

# Package ‘replyr’

July 24, 2019

**Type** Package

**Title** Patches to Use 'dplyr' on Remote Data Sources

**Version** 1.0.4

**Date** 2019-07-24

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

**URL** <https://github.com/WinVector/replyr/>,  
<https://winvector.github.io/replyr/>

**BugReports** <https://github.com/WinVector/replyr/issues>

**Description** Patches to use 'dplyr' on remote data sources ('SQL' databases, 'Spark' 2.0.0 and above) in a reliable ``generic" fashion (generic meaning user code works similarly on all such sources, without needing per-source adaption). Due to the fluctuating nature of 'dplyr'/'dbplyr'/'rlang' APIs this package is going into maintenance mode. Most of the 'replyr' functions are already done better by one of the non-monolithic replacement packages: 'wrapr', 'seplyr', 'rquery', or 'cdata'.

**License** GPL-2 | GPL-3

**LazyData** TRUE

**Depends** R (>= 3.4.0), wrapr (>= 1.8.8)

**Imports** dplyr (>= 0.7.0), rlang (>= 0.2.0), dbplyr, DBI

**RoxygenNote** 6.1.1

**Suggests** testthat, knitr, rmarkdown, sparklyr, igraph, DiagrammeR,  
htmlwidgets, webshot, RSQLite

**VignetteBuilder** knitr

**ByteCompile** true

**NeedsCompilation** no

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

**Repository** CRAN

**Date/Publication** 2019-07-24 19:10:02 UTC

**R topics documented:**

addConstantColumn . . . . .	2
buildJoinPlan . . . . .	3
dplyr_src_to_db_handle . . . . .	4
example_employeeAndDate . . . . .	4
executeLeftJoinPlan . . . . .	5
expandColumn . . . . .	6
gapply . . . . .	7
inspectDescrAndJoinPlan . . . . .	9
keysAreUnique . . . . .	11
key_inspector_all_cols . . . . .	11
key_inspector_postgresql . . . . .	12
key_inspector_sqlite . . . . .	13
makeJoinDiagramSpec . . . . .	13
replyr . . . . .	14
replyr_add_ids . . . . .	15
replyr_apply_f_mapped . . . . .	16
replyr_arrange . . . . .	17
replyr_bind_rows . . . . .	18
replyr_check_ranks . . . . .	19
replyr_coalesce . . . . .	20
replyr_colClasses . . . . .	21
replyr_copy_from . . . . .	22
replyr_copy_to . . . . .	22
replyr_dim . . . . .	23
replyr_filter . . . . .	24
replyr_get_src . . . . .	25
replyr_group_by . . . . .	25
replyr_hasrows . . . . .	26
replyr_has_table . . . . .	27
replyr_inTest . . . . .	27
replyr_is_local_data . . . . .	28
replyr_is_MySQL_data . . . . .	29
replyr_is_Spark_data . . . . .	29
replyr_list_tables . . . . .	30
replyr_mapRestrictCols . . . . .	31
replyr_ncol . . . . .	32
replyr_nrow . . . . .	33
replyr_quantile . . . . .	33
replyr_quantilec . . . . .	34
replyr_rename . . . . .	35
replyr_reverseMap . . . . .	35
replyr_select . . . . .	36
replyr_split . . . . .	37
replyr_summary . . . . .	38
replyr_testCols . . . . .	39
replyr_union_all . . . . .	40

`addConstantColumn` 3

<code>replyr_uniqueValues</code> . . . . .	41
<code>tableDescription</code> . . . . .	41
<code>topoSortTables</code> . . . . .	42
<code>%land%</code> . . . . .	43

---

`addConstantColumn` *Add constant to a table.*

---

### Description

Work around different treatment of character types across remote data sources when adding a constant column to a table. Deals with issues such as Postgresql requiring a character-cast and MySQL not allowing such.

### Usage

```
addConstantColumn(d, colName, val, ...,
  tempNameGenerator = mk_tmp_name_source("replyr_addConstantColumn"))
```

### Arguments

`d` data.frame like object to add column to.

`colName` character, name of column to add.

`val` scalar, value to add.

`...` force later arguments to be bound by name.

`tempNameGenerator` temp name generator produced by `wrpr::mk_tmp_name_source`, used to record `dplyr::compute()` effects.

### Value

table with new column added.

### Examples

```
d <- data.frame(x= c(1:3))
addConstantColumn(d, 'newCol', 'newVal')
```

---

buildJoinPlan	<i>Build a join plan</i>
---------------	--------------------------

---

## Description

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

## Usage

```
buildJoinPlan(tDesc, ..., check = TRUE)
```

## Arguments

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

## Value

detailed column join plan (appropriate for editing)

## See Also

`tableDescription`, `inspectDescrAndJoinPlan`, `makeJoinDiagramSpec`, `executeLeftJoinPlan`

## Examples

```
d <- data.frame(id=1:3, weight= c(200, 140, 98))
tDesc <- rbind(tableDescription('d1', d),
              tableDescription('d2', d))
tDesc$keys[[1]] <- list(PrimaryKey= 'id')
tDesc$keys[[2]] <- list(PrimaryKey= 'id')
buildJoinPlan(tDesc)
```

---

`dplyr_src_to_db_handle`

*Obsolete with dplyr 0.7.0 and forward*

---

**Description**

Obsolete with dplyr 0.7.0 and forward

**Usage**

`dplyr_src_to_db_handle(dplyr_src)`

**Arguments**

`dplyr_src`      remote data handle

**Value**

`dplyr_src`

---

`example_employeeAndDate`

*build some example tables*

---

**Description**

build some example tables

**Usage**

`example_employeeAndDate(con)`

**Arguments**

`con`              db connection

**Value**

example tables

---

```
executeLeftJoinPlan
```

*Execute an ordered sequence of left joins.*

---

## Description

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

## Usage

```
executeLeftJoinPlan(tDesc, columnJoinPlan, ..., checkColumns = FALSE,
  computeFn = function(x, name) { dplyr::compute(x, name = name) },
  eagerCompute = TRUE, checkColClasses = FALSE, verbose = FALSE,
  dryRun = FALSE,
  tempNameGenerator = mk_tmp_name_source("executeLeftJoinPlan"))
```

## Arguments

<code>tDesc</code>	description of tables, either a <code>data.frame</code> from <code>tableDescription</code> , or a list mapping from names to handles/frames. Only used to map table names to data.
<code>columnJoinPlan</code>	columns to join, from <code>buildJoinPlan</code> (and likely altered by user). Note: no column names must intersect with names of the form <code>table_CLEANEDTABNAME_present</code> .
<code>...</code>	force later arguments to bind by name.
<code>checkColumns</code>	logical if TRUE confirm column names before starting joins.
<code>computeFn</code>	function to call to try and materialize intermediate results.
<code>eagerCompute</code>	logical if TRUE materialize intermediate results with <code>computeFn</code> .
<code>checkColClasses</code>	logical if true check for exact class name matches
<code>verbose</code>	logical if TRUE print more.
<code>dryRun</code>	logical if TRUE do not perform joins, only print steps.
<code>tempNameGenerator</code>	temp name generator produced by <code>wrpr::mk_tmp_name_source</code> , used to record <code>dplyr::compute()</code> effects.

## Value

joined table

## See Also

`tableDescription`, `buildJoinPlan`, `inspectDescrAndJoinPlan`, `makeJoinDiagramSpec`

**Examples**

```

# example data
meas1 <- data.frame(id= c(1,2),
                    weight= c(200, 120),
                    height= c(60, 14))
meas2 <- data.frame(pid= c(2,3),
                    weight= c(105, 110),
                    width= 1)

# get the initial description of table defs
tDesc <- rbind(tableDescription('meas1', meas1),
              tableDescription('meas2', 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 <- buildJoinPlan(tDesc)
# decide we don't want the width column
columnJoinPlan$want[columnJoinPlan$resultColumn=='width'] <- FALSE
# double check our plan
if(!is.null(inspectDescrAndJoinPlan(tDesc, columnJoinPlan,
                                   checkColClasses= TRUE))) {
  stop("bad join plan")
}

# execute the left joins
executeLeftJoinPlan(tDesc, columnJoinPlan,
                   checkColClasses= TRUE,
                   verbose= TRUE)

# also good
executeLeftJoinPlan(list('meas1'=meas1, 'meas2'=meas2),
                   columnJoinPlan,
                   checkColClasses= TRUE,
                   verbose= TRUE)

```

---

expandColumn

*Expand a column of vectors into one row per value of each vector.*


---

**Description**

Similar to `tidyr::unnest` but lands rowids and value ids, and can work on remote data sources. Fairly expensive per-row operation, not suitable for big data.

**Usage**

```

expandColumn(data, colName, ..., rowidSource = NULL, rowidDest = NULL,
            idxDest = NULL,
            tempNameGenerator = mk_tmp_name_source("replyr_expandColumn"))

```

**Arguments**

<code>data</code>	data.frame to work with.
<code>colName</code>	character name of column to expand.
<code>...</code>	force later arguments to be bound by name
<code>rowidSource</code>	optional character name of column to take row indices from (rowidDest must be NULL to use this).
<code>rowidDest</code>	optional character name of column to write row indices to (must not be an existing column name, rowidSource must be NULL to use this).
<code>idxDest</code>	optional character name of column to write value indices to (must not be an existing column name).
<code>tempNameGenerator</code>	temp name generator produced by <code>wrapp::mk_tmp_name_source</code> , used to record <code>dplyr::compute()</code> effects.

**Value**

expanded data frame where each value of `colName` column is in a new row.

**Examples**

```
d <- data.frame(name= c('a','b'))
d$value <- list(c('x','y'),'z')
expandColumn(d, 'value',
             rowidDest= 'origRowId',
             idxDest= 'valueIndex')
```

---

gapply

*grouped ordered apply*


---

**Description**

Partitions from by values in grouping column, applies a generic transform to each group and then binds the groups back together. Only advised for a moderate number of groups and better if grouping column is an index. This is powerful enough to implement "The Split-Apply-Combine Strategy for Data Analysis" <https://www.jstatsoft.org/article/view/v040i01>

**Usage**

```
gapply(df, gcolumn, f, ..., ocolumn = NULL, decreasing = FALSE,
       partitionMethod = "split", bindrows = TRUE, maxgroups = 100,
       eagerCompute = FALSE, restoreGroup = FALSE,
       tempNameGenerator = mk_tmp_name_source("replyr_gapply"))
```

**Arguments**

<code>df</code>	remote dplyr data item
<code>gcolumn</code>	grouping column
<code>f</code>	transform function or pipeline
<code>...</code>	force later values to be bound by name
<code>ocolumn</code>	ordering column (optional)
<code>decreasing</code>	logical, if TRUE sort in decreasing order by <code>ocolumn</code>
<code>partitionMethod</code>	method to partition the data, one of 'group_by' (depends on <code>f</code> being dplyr compatible), 'split' (only works over local data frames), or 'extract'
<code>bindrows</code>	logical, if TRUE bind the rows back into a data item, else return split list
<code>maxgroups</code>	maximum number of groups to work over (intentionally not enforced if <code>partitionMethod</code> =='group')
<code>eagerCompute</code>	logical, if TRUE call <code>compute</code> on split results
<code>restoreGroup</code>	logical, if TRUE restore group column after apply when <code>partitionMethod</code> %in% c('extract', 'split')
<code>tempNameGenerator</code>	temp name generator produced by <code>wrapr::mk_tmp_name_source</code> , used to record <code>dplyr::compute()</code> effects.

**Details**

Note this is a fairly expensive operator, so it only makes sense to use in situations where `f` itself is fairly complicated and/or expensive.

**Value**

transformed frame

**Examples**

```
d <- data.frame(
  group = c(1, 1, 2, 2, 2),
  order = c(.1, .2, .3, .4, .5),
  values = c(10, 20, 2, 4, 8)
)

# User supplied window functions. They depend on known column names and
# the data back-end matching function names (as cumsum).
cumulative_sum <- function(d) {
  dplyr::mutate(d, cv = cumsum(values))
}
rank_in_group <- function(d) {
  d %>%
    dplyr::mutate(., constcol = 1) %>%
    dplyr::mutate(., rank = cumsum(constcol)) %>%
    dplyr::select(., -constcol)
}
```

```

}

for (partitionMethod in c('group_by', 'split', 'extract')) {
  print(partitionMethod)
  print('cumulative sum example')
  print(
    gapply(
      d,
      'group',
      cumulative_sum,
      ocolumn = 'order',
      partitionMethod = partitionMethod
    )
  )
  print('ranking example')
  print(
    gapply(
      d,
      'group',
      rank_in_group,
      ocolumn = 'order',
      partitionMethod = partitionMethod
    )
  )
  print('ranking example (decreasing)')
  print(
    gapply(
      d,
      'group',
      rank_in_group,
      ocolumn = 'order',
      decreasing = TRUE,
      partitionMethod = partitionMethod
    )
  )
}

```

---

```
inspectDescrAndJoinPlan
```

*check that a join plan is consistent with table descriptions*

---

### Description

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

### Usage

```
inspectDescrAndJoinPlan(tDesc, columnJoinPlan, ...,
  checkColClasses = FALSE)
```

**Arguments**

tDesc description of tables, from tableDescription (and likely altered by user).  
columnJoinPlan columns to join, from buildJoinPlan (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**

tableDescription, buildJoinPlan, makeJoinDiagramSpec, executeLeftJoinPlan

**Examples**

```
# example data
d1 <- data.frame(id= 1:3,
                 weight= c(200, 140, 98),
                 height= c(60, 24, 12))
d2 <- data.frame(pid= 2:3,
                 weight= c(130, 110),
                 width= 1)

# get the initial description of table defs
tDesc <- rbind(tableDescription('d1', d1),
              tableDescription('d2', 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 <- buildJoinPlan(tDesc)
# confirm the plan
inspectDescrAndJoinPlan(tDesc, columnJoinPlan,
                       checkColClasses= TRUE)

# damage the plan
columnJoinPlan$sourceColumn[columnJoinPlan$sourceColumn=='width'] <- 'wd'
# find a problem
inspectDescrAndJoinPlan(tDesc, columnJoinPlan,
                       checkColClasses= TRUE)
```

keysAreUnique      *Check uniqueness of rows with respect to keys.*

---

**Description**

Can be an expensive operation.

**Usage**

```
keysAreUnique(tDesc)
```

**Arguments**

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

**Value**

logical TRUE if keys are unique

**See Also**

`tableDescription`

**Examples**

```
d <- data.frame(x=c(1,1,2,2,3,3), y=c(1,2,1,2,1,2))
tDesc1 <- tableDescription('d1', d)
tDesc2 <- tableDescription('d2', d)
tDesc <- rbind(tDesc1, tDesc2)
tDesc$keys[[2]] <- c(x='x')
keysAreUnique(tDesc)
```

---

key\_inspector\_all\_cols  
*Return all columns as guess at preferred primary keys.*

---

**Description**

Return all columns as guess at preferred primary keys.

**Usage**

```
key_inspector_all_cols(handle)
```

**Arguments**

`handle`            `data handle`

**Value**

map of keys to keys

**See Also**

`tableDescription`

**Examples**

```
d <- data.frame(x=1:3, y=NA)
key_inspector_all_cols(d)
```

---

`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**

```
key_inspector_postgresql(handle)
```

**Arguments**

`handle`            `data handle`

**Value**

map of keys to keys

**See Also**

`tableDescription`

---

```
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

```
key_inspector_sqlite(handle)
```

### Arguments

handle            data handle

### Value

map of keys to keys

### See Also

tableDescription

---

```
makeJoinDiagramSpec
```

*Build a drawable specification of the join diagram*

---

### Description

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

### Usage

```
makeJoinDiagramSpec(columnJoinPlan, ..., groupByKey = TRUE,
  graphOpts = NULL)
```

### Arguments

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

**Value**

grViz diagram spec

**See Also**

tableDescription, buildJoinPlan, executeLeftJoinPlan

**Examples**

```
if (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_employeeAndDate(my_db)
  # fix order by hand, please see replyr::topoSortTables for
  # how to automate this.
  ord <- match(c('employeeanddate', 'orgtable', 'activity', 'revenue'),
              tDesc$tableName)
  tDesc <- tDesc[ord, , drop=FALSE]
  columnJoinPlan <- buildJoinPlan(tDesc, check= FALSE)
  # unify keys
  columnJoinPlan$resultColumn[columnJoinPlan$resultColumn=='id'] <- 'eid'
  # look at plan defects
  print(paste('problems:',
              inspectDescrAndJoinPlan(tDesc, columnJoinPlan)))
  diagramSpec <- makeJoinDiagramSpec(columnJoinPlan)
  # to render as JavaScript:
  #   DiagrammerR::grViz(diagramSpec)
  DBI::dbDisconnect(my_db)
  my_db <- NULL
}
```

**Description**

Methods to reliably use `dplyr` on remote data sources in R (SQL databases, Spark 2.0.0 and above) in a generic fashion.

## Details

replyr is going into maintenance mode. It has been hard to track shifting dplyr/dbplyr/rlang APIs and data structures post dplyr 0.5. Most of what it does is now done better in one of the newer non-monolithic packages:

- Programming and meta-programming tools: wrapr <https://CRAN.R-project.org/package=wrapr>.
- Adapting dplyr to standard evaluation interfaces: seplyr <https://CRAN.R-project.org/package=seplyr>.
- Big data data manipulation: rquery <https://CRAN.R-project.org/package=rquery> and cdata <https://CRAN.R-project.org/package=cdata>.

replyr helps with the following:

- Summarizing remote data (via replyr\_summarize).
- Facilitating writing "source generic" code that works similarly on multiple 'dplyr' data sources.
- Providing big data versions of functions for splitting data, binding rows, pivoting, adding row-ids, ranking, and completing experimental designs.
- Packaging common data manipulation tasks into operators such as the gapply function.
- Providing support code for common SparklyR tasks, such as tracking temporary handle IDs.

replyr is in maintenance mode. Better version of the functionality have been ported to the following packages: wrapr, cdata, rquery, and seplyr.

To learn more about replyr, please start with the vignette: `vignette('replyr', 'replyr')`

---

replyr\_add\_ids      *Add unique ids to rows. Note: re-arranges rows in many cases.*

---

## Description

Add unique ids to rows. Note: re-arranges rows in many cases.

## Usage

```
replyr_add_ids(df, idColName, env = parent.frame(),
  local_short_cut = TRUE)
```

## Arguments

df	data.frame object to work with
idColName	name of column to add
env	environment to evaluate in (not used).
local_short_cut	logical, if TRUE use base R on local data.

**Examples**

```
replyr_add_ids(data.frame(x=c('a','b')), 'id', local_short_cut = FALSE)
```

---

```
replyr_apply_f_mapped
```

*Apply a function to a re-mapped data frame.*

---

**Description**

Apply a function to a re-mapped data frame.

**Usage**

```
replyr_apply_f_mapped(d, f, nmap, ..., restrictMapIn = FALSE,
  rmap = replyr::replyr_reverseMap(nmap), restrictMapOut = FALSE)
```

**Arguments**

d	data.frame to work on
f	function to apply.
nmap	named list mapping with keys specifying new column names, and values as original column names.
...	force later arguments to bind by name
restrictMapIn	logical if TRUE restrict columns when mapping in.
rmap	reverse map (for after f is applied).
restrictMapOut	logical if TRUE restrict columns when mapping out.

**See Also**

```
let, replyr_reverseMap, replyr_mapRestrictCols
```

**Examples**

```
# an external function with hard-coded column names
DecreaseRankColumnByOne <- function(d) {
  d$RankColumn <- d$RankColumn - 1
  d
}

# our example data, with different column names
d <- data.frame(Sepal_Length=c(5.8,5.7),
```

```
      Sepal_Width=c(4.0,4.4),
      Species='setosa',rank=c(1,2))
print(d)

# map our data to expected column names so we can use function
nmap <- c(GroupColumn='Species',
          ValueColumn='Sepal_Length',
          RankColumn='rank')
print(nmap)

dF <- replyr_apply_f_mapped(d, DecreaseRankColumnByOne, nmap)
print(dF)
```

---

replyr\_arrange      *arrange by a single column*

---

### Description

arrange by a single column

### Usage

```
replyr_arrange(.data, colname, descending = FALSE)
```

### Arguments

.data	data object to work on
colname	character column name
descending	logical if true sort descending (else sort ascending)

### Examples

```
d <- data.frame(Sepal_Length= c(5.8,5.7),
               Sepal_Width= c(4.0,4.4))
replyr_arrange(d, 'Sepal_Length', descending= TRUE)
```

---

```
replyr_bind_rows Bind a list of items by rows (can't use dplyr::bind_rows or
dplyr::combine on remote sources). Columns are intersected.
```

---

### Description

Bind a list of items by rows (can't use `dplyr::bind_rows` or `dplyr::combine` on remote sources). Columns are intersected.

### Usage

```
replyr_bind_rows(lst, ..., useDplyrLocal = TRUE, useSparkRbind = TRUE,
  useUnionALL = TRUE,
  tempNameGenerator = mk_tmp_name_source("replyr_bind_rows"))
```

### Arguments

`lst` list of items to combine, must be all in same dplyr data service

`...` force other arguments to be used by name

`useDplyrLocal` logical if TRUE use dplyr for local data.

`useSparkRbind` logical if TRUE try to use rbind on Sparklyr data

`useUnionALL` logical if TRUE try to use union all binding

`tempNameGenerator` temp name generator produced by `wrpr::mk_tmp_name_source`, used to record `dplyr::compute()` effects.

### Value

single data item

### Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  # my_db <- sparklyr::spark_connect(master = "local")
  d <- replyr_copy_to(my_db, data.frame(x = 1:2), 'd',
    temporary = TRUE)
  # dplyr::bind_rows(list(d, d))
  # # Argument 1 must be a data frame or a named atomic vector,
  # # not a tbl_dbi/tbl_sql/tbl_lazy/tbl
  print(replyr_bind_rows(list(d, d)))
  DBI::dbDisconnect(my_db)
}
```

---

replyr\_check\_ranks *confirm data has good ranked groups*

---

### Description

confirm data has good ranked groups

### Usage

```
replyr_check_ranks(x, GroupColumnName, ValueColumnName, RankColumnName,
  ..., decreasing = FALSE,
  tempNameGenerator = mk_tmp_name_source("replyr_check_ranks"))
```

### Arguments

`x` data item to work with

`GroupColumnName` column to group by

`ValueColumnName` column determining order

`RankColumnName` column having proposed rank (function of order)

`...` force later arguments to bind by name

`decreasing` if true make order decreasing instead of increasing.

`tempNameGenerator` temp name generator produced by `wrapr::mk_tmp_name_source`, used to record `dplyr::compute()` effects.

### Value

summary of quality of ranking.

### Examples

```
d <- data.frame(Sepal_Length=c(5.8,5.7), Sepal_Width=c(4.0,4.4),
  Species='setosa', rank=c(1,2))
replyr_check_ranks(d, 'Species', 'Sepal_Length', 'rank', decreasing=TRUE)
```

---

replyr\_coalesce     *Augment a data frame by adding additional rows.*

---

## Description

Note: do not count on order of resulting data. Also only added rows are altered by the fill instructions.

## Usage

```
replyr_coalesce(data, support, ..., fills = NULL, newRowColumn = NULL,
  copy = TRUE,
  tempNameGenerator = mk_tmp_name_source("replyr_coalesce"))
```

## Arguments

data	data.frame data to augment
support	data.frame rows of unique key-values into data
...	not used, force later arguments to bind by name
fills	list default values to fill in columns
newRowColumn	character if not null name to use for new row indicator
copy	logical if TRUE copy support to data's source
tempNameGenerator	temp name generator produced by wrapr::mk_tmp_name_source, used to record dplyr::compute() effects.

## Value

augmented data

## Examples

```
# single column key example
data <- data.frame(year = c(2005,2007,2010),
  count = c(6,1,NA),
  name = c('a','b','c'),
  stringsAsFactors = FALSE)
support <- data.frame(year=2005:2010)
filled <- replyr_coalesce(data, support,
  fills=list(count=0))
filled <- filled[order(filled$year), ]
filled

# complex key example
data <- data.frame(year = c(2005,2007,2010),
```

```
count = c(6,1,NA),
name = c('a','b','c'),
stringsAsFactors = FALSE)
support <- expand.grid(year=2005:2010,
name= c('a','b','c','d'),
stringsAsFactors = FALSE)
filled <- replyr_coalesce(data, support,
fills=list(count=0))
filled <- filled[order(filled$year, filled$name), ]
filled
```

---

replyr\_colClasses *Get column classes.*

---

### Description

Get column classes.

### Usage

```
replyr_colClasses(x)
```

### Arguments

x                   tbl or item that can be coerced into such.

### Value

list of column classes.

### Examples

```
d <- data.frame(x=c(1,2))
replyr_colClasses(d)
```

---

replayr\_copy\_from *Bring remote data back as a local data frame tbl.*

---

### Description

Bring remote data back as a local data frame tbl.

### Usage

```
replayr_copy_from(d, maxrow = 1e+06)
```

### Arguments

d	remote dplyr data item
maxrow	max rows to allow (stop otherwise, set to NULL to allow any size).

### Value

local tbl.

### Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {  
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")  
  RSQLite::initExtension(my_db)  
  d <- replayr_copy_to(my_db, data.frame(x=c(1,2)), 'd')  
  d2 <- replayr_copy_from(d)  
  print(d2)  
  DBI::dbDisconnect(my_db)  
}
```

---

replayr\_copy\_to *Copy data to remote service.*

---

### Description

Copy data to remote service.

### Usage

```
replayr_copy_to(dest, df, name = paste(deparse(substitute(df)), collapse =  
  " "), ..., rowNumberColumn = NULL, temporary = FALSE,  
  overwrite = TRUE, maxrow = 1e+06)
```

**Arguments**

dest	remote data source
df	local data frame
name	name for new remote table
...	force later values to be bound by name
rowNumberColumn	if not null name to add row numbers to
temporary	logical, if TRUE try to create a temporary table
overwrite	logical, if TRUE try to overwrite
maxrow	max rows to allow in a remote to remote copy.

**Value**

remote handle

**Examples**

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  RSQLite::initExtension(my_db)
  d <- replyr_copy_to(my_db, data.frame(x=c(1,2)), 'd')
  print(d)
  DBI::dbDisconnect(my_db)
}
```

---

replyr_dim	<i>Compute dimensions of a data.frame (work around <a href="https://github.com/rstudio/sparklyr/issues/976">https://github.com/rstudio/sparklyr/issues/976</a> ).</i>
------------	---

---

**Description**

Compute dimensions of a data.frame (work around <https://github.com/rstudio/sparklyr/issues/976> ).

**Usage**

```
replyr_dim(x)
```

**Arguments**

x tbl or item that can be coerced into such.

**Value**

dimensions (including rows)

**Examples**

```
d <- data.frame(x=c(1,2))
replyr_dim(d)
```

---

<code>replyr_filter</code>	<i>Filter a tbl on a column having values in a given set.</i>
----------------------------	---

---

**Description**

Filter a tbl on a column having values in a given set.

**Usage**

```
replyr_filter(x, cname, values, ..., verbose = TRUE,
             tempNameGenerator = mk_tmp_name_source("replyr_filter"))
```

**Arguments**

<code>x</code>	tbl or item that can be coerced into such.
<code>cname</code>	name of the column to test values of.
<code>values</code>	set of values to check set membership of.
<code>...</code>	force later arguments to bind by name.
<code>verbose</code>	logical if TRUE echo warnings
<code>tempNameGenerator</code>	temp name generator produced by <code>wrpr::mk_tmp_name_source</code> , used to record <code>dplyr::compute()</code> effects.

**Value**

new tbl with only rows where `cname` value is in `values` set.

**Examples**

```
values <- c('a', 'c')
d <- data.frame(x=c('a', 'a', 'b', 'b', 'c', 'c'), y=1:6,
               stringsAsFactors=FALSE)
replyr_filter(d, 'x', values)
```

---

`replyr_get_src`      *Get the "remote data source" where a data.frame like object lives.*

---

**Description**

Get the "remote data source" where a data.frame like object lives.

**Usage**

```
replyr_get_src(df)
```

**Arguments**

`df`                      data.frame style object

**Value**

source (string if data.frame, tbl, or data.table, NULL if unknown, remote source otherwise)

**Examples**

```
replyr_get_src(data.frame(x=1:2))
```

---

`replyr_group_by`      *group\_by columns*

---

**Description**

See also: <https://gist.github.com/skranz/9681509>

**Usage**

```
replyr_group_by(.data, colnames)
```

**Arguments**

`.data`                      data.frame  
`colnames`                  character vector of column names to group by.

**Value**

`.data` grouped by columns named in `colnames`

**See Also**

group\_by, group\_by\_at

**Examples**

```
d <- data.frame(Sepal_Length= c(5.8,5.7),
                Sepal_Width= c(4.0,4.4),
                Species= 'setosa')
replyr_group_by(d, 'Species')
```

---

replyr\_hasrows      *Check if a table has rows.*

---

**Description**

Check if a table has rows.

**Usage**

```
replyr_hasrows(d)
```

**Arguments**

d                      tbl or item that can be coerced into such.

**Value**

number of rows

**Examples**

```
d <- data.frame(x=c(1,2))
replyr_hasrows(d)
```

---

replayr\_has\_table    *check for a table*

---

### Description

Work around connection v.s. handle issues <https://github.com/tidyverse/dplyr/issues/2849>

### Usage

```
replayr_has_table(con, name)
```

### Arguments

con	connection
name	character name to check for

### Value

TRUE if table present

### Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  RSQLite::initExtension(my_db)
  d <- replayr_copy_to(my_db, data.frame(x=c(1,2)), 'd')
  print(d)
  print(replayr_has_table(my_db, 'd'))
  DBI::dbDisconnect(my_db)
}
```

---

replayr\_inTest    *Product a column noting if another columns values are in a given set.*

---

### Description

Product a column noting if another columns values are in a given set.

### Usage

```
replayr_inTest(x, cname, values, nname, ...,
  tempNameGenerator = mk_tmp_name_source("replayr_inTest"),
  verbose = TRUE)
```

**Arguments**

x	tbl or item that can be coerced into such.
cname	name of the column to test values of.
values	set of values to check set membership of.
nname	name for new column
...	force later parameters to bind by name
tempNameGenerator	temp name generator produced by wrapr::mk_tmp_name_source, used to record dplyr::compute() effects.
verbose	logical if TRUE echo warnings

**Value**

table with membership indications.

**Examples**

```
values <- c('a', 'c')
d <- data.frame(x=c('a', 'a', 'b', NA, 'c', 'c'), y=1:6,
               stringsAsFactors=FALSE)
replayr_inTest(d, 'x', values, 'match')
```

---

```
replayr_is_local_data
Test if data is local.
```

---

**Description**

Test if data is local.

**Usage**

```
replayr_is_local_data(d)
```

**Arguments**

d	data frame
---	------------

**Value**

TRUE if local data (data.frame, tbl/tibble)

**Examples**

```
replayr_is_local_data(data.frame(x=1:3))
```

---

```
replayr_is_MySQL_data
```

*Test if data is MySQL.*

---

**Description**

Test if data is MySQL.

**Usage**

```
replayr_is_MySQL_data(d)
```

**Arguments**

d                    data frame

**Value**

TRUE if Spark data

**Examples**

```
replayr_is_MySQL_data(data.frame(x=1:3))
```

---

```
replayr_is_Spark_data
```

*Test if data is Spark.*

---

**Description**

Test if data is Spark.

**Usage**

```
replayr_is_Spark_data(d)
```

**Arguments**

d                    data frame

**Value**

TRUE if Spark data

**Examples**

```
replyr_is_Spark_data(data.frame(x=1:3))
```

---

```
replyr_list_tables list tables
```

---

**Description**

Work around connection v.s. handle issues <https://github.com/tidyverse/dplyr/issues/2849>

**Usage**

```
replyr_list_tables(con)
```

**Arguments**

con                    connection

**Value**

list of tables names

**Examples**

```
if (requireNamespace("RSQLite", quietly = TRUE)) {  
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")  
  RSQLite::initExtension(my_db)  
  d <- replyr_copy_to(my_db, data.frame(x=c(1,2)), 'd',  
                     overwrite=TRUE, temporary=TRUE)  
  
  print(d)  
  print(replyr_list_tables(my_db))  
  DBI::dbDisconnect(my_db)  
}
```

---

```
replyr_mapRestrictCols
```

*Map names of columns to known values and drop other columns.*

---

## Description

Restrict a data item's column names and re-name them in bulk.

## Usage

```
replyr_mapRestrictCols(x, nmap, ..., restrict = FALSE, reverse = FALSE)
```

## Arguments

<code>x</code>	data item to work on
<code>nmap</code>	named list mapping with keys specifying new column names, and values as original column names.
<code>...</code>	force later arguments to bind by name
<code>restrict</code>	logical if TRUE restrict to columns mentioned in <code>nmap</code> .
<code>reverse</code>	logical if TRUE apply the inverse of <code>nmap</code> instead of <code>nmap</code> .

## Details

Something like `replyr::replyr_mapRestrictCols` is only useful to get control of a function that is not parameterized (in the sense it has hard-coded column names inside its implementation that don't match column names in our data).

## Value

data item with columns renamed (and possibly restricted).

## See Also

```
let, replyr_reverseMap, replyr_apply_f_mapped
```

## Examples

```
# an external function with hard-coded column names
DecreaseRankColumnByOne <- function(d) {
  d$RankColumn <- d$RankColumn - 1
  d
}

# our example data, with different column names
d <- data.frame(Sepal_Length=c(5.8,5.7),
               Sepal_Width=c(4.0,4.4),
```

```
      Species='setosa',rank=c(1,2))
print(d)

# map our data to expected column names so we can use function
nmap <- c(GroupColumn='Species',
          ValueColumn='Sepal_Length',
          RankColumn='rank')
print(nmap)
dm <- replyr_mapRestrictCols(d,nmap)
print(dm)

# can now apply code that expects hard-coded names.
dm <- DecreaseRankColumnByOne(dm)

# map back to our original column names (for the columns we retained)
# Note: can only map back columns that were retained in first mapping.
replyr_mapRestrictCols(dm, nmap, reverse=TRUE)
```

---

replyr_ncol	<i>Compute number of columns of a data.frame (work around <a href="https://github.com/rstudio/sparklyr/issues/976">https://github.com/rstudio/sparklyr/issues/976</a> ).</i>
-------------	--

---

## Description

Compute number of columns of a data.frame (work around <https://github.com/rstudio/sparklyr/issues/976> ).

## Usage

```
replyr_ncol(x)
```

## Arguments

x                   tbl or item that can be coerced into such.

## Value

number of columns

## Examples

```
d <- data.frame(x=c(1,2))
replyr_ncol(d)
```

---

replyr_nrow	<i>Compute number of rows of a tbl.</i>
-------------	---

---

### Description

Number of row in a table. This function is not "group aware" it returns the total number of rows, not rows per dplyr group. Also `replyr_nrow` depends on data being returned to count, so some corner cases (such as zero columns) will count as zero rows. In particular work around dplyr issue 2871 <https://github.com/tidyverse/dplyr/issues/2871>.

### Usage

```
replyr_nrow(x)
```

### Arguments

`x` tbl or item that can be coerced into such.

### Value

number of rows

### Examples

```
d <- data.frame(x=c(1,2))
replyr_nrow(d)
```

---

replyr_quantile	<i>Compute quantiles on remote column (NA's filtered out) using binary search.</i>
-----------------	--

---

### Description

NA's filtered out and does not break ties the same as `stats::quantile`.

### Usage

```
replyr_quantile(x, cname, probs = seq(0, 1, 0.25), ...,
  tempNameGenerator = mk_tmp_name_source("replyr_quantile"))
```

**Arguments**

x	tbl or item that can be coerced into such.
cname	column name to compute over
probs	numeric vector of probabilities with values in [0,1].
...	force later arguments to be bound by name.
tempNameGenerator	temp name generator produced by wrapr::mk_tmp_name_source, used to record dplyr::compute() effects.

**Examples**

```
d <- data.frame(xvals=rev(1:1000))
replyr_quantile(d, 'xvals')
```

---

replyr_quantilec	<i>Compute quantiles on remote column (NA's filtered out) using cum-sum.</i>
------------------	--

---

**Description**

NA's filtered out and does not break ties the same as stats::quantile.

**Usage**

```
replyr_quantilec(x, cname, probs = seq(0, 1, 0.25), ...,
  tempNameGenerator = mk_tmp_name_source("replyr_quantilec"))
```

**Arguments**

x	tbl or item that can be coerced into such.
cname	column name to compute over (not 'n' or 'csum')
probs	numeric vector of probabilities with values in [0,1].
...	force later arguments to bind by name.
tempNameGenerator	temp name generator produced by wrapr::mk_tmp_name_source, used to record dplyr::compute() effects.

**Examples**

```
d <- data.frame(xvals=rev(1:1000))
replyr_quantilec(d, 'xvals')
```

---

replyr_rename	<i>Rename a column</i>
---------------	------------------------

---

**Description**

Rename a column

**Usage**

```
replyr_rename(.data, ..., newName, oldName)
```

**Arguments**

.data	data object to work on
...	force later arguments to bind by name
newName	character new column name
oldName	character old column name

**Examples**

```
d <- data.frame(Sepal_Length= c(5.8,5.7),
                Sepal_Width= c(4.0,4.4),
                Species= 'setosa', rank=c(1,2))
replyr_rename(d, newName = 'family', oldName = 'Species')
```

---

replyr_reverseMap	<i>Reverse a name assignment map (which are written NEW-NAME=OLDNAME).</i>
-------------------	--

---

**Description**

Reverse a name assignment map (which are written NEWNAME=OLDNAME).

**Usage**

```
replyr_reverseMap(nmap)
```

**Arguments**

nmap	named list mapping with keys specifying new column names, and values as original column names.
------	--

**Value**

inverse map

**See Also**

let, replyr\_apply\_f\_mapped, replyr\_mapRestrictCols

**Examples**

```
mp <- c(A='x', B='y')
print(mp)
replyr_reverseMap(mp)
```

---

replyr_select	<i>select columns</i>
---------------	-----------------------

---

**Description**

select columns

**Usage**

```
replyr_select(.data, colnames)
```

**Arguments**

.data	data object to work on
colnames	character column names

**Examples**

```
d <- data.frame(Sepal_Length= c(5.8,5.7),
               Sepal_Width= c(4.0,4.4),
               Species= 'setosa', rank=c(1,2))
replyr_select(d, c('Sepal_Length', 'Species'))
```

---

replyr_split	<i>split a data item by values in a column.</i>
--------------	---

---

### Description

Partitions from by values in grouping column, and returns list. Only advised for a moderate number of groups and better if grouping column is an index. This plus lapply and replyr::bind\_rows is powerful enough to implement "The Split-Apply-Combine Strategy for Data Analysis" <https://www.jstatsoft.org/article/view/v040>

### Usage

```
replyr_split(df, gcolumn, ..., ocolumn = NULL, decreasing = FALSE,
  partitionMethod = "extract", maxgroups = 100, eagerCompute = FALSE)
```

### Arguments

df	remote dplyr data item
gcolumn	grouping column
...	force later values to be bound by name
ocolumn	ordering column (optional)
decreasing	if TRUE sort in decreasing order by ocolumn
partitionMethod	method to partition the data, one of 'split' (only works over local data frames), or 'extract'
maxgroups	maximum number of groups to work over
eagerCompute	if TRUE call compute on split results

### Value

list of data items

### Examples

```
d <- data.frame(group=c(1,1,2,2,2),
  order=c(.1,.2,.3,.4,.5),
  values=c(10,20,2,4,8))
dSplit <- replyr_split(d, 'group', partitionMethod='extract')
dApp <- lapply(dSplit, function(di) data.frame(as.list(colMeans(di))))
replyr_bind_rows(dApp)
```

---

```
replyr_summary      Compute usable summary of columns of tbl.
```

---

### Description

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

### Usage

```
replyr_summary(x, ..., countUniqueNum = FALSE,
               countUniqueNonNum = FALSE, cols = NULL, compute = TRUE)
```

### Arguments

<code>x</code>	tbl or item that can be coerced into such.
<code>...</code>	force additional arguments to be bound by name.
<code>countUniqueNum</code>	logical, if true include unique non-NA counts for numeric cols.
<code>countUniqueNonNum</code>	logical, if true include unique non-NA counts for non-numeric cols.
<code>cols</code>	if not NULL set of columns to restrict to.
<code>compute</code>	logical if TRUE call compute before working

### Details

Can be slow compared to `dplyr::summarize_all()` (but serves a different purpose). Also, for numeric columns includes NaN in nna count (as is typical for R, e.g., `is.na(NaN)`). And note: `replyr_summary()` currently skips "raw" columns.

### Value

summary of columns.

### See Also

`rsummary`

### Examples

```
d <- data.frame(p= c(TRUE, FALSE, NA),
               r= I(list(1, 2, 3)),
               s= NA,
               t= as.raw(3:5),
               w= 1:3,
               x= c(NA, 2, 3),
```

```

        y= factor(c(3,5,NA)),
        z= c('a',NA,'z'),
        stringsAsFactors=FALSE)
# sc <- sparklyr::spark_connect(version='2.2.0',
#                               master = "local")
# dS <- replyr_copy_to(sc, dplyr::select(d, -r, -t), 'dS',
#                     temporary=TRUE, overwrite=TRUE)
# replyr_summary(dS)
# sparklyr::spark_disconnect(sc)
if (requireNamespace("RSQLite", quietly = TRUE)) {
  my_db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  RSQLite::initExtension(my_db)
  dM <- replyr_copy_to(my_db, dplyr::select(d, -r, -t), 'dM',
                    temporary=TRUE, overwrite=TRUE)
  print(replyr_summary(dM))
  DBI::dbDisconnect(my_db)
}
dSq <- list(1,2,3)
replyr_summary(d)

```

---

replyr\_testCols      *Run test on columns.*

---

## Description

Applies user function to head of each column. Good for determining things such as column class.

## Usage

```
replyr_testCols(x, f, n = 6L)
```

## Arguments

x	tbl or item that can be coerced into such.
f	test function (returning logical, not depending on data length).
n	number of rows to use in calculation.

## Value

logical vector of results.

## Examples

```

d <- data.frame(x=c(1,2),y=c('a','b'))
replyr_testCols(d,is.numeric)

```

---

replyr\_union\_all     *Union two tables.*

---

### Description

Spark 2\* union\_all has issues (<https://github.com/WinVector/replyr/blob/master/issues/UnionIssue.md>), and exposed union\_all semantics differ from data-source back-end to back-end. This is an attempt to provide a join-based replacement.

### Usage

```
replyr_union_all(tabA, tabB, ..., useDplyrLocal = TRUE,
  useSparkRbind = TRUE,
  tempNameGenerator = mk_tmp_name_source("replyr_union_all"))
```

### Arguments

tabA	not-NULL table with at least 1 row.
tabB	not-NULL table with at least 1 row on same data source as tabA and common columns.
...	force later arguments to be bound by name.
useDplyrLocal	logical if TRUE use dplyr::bind_rows for local data.
useSparkRbind	logical if TRUE try to use rbind on Sparklyr data
tempNameGenerator	temp name generator produced by wrapr::mk_tmp_name_source, used to record dplyr::compute() effects.

### Value

table with all rows of tabA and tabB (union\_all).

### Examples

```
d1 <- data.frame(x = c('a','b'), y = 1, stringsAsFactors= FALSE)
d2 <- data.frame(x = 'c', z = 1, stringsAsFactors= FALSE)
replyr_union_all(d1, d2, useDplyrLocal= FALSE)
```

```
replyr_uniqueValues
```

*Compute number of unique values for each level in a column.*

---

### Description

Compute number of unique values for each level in a column.

### Usage

```
replyr_uniqueValues(x, cname)
```

### Arguments

`x` tbl or item that can be coerced into such.  
`cname` name of columns to examine, must not be equal to 'replyr\_private\_value\_n'.

### Value

unique values for the column.

### Examples

```
d <- data.frame(x=c(1,2,3,3))  
replyr_uniqueValues(d, 'x')
```

---

```
tableDescription Build a nice description of a table.
```

---

### Description

Please see <http://www.win-vector.com/blog/2017/05/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

```
tableDescription(tableName, handle, ...,  
  keyInspector = key_inspector_all_cols)
```

**Arguments**

tableName	name of table to add to join plan.
handle	table or table handle to add to join plan (can already be in the plan).
...	force later arguments to bind by name.
keyInspector	function that determines preferred primary key set for table.

**Details**

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

**Value**

table describing the data.

**See Also**

`buildJoinPlan`, `keysAreUnique`, `makeJoinDiagramSpec`, `executeLeftJoinPlan`

**Examples**

```
d <- data.frame(x=1:3, y=NA)
tableDescription('d', d)
```

---

topoSortTables      *Topologically sort join plan so values are available before uses.*

---

**Description**

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

**Usage**

```
topoSortTables(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**

```

if (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_employeeAndDate(my_db)
  columnJoinPlan <- buildJoinPlan(tDesc, check= FALSE)
  # unify keys
  columnJoinPlan$resultColumn[columnJoinPlan$resultColumn=='id'] <- 'eid'
  # look at plan defects
  print(paste('problems:',
              inspectDescrAndJoinPlan(tDesc, columnJoinPlan)))

  # fix plan
  if(requireNamespace('igraph', quietly = TRUE)) {
    sorted <- topoSortTables(columnJoinPlan, 'employeeanddate')
    print(paste('problems:',
                inspectDescrAndJoinPlan(tDesc, sorted$columnJoinPlan)))
    # plot(sorted$dependencyGraph)
  }
  DBI::dbDisconnect(my_db)
  my_db <- NULL
}

```

---

%land%

*Land a value to variable from a pipeline.*

---

**Description**

%land% and %->% ("writearrow") copy a pipeline value to a variable on the right hand side.  
 %land\_% and %->\_% copy a pipeline value to a variable named by the value referenced by its  
 right hand side argument.

**Usage**

value %land% name

value %->% name

value %->\_% name

value %land\_% name

**Arguments**

value	value to write
name	variable to write to

**Details**

Technically these operators are not "-> assignment", so they might not be specifically prohibited in an oppugnant reading of some style guides.

**Value**

value

**Examples**

```
sin(7) %->% z1
sin(7) %->_ % 'z2'
varname <- 'z3'
sin(7) %->_ % varname
```