeline**: pipeline with one source table
- **free_table_key**: name of table to consider free (input) to the pipeline
- **strict**: logical, if TRUE excess columns are considered an error

Value

`relop_arrow` wrapping of pipeline
assign_slice

Assign a value to a slice of data (set of rows meeting a condition, and specified set of columns).

Description

Uses if_else_block.

Usage

assign_slice(source, testexpr, columns, value, env = parent.frame())

Arguments

  source          optree relop node or data.frame.
  testexpr        character containing the test expression.
  columns         character vector of column names to alter.
  value           value to set in matching rows and columns (scalar).
  env             environment to look to.

Details

Note: ifebtest_* is a reserved column name for this procedure.

Value

optree or data.frame.

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ':memory:');
  d <- rq_copy_to(
    my_db, 'd',
    data.frame(i = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                a = c(0, 0, 1, 1, 1, 1, 1, 1, 1, 1),
                b = c(0, 1, 0, 1, 1, 1, 1, 1, 1, 1),
                r = runif(10)),
    temporary=TRUE, overwrite=TRUE)
  optree <- d %>%
    assign_slice(.,
                  testexpr = qe(r<0.5),
                  columns = qc(a, b),
                  value = 2)
  cat(format(optree))
build_join_plan

```r
sql <- to_sql(optree, my_db)
cat(sql)

print(DBI::dbGetQuery(my_db, sql))

DBI::dbDisconnect(my_db)
```

---

**Description**

Please see vignette('DependencySorting', package = 'rquery') and vignette('joinController', package = 'rquery') for more details.

**Usage**

```r
build_join_plan(tDesc, ..., check = TRUE)
```

**Arguments**

- `tDesc` description of tables from `describe_tables` (and likely altered by user). Note: no column names must intersect with names of the form table_CLEANEDTABNAME_present.
- `...` force later arguments to bind by name.
- `check` logical, if TRUE check the join plan for consistency.

**Value**

detailed column join plan (appropriate for editing)

**See Also**

`describe_tables, inspect_join_plan, graph_join_plan, actualize_join_plan`

**Examples**

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  d <- data.frame(id=1:3, weight= c(200, 140, 98))
  DBI::dbWriteTable(my_db,"d1", d)
  DBI::dbWriteTable(my_db,"d2", d)
  tDesc <- describe_tables(my_db, c("d1", "d2"))
  tDesc$keys[[1]] <- list(PrimaryKey = 'id')
  tDesc$keys[[2]] <- list(PrimaryKey = 'id')
  print(build_join_plan(tDesc))
}
columns_used

Return columns used

Description

Return columns used

Usage

columns_used(x, ..., using = NULL)

Arguments

x rquery operation tree.

... generic additional arguments (not used)

using character, if not NULL set of columns used from above.

Value

vector of table qualified column names.

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "/:memory:"
  d1 <- rq_copy_to(my_db, 'd1',
                   data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
                   data.frame(AUC = 0.6, D = 0.3))
  optree <- natural_join(d1, d2, by = "AUC")
  cat(format(optree))
  print(columns_used(optree))
  DBI::dbDisconnect(my_db)
}
**Description**

Return column names

**Usage**

```r
column_names(x, ...)
```

**Arguments**

- `x` rquery operation tree.
- `...` generic additional arguments

**Value**

vector of column names

**Examples**

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), " :memory:
  d1 <- rq_copy_to(my_db, 'd1',
                data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
                data.frame(AUC = 0.6, D = 0.3))
  optree <- natural_join(d1, d2, by = "AUC")
  cat(format(optree))
  print(column_names(optree))
  DBI::dbDisconnect(my_db)
}
```

**Description**

Hyderdrive (science fiction show) synonym for `execute`
Usage

\[
\text{commencify}(\text{source}, \text{optree}, \ldots, \text{limit} = \text{NULL}, \text{source_limit} = \text{NULL}, \text{overwrite} = \text{TRUE}, \text{temporary} = \text{TRUE}, \allowcolor{blue}{\text{allow_executor} = \text{TRUE}}, \text{temp_source} = \text{mk_tmp_name_source("rquery_ex")}, \text{env} = \text{parent.frame}())
\]

Arguments

- **source**: data.frame or database connection (rquery_db_info class or DBI connections preferred).
- **optree**: relop operation tree.
- **...**: force later arguments to bind by name.
- **limit**: numeric, if set limit to this many rows during data bring back (not used when landing a table).
- **source_limit**: numeric if not NULL limit sources to this many rows.
- **overwrite**: logical if TRUE drop an previous table.
- **temporary**: logical if TRUE try to create a temporary table.
- **allow_executor**: logical if TRUE allow any executor set as rquery.rquery_executor to be used.
- **temp_source**: temporary name generator.
- **env**: environment to work in.

Value

data.frame

See Also

- execute

Examples

# WARNING: example tries to change rquery.rquery_db_executor option to RSQLite and back.
if (requireNamespace("DBI", quietly = TRUE) & requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "mem:memory:"
  old_o <- options(list("rquery.rquery_db_executor" = list(db = my_db)))
  d <- rq_copy_to(my_db, "d",
                  data.frame(AUC = 0.6, R2 = 0.2))
  optree <- extend_se(d, c("v" := "AUC + R2", "x" := pmax(AUC,v)))
}
print(optree)
cat(format(optree))

v <- execute(my_db, optree)
print(v)

v2 <- execute(data.frame(AUC = 1, R2 = 2), optree)
print(v2)

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

complete_design

Complete an experimental design.

Description

Complete an experimental design.

Usage

complete_design(design_table, data_table)

Arguments

design_table  optree or for experimental design.
data_table     optree for data.

Value

joined and annotated table optree.

Examples

if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
)
  # example experimental design
  values <- list(nums = 1:3, lets = c("a", "b"))
  design <- expand_grid(my_db, values)

  # not quite matching data
  data <- build_frame(
```r
"nums", "lets" |  
1L , "a" |  
1L , "b" |  
77L , "a" | # out of place ID  
2L , "b" |  
3L , "a" | # duplicated  
3L , "a" |  
3L , "b" )
data$row_number <- seq_len(nrow(data))
data <- rq_copy_to(my_db, "data", data)

# compare/augment
res <- complete_design(design, data)
cat(format(res))
res <- materialize(my_db, res)

print("completed data design")
print(execute(my_db, res))

# look for dups (can use extend_se(partation) on
# databases with window fns.
print("duplicate key rows:")
res %>%
  project_se(.,,
    groupby = column_names(design),
    "count" %:>="SUM(1)") %>%
  select_rows_se(.,, "count>1") %>%
  execute(my_db, .) %>%
  print(.)

# look for data that was not in design
print("data rows not in design:")
data %>%
  natural_join(., res,
    jointype = "LEFT",
    by = column_names(design)) %>%
  select_rows_se(., "is.na(row_in_design_table)") %>%
  execute(my_db, .) %>%
  print(.)

DBI::dbDisconnect(my_db)
}
```

#### convert_yaml_to_pipeline

Convert a series of simple objects (from YAML deserialization) to an rquery pipeline.
count_null_cols

Description

Convert a series of simple objects (from YAML deserialization) to an rquery pipeline.

Usage

convert_yaml_to_pipeline(rep, ..., source = NULL, env = parent.frame())

Arguments

rep input objects
... not used, force later arguments to bind by name
source input rquery node
env environment to evaluate in

Value

rquery operator tree

count_null_cols Count NULLs per row for given column set.

Description

Build a query that counts the number of nulls in each row.

Usage

count_null_cols(source, cols, count)

Arguments

source incoming rel_op tree or data.frame.
cols character, columns to track
count character, column to write count in.

Value

rel_op node or data.frame (depending on input).

See Also

null_replace, mark_null_cols
Examples

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

  d <- rq_copy_to(my_db, 'd',
                  data.frame(AUC = c(0.6, 0.5, NA),
                              R2 = c(1.0, 0.9, NA)))
  op_tree <- d %>% count_null_cols(., c("AUC", "R2"), "nnull")
  cat(format(op_tree))
  sql <- to_sql(op_tree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))

  # ad-hoc mode
  data.frame(AUC=c(1, NA, 0.5), R2=c(NA, 1, 0)) %>%
             op_tree %>%
             print(.)

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

---

**db_td**

*Construct a table description from a database source.*

**Description**

Build structures (table name, column names, and quoting strategy) needed to represent data from a remote table.

**Usage**

```r
db_td(db, table_name, ..., qualifiers = NULL, limit_was = 6L)
dbi_table(db, table_name, ..., qualifiers = NULL, limit_was = 6L)
```

**Arguments**

- `db`  
  - database connection
- `table_name`  
  - name of table
- `...`  
  - not used, force later argument to bind by name
db_td

qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

limit_was optional, row limit used to produce head_sample. If NULL no head_sample is produced and rq_colnames is used to get column names.

Details

Note: in examples we use rq_copy_to() to create data. This is only for the purpose of having easy portable examples. With big data the data is usually already in the remote database or Spark system. The task is almost always to connect and work with this pre-existing remote data and the method to do this is db_td which builds a reference to a remote table given the table name.

Value

a relop representation of the data

Functions

• dbi_table(): old name for db_td

See Also

mk_td, local_td, rq_copy_to, materialize, execute, to_sql

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  rq_copy_to(my_db,
    'd',
    data.frame(AUC = 0.6, R2 = 0.2),
    overwrite = TRUE,
    temporary = TRUE)
  d <- db_td(my_db, 'd')
  print(d)
  sql <- to_sql(d, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  cols <- columns_used(d)
  print(cols)
  sql2 <- to_sql(d, my_db, using = "AUC")
  cat(sql2)
  print(DBI::dbGetQuery(my_db, sql2))
  DBI::dbDisconnect(my_db)
}
```
**describe_tables**  
*Build a nice description of a table.*

**Description**

Please see [https://win-vector.com/2017/05/26/managing-spark-data-handles-in-r/](https://win-vector.com/2017/05/26/managing-spark-data-handles-in-r/) for details. Note: one usually needs to alter the keys column which is just populated with all columns.

**Usage**

```r
describe_tables(db, tablenames, ..., keyInspector = key_inspector_all_cols)
```

**Arguments**

- `db` database handle
- `tablenames` character, names of tables to describe.
- `...` force later arguments to bind by name.
- `keyInspector` function that determines preferred primary key set for tables.

**Details**

Please see `vignette('DependencySorting', package = 'rquery')` and `vignette('joinController', package = 'rquery')` for more details.

**Value**

table describing the data.

**See Also**

`build_join_plan`, `graph_join_plan`, `actualize_join_plan`

**Examples**

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  ex <- example_employee_date(my_db)
  print(describe_tables(my_db, ex$tableName,
                        keyInspector = key_inspector_sqlite))
  DBI::dbDisconnect(my_db)
}
```
drop_columns  Make a drop columns node (not a relational operation).

Description

Note: must keep at least one column.

Usage

drop_columns(source, drops, ..., strict = FALSE, env = parent.frame())

Arguments

source source to drop columns from.

drops list of distinct column names.

... force later arguments to bind by name

strict logical, if TRUE do check columns to be dropped are actually present.

env environment to look to.

Value
drop columns node.

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  d <- rq_copy_to(my_db, 'd',
                  data.frame(AUC = 0.6, R2 = 0.2))
  optree <- drop_columns(d, 'AUC')
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```r
execute

**Example Usage**

```r
execute(ops, ..., env = parent.frame())
```

**Arguments**

- `ops` : rquery pipeline with tables formed by `wrap()`.
- `...` : not used, force later argument to be referred by name.
- `env` : environment to work in.

**Value**

data.frame result

**Example**

```r
if(requireNamespace("rqdatatable")) {
  d <- data.frame(x = 1:3, y = 4:6)
  d %.>%
    wrap(.) %.>%
    extend(., z := x + y) %.>%
    execute(.)
}
```
execute

Usage

execute(
  source,
  optree,
  ...
  limit = NULL,
  source_limit = NULL,
  overwrite = TRUE,
  temporary = TRUE,
  allow_executor = TRUE,
  temp_source = mk_tmp_name_source("rquery_ex"),
  env = parent.frame()
)

Arguments

source data.frame or database connection (rquery_db_info class or DBI connections preferred).

optree relop operation tree.

... force later arguments to bind by name.

limit numeric, if set limit to this many rows during data bring back (not used when landing a table).

source_limit numeric if not NULL limit sources to this many rows.

overwrite logical if TRUE drop an previous table.

temporary logical if TRUE try to create a temporary table.

allow_executor logical if TRUE allow any executor set as rquery.rquery_executor to be used.

temp_source temporary name generator.

env environment to work in.

Value
data.frame

See Also

materialize, db_td, to_sql, rq_copy_to, mk_td

Examples

# WARNING: example tries to change rquery.rquery_db_executor option to RSQLite and back.
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  old_o <- options(list("rquery.rquery_db_executor" = list(db = my_db)))
  d <- rq_copy_to(my_db, 'd',
                  data.frame(AUC = 0.6, R2 = 0.2))
  optree <- extend_se(d, c("v" %= "AUC + R2", "x" %= "pmax(AUC,v)")))
```r
print(optree)
cat(format(optree))

v <- execute(my_db, optree)
print(v)

v2 <- execute(data.frame(AUC = 1, R2 = 2), optree)
print(v2)

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

---

**expand_grid**

*Cross product vectors in database.*

---

**Description**

Cross product vectors in database.

**Usage**

```r
expand_grid(
  db,
  values,
  ...,  # force later arguments to bind by name.
  temporary = TRUE,
  table_name = (wrapr::mk_tmp_name_source("eg"))(),
  qualifiers = NULL
)
```

**Arguments**

- `db`: database handle
- `values`: named list of value vectors.
- `...`: force later arguments to bind by name.
- `temporary`: logical if TRUE try to make temporary table.
- `table_name`: name to land result as.
- `qualifiers`: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

**Value**

table handle.
\textbf{extend}  

\textit{Extend data by adding more columns.}

**Description**

Create a node similar to a Codd extend relational operator (add derived columns).

**Usage**

\begin{verbatim}
extend(
    source,
    ...,  
    partitionby = NULL,
    orderby = NULL,
    reverse = NULL,
    display_form = NULL,
    env = parent.frame()
)
\end{verbatim}

\begin{verbatim}
extend_nse(
    source,
    ...,  
    partitionby = NULL,
    orderby = NULL,
    reverse = NULL,
    display_form = NULL,
    env = parent.frame()
)
\end{verbatim}

**Arguments**

\begin{itemize}
  \item \texttt{source} source to select from.
  \item \texttt{...} new column assignment expressions.
  \item \texttt{partitionby} partitioning (window function) terms.
\end{itemize}

**Examples**

\begin{verbatim}
if (requireNamespace("DBI", quietly = TRUE) &&
    requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  values <- list(nums = 1:3, lets = c("a", "b"))
  res <- expand_grid(my_db, values)
  print(res)
  execute(my_db, res)
  DBI::dbDisconnect(my_db)
}
\end{verbatim}
extend_se

orderby ordering (in window function) terms.
reverse reverse ordering (in window function) terms.
display_form character presentation form
env environment to look for values in.

Details

Partitionby and orderby can only be used with a database that supports window-functions (such as PostgreSQL, Spark, and so on).

Supports bquote().-style name abstraction with the extension that - promotes strings to names (please see here: https://github.com/WinVector/rquery/blob/master/Examples/Substitution/Substitution.md).

Note: if any window/aggregation functions are present then at least one of partitionby or orderby must be non empty. For this purpose partitionby=1 is allowed and means "single partition on the constant 1."

Value

extend node.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "memory:"
  d <- rq_copy_to(my_db, 'd',
                   data.frame(AUC = 0.6, R2 = 0.2))
  NEWCOL <- as.name("v")
  NEWVALUE = "zz"
  optree <- extend(d, .(NEWCOL) %:=% ifelse(AUC>0.5, R2, 1.0), .(NEWVALUE) %:=% 6)
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

---

extend_se

Extend data by adding more columns.

Description

Create a node similar to a Codd extend relational operator (add derived columns).
Usage

```r
extend_se(
  source,
  assignments,
  ..., 
  partitionby = NULL,
  orderby = NULL,
  reverse = NULL,
  display_form = NULL,
  env = parent.frame()
)
```

Arguments

- `source`: source to select from.
- `assignments`: new column assignment expressions.
- `...`: force later arguments to bind by name
- `partitionby`: partitioning (window function) terms.
- `orderby`: ordering (in window function) terms.
- `reverse`: reverse ordering (in window function) terms.
- `display_form`: character presentation form
- `env`: environment to look for values in.

Details

Partitionby and orderby can only be used with a database that supports window-functions (such as PostgreSQL, Spark and so on).

Note: if any window/aggregation functions are present then at least one of partitionby or orderby must be non empty. For this purpose partitionby=1 is allowed and means "single partition on the constant 1."

Value

`extend_node`

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- rq_copy_to(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2))
  optree <- extend_se(d, c("v" := "AUC + R2", "x" := pmax(AUC,v))
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
}
### getDBOption

Get a database connection option.

**Description**

Note: we are moving away from global options to options in the DB handle.

**Usage**

```r
getDBOption(db, optname, default, connection_options = list())
```

**Arguments**

- `db` database connection handle.
- `optname` character, single option name.
- `default` what to return if not set.
- `connection_options` name list of per connection options.

**Value**

option value

---

### format_node

Format a single node for printing.

**Description**

Format a single node for printing.

**Usage**

```r
format_node(node)
```

**Arguments**

- `node` node of operator tree to be formatted

**Value**

character display form of the node
Examples

```r
if(requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  print(getDBOption(my_db, "use_DBI_dbExecute"))
  DBI::dbDisconnect(my_db)
}
```

---

**graph_join_plan**  
*Build a draw-able specification of the join diagram*

**Description**

Please see vignette('DependencySorting', package = 'rquery') and vignette('joinController', package= 'rquery') for more details.

**Usage**

```r
graph_join_plan(columnJoinPlan, ..., groupByKeys = TRUE, graphOpts = NULL)
```

**Arguments**

- `columnJoinPlan`  
  join plan
- `...`  
  force later arguments to bind by name
- `groupByKeys`  
  logical if true build key-equivalent sub-graphs
- `graphOpts`  
  options for graphViz

**Value**

`grViz` diagram spec

**See Also**

describe_tables, build_join_plan, actualize_join_plan

**Examples**

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  # note: employeeanddate is likely built as a cross-product
  # join of an employee table and set of dates of interest
  # before getting to the join controller step. We call
  # such a table "row control" or "experimental design."
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  RSQLite::initExtension(my_db)
  tDesc <- example_employee_date(my_db)
}
# fix order by hand, please see rquery::topo_sort_tables for
# how to automate this.
ord <- match(c('employeeanddate', 'orgtable', 'activity', 'revenue'),
  tDesc$tableName)
tDesc <- tDesc[ord, , drop=FALSE]
columnJoinPlan <- build_join_plan(tDesc, check= FALSE)
# unify keys
columnJoinPlan$resultColumn[columnJoinPlan$resultColumn=='id'] <- 'eid'
# look at plan defects
print(paste('problems:',
  inspect_join_plan(tDesc, columnJoinPlan)))
diagramSpec <- graph_join_plan(columnJoinPlan)
# # to render as JavaScript:
# if(requireNamespace("DiagrammeR", quietly = TRUE)) {
#   print(DiagrammeR::grViz(diagramSpec))
# }
DBI::dbDisconnect(my_db)
my_db <- NULL
}

### if_else_block

**Build a sequence of statements simulating an if/else block: if(){}else{}.**

**Description**

This device uses expression-ifelse(,,) to simulate the more powerful per-row block-if(){}else{}. The difference is expression-ifelse(,,) can choose per-row what value to express, whereas block-if(){}else{} can choose per-row where to assign multiple values. By simulation we mean: a sequence of quoted mutate expressions are emitted that implement the transform. These expressions can then be optimized into a minimal number of no-dependency blocks by `extend_se` for efficient execution. The idea is the user can write legible code in this notation, and the translation turns it into safe and efficient code suitable for execution either on data.frames or at a big data scale using RPostgreSQL or sparklyr.

**Usage**

```r
if_else_block(testexpr, ..., thenexprs = NULL, elseexprs = NULL)
```

**Arguments**

- `testexpr` character containing the test expression.
- `...` force later arguments to bind by name.
- `thenexprs` named character then assignments (altering columns, not creating).
- `elseexprs` named character else assignments (altering columns, not creating).
Details

Note: ifebtest_* is a reserved column name for this procedure.

Value

sequence of statements for extend_se().

See Also

if_else_op

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  # Example: clear one of a or b in any row where both are set.
  # Land random selections early to avoid SQLite bug.
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  d <- rq_copy_to(
    my_db,
    'd',
    data.frame(i = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
                a = c(0, 0, 1, 1, 1, 1, 1, 1, 1, 1),
                b = c(0, 1, 0, 1, 1, 1, 1, 1, 1, 1),
                r = runif(10),
                edited = 0),
    temporary=TRUE, overwrite=TRUE)

  program <- if_else_block(
    testexpr = qe((a+b)>1),
    thenexprs = c(
      if_else_block(
        testexpr = qe(r >= 0.5),
        thenexprs = qae(a %:=% 0),
        elseexprs = qae(b %:=% 0)),
      qae(edited %:=% 1)))

  print(program)

  optree <- extend_se(d, program)
  cat(format(optree))

  sql <- to_sql(optree, my_db)
  cat(sql)

  print(DBI::dbGetQuery(my_db, sql))

  # Why we need to land the random selection early
  # for SQLite:
  q <- "SELECT r AS r1, r AS r2 FROM ( 
    SELECT random() AS r FROM ( 
      SELECT * from ( VALUES(1),(2) ) 
    ) a 
  ) b"

  print(query(optree, my_db, q))
}


Build a relop node simulating a per-row block-if(){}else{}.

Description

This device uses expression-ifelse(,,) to simulate the more powerful per-row block-if(){}else{}. The difference is expression-ifelse(,,) can choose per-row what value to express, whereas block-if(){}else() can choose per-row where to assign multiple values. By simulation we mean: a sequence of quoted mutate expressions are emitted that implement the transform. These expressions can then be optimized into a minimal number of no-dependency blocks by `extend_se` for efficient execution. The idea is the user can write legible code in this notation, and the translation turns it into safe and efficient code suitable for execution either on `data.frames` or at a big data scale using `RPostgreSQL` or `sparklyr`.

Usage

```r
if_else_op(
  source,
  testexpr,
  ..., # force later arguments to bind by name.
  thenexprs = NULL,
  elseexprs = NULL,
  env = parent.frame()
)
```

Arguments

- `source`: optree relop node or `data.frame`.
- `testexpr`: character containing the test expression.
- `thenexprs`: named character then assignments (altering columns, not creating).
- `elseexprs`: named character else assignments (altering columns, not creating).
- `env`: environment to look to.

Details

Note: `ifebtest_*` is a reserved column name for this procedure.
inspect_join_plan

Value

operator tree or data.frame.

See Also

if_else_block

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  # Example: clear one of a or b in any row where both are set.
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ".memory:"
  d <- rq_copy_to(
    my_db,
    'd',
    data.frame(i = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
    a = c(0, 0, 1, 1, 1, 1, 1, 1, 1, 1),
    b = c(0, 1, 0, 1, 1, 1, 1, 1, 1, 1),
    edited = NA),
    temporary=True, overwrite=True)
  optree <- d %.>%
    if_else_op(.,
      testexpr = qe((a+b)>1),
      thenexprs = qae(a %:=% 0,
        b %:=% 0,
        edited %:=% 1),
      elseexprs = qae(edited %:=% 0))
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}

inspect_join_plan

check that a join plan is consistent with table descriptions.

Description

Please see vignette('DependencySorting', package = 'rquery') and vignette('joinController', package= 'rquery') for more details.
Usage

inspect_join_plan(tDesc, columnJoinPlan, ..., checkColClasses = FALSE)

Arguments

tDesc description of tables, from describe_tables (and likely altered by user).

columnJoinPlan columns to join, from build_join_plan (and likely altered by user). Note: no column names must intersect with names of the form table\_CLEANEDTABNAME\_present.

... force later arguments to bind by name.

checkColClasses logical if true check for exact class name matches

Value

NULL if okay, else a string

See Also

describe_tables, build_join_plan, graph_join_plan, actualize_join_plan

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  # example data
  DBI::dbWriteTable(my_db, 
    "d1",
    data.frame(id = 1:3,
               weight = c(200, 140, 98),
               height = c(60, 24, 12)))
  DBI::dbWriteTable(my_db, 
    "d2",
    data.frame(pid = 2:3,
                weight = c(130, 110),
                width = 1))
  
  # get the initial description of table defs
  tDesc <- describe_tables(my_db, qc(d1, d2))
  # declare keys (and give them consistent names)
  tDesc$keys[[1]] <- list(PrimaryKey = 'id')
  tDesc$keys[[2]] <- list(PrimaryKey = 'pid')
  # build the join plan
  columnJoinPlan <- build_join_plan(tDesc)
  # confirm the plan
  print(inspect_join_plan(tDesc, columnJoinPlan,
                          checkColClasses = TRUE))
  # damage the plan
  columnJoinPlan$sourceColumn[columnJoinPlan$sourceColumn == 'width'] <- 'wd'
  # find a problem
  print(inspect_join_plan(tDesc, columnJoinPlan, ...))
}
key_inspector_all_cols

Return all columns as guess of preferred primary keys.

Description

Return all columns as guess of preferred primary keys.

Usage

key_inspector_all_cols(db, tablename)

Arguments

db database handle
tablename character, name of table

Value

map of keys to keys

See Also

describe_tables

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory::")
  DBI::dbWriteTable(my_db, "d",
                    data.frame(x=1:3, y=NA))
  print(key_inspector_all_cols(my_db, "d"))
  DBI::dbDisconnect(my_db)
}
### key_inspector_postgresql

*Return all primary key columns as guess at preferred primary keys for a PostgreSQL handle.*

#### Description

Return all primary key columns as guess at preferred primary keys for a PostgreSQL handle.

#### Usage

```python
key_inspector_postgresql(db, tablename)
```

#### Arguments

- `db`  
  database handle
- `tablename`  
  character, name of table

#### Value

map of keys to keys

#### See Also

`describe_tables`

---

### key_inspector_sqlite

*Return all primary key columns as guess at preferred primary keys for a SQLite handle.*

#### Description

Return all primary key columns as guess at preferred primary keys for a SQLite handle.

#### Usage

```python
key_inspector_sqlite(db, tablename)
```

#### Arguments

- `db`  
  database handle
- `tablename`  
  character, name of table

#### Value

map of keys to keys
local_td

See Also
describe_tables

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  DBI::dbExecute(my_db, "
    CREATE TABLE orgtable (
      eid TEXT,
      date INTEGER,
      dept TEXT,
      location TEXT,
      PRIMARY KEY (eid, date)
    )
  ")
  print(key_inspector_sqlite(my_db, "orgtable"))
  DBI::dbDisconnect(my_db)
}
```

local_td

Construct a table description of a local data.frame.

Description

Construct a table description of a local data.frame.

Usage

```r
local_td(
  d,
  ..., 
  name = NULL,
  name_source = wrapr::mk_tmp_name_source("rqltd"),
  env = parent.frame()
)
```

Arguments

data.frame or name of data.frame to use as a data source.

... not used, force later arguments to be optional.

name if not null name to user for table.

name_source temporary name source.

env environment to work in.
lookup_by_column

Value

a relop representation of the data

See Also

db_td, mk_td

Examples

```r
d <- data.frame(x = 1)
local_td(d)
local_td("d")
local_td(as.name("d"))
local_td(data.frame(x = 1))
d %>% local_td # needs wrapr 1.5.0 or newer to capture name
```

---

lookup_by_column  Use one column to pick values from other columns.

Description

The pick column selects values from the columns it names (per-row).

Usage

```r
lookup_by_column(
  source,
  pick,
  result,
  ...
  tmp_name_source = wrapr::mk_tmp_name_source("qn"),
  temporary = TRUE,
  qualifiers = NULL,
  f_dt_factory = NULL
)
```

Arguments

- **source** source to select from (relop or data.frame).
- **pick** character scalar, name of column to control value choices.
- **result** character scalar, name of column to place values in.
- **...** force later arguments to be bound by name
- **tmp_name_source** wrapr::mk_tmp_name_source(), temporary name generator.


make_assignments

Make a list of assignments, applying many functions to many columns.

Description

Make a list of assignments, applying each function to each column named. Intended to be used as an argument in extend_se() or project_se().

temporary logical, if TRUE use temporary tables.

qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

f_dt_factory optional signature f_dt_factory(pick, result) returns function with signature f_dt(d, nd) where d is a data.table. The point is the function must come from a data.table enabled package. Please see rqdatatable::make_dt_lookup_by_column for an example.

Examples

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

# library("rqdatatable")
# df %.>%
#  lookup_by_column(., "choice", "derived")

if (requireNamespace("DBI", quietly = TRUE) &&
requireNamespace("RSQLite", quietly = TRUE)) {
  db <- DBI::dbConnect(RSQLite::SQLite(),
"::memory:"
RSQLite::initExtension(db)
  dr <- rq_copy_to(db, "dRemote", df,
    overwrite = TRUE,
    temporary = TRUE)

  ops <- dr %.>%
    lookup_by_column(., "choice", "derived"
  cat(format(ops))

  execute(db, ops) %.>%
    print(.)

  DBI::dbDisconnect(db)
}
map_column_values

Usage

make_assignments(columns, funs, ..., sep = " ", prefix = TRUE)

Arguments

columns character, vector of column names to take values from.
funs character, names of functions to apply.
... not used, forced later parameters to bind by name
sep character, naming separator
prefix logical, if TRUE place function names prior, else after in results.

Examples

assignments <- make_assignments(c("x", "y"), c("mean", med = "median"))
print(assignments)
ops <- mk_td('d', c('x', 'y')) %.>% project_se(., assignments)
cat(format(ops))

map_column_values Remap values in a set of columns.

Description

Remap values in a set of columns.

Usage

map_column_values(source, colmap, ..., null_default = FALSE)

Arguments

source optree relop node or data.frame.
colmap data.frame with columns column_name, old_value, new_value.
... force later arguments to bind by name.
null_default logical, if TRUE map non-matching values to NULL (else they map to self).

Value

implementing optree or altered data.frame
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(),
                         "::memory:")

d <- rq_copy_to(my_db, 'd',
               data.frame(a = c("1", "2", "1", "3"),
                          b = c("1", "1", "3", "2"),
                          c = c("1", "2", "3", "4"),
                          stringsAsFactors = FALSE),
               temporary = TRUE,
               overwrite = TRUE)

mp <- build_frame(
                  "column_name", "old_value", "new_value" |
                  "a" , "1" , "10" |
                  "a" , "2" , "20" |
                  "b" , "1" , "100" |
                  "b" , "3" , "300" )

# example
op_tree <- d %>%
  map_column_values(.mp)

print(DBI::dbGetQuery(my_db, sql))
}
```

**mark_null_cols**

Indicate NULLs per row for given column set.

**Description**

Build a query that creates NULL indicators for nulls in selected columns.

**Usage**

```r
mark_null_cols(source, cols)
```

**Arguments**

- **source**: incoming rel_op tree or data.frame.
- **cols**: named character, values are columns to track, names are where to land indicators.
Value

rel_op node or data.frame (depending on input).

See Also

null_replace, count_null_cols

Examples

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

d <- rq_copy_to(my_db,
  #quotesingle.Var
d
  #quotemeasure.end
  data.frame(AUC = c(0.6, 0.5, NA),
             R2 = c(1.0, 0.9, NA)))

  op_tree <- d %.>% mark_null_cols(., qc(AUC_NULL, R2_NULL) %:=% qc(AUC, R2))

cat(format(op_tree))
sql <- to_sql(op_tree, my_db)
cat(sql)
print(DBI::dbGetQuery(my_db, sql))

  # ad-hoc mode
  data.frame(AUC=c(1,NA,0.5), R2=c(NA,1,0)) %.>%
  op_tree %.>%
  print(.)

  # cleanup
  options(old_o)
  DBI::dbDisconnect(my_db)
}

---

**materialize**

**Materialize an optree as a table.**

**Description**

Run the data query as a CREATE TABLE AS. Think of as a function that can be applied to relop trees, not as a component to place in pipelines.
Usage

materialize(
  db,
  optree,
  table_name = mk_tmp_name_source("rquery_mat")(),
  ..., limit = NULL,
  source_limit = NULL,
  overwrite = TRUE,
  temporary = FALSE,
  qualifiers = NULL
)

Arguments

db database connection (rquery_db_info class or DBI connections preferred).
optree relop operation tree.
table_name character, name of table to create.
... force later arguments to bind by name.
limit numeric if not NULL result limit (to use this, last statement must not have a limit).
source_limit numeric if not NULL limit sources to this many rows.
overwrite logical if TRUE drop an previous table.
temporary logical if TRUE try to create a temporary table.
qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Value
table description

See Also

db_td, execute, to_sql, rq_copy_to, mk_td

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"

  d <- rq_copy_to(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2),
    temporary = TRUE, overwrite = TRUE)
  optree <- extend_se(d, c("v" %=:="AUC + R2", "x" %=:="pmax(AUC,v)"))
  cat(format(optree))
  res <- materialize(my_db, optree, "example")
}
```r
materialize_node

Create a materialize node.

Description

Write results into a specified table. Result is transient, lives only for the duration of the pipeline calculation. This node is only used to break up or un-nest calculations, not for value sharing or re-use.

Usage

materialize_node(
  source,
  table_name = (wrapt::mk_tmp_name_source("rquerymn"))(),
  ..., 
  qualifiers = NULL
)

Arguments

source      source to work from (relop node)
table_name  character, name of caching table
...         not used, force later argument to bind by name
qualifiers  optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Details

Note this node can not be used in multiple paths in the same rel_op tree as it re-uses table names and re-computes each time called.

Value

relop materialize_node

See Also

rsummary_node, non_sql_node
```
**Description**

Build minimal structures (table name and column names) needed to represent data from a remote table.

**Usage**

```r
mk_td(
  table_name,  # character, name of table
  columns,     # character, column names of table (non-empty and unique values).
  ...,         # not used, force later argument to bind by name
  qualifiers = NULL,  # optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
  q_table_name = NULL,  # optional character, qualified table name, note: has to be re-generated for different DB connections.
  head_sample = NULL,   # optional, head_sample of table as an example
  limit_was = NULL      # optional, row limit used to produce head_sample.
)
```

**Arguments**

- `table_name`: character, name of table
- `columns`: character, column names of table (non-empty and unique values).
- `...`: not used, force later argument to bind by name
- `qualifiers`: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
- `q_table_name`: optional character, qualified table name, note: has to be re-generated for different DB connections.
- `head_sample`: optional, head_sample of table as an example
- `limit_was`: optional, row limit used to produce head_sample.

**Details**

Generate a query that returns contents of a table, we could try to eliminate this (replace the query with the table name), but there are features one can work with with the query in place and SQL optimizers likely make this zero-cost anyway.
natural_join

Make a natural_join node.

Description

Natural join is a join by identity on all common columns specified in the by argument. Any common columns not specified in the by argument are coalesced into a single column preferring the first or "a" table.

Usage

natural_join(a, b, ..., by, jointype = "INNER", env = parent.frame())
natural_join

Arguments

- **a**: source to select from.
- **b**: source to select from.
- **...**: force later arguments to bind by name
- **by**: character, set of columns to match. If by is a named character vector the right table will have columns renamed.
- **jointype**: type of join ('INNER', 'LEFT', 'RIGHT', 'FULL').
- **env**: environment to look to.

Value

natural_join node.

Examples

```r
if(requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), 
    ":memory:")

d1 <- rq_copy_to(
  my_db, 'd1',
  build_frame(
    "key", "val", "val1" | 
    "a" , 1 , 10 | 
    "b" , 2 , 11 | 
    "c" , 3 , 12 ))

d2 <- rq_copy_to(
  my_db, 'd2',
  build_frame(
    "key", "val", "val2" | 
    "a" , 5 , 13 | 
    "b" , 6 , 14 | 
    "d" , 7 , 15 ))

# key matching join
optree <- natural_join(d1, d2, 
  jointype = "LEFT", by = 'key')
execute(my_db, optree) %>%
  print(.)

DBI::dbDisconnect(my_db)
}
```
non_sql_node

Wrap a non-SQL node.

Description
Note: non-SQL nodes are allowed to delete/overwrite both both the incoming and outgoing tables, so do not point them to non-temporary structures. Also they tend to land all columns (losing narrowing optimization), so can be expensive and should be used sparingly. Finally their result can only be used once in a pipeline (else they will try to clobber their own result).

Usage

non_sql_node(
  source,
  ..., 
  f_db = NULL,
  f_df = NULL,
  f_dt = NULL,
  incoming_table_name,
  incoming_qualifiers = NULL,
  outgoing_table_name,
  outgoing_qualifiers = NULL,
  columns_produced,
  display_form = "non_sql_node",
  orig_columns = TRUE,
  temporary = TRUE,
  check_result_details = TRUE,
  env = parent.frame()
)  

Arguments

source  source to work from (data.frame or relop node)
...
force later arguments to bind by name
f_db database implementation signature: f_db(db, incoming_table_name, outgoing_table_name, nd, ...) (db being a database handle)

f_df data.frame implementation signature: f_df(data.frame, nd) (NULL defaults to taking from database).

f_dt data.table implementation signature: f_dt(data.table, nd) (NULL defaults f_df).
incoming_table_name character, name of incoming table
incoming_qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

outgoing_table_name character, name of produced table
normalize_cols

outgoing_qualifiers
optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
columns_produced
character, names of additional columns produced
display_form
character, how to print node
orig_columns
logical if TRUE select all original columns.
temporary
logical, if TRUE mark tables temporary.
check_result_details
logical, if TRUE enforce result type and columns.
env
environment to look to.

Value
non-sql node.

See Also
rsummary_node, quantile_node

normalize_cols Build an optree pipeline that normalizes a set of columns so each column sums to one in each partition.

Description
This is an example of building up a desired pre-prepared pipeline fragment from relop nodes.

Usage
normalize_cols(source, columns, ..., partitionby = NULL, env = parent.frame())

Arguments
source relop tree or data.frame source.
columns character, columns to normalize.
... force later arguments to bind by name.
partitionby partitioning (window function) column names to define partitions.
env environment to look for values in.
Examples

# by hand logistic regression example
scale <- 0.237
d <- mk_td("survey_table",
c("subjectID", "surveyCategory", "assessmentTotal"))
optree <- d %>%
extend(.,
  probability %:=
    exp(assessmentTotal * scale)) %>%
normalize_cols(.,
  "probability",
  partitionby = 'subjectID') %>%
pick_top_k(.,
  partitionby = 'subjectID',
  orderby = c("probability", "surveyCategory"),
  reverse = c("probability")) %>%
rename_columns(.,
  'diagnosis' %:= 'surveyCategory') %>%
select_columns(.,
c("subjectID",
  'diagnosis',
  "probability")) %>%
orderby(., 'subjectID')
cat(format(optree))

null_replace

Create a null_replace node.

Description

Replace NA/NULL is specified columns with the given replacement value.

Usage

null_replace(src, cols, value, ..., note_col = NULL, env = parent.frame())

Arguments

src relop or data.frame data source.
cols character, columns to work on.
value scalar, value to write.
... force later arguments to bind by name.
note_col character, if not NULL record number of columns altered per-row in this column.
env environment to look to.
Value

null_replace node or data.frame.

See Also

count_null_cols, mark_null_cols

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d1 <- rq_copy_to(my_db, "d1",
      data.frame(A = c(NA, 2, 3, NA), B = c(3, NA, 4, NA)))
  optree <- null_replace(d1, qc(A, B),
    0.0, note_col = "alterations"
  )
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

op_diagram

Build a diagram of a optree pipeline.

Description

Build a diagram of a optree pipeline.

Usage

```r
op_diagram(optree, ..., merge_tables = FALSE, show_table_columns = TRUE)
```

Arguments

```r
optree  operation tree pipeline (or list of such).
...
merge_tables  logical, if TRUE merge all same table references into one node. rel_op nodes that declare a materialize_as name will be cross-linked.
show_table_columns  logical, if TRUE show table columns.
```

Value

character DiagrammeR::grViz() ready text.
Examples

```r
d <- mk_td('d',
    columns = qc(AUC, R2))
optree <- d %.>%
    extend(., v %:=% ifelse(AUC>0.5, R2, 1.0)) %.>%
    quantile_node(.) %.>%
    natural_join(., d, jointype = "LEFT", by = "AUC") %.>%
    orderby(., "AUC")

cat(format(optree))
cat(op_diagram(optree))

# if(requireNamespace("DiagrammeR", quietly = TRUE)) {
#   op_diagram(., merge_tables = TRUE) %.>%
#   DiagrammeR::grViz(.) %.>%
#   print(.)
#   # or to render to png
#   op_diagram(., merge_tables = TRUE) %.>%
#   DiagrammeR::DiagrammeR(diagram = ., type = "grViz") %.>%
#   DiagrammeRsvg::export_svg(.) %.>%
#   charToRaw(.) %.>%
#   rsvg::rsvg_png(., file = "diagram1.png")
# }
```

**orderby**

*Make an orderby node (not a relational operation).*

**Description**

Order a table by a set of columns (not general expressions) and limit number of rows in that order.

**Usage**

```r
orderby(
    source,
    cols = NULL,
    ..., reverse = NULL,
    limit = NULL,
    env = parent.frame()
)
```
order_expr

Arguments

- `source`: source to select from.
- `cols`: order by named columns ascending.
- `...`: force later arguments to be bound by name.
- `reverse`: character, which columns to reverse ordering of top descending.
- `limit`: number limit row count.
- `env`: environment to look to.

Details

Note: this is a relational operator in that it takes a table that is a relation (has unique rows) to a table that is still a relation. However, most relational systems do not preserve row order in storage or between operations. So without the limit set this is not a useful operator except as a last step prior to pulling data to an in-memory data.frame (which does preserve row order).

Value

`order_by` node.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- rq_copy_to(my_db, 'd',
                  data.frame(AUC = 0.6, R2 = 0.2))
  optree <- orderby(d, cols = "AUC", reverse = "AUC", limit=4)
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

order_expr

Make a `order_expr` node.

Description

`order_expr()` uses `bquote()` .()-style escaping.

Usage

```r
order_expr(source, expr, env = parent.frame())
order_expr_nse(source, expr, env = parent.frame())
```
order_expr_se

Arguments

source  source to select from.
expr    expression to order_expr.
env     environment to look to.

Value

select columns node.

Examples

if (requireNamespace("DBI", quietly = TRUE) & requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  d <- rq_copy_to(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2, z = 3))
  TARGETCOL = as.name("AUC")
  optree <- order_expr(d, .(TARGETCOL)/R2) .%>%
    select_columns(., "R2")
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}

order_expr_se  Make a order_expr node.

Description

Make a order_expr node.

Usage

order_expr_se(source, expr, env = parent.frame())

Arguments

source  source to select from.
expr    expression to order_expr in ascending order.
env     environment to look for values in.

Value

select columns node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "memory:")
  d <- rq_copy_to(my_db, 'd',
                   data.frame(AUC = 0.6, R2 = 0.2))
  optree <- order_expr_se(d, "AUC/R2")
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

order_rows

Make an orderby node (not a relational operation).

Description

Order a table by a set of columns (not general expressions) and limit number of rows in that order.

Usage

```r
order_rows(
  source,
  cols = NULL,
  ...,
  reverse = NULL,
  limit = NULL,
  env = parent.frame()
)
```

Arguments

- `source`: source to select from.
- `cols`: order by columns ascending.
- `...`: force later arguments to be bound by name
- `reverse`: character, which columns to reverse ordering of to descending.
- `limit`: number limit row count.
- `env`: environment to look to.

Details

Note: this is a relational operator in that it takes a table that is a relation (has unique rows) to a table that is still a relation. However, most relational systems do not preserve row order in storage or between operations. So without the limit set this is not a useful operator except as a last step prior to pulling data to an in-memory data.frame (which does preserve row order).
pick_top_k

Build an optree pipeline that selects up to the top k rows from each group in the given order.

Description

This is an example of building up a desired pre-prepared pipeline fragment from relop nodes.

Usage

```r
pick_top_k(
  source,
  ..., force later arguments to bind by name.
  partitionby = NULL,
  orderby = NULL,
  reverse = NULL,
  k = 1L,
  order_expression = "row_number()",
  order_column = "row_number",
  keep_order_column = TRUE,
  env = parent.frame()
)
```

Arguments

- **source**: relop tree or data.frame source.
- **...**: force later arguments to bind by name.
- **partitionby**: partitioning (window function) column names.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- rq_copy_to(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2))
  optree <- order_rows(d, cols = "AUC", reverse = "AUC", limit=4)
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}

```
pre_sql_sub_expr

orderby character, ordering (in window function) column names.
reverse character, reverse ordering (in window function) of these column names.
k integer, number of rows to limit to in each group.
order_expression character, command to compute row-order/rank.
order_column character, column name to write per-group rank in (no ties).
keep_order_column logical, if TRUE retain the order column in the result.
env environment to look for values in.

Examples

# by hand logistic regression example
scale <- 0.237
d <- mk_td("survey_table",
  c("subjectID", "surveyCategory", "assessmentTotal")))
optree <- d %>%
  extend(.,
    probability %:=%
      exp(assessmentTotal * scale)) %>%
  normalize_cols(.,
    "probability",
    partitionby = "subjectID") %>%
  pick_top_k(.,
    partitionby = "subjectID",
    orderby = c("probability", "surveyCategory"),
    reverse = c("probability", "surveyCategory")) %>%
  rename_columns(.,
    "diagnosis" %:=%
      "surveyCategory") %>%
  select_columns(., c("subjectID",
    "diagnosis",
    "probability")) %>%
  orderby(., "subjectID")
cat(format(optree))

Description

represents an expression. Unnamed list of pre_sql_terms and character.

Usage

pre_sql_sub_expr(terms, info = NULL)
Arguments

terms
info

Value

pre_sql_sub_expr

Description

Supports bquote() .()-style name abstraction including .(-) notation to promote strings to names (please see here: https://github.com/WinVector/rquery/blob/master/Examples/Substitution/Substitution.md).

Usage

project(source, ..., groupby = c(), env = parent.frame())

project_nse(source, ..., groupby = c(), env = parent.frame())

aggregate_nse(source, ..., groupby = c(), env = parent.frame())

Arguments

source
...
groupby
eve

Value

project node.

Examples

if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(),
    ":memory:"
  )
  d <- rq_copy_to(
    my_db, 'd',
    data.frame(group = c('a', 'a', 'b', 'b'),
      val = 1:4,
    )
  )
```r
stringsAsFactors = FALSE)

op_tree <- d %.>%
  project(., groupby = "group", vmax %:=% max(val))
cat(format(op_tree))
sql <- to_sql(op_tree, my_db)
cat(sql)
execute(my_db, op_tree) %.>%
  print(.)

op_tree <- d %.>%
  project(., groupby = NULL, vmax %:=% max(val))
cat(format(op_tree))
sql <- to_sql(op_tree, my_db)
cat(sql)
execute(my_db, op_tree) %.>%
  print(.)

DBI::dbDisconnect(my_db)
}
```

---

**project_se**

*project data by grouping, and adding aggregate columns.*

**Description**

project data by grouping, and adding aggregate columns.

**Usage**

```r
project_se(source, assignments, ..., groupby = c(), env = parent.frame())
aggregate_se(source, assignments, ..., groupby = c(), env = parent.frame())
```

**Arguments**

- **source**: source to select from.
- **assignments**: new column assignment expressions.
- **...**: not used, force later arguments to be by name
- **groupby**: grouping columns.
- **env**: environment to look for values in.

**Value**

project node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
    my_db <- DBI::dbConnect(RSQLite::SQLite(),
                           "::memory:"
    d <- rq_copy_to(
        my_db, 'd',
        data.frame(group = c('a', 'a', 'b', 'b'),
                    val = 1:4,
                    stringsAsFactors = FALSE))

    op_tree <- d %.>%
                  project_se(., groupby = "group", "vmax" %:=% "max(val)")
    cat(format(op_tree))
    sql <- to_sql(op_tree, my_db)
    cat(sql)
    execute(my_db, op_tree) %.>%
        print(.)

    op_tree <- d %.>%
                  project_se(., groupby = NULL, "vmax" %:=% "max(val)")
    cat(format(op_tree))
    sql <- to_sql(op_tree, my_db)
    cat(sql)
    execute(my_db, op_tree) %.>%
        print(.)

    DBI::dbDisconnect(my_db)
}
```

quantile_cols

Compute quantiles of specified columns (without interpolation, needs a database with window functions).

**Description**

Compute quantiles of specified columns (without interpolation, needs a database with window functions).

**Usage**

```r
quantile_cols(
    db, incoming_table_name, ...
    probs = seq(0, 1, 0.25),
    probs_name = "quantile_probability",
    cols = rq_colnames(db, incoming_table_name),
```
**quantile_node**

```r
     qualifiers = NULL

Arguments

- `db`: database connection
- `incoming_table_name`: name of table to compute quantiles of
- `...`: force later arguments to bind by name
- `probs`: numeric, probabilities to compute quantiles of
- `probs_name`: character name for probability column
- `cols`: character, columns to compute quantiles of
- `qualifiers`: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Value

data.frame of quantiles

See Also

- `quantile_node.rsummary`

---

**quantile_node**

*Compute quantiles over non-NULL values (without interpolation, needs a database with window functions).*

Description

Please see [https://github.com/WinVector/rquery/blob/master/extras/Summary_Example.md](https://github.com/WinVector/rquery/blob/master/extras/Summary_Example.md) for an example.

Usage

```r
quantile_node(
  source,
  cols = NULL,
  ...,
  probs_name = "quantile_probability",
  probs = seq(0, 1, 0.25),
  tmp_name_source = wrapr::mk_tmp_name_source("qn"),
  temporary = TRUE,
  qualifiers = NULL
)
```
quote_identifier

Arguments

source: source to select from (relop or data.frame).
cols: character, compute quantiles for these columns (NULL indicates all columns).
...: force later arguments to be bound by name
probs_name: character, column name to write probs in.
probs: numeric quantiles to compute
tmp_name_source: wrapr::mk_tmp_name_source(), temporary name generator.
temporary: logical, if TRUE use temporary tables
qualifiers: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

details

This is a non_sql_node, so please see non_sql_node for some of the issues for this node type.

Value

table of quantiles

See Also

quantile_cols, rsummary, non_sql_node

quote_identifier

Quote an identifier.

description

Quote an identifier.

usage

quote_identifier(x, id)

arguments

x: database handle or rquery_db_info object.
id: character to quote

value

quoted identifier
quote_literal

Description
Quote a value

Usage
quote_literal(x, o)

Arguments
x  database handle or rquery_db_info object.
o  value to quote

Value
quoted string

quote_string

Description
Quote a string

Usage
quote_string(x, s)

Arguments
x  database handle or rquery_db_info object.
s  character to quote

Value
quoted string
quote_table_name

Quote a table name.

Description
Quote a table name.

Usage
quote_table_name(x, id, ..., qualifiers = character(0))

Arguments
x database handle or rquery_db_info object.
id character to quote
... not used, force later arguments to bind by name.
qualifiers named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Value
quoted identifier

rename_columns

Make a rename columns node (copies columns not renamed).

Description
Make a rename columns node (copies columns not renamed).

Usage
code{
rename_columns(source, cmap, env = parent.frame())
}

Arguments
source source to rename from.
cmap map written as new column names as keys and old column names as values.
env environment to look to.

Value
tmprename columns node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "':memory:'")
  d <- rq_copy_to(my_db, 'd',
                   data.frame(AUC = 0.6, R2 = 0.2, z = 3))
  op_tree <- rename_columns(d, c('R2' := 'AUC', 'AUC' := 'R2'))
  cat(format(op_tree))
  sql <- to_sql(op_tree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

---

**row_counts**  
*Build an optree pipeline counts rows.*

**Description**

This is an example of building up a desired pre-prepared pipeline fragment from relop nodes.

**Usage**

```r
row_counts(source, ..., groupby = character(0), env = parent.frame())
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>relop tree or data.frame source.</td>
</tr>
<tr>
<td>...</td>
<td>force later arguments to bind by name.</td>
</tr>
<tr>
<td>groupby</td>
<td>partitioning (window function) column names.</td>
</tr>
<tr>
<td>env</td>
<td>environment to look for values in.</td>
</tr>
</tbody>
</table>

**Examples**

```r
# by hand logistic regression example
d <- mk_td("survey_table",
            c("subjectID", "surveyCategory", "assessmentTotal"))
optree <- d %.>%
row_counts(d, groupby = "subjectID")
cat(format(opTree))
```
**rquery_apply_to_data_frame**

Execute optree in an environment where d is the only data.

**Description**

Default DB uses RSQLite (so some functions are not supported).

**Usage**

```r
rquery_apply_to_data_frame(
  d,
  optree,
  ...,  
  limit = NULL,
  source_limit = NULL,
  allow_executor = TRUE,
  env = parent.frame()
)
```

**Arguments**

- **d**: data.frame or named list of data.frames.
- **optree**: rquery rel_op operation tree.
- **...**: force later arguments to bind by name.
- **limit**: integer, if not NULL limit result to no more than this many rows.
- **source_limit**: numeric if not NULL limit sources to this many rows.
- **allow_executor**: logical if TRUE allow any executor set as rquery.rquery_executor to be used.
- **env**: environment to look to.

**Value**

data.frame result

**Examples**

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

    optree <- mk_td("d", c("AUC", "R2", "D")) %>%
              extend(., c %:% sqrt(R2)) %>%
              orderby(., cols = "R2", reverse = "R2")
}
```
Build a db information stand-in

Description
Build a db information stand-in

Usage
rquery_db_info(
  ...,
  connection = NULL,
  is_dbi = FALSE,
  identifier_quote_char = "\\",
  string_quote_char = "\"",
  overrides = NULL,
  note = "",
  connection_options = rq_connection_advice(connection),
  db_methods = rquery_default_methods()
)

Arguments

  ... force all arguments to be by name.
  connection connection handle to database or Spark.
  is_dbi if TRUE the database connection can be used with DBI.
  identifier_quote_char character, quote to put around identifiers.
rq_colnames

string_quote_char
class, quote to put around strings.

overrides
named list of functions to place in info.

note
character note to add to display form.

connection_options
named list of per-connection options.

db_methods
named list of to_sql methods.

Value
rquery_db_info object

rquery_default_db_info

An example rquery_db_info object useful for formatting SQL without a database connection.

Description
An example rquery_db_info object useful for formatting SQL without a database connection.

Usage
rquery_default_db_info()

Value

a rquery_db_info without a connection and vanilla settings.

rq_colnames
List table column names.

Description
List table column names.

Usage
rq_colnames(db, table_name, ..., qualifiers = NULL)

Arguments
db
Connection handle

table_name
character table name

...
not used, force later argument to bind by name

qualifiers
optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
rq_coltypes

Value

character list of column names

Get column types by example values as a data.frame.

Description

Example values not necessarily all from same row. Taking values from different rows is to try to work around NA not carrying type/class info in many cases.

Usage

rq_coltypes(
  db,
  table_name,
  ..., 
  qualifiers = NULL,
  prefer_not_NA = FALSE,
  force_check = FALSE
)

Arguments

db Connection handle.
table_name character table name referring to a non-empty table.
... force later arguments to bind by name.
qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
prefer_not_NA logical, if TRUE try to find an non-NA example for all columns (FALSE just for logical columns).
force_check logical, if TRUE perform checks regardless of check_logical_column_types option setting.

Value

single row data.frame with example values, not all values necessarily from same database row.

Examples

if(requireNamespace("DBI", quietly = TRUE) & requireNamespace("RSQLite", quietly = TRUE)) {
  db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"

  # getDBOption(db, "check_logical_column_types", FALSE)
  # options(rq_connection_tests(db))
Get advice for a DB connection (beyond tests).

These settings are set by the package maintainers based on experience with specific databases.

Usage

rq_connection_advice(db)

Arguments

db
database connection handle

Value

named list of options
rq_connection_name

Build a canonical name for a db connection class.

Description

Build a canonical name for a db connection class.

Usage

rq_connection_name(db)

Arguments

db Database connection handle.

Value

character, key version of handle for option lookups.

Examples

```r
if(requireNamespace("DBI", quietly = TRUE) & requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  print(rq_connection_name(my_db))
  print(rq_connection_advice(my_db))
  DBI::dbDisconnect(my_db)
}
```
rq_connection_tests  Try and test database for some option settings.

Description

These settings are estimated by experiments. This is not the full set of options- but just the ones tested here.

Usage

rq_connection_tests(db, ..., overrides = NULL, use_advice = TRUE)

Arguments

db  database connection handle.
...
force later arguments to bind by name.
overrides  named character vector or list, options (just name, not DB qualification) to force
use_advice  logical if TRUE incorporate hard-coded advice.

Details

Note: tests are currently run in the default schema. Also it is normal to see some warning/error messages as different database capabilities are tested.

Value

named list of options

See Also

rq_connection_advice

Examples

if(requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  print(rq_connection_name(my_db))
  print(rq_connection_tests(my_db,
    overrides = c("use_DBI_dbExistsTable" = FALSE)))
  # the following would set options
  # print(options(rq_connection_tests(my_db)))
  DBI::dbDisconnect(my_db)
}
Copy local R table to remote data handle.

Description

Copy local R table to remote data handle.

Usage

rq_copy_to(
  db,
  table_name,
  d,
  ...
  qualifiers = NULL,
  overwrite = FALSE,
  temporary = TRUE,
  rowidcolumn = NULL
)

Arguments

db database connection handle.

name of table to create.

data.frame to copy to database.

force later argument to be by name

optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

logical, if TRUE try to overwrite existing table.

logical, if TRUE try to mark table as temporary.

character, name to land row-ids.

Value

a relop representation of the data

See Also
db_td, mk_td, materialize, execute, to_sql
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- rq_copy_to(db, 'd',
                 data.frame(AUC = 0.6, R2 = 0.2))

  sql <- to_sql(d, db)
  cat(sql)
  print(DBI::dbGetQuery(db, "SELECT * FROM d"))
  DBI::dbDisconnect(db)
}
```

---

**rq_execute**  
Execute a query, typically an update that is not supposed to return results.

Description

Execute a query, typically an update that is not supposed to return results.

Usage

```r
rq_execute(db, q)
```

Arguments

- `db`:
  - database connection handle

- `q`:
  - character query

Value

nothing

See Also

`db_td`
rq_get_query

Execute a get query, typically a non-update that is supposed to return results.

Description
Execute a get query, typically a non-update that is supposed to return results.

Usage
rq_get_query(db, q)

Arguments
- db: database connection handle
- q: character query

Value
nothing

See Also
db_td

rq_head
Get head of db table

Description
Get head of db table

Usage
rq_head(db, table_name, ..., qualifiers = NULL, limit = 6L)

Arguments
- db: Connection handle
- table_name: character table name
- ...: not used, force later argument to bind by name
- qualifiers: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.
- limit: integer, how many rows to take
rq_remove_table

Value

first few rows

---

rq_nrow  Count rows and return as numeric

Description

Count rows and return as numeric

Usage

rq_nrow(db, table_name, ..., qualifiers = NULL)

Arguments

- **db**: database connection
- **table_name**: character, name of table
- **...**: not used, force later argument to bind by name
- **qualifiers**: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Value

numeric row count

See Also

db_td

---

rq_remove_table  Remove table

Description

Remove table

Usage

rq_remove_table(db, table_name, ..., qualifiers = NULL)
**rq_table_exists**

**Arguments**

- `db` database connection.
- `table_name` character, name of table to create.
- `...` not used, force later argument to bind by name
- `qualifiers` optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

**Value**

logical TRUE if table existed, else FALSE

**See Also**

`db_td`

**Description**

Check if a table exists.

**Usage**

```r
rq_table_exists(db, table_name, ..., qualifiers = NULL)
```

**Arguments**

- `db` Connection handle
- `table_name` character table name
- `...` not used, force later argument to bind by name
- `qualifiers` optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

**Value**

logical TRUE if table exists.

**See Also**

`db_td`
rstr

Quick look at remote data

Description
Quick look at remote data

Usage

```r
rstr(
  my_db,
  tableName,
  ..., 
  displayRows = 10,
  countRows = TRUE,
  qualifiers = NULL
)
```

```r
rlook(
  my_db,
  tableName,
  ..., 
  displayRows = 10,
  countRows = TRUE,
  qualifiers = NULL
)
```

Arguments

- `my_db`: database handle
- `tableName`: name of table to look at
- `...`: not used, force later arguments to bind by name
- `displayRows`: number of rows to sample
- `countRows`: logical, if TRUE return row count.
- `qualifiers`: optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Value

str view of data

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
```
rsummary <- DBI::dbConnect(RSQLite::SQLite(), "::memory::")
DBI::dbWriteTable(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2),
    overwrite = TRUE,
    temporary = TRUE)

rlook(my_db, 'd')
DBI::dbDisconnect(my_db)
}

rsummary

rs summary

Compute usable summary of columns of remote table.

Description

Compute per-column summaries and return as a data.frame. Warning: can be an expensive operation.

Usage

rsummary(
    db,
    tableName,
    ...,
    countUniqueNum = FALSE,
    quartiles = FALSE,
    cols = NULL,
    qualifiers = NULL
)

Arguments

db database connection.
tableName name of table.
... force additional arguments to be bound by name.
countUniqueNum logical, if TRUE include unique non-NA counts for numeric cols.
quartiles logical, if TRUE add Q1 (25%), median (50%), Q3 (75%) quartiles.
cols if not NULL set of columns to restrict to.
qualifiers optional named ordered vector of strings carrying additional db hierarchy terms, such as schema.

Details

For numeric columns includes NaN in nna count (as is typical for R, e.g., is.na(NaN)).
Value

data.frame summary of columns.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) &&
     requireNamespace("RSQLite", quietly = TRUE)) {
  d <- data.frame(p = c(TRUE, FALSE, NA),
                   s = NA,
                   w = 1:3,
                   x = c(NA, 2, 3),
                   y = factor(c(3, 5, NA)),
                   z = c('a', NA, 'a'),
                   stringsAsFactors=FALSE)
  db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
                      RSQLite::initExtension(db)
                      rq_copy_to(db, "dRemote", d,
                                  overwrite = TRUE, temporary = TRUE)
  print(rsummary(db, "dRemote"))
  DBI::dbDisconnect(db)
}
```

---

**rsummary_node**  
Create an rsumary relop operator node.

**Description**

This is a non_sql_node, so please see non_sql_node for some of the issues for this node type.

**Usage**

```r
rsummary_node(
  source,  
  ...,  
  quartiles = FALSE,  
  tmp_name_source = wrapr::mk_tmp_name_source("sn"),  
  temporary = TRUE
)
```

**Arguments**

- **source**: incoming source (relop node or data.frame).
- **...**: force later arguments to be by name
- **quartiles**: logical, if TRUE add Q1 (25%), median (50%), Q3 (75%) quartiles.
- **tmp_name_source**: wrapr::mk_tmp_name_source(), temporary name generator.
- **temporary**: logical, if TRUE use temporary tables
select_columns

Make a select columns node (not a relational operation).

Description

Make a select columns node (not a relational operation).

Usage

select_columns(source, columns, env = parent.frame())
Arguments

- **source**: source to select from.
- **columns**: list of distinct column names.
- **env**: environment to look to.

Value

select columns node.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  d <- rq_copy_to(my_db, 'd',
                  data.frame(AUC = 0.6, R2 = 0.2))
  optree <- select_columns(d, 'AUC')
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

---

**select_rows**

Make a select rows node.

Description

Supports bquote() .()-style name abstraction including .() notation to promote strings to names (please see here: [https://github.com/WinVector/rquery/blob/master/Examples/Substitution/Substitution.md](https://github.com/WinVector/rquery/blob/master/Examples/Substitution/Substitution.md)).

Usage

```r
select_rows(source, expr, env = parent.frame())

select_rows_nse(source, expr, env = parent.frame())
```

Arguments

- **source**: source to select from.
- **expr**: expression to select rows.
- **env**: environment to look to.
Value

select rows node.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "::memory:"
  d <- rq_copy_to(my_db, 'd',
    data.frame(AUC = 0.6, R2 = 0.2, z = 3))
  TARGETCOL = as.name("AUC")
  optree <- select_rows(d, .(TARGETCOL) >= 0.5) %.>%
    select_columns(., "R2")
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

select_rows_se  

Make a select rows node.

Description

Make a select rows node.

Usage

`select_rows_se(source, expr, env = parent.frame())`

Arguments

- `source`: source to select from.
- `expr`: expression to select rows.
- `env`: environment to look for values in.

Value

select rows node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d <- rq_copy_to(my_db, 'd',
                 data.frame(AUC = 0.6, R2 = 0.2))
  optree <- select_rows_se(d, "AUC >= 0.5")
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

---

**setDBOpt**

Set a database connection option.

**Description**

If db is of class rquery_db_info it sets the appropriate connection option, not the global state.

**Usage**

```r
setDBOpt(db, optname, val)
```

**Arguments**

- `db`: rquery_db_info instance
- `optname`: character, single option name.
- `val`: value to set

**Value**

- `db`

---

**setDBOption**

Set a database connection option.

**Description**

Note: we are moving away from global options to options in the DB handle. Prefer `setDBOpt`.

**Usage**

```r
setDBOption(db, optname, val)
```
Arguments

- **db**
  database connection handle.
- **optname**
  character, single option name.
- **val**
  value to set

Value

- original options value

---

**set_indicator**

*Make a set indicator node.*

Description

Create a new column indicating the membership of another column in a given set.

Usage

```r
set_indicator(
  source,
  rescol,
  testcol,
  testvalues,
  ...,
  translate_quotes = FALSE,
  env = parent.frame()
)
```

Arguments

- **source**
  source to select from.
- **rescol**
  name of column to land indicator in.
- **testcol**
  name of column to check.
- **testvalues**
  values to check for.
- **...**
  force later arguments to bind by name
- **translate_quotes**
  logical if TRUE translate quotes to SQL choice (simple replacement, no escaping).
- **env**
  environment to look to.

Value

- set_indicator node.
sql_expr_set

Build a query that applies a SQL expression to a set of columns.

Description

Build a query that applies a SQL expression to a set of columns.

Usage

sql_expr_set(source, cols, expr)
sql_expr_set

Arguments

source
incoming rel_op tree or data.frame.

cols
character, columns to operate in. If a named array names are where results are landed, values names of value columns.

expr
character or list of character and names, expression to apply to columns "." stands for column value to use.

Value

rel_op node or data.frame (depending on input).

See Also

null_replace, count_null_cols, mark_null_cols

Examples

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

d <- rq_copy_to(my_db, 'd',
               data.frame(AUC = c(NA, 0.5, NA),
                           R2 = c(1.0, 0.9, NA),
                           delta = 3,
                           cat = c("a", NA, "c"),
                           stringsAsFactors = FALSE))

# example
op_tree <- d %.>% sql_expr_set(., qc(AUC, R2), ". + 1")
cat(format(op_tree))
sql <- to_sql(op_tree, my_db)
cat(sql)
print(DBI::dbGetQuery(my_db, sql))

# ex2 names (but not marked as names)
op_tree2 <- d %.>% sql_expr_set(., qc(AUC, R2), ". + 1 + delta")
cat(to_sql(op_tree2, my_db))

# ex3 names (also so marked)
op_tree3 <- d %.>% sql_expr_set(., qc(AUC, R2),
                                  list(". + 1 +", as.name("delta")))
cat(to_sql(op_tree3, my_db))

# cleanup
options(old_o)
DBI::dbDisconnect(my_db)
sql_node

Make a general SQL node.

Description

Make a general SQL node.

Usage

sql_node(
  source,
  exprs,
  ..., 
  mods = NULL,
  orig_columns = TRUE,
  expand_braces = TRUE,
  translate_quotes = TRUE,
  env = parent.frame()
)

Arguments

source       source to work from.
exprs        SQL expressions
...          force later arguments to bind by name
mods          SQL modifiers (GROUP BY, ORDER BY, and so on)
orig_columns  logical if TRUE select all original columns.
expand_braces logical if TRUE use col notation to ensure col is a column name.
translate_quotes logical if TRUE translate quotes to SQL choice (simple replacement, no escaping).
env           environment to look to.

Value

sql node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  # example database connection
  my_db <- DBI::dbConnect(RSQLite::SQLite(),
                          "::memory:"
  )
  # load up example data
  d <- rq_copy_to(
    my_db, 'd',
    data.frame(v1 = c(1, 2, NA, 3),
                v2 = c(NA, "b", NA, "c"),
                v3 = c(NA, NA, 7, 8),
                stringsAsFactors = FALSE)
  )
  # look at table
  execute(my_db, d)
  # get list of columns
  vars <- column_names(d)
  print(vars)

  # 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)
  cat(paste(unlist(expr), collapse = " "))

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

  # examine produced SQL
  sql <- to_sql(op_tree, my_db)
  cat(sql)

  # execute
  execute(my_db, op_tree) %>%
    print(.)

  # whole process wrapped in convenience node
  op_tree2 <- d %>%
    count_null_cols(., vars, "nnull")
  execute(my_db, op_tree2) %>%
    print(.)
}
```
# sql_node also allows marking variable in quoted expressions
ops <- d %.>%
  sql_node(., qae(sqrt_v1 = sqrt(.[v1])))
execute(my_db, ops) %.>%
  print(.)
# marking variables allows for error-checking of column names
tryCatch({
  ops <- d %.>%
  sql_node(., qae(sqrt_v1 = sqrt(.[v1_misspelled])))
},
  error = function(e) {print(e)})
DBI::dbDisconnect(my_db)

<table>
<thead>
<tr>
<th>tables_used</th>
<th>Return vector of table names used.</th>
</tr>
</thead>
</table>

**Description**

Return vector of table names used.

**Usage**

```r
tables_used(node, ...)
```

**Arguments**

- **node**: rquery tree to examine.
- **...**: (not used)

**Value**

names of tables used.

**Examples**

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "memory:"
  d1 <- rq_copy_to(my_db, 'd1',
    data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
    data.frame(AUC = 0.6, D = 0.3))
  optree <- natural_join(d1, d2, by = "AUC")
  cat(format(optree))
  print(tables_used(optree))
}
```
Description

Theta join is a join on an arbitrary predicate.

Usage

theta_join(a, b, expr, ..., jointype = "INNER", suffix = c("_a", "_b"), env = parent.frame())

theta_join_nse(a, b, expr, ..., jointype = "INNER", suffix = c("_a", "_b"), env = parent.frame())

Arguments

a source to select from.
b source to select from.
expr unquoted join condition
... force later arguments to be by name
jointype type of join (‘INNER’, ‘LEFT’, ‘RIGHT’, ‘FULL’).
suffix character length 2, suffices to disambiguate columns.
env environment to look for values in.

Value

theta_join node.
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d1 <- rq_copy_to(my_db, 'd1',
                   data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
                   data.frame(AUC2 = 0.4, R2 = 0.3))
  optree <- theta_join(d1, d2, AUC >= AUC2)
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

---

**theta_join_se**  
*Make a theta_join node.*

**Description**

Theta join is a join on an arbitrary predicate.

**Usage**

```r
theta_join_se(
  a,
  b,
  expr,
  ...
, jointype = "INNER",
  suffix = c("_a", "_b"),
  env = parent.frame()
)
```

**Arguments**

- `a` source to select from.
- `b` source to select from.
- `expr` quoted join conditions
- `...` force later arguments to be by name
- `jointype` type of join ("INNER", "LEFT", "RIGHT", "FULL").
- `suffix` character length 2, suffices to disambiguate columns.
- `env` environment to look for values in.
Value

theta_join node.

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d1 <- rq_copy_to(my_db, 'd1',
    data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
    data.frame(AUC2 = 0.4, R2 = 0.3))
  optree <- theta_join_se(d1, d2, "AUC >= AUC2")
  cat(format(optree))
  sql <- to_sql(optree, my_db)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
```

Description

Depends on igraph package. Please see vignette('DependencySorting', package = 'rquery') and vignette('joinController', package = 'rquery') for more details.

Usage

topo_sort_tables(columnJoinPlan, leftTableName, ...)

Arguments

columnJoinPlan join plan
leftTableName which table is left
... force later arguments to bind by name

Value

list with dependencyGraph and sorted columnJoinPlan
Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE) &&
    requireNamespace('igraph', quietly = TRUE)) {
  # note: employeeanddate is likely built as a cross-product
  # join of an employee table and set of dates of interest
  # before getting to the join controller step. We call
  # such a table "row control" or "experimental design."
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  RSQLite::initExtension(my_db)
  tDesc <- example_employee_date(my_db)
  columnJoinPlan <- build_join_plan(tDesc, check= FALSE)
  # unify keys
  columnJoinPlan$resultColumn[columnJoinPlan$resultColumn=='id'] <- 'eid'
  # look at plan defects
  print(paste('problems:',
    inspect_join_plan(tDesc, columnJoinPlan)))
  # fix plan
  sorted <- topo_sort_tables(columnJoinPlan, 'employeeanddate')
  print(paste('problems:',
    inspect_join_plan(tDesc, sorted$columnJoinPlan)))
  print(plot(sorted$dependencyGraph))
  DBI::dbDisconnect(my_db)
  my_db <- NULL
}
```

to_sql

`to_sql` Return SQL implementation of operation tree.

Description

Add to last argument and pass all others through.

Usage

```
to_sql(
  x,
  db,
  ...
) limit = NULL,
source_limit = NULL,
indent_level = 0,
 tnum = mk_tmp_name_source("tsql"),
append_cr = TRUE,
using = NULL
) ```
Arguments

x rquery operation tree.
db DBI database handle or rquery_db_info object.
... generic additional arguments (not used).
limit numeric if not NULL limit result to this many rows.
source_limit numeric if not NULL limit sources to this many rows.
indent_level level to indent.
tnum temp sub-query name generator.
append_cr logical if TRUE end with CR.
using character, if not NULL set of columns used from above.

Value

SQL command

See Also
db_td, materialize, execute, rq_copy_to, mk_td

Examples

```r
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
  d1 <- rq_copy_to(my_db, 'd1',
      data.frame(AUC = 0.6, R2 = 0.2))
  d2 <- rq_copy_to(my_db, 'd2',
      data.frame(AUC = 0.6, D = 0.3))
  optree <- natural_join(d1, d2, by = "AUC")
  cat(format(optree))
  print(to_sql(optree, my_db))
  DBI::dbDisconnect(my_db)
}
```

Description

Convert an rquery op diagram to a simple representation, appropriate for conversion to YAML.
Usage

to_transport_representation(ops, ..., convert_named_vectors_to_lists = TRUE)

Arguments

ops rquery operator dag

... not used, force later arguments to be by name

custom_named_vectors_to_lists

logical, if TRUE convert named vectors to lists

Value

representation structure

unionall

Make an unionall node (not a relational operation).

Description

Concatenate tables by rows.

Usage

unionall(sources, env = parent.frame())

Arguments

sources list of relop trees or list of data.frames

env environment to look to.

Value

order_by node or altered data.frame.

Examples

if (requireNamespace("DBI", quietly = TRUE) & requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), "memory:"
  d <- rq_copy_to(my_db, 'd',
                 data.frame(AUC = 0.6, R2 = 0.2))
  optree <- unionall(list(d, d, d))
  cat(format(optree))
  sql <- to_sql(optree, my_db, limit = 2)
  cat(sql)
  print(DBI::dbGetQuery(my_db, sql))
  DBI::dbDisconnect(my_db)
}
Wrap a data frame for later execution.

Description

Create a table description that includes the actual data. Prevents wastefull table copies in immediate pipelines. Used with ex().

Usage

wrap(d, ..., table_name = NULL, env = parent.frame())

Arguments

d data.frame
... not used, force later argument to be referred by name
table_name character, name of table
env environment to work in.

Value

a table description, with data attached

Examples

if(requireNamespace('rqdatatable')) {
  d <- data.frame(x = 1:3, y = 4:6)
  d %.>%
    wrap(.) %.>%
    extend(., z := x + y) %.>%
    ex(.)
}

Index

actualize_join_plan, 5, 13, 22, 31, 36
affine_transform, 6
aggregate_nse (project), 60
aggregate_se (project_se), 61
apply_right.rellop, 7
apply_right_S4, ANY, rquery_db_info-method, 9
apply_right_S4, data.frame, relop_arrow-method, 10
apply_right_S4, relop_arrow, relop_arrow-method, 10
arrow, 11
assign_slice, 12
build_join_plan, 5, 13, 22, 31, 36
column_names, 15
columns_used, 14
commencify, 15
complete_design, 17
convert_yaml_to_pipeline, 18
count_null_cols, 19, 44, 53, 89
db_td, 20, 25, 40, 45, 48, 75–79, 97
dbi_table (db_td), 20
describe_tables, 5, 13, 22, 31, 36
drop_columns, 23
ex, 24
execute, 15, 16, 21, 24, 45, 48, 75, 97
expand_grid, 26
extend, 27
extend_nse (extend), 27
extend_se, 28, 32, 34
format_node, 30
getDBOption, 30
graph_join_plan, 5, 13, 22, 31, 36
if_else_block, 32, 35
if_else_op, 33, 34
inspect_join_plan, 5, 13, 35
key_inspector_all_cols, 37
key_inspector_postgresql, 38
key_inspector_sqlite, 38
local_td, 21, 39, 48
lookup_by_column, 40
make_assignments, 41
map_column_values, 42
mark_null_cols, 19, 43, 53, 89
materialize, 21, 25, 44, 48, 75, 97
materialize_node, 46
mk_td, 21, 25, 40, 45, 47, 75, 97
natural_join, 48
non_sql_node, 46, 50, 64, 82, 83
normalize_cols, 51
null_replace, 19, 44, 52, 89
op_diagram, 53
order_expr, 55
order_expr_nse (order_expr), 55
order_expr_se, 56
order_rows, 57
orderby, 54
pick_top_k, 58
pre_sql_sub_expr, 59
project, 60
project_nse (project), 60
project_se, 61
quantile_cols, 62, 64
quantile_node, 51, 63, 63, 83
quote_identifier, 64
quote_literal, 65
quote_string, 65
quote_table_name, 66
rename_columns, 66
rlook(rstr), 80
row_counts, 67
rq_colnames, 70
rq_coltypes, 71
rq_connection_advice, 72, 74
rq_connection_name, 73
rq_connection_tests, 73, 74
rq_copy_to, 21, 25, 45, 48, 75, 97
rq_execute, 76
rq_get_query, 77
rq_head, 77
rq_nrow, 78
rq_remove_table, 78
rq_table_exists, 79
rquery (rquery-package), 4
rquery-package, 4
rquery_apply_to_data_frame, 8, 68
rquery_db_info, 69
rquery_default_db_info, 70
rstr, 80
rsummary, 63, 64, 81
rsummary_node, 46, 51, 82
select_columns, 83
select_rows, 84
select_rows_nse (select_rows), 84
select_rows_se, 85
set_indicator, 87
setDBOpt, 86, 86
setDBOption, 86
sql_expr_set, 88
sql_node, 90
table_source (mk_td), 47
tables_used, 92
theta_join, 93
theta_join_nse (theta_join), 93
theta_join_se, 94
to_sql, 21, 25, 45, 48, 75, 96
to_transport_representation, 97
topo_sort_tables, 95
unionall, 98
wrap, 99