

Package ‘dm’

July 30, 2020

Title Relational Data Models

Version 0.1.6

Date 2020-07-29

Description Provides tools for working with multiple related tables, stored as data frames or in a relational database. Multiple tables (data and metadata) are stored in a compound object, which can then be manipulated with a pipe-friendly syntax.

License MIT + file LICENSE

URL <https://krlmlr.github.io/dm>, <https://github.com/krlmlr/dm>

BugReports <https://github.com/krlmlr/dm/issues>

Depends R (>= 3.3)

Imports backports, cli (>= 2.0.0), DBI, dplyr (>= 1.0.0), ellipsis, glue, igraph, lifecycle, magrittr, memoise, methods, pillar, purrr, rlang (>= 0.4.0), tibble, tidyr (>= 1.0.0), tidyselect (>= 1.0.0), vctrs (>= 0.3.2)

Suggests brio, crayon, dbplyr, DiagrammeR, DiagrammeRsvg, digest, fansi, keyring, knitr, nycflights13, odbc, RMariaDB, rmarkdown, RPostgres, RSQLite, testthat (>= 2.1.0), tidyverse, withr

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1.9000

NeedsCompilation no

Author Tobias Schieferdecker [aut],
Kirill Müller [aut, cre] (<<https://orcid.org/0000-0002-1416-3412>>),
Darko Bergant [aut],
Katharina Brunner [ctb],
James Wondrasek [ctb],
energie360° AG [fnd],
cynkra GmbH [fnd, cph]

Maintainer Kirill Müller <kr1mlr+r@mailbox.org>

Repository CRAN

Date/Publication 2020-07-30 00:20:06 UTC

R topics documented:

check_key	3
check_set_equality	4
check_subset	5
copy_dm_to	5
decompose_table	7
dm	9
dm_add_fk	11
dm_add_pk	12
dm_add_tbl	14
dm_bind	15
dm_disambiguate_cols	16
dm_draw	17
dm_enum_fk_candidates	19
dm_examine_constraints	21
dm_filter	22
dm_financial	24
dm_flatten_to_tbl	24
dm_from_src	26
dm_get_all_fks	27
dm_get_all_pks	28
dm_get_filters	29
dm_get_fk	29
dm_get_pk	30
dm_get_referencing_tables	31
dm_has_fk	32
dm_has_pk	33
dm_is_referenced	33
dm_join_to_tbl	34
dm_nrow	35
dm_nycflights13	36
dm_paste	36
dm_ptype	38
dm_rename	38
dm_rm_tbl	39
dm_select	40
dm_select_tbl	41
dm_zoom_to	42
dplyr_join	44
dplyr_table_manipulation	45
enum_pk_candidates	47
examine_cardinality	48

<code>check_key</code>	3
<code>head.zoomed_dm</code>	50
<code>pull_tbl</code>	51
<code>reunite_parent_child</code>	51
<code>rows-db</code>	53
<code>rows-dm</code>	54
<code>rows_truncate</code>	57
<code>tbl.dm</code>	58
<code>tidyr_table_manipulation</code>	59
Index	60

<code>check_key</code>	<i>Check if column(s) can be used as keys</i>
------------------------	---

Description

`check_key()` accepts a data frame and, optionally, columns. It throws an error if the specified columns are NOT a unique key of the data frame. If the columns given in the ellipsis ARE a key, the data frame itself is returned silently, so that it can be used for piping.

Usage

```
check_key(.data, ...)
```

Arguments

<code>.data</code>	The data frame whose columns should be tested for key properties.
<code>...</code>	<p>The names of the columns to be checked.</p> <p>One or more unquoted expressions separated by commas. Variable names can be treated as if they were positions, so you can use expressions like <code>x:y</code> to select ranges of variables.</p> <p>The arguments in <code>...</code> are automatically quoted and evaluated in a context where column names represent column positions. They also support unquoting and splicing. See <code>vignette("programming")</code> for an introduction to these concepts.</p> <p>See <code>select</code> helpers for more details and examples about <code>tidyselect</code> helpers such as <code>starts_with()</code>, <code>everything()</code>, ...</p>

Value

Returns `.data`, invisibly, if the check is passed. Otherwise an error is thrown and the reason for it is explained.

Examples

```
data <- tibble::tibble(a = c(1, 2, 1), b = c(1, 4, 1), c = c(5, 6, 7))
# this is failing:
try(check_key(data, a, b))

# this is passing:
check_key(data, a, c)
```

check_set_equality *Check column values for set equality*

Description

check_set_equality() is a wrapper of check_subset(). It tests if one value set is a subset of another and vice versa, i.e., if both sets are the same. If not, it throws an error.

Usage

```
check_set_equality(t1, c1, t2, c2)
```

Arguments

t1	The data frame that contains column c1.
c1	The column of t1 that should only contain values that are also present in column c2 of data frame t2.
t2	The data frame that contains column c2.
c2	The column of t2 that should only contain values that are also present in column c1 of data frame t1.

Value

Returns t1, invisibly, if the check is passed. Otherwise an error is thrown and the reason for it is explained.

Examples

```
data_1 <- tibble::tibble(a = c(1, 2, 1), b = c(1, 4, 1), c = c(5, 6, 7))
data_2 <- tibble::tibble(a = c(1, 2, 3), b = c(4, 5, 6), c = c(7, 8, 9))
# this is failing:
try(check_set_equality(data_1, a, data_2, a))

data_3 <- tibble::tibble(a = c(2, 1, 2), b = c(4, 5, 6), c = c(7, 8, 9))
# this is passing:
check_set_equality(data_1, a, data_3, a)
```

check_subset	<i>Check column values for subset</i>
--------------	---------------------------------------

Description

check_subset() tests if the values of the chosen column c1 of data frame t1 are a subset of the values of column c2 of data frame t2.

Usage

```
check_subset(t1, c1, t2, c2)
```

Arguments

t1	The data frame that contains column c1.
c1	The column of t1 that should only contain the values that are also present in column c2 of data frame t2.
t2	The data frame that contains column c2.
c2	The column of the second data frame that has to contain all values of c1 to avoid an error.

Value

Returns t1, invisibly, if the check is passed. Otherwise an error is thrown and the reason for it is explained.

Examples

```
data_1 <- tibble::tibble(a = c(1, 2, 1), b = c(1, 4, 1), c = c(5, 6, 7))
data_2 <- tibble::tibble(a = c(1, 2, 3), b = c(4, 5, 6), c = c(7, 8, 9))
# this is passing:
check_subset(data_1, a, data_2, a)

# this is failing:
try(check_subset(data_2, a, data_1, a))
```

copy_dm_to	<i>Copy data model to data source</i>
------------	---------------------------------------

Description

copy_dm_to() takes a [dplyr::src_dbi](#) object or a [DBI::DBIConnection](#) object as its first argument and a [dm](#) object as its second argument. The latter is copied to the former. By default, temporary tables will be created and the key constraints will be set (currently only on MSSQL and Postgres databases).

Usage

```
copy_dm_to(
  dest,
  dm,
  ...,
  types = NULL,
  overwrite = NULL,
  indexes = NULL,
  unique_indexes = NULL,
  set_key_constraints = TRUE,
  unique_table_names = NULL,
  table_names = NULL,
  temporary = TRUE
)
```

Arguments

<code>dest</code>	An object of class "src" or "DBIConnection".
<code>dm</code>	A dm object.
<code>...</code>	Passed on to <code>dplyr::copy_to()</code> , which is used on each table.
<code>overwrite</code> , <code>types</code> , <code>indexes</code> , <code>unique_indexes</code>	Must remain NULL.
<code>set_key_constraints</code>	Boolean variable, if TRUE will mirror dm key constraints on a database.
<code>unique_table_names</code>	Deprecated.
<code>table_names</code>	<p>Desired names for the tables on <code>dest</code>; the names within the <code>dm</code> remain unchanged. Can be NULL, a named character vector, a function or a one-sided formula.</p> <p>If left NULL (default), the names will be determined automatically depending on the <code>temporary</code> argument:</p> <ol style="list-style-type: none"> <code>temporary = TRUE</code> (default): unique table names based on the names of the tables in the <code>dm</code> are created. <code>temporary = FALSE</code>: the table names in the <code>dm</code> are used as names for the tables on <code>dest</code>. <p>If a function or one-sided formula, <code>table_names</code> is converted to a function using <code>rlang::as_function()</code>. This function is called with the unquoted table names of the <code>dm</code> object as the only argument. The output of this function is processed by <code>DBI::dbQuoteIdentifier()</code>, that result should be a vector of identifiers of the same length as the original table names.</p> <p>Use a variant of <code>table_names = ~ DBI::SQL(dplyr::in_schema("schema_name", .x))</code> to specify the same schema for all tables. Use <code>table_names = identity</code> with <code>temporary = TRUE</code> to avoid giving temporary tables unique names.</p> <p>If a named character vector, the names of this vector need to correspond to the table names in the <code>dm</code>, and its values are the desired names on <code>dest</code>. The value</p>

is processed by `DBI::dbQuoteIdentifier()`, that result should be a vector of identifiers of the same length as the original table names.

Use qualified names corresponding to your database's syntax to specify e.g. database and schema for your tables.

`temporary` Boolean variable, if TRUE, only temporary tables will be created. These tables will vanish when disconnecting from the database.

Details

No tables will be overwritten; passing `overwrite = TRUE` to the function will give an error. Types are determined separately for each table, setting the `types` argument will also throw an error. The arguments are included in the signature to avoid passing them via the `...` ellipsis.

Value

A `dm` object on the given `src` with the same table names as the input `dm`.

Examples

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

# Copy to temporary tables, unique table names by default:
temp_dm <- copy_dm_to(
  con,
  dm_nycflights13(),
  set_key_constraints = FALSE
)

# Persist, explicitly specify table names:
persistent_dm <- copy_dm_to(
  con,
  dm_nycflights13(),
  temporary = FALSE,
  table_names = ~ paste0("flights_", .x)
)
dbplyr::remote_name(persistent_dm$planes)

DBI::dbDisconnect(con)
```

Description

Questioning

Perform table surgery by extracting a 'parent table' from a table, linking the original table and the new table by a key, and returning both tables.

`decompose_table()` accepts a data frame, a name for the 'ID column' that will be newly created, and the names of the columns that will be extracted into the new data frame.

It creates a 'parent table', which consists of the columns specified in the ellipsis, and a new 'ID column'. Then it removes those columns from the original table, which is now called the 'child table', and adds the 'ID column'.

Usage

```
decompose_table(.data, new_id_column, ...)
```

Arguments

<code>.data</code>	Data frame from which columns ... are to be extracted.
<code>new_id_column</code>	Name of the identifier column (primary key column) for the parent table. A column of this name is also added in 'child table'.
<code>...</code>	The columns to be extracted from the <code>.data</code> . One or more unquoted expressions separated by commas. You can treat variable names as if they were positions, so you can use expressions like <code>x:y</code> to select ranges of variables. The arguments in ... are automatically quoted and evaluated in a context where column names represent column positions. They also support unquoting and splicing. See <code>vignette("programming")</code> for an introduction to those concepts. See <code>select</code> helpers for more details, and the examples about <code>tidyselect</code> helpers, such as <code>starts_with()</code> , <code>everything()</code> , ...

Value

A named list of length two:

- entry "child_table": the child table with column `new_id_column` referring to the same column in `parent_table`,
- entry "parent_table": the "lookup table" for `child_table`.

Life cycle

This function is marked "questioning" because it feels more useful when applied to a table in a `dm` object.

See Also

Other table surgery functions: [reunite_parent_child\(\)](#)

Examples

```
decomposed_table <- decompose_table(mtcars, new_id, am, gear, carb)
decomposed_table$child_table
decomposed_table$parent_table
```

 dm

Data model class

Description

The `dm` class holds a list of tables and their relationships. It is inspired by [datamodelr](#), and extends the idea by offering operations to access the data in the tables.

`dm()` creates a `dm` object from `tbl` objects (tibbles or lazy data objects).

`new_dm()` is a low-level constructor that creates a new `dm` object.

- If called without arguments, it will create an empty `dm`.
- If called with arguments, no validation checks will be made to ascertain that the inputs are of the expected class and internally consistent; use `validate_dm()` to double-check the returned object.

`validate_dm()` checks the internal consistency of a `dm` object.

`dm_get_src()` returns the **dplyr** source for a `dm` object. All tables in a `dm` object must be from the same source, i.e. either they are all data frames, or they all are stored on the same database.

`dm_get_con()` returns the `DBI::DBIConnection` for `dm` objects. This works only if the tables are stored on a database, otherwise an error is thrown.

`dm_get_tables()` returns a named list of **dplyr** `tbl` objects of a `dm` object. Filtering expressions are NOT evaluated at this stage. To get a filtered table, use `dm_apply_filters_to_tbl()`, to apply filters to all tables use `dm_apply_filters()`

`is_dm()` returns TRUE if the input is of class `dm`.

`as_dm()` coerces objects to the `dm` class

Usage

```
dm(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))
```

```
new_dm(tables = list())
```

```
validate_dm(x)
```

```
dm_get_src(x)
```

```
dm_get_con(x)
```

```
dm_get_tables(x)
```

`is_dm(x)`

`as_dm(x)`

Arguments

<code>...</code>	Tables to add to the dm object. If no names are provided, the tables are auto-named.
<code>.name_repair</code>	Options for name repair. Forwarded as <code>repair</code> to <code>vctrs::vec_as_names()</code> .
<code>tables</code>	A named list of the tables (tibble-objects, not names), to be included in the dm object.
<code>x</code>	An object.

Value

For `dm()`, `new_dm()`, `as_dm()`: A dm object.

For `validate_dm()`: Returns the dm, invisibly, after finishing all checks.

For `dm_get_src()`: the **dplyr** source for a dm object, or NULL if the dm object contains data frames.

For `dm_get_con()`: The `DBI::DBIConnection` for dm objects.

For `dm_get_tables()`: A named list with the tables constituting the dm.

For `is_dm()`: Boolean, is this object a dm.

See Also

- `dm_from_src()` for connecting to all tables in a database and importing the primary and foreign keys
- `dm_add_pk()` and `dm_add_fk()` for adding primary and foreign keys
- `copy_dm_to()` for DB interaction
- `dm_draw()` for visualization
- `dm_join_to_tbl()` for flattening
- `dm_filter()` for filtering
- `dm_select_tbl()` for creating a dm with only a subset of the tables
- `dm_nycflights13()` for creating an example dm object
- `decompose_table()` for table surgery
- `check_key()` and `check_subset()` for checking for key properties
- `examine_cardinality()` for checking the cardinality of the relation between two tables

Examples

```
dm(trees, mtcars)
new_dm(list(trees = trees, mtcars = mtcars))
as_dm(list(trees = trees, mtcars = mtcars))
```

```

dm_nycflights13() %>% tbl("airports")
dm_nycflights13() %>% src_tbls()
dm_nycflights13() %>% dm_get_src()

copy_dm_to(
  dbplyr::src_memdb(),
  dm_nycflights13(),
  unique_table_names = TRUE
) %>%
  dm_get_con()

dm_nycflights13() %>% dm_get_tables()
dm_nycflights13() %>% dm_get_filters()
dm_nycflights13() %>% validate_dm()
is_dm(dm_nycflights13())
dm_nycflights13()["airports"]
dm_nycflights13()[["airports"]]
dm_nycflights13()$airports

```

dm_add_fk

*Add/remove foreign keys***Description**

dm_add_fk() marks the specified columns as the foreign key of table `table` with respect to the primary key of table `ref_table`. If `check == TRUE`, then it will first check if the values in columns `columns` are a subset of the values of the primary key in table `ref_table`.

dm_rm_fk() can remove either one reference between two tables, or all references at once, if argument `columns = NULL`. All arguments may be provided quoted or unquoted.

Usage

```
dm_add_fk(dm, table, columns, ref_table, check = FALSE)
```

```
dm_rm_fk(dm, table, columns, ref_table)
```

Arguments

dm	A dm object.
table	A table in the dm.
columns	For dm_add_fk(): The columns of <code>table</code> which are to become the foreign key columns that reference the primary key of <code>ref_table</code> . For dm_rm_fk(): The columns of <code>table</code> that should no longer be referencing the primary key of <code>ref_table</code> . If <code>NULL</code> , all columns will be evaluated.
ref_table	For dm_add_fk(): The table which <code>table</code> will be referencing. This table needs to have a primary key set. For dm_rm_fk(): The table that <code>table</code> is referencing.

check Boolean, if TRUE, a check will be performed to determine if the values of column are a subset of the values of the primary key column of ref_table.

Value

For dm_add_fk(): An updated dm with an additional foreign key relation.

For dm_rm_fk(): An updated dm without the given foreign key relation.

Compound keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. The syntax of these functions will be extended but will remain compatible with current semantics.

See Also

Other foreign key functions: [dm_enum_fk_candidates\(\)](#), [dm_get_all_fks\(\)](#), [dm_get_fk\(\)](#), [dm_has_fk\(\)](#)

Other foreign key functions: [dm_enum_fk_candidates\(\)](#), [dm_get_all_fks\(\)](#), [dm_get_fk\(\)](#), [dm_has_fk\(\)](#)

Examples

```
nycflights_dm <- dm(
  planes = nycflights13::planes,
  flights = nycflights13::flights
)

nycflights_dm %>%
  dm_draw()

nycflights_dm %>%
  dm_add_pk(planes, tailnum) %>%
  dm_add_fk(flights, tailnum, planes) %>%
  dm_draw()

dm_nycflights13(cycle = TRUE) %>%
  dm_rm_fk(flights, dest, airports) %>%
  dm_draw()
```

Description

dm_add_pk() marks the specified columns as the primary key of the specified table. If check == TRUE, then it will first check if the given combination of columns is a unique key of the table. If force == TRUE, the function will replace an already set key.

dm_rm_pk() removes a primary key from a table and leaves the dm object otherwise unaltered. Foreign keys that point to the table from other tables, can be optionally removed as well.

Usage

```
dm_add_pk(dm, table, columns, check = FALSE, force = FALSE)
```

```
dm_rm_pk(dm, table, rm_referencing_fks = FALSE)
```

Arguments

dm	A dm object.
table	A table in the dm.
columns	Table columns, unquoted.
check	Boolean, if TRUE, a check is made if the combination of columns is a unique key of the table.
force	Boolean, if FALSE (default), an error will be thrown if there is already a primary key set for this table. If TRUE, a potential old pk is deleted before setting a new one.
rm_referencing_fks	Boolean: if FALSE (default), will throw an error if there are foreign keys addressing the primary key that is to be removed. If TRUE, the function will remove, in addition to the primary key of the table argument, also all foreign key constraints that are pointing to it.

Value

For dm_add_pk(): An updated dm with an additional primary key.

For dm_rm_pk(): An updated dm without the indicated primary key.

Compound keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. The syntax of these functions will be extended but will remain compatible with current semantics.

See Also

Other primary key functions: [dm_get_all_pks\(\)](#), [dm_get_pk\(\)](#), [dm_has_pk\(\)](#), [enum_pk_candidates\(\)](#)

Examples

```

nycflights_dm <- dm(
  planes = nycflights13::planes,
  airports = nycflights13::airports
)

nycflights_dm %>%
  dm_draw()

# the following works
nycflights_dm %>%
  dm_add_pk(planes, tailnum) %>%
  dm_add_pk(airports, faa, check = TRUE) %>%
  dm_draw()

# the following throws an error:
try(
  nycflights_dm %>%
    dm_add_pk(planes, manufacturer, check = TRUE)
)

dm_nycflights13() %>%
  dm_rm_pk(airports, rm_referencing_fks = TRUE) %>%
  dm_draw()

```

`dm_add_tbl`*Add tables to a dm*

Description

`dm_add_tbl()` adds one or more tables to a `dm`. It uses `mutate()` semantics.

Usage

```
dm_add_tbl(dm, ..., repair = "unique", quiet = FALSE)
```

Arguments

<code>dm</code>	A <code>dm</code> object.
<code>...</code>	One or more tables to add to the <code>dm</code> . If no explicit name is given, the name of the expression is used.
<code>repair</code>	Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown.

- Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.
- Unique names are unique. A suffix is appended to duplicate names to make them unique.
- Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error.

The "check_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria.

`quiet` By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set `quiet` to `TRUE` to silence the messages.

Value

The initial dm with the additional table(s).

See Also

[dm_rm_tbl\(\)](#)

Examples

```
dm() %>%
  dm_add_tbl(mtcars, flowers = iris)

# renaming table names if necessary (depending on the `repair` argument)
dm() %>%
  dm_add_tbl(new_tbl = mtcars, new_tbl = iris)
```

dm_bind

Merge several dm

Description

Create a single dm from two or more dm objects.

Usage

```
dm_bind(..., repair = "check_unique", quiet = FALSE)
```

Arguments

`...` dm objects to bind together.

`repair` Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown.

- Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.
- Unique names are unique. A suffix is appended to duplicate names to make them unique.
- Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error.

The "check_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria.

`quiet` By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set `quiet` to `TRUE` to silence the messages.

Details

The dm objects have to share the same src. By default table names need to be unique.

Value

dm containing the tables and key relations of all dm objects.

Examples

```
dm_1 <- dm_nycflights13()
dm_2 <- dm(mtcars, iris)
dm_bind(dm_1, dm_2)
```

`dm_disambiguate_cols` *Resolve column name ambiguities*

Description

This function ensures that all columns in a dm have unique names.

Usage

```
dm_disambiguate_cols(dm, sep = ".", quiet = FALSE)
```

Arguments

<code>dm</code>	A dm object.
<code>sep</code>	The character variable that separates the names of the table and the names of the ambiguous columns.
<code>quiet</code>	Boolean. By default, this function lists the renamed columns in a message, pass <code>TRUE</code> to suppress this message.

Details

The function first checks if there are any column names that are not unique. If there are, those columns will be assigned new, unique, names by prefixing their existing name with the name of their table and a separator. Columns that act as primary or foreign keys will not be renamed because only the foreign key column will remain when two tables are joined, making that column name "unique" as well.

Value

A dm whose column names are unambiguous.

Examples

```
dm_nycflights13() %>%
  dm_disambiguate_cols()
```

dm_draw

Draw a diagram of the data model

Description

dm_draw() uses **DiagrammeR** to draw diagrams.

dm_set_colors() allows to define the colors that will be used to display the tables of the data model. The colors can either be either specified with hex color codes or using the names of the built-in R colors. An overview of the colors corresponding to the standard color names can be found at the bottom of <http://rpubs.com/kr1m1r/colors>.

dm_get_colors() returns the colors defined for a data model.

dm_get_available_colors() returns an overview of the names of the available colors These are the standard colors also returned by `grDevices::colors()` plus a default table color with the name "default".

Usage

```
dm_draw(
  dm,
  rankdir = "LR",
  col_attr = "column",
  view_type = "keys_only",
  columnArrows = TRUE,
  graph_attrs = "",
  node_attrs = "",
  edge_attrs = "",
  focus = NULL,
  graph_name = "Data Model"
```

```
)
dm_set_colors(dm, ...)
dm_get_colors(dm)
dm_get_available_colors()
```

Arguments

dm	A dm object.
rankdir	Graph attribute for direction (e.g., 'BT' = bottom → top).
col_attr	Column attributes to display. By default only the column name ("column") is displayed.
view_type	Can be "keys_only" (default), "all" or "title_only". It defines the level of details for rendering tables (only primary and foreign keys, all columns, or no columns).
columnArrows	Edges from columns to columns (default: TRUE).
graph_attrs	Additional graph attributes.
node_attrs	Additional node attributes.
edge_attrs	Additional edge attributes.
focus	A list of parameters for rendering (table filter).
graph_name	The name of the graph.
...	Colors to set in the form color = table. Allowed colors are all hex coded colors (quoted) and the color names from dm_get_available_colors() . tidyselect is supported, see dplyr::select() for details on the semantics.

Value

For [dm_draw\(\)](#): returns an object of class [grViz](#) (see also [DiagrammeR::grViz\(\)](#)), which, when printed, produces the output seen in the viewer as a side effect.

For [dm_set_colors\(\)](#): the updated data model.

For [dm_get_colors\(\)](#), a two-column tibble with one row per table.

For [dm_get_available_colors\(\)](#), a vector with the available colors.

Examples

```
dm_nycflights13() %>%
  dm_draw()

dm_nycflights13(cycle = TRUE) %>%
  dm_draw(view_type = "title_only")

head(dm_get_available_colors())
length(dm_get_available_colors())
```

```
dm_nycflights13() %>%
  dm_get_colors()

dm_nycflights13(color = FALSE) %>%
  dm_set_colors(
    darkblue = starts_with("air"),
    "#5986C4" = flights
  ) %>%
  dm_draw()

# Splicing is supported:
nyc_cols <-
  dm_nycflights13() %>%
  dm_get_colors()
nyc_cols

dm_nycflights13(color = FALSE) %>%
  dm_set_colors(!!!nyc_cols) %>%
  dm_draw()
```

dm_enum_fk_candidates *Foreign key candidates*

Description

Questioning

Determine which columns would be good candidates to be used as foreign keys of a table, to reference the primary key column of another table of the `dm` object.

Usage

```
dm_enum_fk_candidates(dm, table, ref_table)

enum_fk_candidates(zoomed_dm, ref_table)
```

Arguments

<code>dm</code>	A <code>dm</code> object.
<code>table</code>	The table whose columns should be tested for suitability as foreign keys.
<code>ref_table</code>	A table with a primary key.
<code>zoomed_dm</code>	A <code>dm</code> with a zoomed table.

Details

dm_enum_fk_candidates() first checks if ref_table has a primary key set, if not, an error is thrown.

If ref_table does have a primary key, then a join operation will be tried using that key as the by argument of join() to match it to each column of table. Attempting to join incompatible columns triggers an error.

The outcome of the join operation determines the value of the why column in the result:

- an empty value for a column of table that is a suitable foreign key candidate
- the count and percentage of missing matches for a column that is not suitable
- the error message triggered for unsuitable candidates that may include the types of mismatched columns

enum_fk_candidates() works like dm_enum_fk_candidates() with the zoomed table as table.

Value

A tibble with the following columns:

columns columns of table,

candidate boolean: are these columns a candidate for a foreign key,

why if not a candidate for a foreign key, explanation for for this.

Life cycle

These functions are marked "questioning" because we are not yet sure about the interface, in particular if we need both dm_enum...() and enum...() variants. Changing the interface later seems harmless because these functions are most likely used interactively.

See Also

Other foreign key functions: [dm_add_fk\(\)](#), [dm_get_all_fks\(\)](#), [dm_get_fk\(\)](#), [dm_has_fk\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_enum_fk_candidates(flights, airports)
```

```
dm_nycflights13() %>%
  dm_zoom_to(flights) %>%
  enum_fk_candidates(airports)
```

`dm_examine_constraints`*Validate your data model*

Description

This function returns a tibble with information about which key constraints are met (`is_key = TRUE`) or violated (`FALSE`). The printing for this object is special, use `as_tibble()` to print as a regular tibble.

Usage

```
dm_examine_constraints(dm)
```

Arguments

`dm` A dm object.

Details

For the primary key constraints, it is tested if the values in the respective columns are all unique. For the foreign key constraints, the tests check if for each foreign key constraint, the values of the foreign key column form a subset of the values of the referenced column.

Value

A tibble with the following columns:

`table` the table in the dm,
`kind` "PK" or "FK",
`columns` the table columns that define the key,
`ref_table` for foreign keys, the referenced table,
`is_key` logical,
`problem` if `is_key = FALSE`, the reason for that.

Examples

```
dm_nycflights13() %>%  
  dm_examine_constraints()
```

dm_filter

*Filtering***Description****Questioning**

Filtering a table of a `dm` object may affect other tables that are connected to it directly or indirectly via foreign key relations.

`dm_filter()` can be used to define filter conditions for tables using syntax that is similar to `dplyr::filter()`. These conditions will be stored in the `dm`, and executed immediately for the tables that they are referring to.

With `dm_apply_filters()`, all tables will be updated according to the filter conditions and the foreign key relations.

`dm_apply_filters_to_tbl()` retrieves one specific table of the `dm` that is updated according to the filter conditions and the foreign key relations.

Usage

```
dm_filter(dm, table, ...)
```

```
dm_apply_filters(dm)
```

```
dm_apply_filters_to_tbl(dm, table)
```

Arguments

<code>dm</code>	A <code>dm</code> object.
<code>table</code>	A table in the <code>dm</code> .
<code>...</code>	Logical predicates defined in terms of the variables in <code>.data</code> , passed on to <code>dplyr::filter()</code> . Multiple conditions are combined with <code>&</code> or <code>,</code> . Only the rows where the condition evaluates to <code>TRUE</code> are kept. The arguments in <code>...</code> are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See <code>vignette("programming", package = "dplyr")</code> for an introduction to these concepts.

Details

The effect of the stored filter conditions on the tables related to the filtered ones is only evaluated in one of the following scenarios:

1. Calling `dm_apply_filters()` or `compute()` (method for `dm` objects) on a `dm`: each filtered table potentially reduces the rows of all other tables connected to it by foreign key relations (cascading effect), leaving only the rows with corresponding key values. Tables that are not connected to any table with an active filter are left unchanged. This results in a new `dm` class object without any filter conditions.

- Calling `dm_apply_filters_to_tbl()`: the remaining rows of the requested table are calculated by performing a sequence of semi-joins (`dplyr::semi_join()`) starting from each table that has been filtered to the requested table (similar to 1. but only for one table).

Several functions of the `dm` package will throw an error if filter conditions exist when they are called.

Value

For `dm_filter()`: an updated `dm` object (filter executed for given table, and condition stored).

For `dm_apply_filters()`: an updated `dm` object (filter effects evaluated for all tables).

For `dm_apply_filters_to_tbl()`, a table.

Life cycle

These functions are marked "questioning" because it feels wrong to tightly couple filtering with the data model. On the one hand, an overview of active filters is useful when specifying the base data set for an analysis in terms of column selections and row filters. However, these filter condition should be only of informative nature and never affect the results of other operations. We are working on formalizing the semantics of the underlying operations in order to present them in a cleaner interface.

Use `dm_zoom_to()` and `dplyr::filter()` to filter rows without registering the filter.

Examples

```
dm_nyc <- dm_nycflights13()
dm_nyc_filtered <-
  dm_nycflights13() %>%
  dm_filter(airports, name == "John F Kennedy Intl")

dm_apply_filters_to_tbl(dm_nyc_filtered, flights)

dm_nyc_filtered %>%
  dm_apply_filters()

# If you want to keep only those rows in the parent tables
# whose primary key values appear as foreign key values in
# `flights`, you can set a `TRUE` filter in `flights`:
dm_nyc %>%
  dm_filter(flights, 1 == 1) %>%
  dm_apply_filters() %>%
  dm_nrow()

# note that in this example, the only affected table is
# `airports` because the departure airports in `flights` are
# only the three New York airports.

dm_nyc %>%
  dm_filter(planes, engine %in% c("Reciprocating", "4 Cycle")) %>%
```

```
compute()
```

dm_financial	<i>Creates a dm object for the Financial data</i>
--------------	---

Description

Experimental

dm_financial() creates an example `dm` object from the tables at <https://relational.fit.cvut.cz/dataset/Financial>. The connection is established once per session, subsequent calls return the same connection.

dm_financial_sqlite() copies the data to a temporary SQLite database. The data is downloaded once per session, subsequent calls return the same database. The trans table is excluded due to its size.

Usage

```
dm_financial()
```

```
dm_financial_sqlite()
```

Value

A `dm` object.

Examples

```
dm_financial() %>%
  dm_draw()
```

dm_flatten_to_tbl	<i>Flatten a part of a dm into a wide table</i>
-------------------	---

Description

dm_flatten_to_tbl() and dm_squash_to_tbl() gather all information of interest in one place in a wide table. Both functions perform a disambiguation of column names and a cascade of joins.

Usage

```
dm_flatten_to_tbl(dm, start, ..., join = left_join)
```

```
dm_squash_to_tbl(dm, start, ..., join = left_join)
```


Arguments

dm	A <code>dm</code> object.
start	The table from which all outgoing foreign key relations are considered when establishing a processing order for the joins. An interesting choice could be for example a fact table in a star schema.
...	Unquoted names of the tables to be included in addition to the <code>start</code> table. The order of the tables here determines the order of the joins. If the argument is empty, all tables that can be reached will be included. If this includes tables that are not direct neighbors of <code>start</code> , it will only work with <code>dm_squash_to_tbl()</code> (given one of the allowed join-methods). <code>tidyselect</code> is supported, see <code>dplyr::select()</code> for details on the semantics.
join	The type of join to be performed, see <code>dplyr::join()</code> .

Details

With ... left empty, this function will join together all the tables of your `dm` object that can be reached from the `start` table, in the direction of the foreign key relations (pointing from the child tables to the parent tables), using the foreign key relations to determine the argument by for the necessary joins. The result is one table with unique column names. Use the ... argument if you would like to control which tables should be joined to the `start` table.

How does filtering affect the result?

Case 1, either no filter conditions are set in the `dm`, or set only in the part that is unconnected to the `start` table: The necessary disambiguations of the column names are performed first. Then all involved foreign tables are joined to the `start` table successively, with the join function given in the `join` argument.

Case 2, filter conditions are set for at least one table that is connected to `start`: First, disambiguation will be performed if necessary. The `start` table is then calculated using `tbl(dm, "start")`. This implies that the effect of the filters on this table is taken into account. For `right_join`, `full_join` and `nest_join`, an error is thrown if any filters are set because filters will not affect the right hand side tables and the result will therefore be incorrect in general (calculating the effects on all RHS-tables would also be time-consuming, and is not supported; if desired, call `dm_apply_filters()` first to achieve that effect). For all other join types, filtering only the `start` table is enough because the effect is passed on by successive joins.

Mind that calling `dm_flatten_to_tbl()` with `join = right_join` and no table order determined in the ... argument will not lead to a well-defined result if two or more foreign tables are to be joined to `start`. The resulting table would depend on the order the tables that are listed in the `dm`. Therefore, trying this will result in a warning.

Since `join = nest_join()` does not make sense in this direction (LHS = child table, RHS = parent table: for valid key constraints each nested column entry would be a tibble of one row), an error will be thrown if this method is chosen.

Value

A single table that results from consecutively joining all affected tables to the `start` table.

See Also

Other flattening functions: [dm_join_to_tbl\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_select_tbl(-weather) %>%
  dm_flatten_to_tbl(flights)
```

dm_from_src	<i>Load a dm from a remote data source</i>
-------------	--

Description

`dm_from_src()` creates a [dm](#) from some or all tables in a [src](#) (a database or an environment) or which are accessible via a DBI-Connection. For Postgres and SQL Server databases, primary and foreign keys are imported from the database.

Usage

```
dm_from_src(src = NULL, table_names = NULL, learn_keys = NULL, ...)
```

Arguments

<code>src</code>	A dplyr table source object or a <code>DBI::DBIConnection</code> object is accepted.
<code>table_names</code>	A character vector of the names of the tables to include.
<code>learn_keys</code>	Experimental Set to TRUE to query the definition of primary and foreign keys from the database. Currently works only for Postgres and SQL Server databases. The default attempts to query and issues an informative message.
<code>...</code>	Experimental Additional parameters for the schema learning query. Currently supports <code>schema</code> (default: "public") and <code>table_type</code> (default: "BASE TABLE") for Postgres databases.

Value

A dm object.

Examples

```
con <- DBI::dbConnect(  
  RMariaDB::MariaDB(),  
  username = "guest",  
  password = "relational",  
  dbname = "Financial_ajs",  
  host = "relational.fit.cvut.cz"  
)  
  
dm_from_src(con)  
  
DBI::dbDisconnect(con)
```

dm_get_all_fks	<i>Get foreign key constraints</i>
----------------	------------------------------------

Description

Get a summary of all foreign key relations in a [dm](#).

Usage

```
dm_get_all_fks(dm)
```

Arguments

dm A dm object.

Value

A tibble with the following columns:

child_table child table,
child_fk_cols foreign key column in child table,
parent_table parent table.

Compound keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. Therefore the child_fk_cols column may contain vectors of length greater than one.

See Also

Other foreign key functions: [dm_add_fk\(\)](#), [dm_enum_fk_candidates\(\)](#), [dm_get_fk\(\)](#), [dm_has_fk\(\)](#)

Examples

```
dm_nycflights13() %>%  
  dm_get_all_pks()
```

dm_get_all_pks	<i>Get all primary keys of a dm object</i>
----------------	--

Description

dm_get_all_pks() checks the dm object for set primary keys and returns the tables, the respective primary key columns and their classes.

Usage

```
dm_get_all_pks(dm)
```

Arguments

dm A dm object.

Value

A tibble with the following columns:

table table name,

pk_cols column name(s) of primary key.

Compound keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. Therefore the pk_cols column may contain vectors of length greater than one.

See Also

Other primary key functions: [dm_add_pk\(\)](#), [dm_get_pk\(\)](#), [dm_has_pk\(\)](#), [enum_pk_candidates\(\)](#)

Examples

```
dm_nycflights13() %>%  
  dm_get_all_pks()
```

dm_get_filters	<i>Get filter expressions</i>
----------------	-------------------------------

Description

dm_get_filters() returns the filter expressions that have been applied to a dm object. These filter expressions are not intended for evaluation, only for information.

Usage

```
dm_get_filters(x)
```

Arguments

x An object.

Value

A tibble with the following columns:

table table that was filtered,

filter the filter expression,

zoomed logical, does the filter condition relate to the zoomed table.

dm_get_fk	<i>Foreign key column names</i>
-----------	---------------------------------

Description

dm_get_fk() returns the names of the columns marked as foreign key of table table with respect to table ref_table within a dm object. If no foreign key is set between the tables, an empty character vector is returned.

Usage

```
dm_get_fk(dm, table, ref_table)
```

Arguments

dm A dm object.

table A table in the dm.

ref_table The table that is referenced from table.

Value

A list of character vectors with the column name(s) of `table`, pointing to the primary key of `ref_table`.

Compound keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. Therefore the function may return vectors of length greater than one in the future.

See Also

Other foreign key functions: [dm_add_fk\(\)](#), [dm_enum_fk_candidates\(\)](#), [dm_get_all_fks\(\)](#), [dm_has_fk\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_get_fk(flights, airports)
dm_nycflights13(cycle = TRUE) %>%
  dm_get_fk(flights, airports)
```

dm_get_pk	<i>Primary key column names</i>
-----------	---------------------------------

Description

`dm_get_pk()` returns the names of the columns marked as primary key of a table of a `dm` object. If no primary key is set for the table, an empty character vector is returned.

Usage

```
dm_get_pk(dm, table)
```

Arguments

<code>dm</code>	A dm object.
<code>table</code>	A table in the dm.

Value

A list with character vectors with the column name(s) of the primary keys of `table`.

Compound keys and multiple primary keys

Currently, keys consisting of more than one column are not supported. **This feature** is planned for dm 0.2.0. Therefore the function may return vectors of length greater than one in the future.

Similarly, each table currently can have only one primary key. This restriction may be lifted in the future. For this reason, and for symmetry with `dm_get_fk()`, this function returns a list of character vectors.

See Also

Other primary key functions: `dm_add_pk()`, `dm_get_all_pks()`, `dm_has_pk()`, `enum_pk_candidates()`

Examples

```
dm_nycflights13() %>%  
  dm_get_pk(flights)  
dm_nycflights13() %>%  
  dm_get_pk(planes)
```

`dm_get_referencing_tables`

Get the names of referencing tables

Description

This function returns the names of all tables that point to the primary key of a table.

Usage

```
dm_get_referencing_tables(dm, table)
```

Arguments

<code>dm</code>	A dm object.
<code>table</code>	A table in the dm.

Value

A character vector of the names of the tables that point to the primary key of table.

See Also

Other functions utilizing foreign key relations: `dm_is_referenced()`

Examples

```
dm_nycflights13() %>%
  dm_get_referencing_tables(airports)
dm_nycflights13() %>%
  dm_get_referencing_tables(flights)
```

dm_has_fk	<i>Check if foreign keys exists</i>
-----------	-------------------------------------

Description

dm_has_fk() checks if a foreign key reference exists between two tables in a dm.

Usage

```
dm_has_fk(dm, table, ref_table)
```

Arguments

dm	A dm object.
table	A table in the dm.
ref_table	The table to be checked if it is referred to.

Value

A boolean value: TRUE if a reference from table to ref_table exists, FALSE otherwise.

See Also

Other foreign key functions: [dm_add_fk\(\)](#), [dm_enum_fk_candidates\(\)](#), [dm_get_all_fks\(\)](#), [dm_get_fk\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_has_fk(flights, airports)
dm_nycflights13() %>%
  dm_has_fk(airports, flights)
```

dm_has_pk	<i>Check for primary key</i>
-----------	------------------------------

Description

dm_has_pk() checks if a given table has columns marked as its primary key.

Usage

```
dm_has_pk(dm, table)
```

Arguments

dm	A dm object.
table	A table in the dm.

Value

A logical value: TRUE if the given table has a primary key, FALSE otherwise.

See Also

Other primary key functions: [dm_add_pk\(\)](#), [dm_get_all_pks\(\)](#), [dm_get_pk\(\)](#), [enum_pk_candidates\(\)](#)

Examples

```
dm_nycflights13() %>%  
  dm_has_pk(flights)  
dm_nycflights13() %>%  
  dm_has_pk(planes)
```

dm_is_referenced	<i>Check foreign key reference</i>
------------------	------------------------------------

Description

Is a table of a [dm](#) referenced by another table?

Usage

```
dm_is_referenced(dm, table)
```

Arguments

dm A dm object.
 table A table in the dm.

Value

TRUE if at least one foreign key exists that points to the primary key of the table argument, FALSE otherwise.

See Also

Other functions utilizing foreign key relations: [dm_get_referencing_tables\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_is_referenced(airports)
dm_nycflights13() %>%
  dm_is_referenced(flights)
```

dm_join_to_tbl	<i>Join two tables</i>
----------------	------------------------

Description

A join of a desired type is performed between table_1 and table_2. The two tables need to be directly connected by a foreign key relation. Since this function is a wrapper around [dm_flatten_to_tbl\(\)](#), the LHS of the join will always be a "child table", i.e. a table referencing the other table.

Usage

```
dm_join_to_tbl(dm, table_1, table_2, join = left_join)
```

Arguments

dm A [dm](#) object.
 table_1 One of the tables involved in the join.
 table_2 The second table of the join.
 join The type of join to be performed, see [dplyr::join\(\)](#).

Value

The resulting table of the join.

See Also

Other flattening functions: [dm_flatten_to_tbl\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_join_to_tbl(airports, flights)

# same result is achieved with:
dm_nycflights13() %>%
  dm_join_to_tbl(flights, airports)

# this gives an error, because the tables are not directly linked to each other:
try(
  dm_nycflights13() %>%
    dm_join_to_tbl(airlines, airports)
)
```

dm_nrow	<i>Number of rows</i>
---------	-----------------------

Description

Returns a named vector with the number of rows for each table.

Usage

```
dm_nrow(dm)
```

Arguments

dm A [dm](#) object.

Value

A named vector with the number of rows for each table.

Examples

```
dm_nycflights13() %>%
  dm_filter(airports, faa %in% c("EWR", "LGA")) %>%
  dm_apply_filters() %>%
  dm_nrow()
```

dm_nycflights13	<i>Creates a dm object for the nycflights13 data</i>
-----------------	---

Description

Creates an example `dm` object from the tables in **nycflights13**, along with the references. See [nycflights13::flights](#) for a description of the data. As described in [nycflights13::planes](#), the relationship between the `flights` table and the `planes` tables is "weak", it does not satisfy data integrity constraints.

Usage

```
dm_nycflights13(cycle = FALSE, color = TRUE, subset = TRUE)
```

Arguments

<code>cycle</code>	Boolean. If <code>FALSE</code> (default), only one foreign key relation (from <code>flights\$origin</code> to <code>airports\$faa</code>) between the <code>flights</code> table and the <code>airports</code> table is established. If <code>TRUE</code> , a <code>dm</code> object with a double reference between those tables will be produced.
<code>color</code>	Boolean, if <code>TRUE</code> (default), the resulting <code>dm</code> object will have colors assigned to different tables for visualization with <code>dm_draw()</code> .
<code>subset</code>	Boolean, if <code>TRUE</code> (default), the <code>flights</code> table is reduced to flights with column <code>day</code> equal to 10.

Value

A `dm` object consisting of `nycflights13` tables, complete with primary and foreign keys and optionally colored.

Examples

```
dm_nycflights13() %>%
  dm_draw()
```

dm_paste	<i>Create R code for a dm object</i>
----------	--------------------------------------

Description

`dm_paste()` takes an existing `dm` and emits the code necessary for its creation.

Usage

```
dm_paste(dm, select = NULL, ..., tab_width = 2, options = NULL, path = NULL)
```

Arguments

dm	A dm object.
select	Deprecated, see "select" in the options argument.
...	Must be empty.
tab_width	Indentation width for code from the second line onwards
options	Formatting options. A character vector containing some of: <ul style="list-style-type: none"> "tables": <code>tibble()</code> calls for empty table definitions derived from <code>dm_ptype()</code>, overrides "select". "select": <code>dm_select()</code> statements for columns that are part of the dm. "keys": <code>dm_add_pk()</code> and <code>dm_add_fk()</code> statements for adding keys. "color": <code>dm_set_colors()</code> statements to set color. "all": All options above except "select" Default NULL is equivalent to <code>c("keys", "color")</code>
path	Output file, if NULL the code is printed to the console.

Details

The code emitted by the function reproduces the structure of the dm object. The options argument controls the level of detail: keys, colors, table definitions. Data in the tables is never included, see `dm_ptype()` for the underlying logic.

Value

Code for producing the prototype of the given dm.

Examples

```
dm() %>%
  dm_paste()
```

```
dm_nycflights13() %>%
  dm_paste()
```

```
dm_nycflights13() %>%
  dm_paste(options = "select")
```

dm_ptype	<i>Prototype for a dm object</i>
----------	----------------------------------

Description

Experimental

The prototype contains all tables, all primary and foreign keys, but no data. All tables are truncated and converted to zero-row tibbles. Column names retain their type. This is useful for performing creation and population of a database in separate steps.

Usage

```
dm_ptype(dm)
```

Arguments

dm	A dm object.
----	--------------

Examples

```
dm_financial() %>%  
  dm_ptype()
```

```
dm_financial() %>%  
  dm_ptype() %>%  
  dm_nrow()
```

dm_rename	<i>Rename columns</i>
-----------	-----------------------

Description

Rename the columns of your [dm](#) using syntax that is similar to `dplyr::rename()`.

Usage

```
dm_rename(dm, table, ...)
```

Arguments

dm	A dm object.
table	A table in the dm.
...	<p>One or more unquoted expressions separated by commas. You can treat variable names as if they were positions, and use expressions like <code>x:y</code> to select the ranges of variables.</p> <p>Use named arguments, e.g. <code>new_name = old_name</code>, to rename the selected variables.</p> <p>The arguments in <code>...</code> are automatically quoted and evaluated in a context where column names represent column positions. They also support unquoting and splicing. See <code>vignette("programming", package = "dplyr")</code> for an introduction to those concepts.</p> <p>See <code>select</code> helpers for more details, and the examples about <code>tidyselect</code> helpers, such as <code>starts_with()</code>, <code>everything()</code>, ...</p>

Details

If key columns are renamed, then the meta-information of the dm is updated accordingly.

Value

An updated dm with the columns of `table` renamed.

Examples

```
dm_nycflights13() %>%
  dm_rename(airports, code = faa, altitude = alt)
```

dm_rm_tbl

Remove tables

Description

Removes one or more tables from a [dm](#).

Usage

```
dm_rm_tbl(dm, ...)
```

Arguments

dm	A dm object.
...	One or more unquoted table names to remove from the dm. <code>tidyselect</code> is supported, see <code>dplyr::select()</code> for details on the semantics.

Value

The dm without the removed table(s) that were present in the initial dm.

See Also

[dm_add_tbl\(\)](#), [dm_select_tbl\(\)](#)

Examples

```
dm_nycflights13() %>%
  dm_rm_tbl(airports)
```

dm_select

Select columns

Description

Select columns of your `dm` using syntax that is similar to `dplyr::select()`.

Usage

```
dm_select(dm, table, ...)
```

Arguments

<code>dm</code>	A dm object.
<code>table</code>	A table in the dm.
<code>...</code>	<p>One or more unquoted expressions separated by commas. You can treat variable names as if they were positions, and use expressions like <code>x:y</code> to select the ranges of variables.</p> <p>Use named arguments, e.g. <code>new_name = old_name</code>, to rename the selected variables.</p> <p>The arguments in <code>...</code> are automatically quoted and evaluated in a context where column names represent column positions. They also support unquoting and splicing. See <code>vignette("programming", package = "dplyr")</code> for an introduction to those concepts.</p> <p>See <code>select</code> helpers for more details, and the examples about <code>tidyselect</code> helpers, such as <code>starts_with()</code>, <code>everything()</code>, ...</p>

Details

If key columns are renamed, then the meta-information of the `dm` is updated accordingly. If key columns are removed, then all related relations are dropped as well.

Value

An updated dm with the columns of table reduced and/or renamed.

Examples

```
dm_nycflights13() %>%  
  dm_select(airports, code = faa, altitude = alt)
```

dm_select_tbl	<i>Select and rename tables</i>
---------------	---------------------------------

Description

dm_select_tbl() keeps the selected tables and their relationships, optionally renaming them.

dm_rename_tbl() renames tables.

Usage

```
dm_select_tbl(dm, ...)
```

```
dm_rename_tbl(dm, ...)
```

Arguments

dm A [dm](#) object.

... One or more table names of the tables of the [dm](#) object. tidyselect is supported, see [dplyr::select\(\)](#) for details on the semantics.

Value

The input dm with tables renamed or removed.

See Also

[dm_rm_tbl\(\)](#)

Examples

```
dm_nycflights13() %>%  
  dm_select_tbl(airports, fl = flights)
```

```
dm_nycflights13() %>%  
  dm_rename_tbl(ap = airports, fl = flights)
```

dm_zoom_to

*Mark table for manipulation***Description**

Zooming to a table of a `dm` allows for the use of many `dplyr`-verbs directly on this table, while retaining the context of the `dm` object.

`dm_zoom_to()` zooms to the given table.

`dm_update_zoomed()` overwrites the originally zoomed table with the manipulated table. The filter conditions for the zoomed table are added to the original filter conditions.

`dm_insert_zoomed()` adds a new table to the `dm`.

`dm_discard_zoomed()` discards the zoomed table and returns the `dm` as it was before zooming.

Please refer to `vignette("dm-zoom-to-table", package = "dm")` for a more detailed introduction.

Usage

```
dm_zoom_to(dm, table)
```

```
dm_insert_zoomed(dm, new_tbl_name = NULL, repair = "unique", quiet = FALSE)
```

```
dm_update_zoomed(dm)
```

```
dm_discard_zoomed(dm)
```

Arguments

<code>dm</code>	A <code>dm</code> object.
<code>table</code>	A table in the <code>dm</code> .
<code>new_tbl_name</code>	Name of the new table.
<code>repair</code>	<p>Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown.</p> <ul style="list-style-type: none"> Minimal names are never <code>NULL</code> or <code>NA</code>. When an element doesn't have a name, its minimal name is an empty string. Unique names are unique. A suffix is appended to duplicate names to make them unique. Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error. <p>The "check_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria.</p>
<code>quiet</code>	By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set <code>quiet</code> to <code>TRUE</code> to silence the messages.

Details

Whenever possible, the key relations of the original table are transferred to the resulting table when using `dm_insert_zoomed()` or `dm_update_zoomed()`.

Functions from `dplyr` that are supported for a `zoomed_dm`: `group_by()`, `summarise()`, `mutate()`, `transmute()`, `filter()`, `select()`, `rename()` and `ungroup()`. You can use these functions just like you would with a normal table.

In addition to filtering the zoomed table, the filter condition from `filter()` is also stored in the `dm`. Depending on which function you use to return to a normal `dm`, one of the following happens:

1. `dm_discard_zoomed()`: all filter conditions for the zoomed table are discarded
2. `dm_update_zoomed()`: the filter conditions of the original table and those of the zoomed table are combined
3. `dm_insert_zoomed()`: the filter conditions of the original table stay there and those of the zoomed table are transferred to the new table of the `dm`

Furthermore, the different `join()`-variants from `dplyr` are also supported (apart from `nest_join()`). The `join`-methods for `zoomed_dm` have an extra argument `select` that allows choosing the columns of the RHS table.

And – last but not least – also the `tidyr`-functions `unite()` and `separate()` are supported for `zoomed_dm`.

Value

For `dm_zoom_to()`: A `zoomed_dm` object.

For `dm_insert_zoomed()`, `dm_update_zoomed()` and `dm_discard_zoomed()`: A `dm` object.

Examples

```
flights_zoomed <- dm_zoom_to(dm_nycflights13(), flights)

flights_zoomed

flights_zoomed_transformed <-
  flights_zoomed %>%
  mutate(am_pm_dep = ifelse(dep_time < 1200, "am", "pm")) %>%
  # `by`-argument of `left_join()` can be explicitly given
  # otherwise the key-relation is used
  left_join(airports) %>%
  select(year:dep_time, am_pm_dep, everything())

flights_zoomed_transformed

# replace table `flights` with the zoomed table
flights_zoomed_transformed %>%
  dm_update_zoomed()

# insert the zoomed table as a new table
flights_zoomed_transformed %>%
```

```

dm_insert_zoomed("extended_flights") %>%
  dm_draw()

# discard the zoomed table
flights_zoomed_transformed %>%
  dm_discard_zoomed()

```

dplyr_join

dplyr join methods for zoomed dm objects

Description

Use these methods without the '.zoomed_dm' suffix (see examples).

Usage

```

## S3 method for class 'zoomed_dm'
left_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'zoomed_dm'
inner_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'zoomed_dm'
full_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'zoomed_dm'
right_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'zoomed_dm'
semi_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'zoomed_dm'
anti_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

```

Arguments

x, y	tbls to join. x is the zoomed_dm and y is another table in the dm.
by	If left NULL (default), the join will be performed by via the foreign key relation that exists between the originally zoomed table (now x) and the other table (y). If you provide a value (for the syntax see dplyr::join), you can also join tables that are not connected in the dm.
copy	Disabled, since all tables in a dm are by definition on the same src.
suffix	Disabled, since columns are disambiguated automatically if necessary, changing the column names to table_name.column_name.

`select` Select a subset of the **RHS-table**'s columns, the syntax being `select = c(col_1, col_2, col_3)` (unquoted or quoted). This argument is specific for the join-methods for `zoomed_dm`. The table's by column(s) are automatically added if missing in the selection.

`...` see [dplyr::join](#)

Examples

```
flights_dm <- dm_nycflights13()
dm_zoom_to(flights_dm, flights) %>%
  left_join(airports, select = c(faa, name))

# this should illustrate that tables don't necessarily need to be connected
dm_zoom_to(flights_dm, airports) %>%
  semi_join(airlines, by = "name")
```

dplyr_table_manipulation

dplyr table manipulation methods for zoomed dm objects

Description

Use these methods without the `zoomed_dm` suffix (see examples).

Usage

```
## S3 method for class 'zoomed_dm'
filter(.data, ...)

## S3 method for class 'zoomed_dm'
mutate(.data, ...)

## S3 method for class 'zoomed_dm'
transmute(.data, ...)

## S3 method for class 'zoomed_dm'
select(.data, ...)

## S3 method for class 'zoomed_dm'
rename(.data, ...)

## S3 method for class 'zoomed_dm'
distinct(.data, ..., .keep_all = FALSE)

## S3 method for class 'zoomed_dm'
arrange(.data, ...)
```

```
## S3 method for class 'zoomed_dm'
slice(.data, ..., .keep_pk = NULL)

## S3 method for class 'zoomed_dm'
group_by(.data, ...)

## S3 method for class 'zoomed_dm'
ungroup(x, ...)

## S3 method for class 'zoomed_dm'
summarise(.data, ...)

## S3 method for class 'zoomed_dm'
pull(.data, var = -1, ...)
```

Arguments

<code>.data</code>	object of class <code>zoomed_dm</code>
<code>...</code>	see corresponding function in package dplyr or tidyr
<code>.keep_all</code>	For <code>distinct.zoomed_dm()</code> : see dplyr::distinct
<code>.keep_pk</code>	For <code>slice.zoomed_dm</code> : Logical, if TRUE, the primary key will be retained during this transformation. If FALSE, it will be dropped. By default, the value is NULL, which causes the function to issue a message in case a primary key is available for the zoomed table. This argument is specific for the <code>slice.zoomed_dm()</code> method.
<code>x</code>	For <code>ungroup.zoomed_dm</code> : object of class <code>zoomed_dm</code>
<code>var</code>	A variable specified as: <ul style="list-style-type: none"> • a literal variable name • a positive integer, giving the position counting from the left • a negative integer, giving the position counting from the right. <p>The default returns the last column (on the assumption that's the column you've created most recently).</p> <p>This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).</p>

Examples

```
zoomed <- dm_nycflights13() %>%
  dm_zoom_to(flights) %>%
  group_by(month) %>%
  arrange(desc(day)) %>%
  summarize(avg_air_time = mean(air_time, na.rm = TRUE))
zoomed
dm_insert_zoomed(zoomed, new_tbl_name = "avg_air_time_per_month")
```

enum_pk_candidates	<i>Primary key candidate</i>
--------------------	------------------------------

Description

Questioning

enum_pk_candidates() checks for each column of a table if the column contains only unique values, and is thus a suitable candidate for a primary key of the table.

dm_enum_pk_candidates() performs these checks for a table in a [dm](#) object.

Usage

```
enum_pk_candidates(table)
```

```
dm_enum_pk_candidates(dm, table)
```

Arguments

table A table in the dm.

dm A dm object.

Value

A tibble with the following columns:

columns columns of table,

candidate boolean: are these columns a candidate for a primary key,

why if not a candidate for a primary key column, explanation for this.

Life cycle

These functions are marked "questioning" because we are not yet sure about the interface, in particular if we need both dm_enum...() and enum...() variants. Changing the interface later seems harmless because these functions are most likely used interactively.

See Also

Other primary key functions: [dm_add_pk\(\)](#), [dm_get_all_pks\(\)](#), [dm_get_pk\(\)](#), [dm_has_pk\(\)](#)

Examples

```
nycflights13::flights %>%  
  enum_pk_candidates()
```

```
dm_nycflights13() %>%
  dm_enum_pk_candidates(airports)
```

examine_cardinality *Check table relations*

Description

All `check_cardinality_?_?()` functions test the following conditions:

1. Is `pk_column` is a unique key for `parent_table`?
2. Is the set of values in `fk_column` of `child_table` a subset of the set of values of `pk_column`?
3. Does the relation between the two tables of the data model meet the cardinality requirements?

`examine_cardinality()` also checks the first two points and subsequently determines the type of cardinality.

Usage

```
check_cardinality_0_n(parent_table, pk_column, child_table, fk_column)
```

```
check_cardinality_1_n(parent_table, pk_column, child_table, fk_column)
```

```
check_cardinality_1_1(parent_table, pk_column, child_table, fk_column)
```

```
check_cardinality_0_1(parent_table, pk_column, child_table, fk_column)
```

```
examine_cardinality(parent_table, pk_column, child_table, fk_column)
```

Arguments

<code>parent_table</code>	Data frame.
<code>pk_column</code>	Column of <code>parent_table</code> that has to be one of its unique keys.
<code>child_table</code>	Data frame.
<code>fk_column</code>	Column of <code>child_table</code> that has to be a foreign key to <code>pk_column</code> in <code>parent_table</code> .

Details

All cardinality-functions accept a parent table (data frame), a column name of this table, a child table, and a column name of the child table. The given column of the parent table has to be one of its unique keys (no duplicates are allowed). Furthermore, in all cases, the set of values of the child table's column has to be a subset of the set of values of the parent table's column.

The cardinality specifications `0_n`, `1_n`, `0_1`, `1_1` refer to the expected relation that the child table has with the parent table. The numbers `0`, `1` and `n` refer to the number of values in the column of the child table that correspond to each value of the column of the parent table. `n` means "more than one" in this context, with no upper limit.

0_n means, that each value of the pk_column has at least 0 and at most n corresponding values in the column of the child table (which translates to no further restrictions).

1_n means, that each value of the pk_column has at least 1 and at most n corresponding values in the column of the child table. This means that there is a "surjective" mapping from the child table to the parent table w.r.t. the specified columns, i.e. for each parent table column value there exists at least one equal child table column value.

0_1 means, that each value of the pk_column has at least 0 and at most 1 corresponding values in the column of the child table. This means that there is a "injective" mapping from the child table to the parent table w.r.t. the specified columns, i.e. no parent table column value is addressed multiple times. But not all of the parent table column values have to be referred to.

1_1 means, that each value of the pk_column has exactly 1 corresponding value in the column of the child table. This means that there is a "bijective" ("injective" AND "surjective") mapping between the child table and the parent table w.r.t. the specified columns, i.e. the sets of values of the two columns are equal and there are no duplicates in either of them.

Finally, examine_cardinality() tests for and returns the nature of the relationship (injective, surjective, bijective, or none of these) between the two given columns. If either pk_column is not a unique key of parent_table or the values of fk_column are not a subset of the values in pk_column, the requirements for a cardinality test is not fulfilled. No error will be thrown, but the result will contain the information which prerequisite was violated.

Value

For check_cardinality_?_?(): Functions return parent_table, invisibly, if the check is passed, to support pipes. Otherwise an error is thrown and the reason for it is explained.

For examine_cardinality(): Returns a character variable specifying the type of relationship between the two columns.

Examples

```
d1 <- tibble::tibble(a = 1:5)
d2 <- tibble::tibble(c = c(1:5, 5))
d3 <- tibble::tibble(c = 1:4)
# This does not pass, `c` is not unique key of d2:
try(check_cardinality_0_n(d2, c, d1, a))

# This passes, multiple values in d2$c are allowed:
check_cardinality_0_n(d1, a, d2, c)

# This does not pass, injectivity is violated:
try(check_cardinality_1_1(d1, a, d2, c))

# This passes:
check_cardinality_0_1(d1, a, d3, c)

# Returns the kind of cardinality
examine_cardinality(d1, a, d2, c)
```

head.zoomed_dm **utils** *table manipulation methods for zoomed_dm objects*

Description

Extract the first or last rows from a table. Use these methods without the '.zoomed_dm' suffix (see examples). The methods for regular dm objects extract the first or last tables.

Usage

```
## S3 method for class 'zoomed_dm'
head(x, n = 6L, ...)

## S3 method for class 'zoomed_dm'
tail(x, n = 6L, ...)
```

Arguments

x object of class zoomed_dm

n an integer vector of length up to dim(x) (or 1, for non-dimensioned objects). Values specify the indices to be selected in the corresponding dimension (or along the length) of the object. A positive value of n[i] includes the first/last n[i] indices in that dimension, while a negative value excludes the last/first abs(n[i]), including all remaining indices. NA or non-specified values (when length(n) < length(dim(x))) select all indices in that dimension. Must contain at least one non-missing value.

... arguments to be passed to or from other methods.

Details

see manual for the corresponding functions in **utils**.

Value

A zoomed_dm object.

Examples

```
zoomed <- dm_nycflights13() %>%
  dm_zoom_to(flights) %>%
  head(4)
zoomed
dm_insert_zoomed(zoomed, new_tbl_name = "head_flights")
```

pull_tbl	<i>Retrieve a table</i>
----------	-------------------------

Description

This function has methods for both dm classes:

1. With `pull_tbl.dm()` you can chose which table of the dm you want to retrieve.
2. With `pull_tbl.zoomed_dm()` you will retrieve the zoomed table in the current state.

Usage

```
pull_tbl(dm, table)
```

Arguments

dm	A dm object.
table	One unquoted table name for <code>pull_tbl.dm()</code> , ignored for <code>pull_tbl.zoomed_dm()</code> .

Value

The requested table

Examples

```
# For an unzoomed dm you need to specify the table to pull:
dm_nycflights13() %>%
  pull_tbl(airports)

# If zoomed, pulling detaches the zoomed table from the dm:
dm_nycflights13() %>%
  dm_zoom_to(airports) %>%
  pull_tbl()
```

reunite_parent_child	<i>Merge two tables that are linked by a foreign key relation</i>
----------------------	---

Description**Questioning**

Perform table fusion by combining two tables by a common (key) column, and then removing this column.

`reunite_parent_child()`: After joining the two tables by the column `id_column`, this column will be removed. The transformation is roughly the inverse of what `decompose_table()` does.

`reunite_parent_child_from_list()`: After joining the two tables by the column `id_column`, `id_column` is removed.

This function is almost exactly the inverse of `decompose_table()` (the order of the columns is not retained, and the original row names are lost).

Usage

```
reunite_parent_child(child_table, parent_table, id_column)
```

```
reunite_parent_child_from_list(list_of_parent_child_tables, id_column)
```

Arguments

<code>child_table</code>	Table (possibly created by <code>decompose_table()</code>) that references <code>parent_table</code>
<code>parent_table</code>	Table (possibly created by <code>decompose_table()</code>).
<code>id_column</code>	Identical name of referencing / referenced column in <code>child_table</code> / <code>parent_table</code> .
<code>list_of_parent_child_tables</code>	Cf arguments <code>child_table</code> and <code>parent_table</code> from <code>reunite_parent_child()</code> , but both in a named list (as created by <code>decompose_table()</code>).

Value

A wide table produced by joining the two given tables.

Life cycle

These functions are marked "questioning" because they feel more useful when applied to a table in a dm object.

See Also

Other table surgery functions: [decompose_table\(\)](#)

Examples

```
decomposed_table <- decompose_table(mtcars, new_id, am, gear, carb)
ct <- decomposed_table$child_table
pt <- decomposed_table$parent_table

reunite_parent_child(ct, pt, new_id)
reunite_parent_child_from_list(decomposed_table, new_id)
```

Description

Experimental

These methods provide a framework for manipulating individual rows in existing tables. All operations expect that both existing and new data are presented in two compatible `tbl` objects.

If `y` lives on a different data source than `x`, it can be copied automatically by setting `copy = TRUE`, just like for `dplyr::left_join()`.

On mutable backends like databases, these operations manipulate the underlying storage. In contrast to all other operations, these operations may lead to irreversible changes to the underlying database. Therefore, in-place updates must be requested explicitly with `in_place = TRUE`. By default, an informative message is given. Unlike `compute()` or `copy_to()`, no new tables are created.

The `sql_rows_*`() functions return the SQL used for the corresponding `rows_*`() function with `in_place = FALSE`. `y` needs to be located on the same data source as `x`.

Usage

```
## S3 method for class 'tbl_dbi'
rows_insert(x, y, by = NULL, ..., in_place = NULL, copy = FALSE, check = NULL)

## S3 method for class 'tbl_dbi'
rows_update(x, y, by = NULL, ..., in_place = NULL, copy = FALSE, check = NULL)

sql_rows_insert(x, y, ...)

sql_rows_update(x, y, by, ...)
```

Arguments

<code>x</code>	A pair of data frames or data frame extensions (e.g. a tibble). <code>y</code> must have the same columns of <code>x</code> or a subset.
<code>y</code>	A pair of data frames or data frame extensions (e.g. a tibble). <code>y</code> must have the same columns of <code>x</code> or a subset.
<code>by</code>	An unnamed character vector giving the key columns. The key values must uniquely identify each row (i.e. each combination of key values occurs at most once), and the key columns must exist in both <code>x</code> and <code>y</code> . By default, we use the first column in <code>y</code> , since the first column is a reasonable place to put an identifier variable.
<code>...</code>	Other parameters passed onto methods.
<code>in_place</code>	Should <code>x</code> be modified in place? This argument is only relevant for mutable backends (e.g. databases, <code>data.tables</code>). When <code>TRUE</code> , a modified version of <code>x</code> is returned invisibly; when <code>FALSE</code> , a new object representing the resulting changes is returned.

copy	If <code>x</code> and <code>y</code> are not from the same data source, and <code>copy</code> is <code>TRUE</code> , then <code>y</code> will be copied into the same <code>src</code> as <code>x</code> . This allows you to join tables across <code>srcs</code> , but it is a potentially expensive operation so you must opt into it.
check	Set to <code>TRUE</code> to always check keys, or <code>FALSE</code> to never check. The default is to check only if <code>in_place</code> is <code>TRUE</code> or <code>NULL</code> . Currently these checks are no-ops and need yet to be implemented.

Value

A `tbl` object of the same structure as `x`. If `in_place = TRUE`, the underlying data is updated as a side effect, and `x` is returned, invisibly.

Examples

```
data <- dbplyr::memdb_frame(a = 1:3, b = letters[c(1:2, NA)], c = 0.5 + 0:2)
data

try(rows_insert(data, tibble::tibble(a = 4, b = "z")))
rows_insert(data, tibble::tibble(a = 4, b = "z"), copy = TRUE)
rows_update(data, tibble::tibble(a = 2:3, b = "w"), copy = TRUE, in_place = FALSE)

rows_insert(data, dbplyr::memdb_frame(a = 4, b = "z"), in_place = TRUE)
data
rows_update(data, dbplyr::memdb_frame(a = 2:3, b = "w"), in_place = TRUE)
data
```

rows-dm

*Modifying rows for multiple tables***Description****Experimental**

These functions provide a framework for updating data in existing tables. Unlike `compute()`, `copy_to()` or `copy_dm_to()`, no new tables are created on the database. All operations expect that both existing and new data are presented in two compatible `dm` objects on the same data source.

The functions make sure that the tables in the target `dm` are processed in topological order so that parent (dimension) tables receive insertions before child (fact) tables.

These operations, in contrast to all other operations, may lead to irreversible changes to the underlying database. Therefore, in-place operation must be requested explicitly with `in_place = TRUE`. By default, an informative message is given.

`dm_rows_insert()` adds new records via `rows_insert()`. The primary keys must differ from existing records. This must be ensured by the caller and might be checked by the underlying database. Use `in_place = FALSE` and apply `dm_examine_constraints()` to check beforehand.

`dm_rows_update()` updates existing records via `rows_update()`. Primary keys must match for all records to be updated.

`dm_rows_patch()` updates missing values in existing records via `rows_patch()`. Primary keys must match for all records to be patched.

`dm_rows_upsert()` updates existing records and adds new records, based on the primary key, via `rows_upsert()`.

`dm_rows_delete()` removes matching records via `rows_delete()`, based on the primary key. The order in which the tables are processed is reversed.

`dm_rows_truncate()` removes all records via `rows_truncate()`, only for tables in `dm`. The order in which the tables are processed is reversed.

Usage

```
dm_rows_insert(x, y, ..., in_place = NULL)
```

```
dm_rows_update(x, y, ..., in_place = NULL)
```

```
dm_rows_patch(x, y, ..., in_place = NULL)
```

```
dm_rows_upsert(x, y, ..., in_place = NULL)
```

```
dm_rows_delete(x, y, ..., in_place = NULL)
```

```
dm_rows_truncate(x, y, ..., in_place = NULL)
```

Arguments

<code>x</code>	Target dm object.
<code>y</code>	dm object with new data.
<code>...</code>	Must be empty.
<code>in_place</code>	Should <code>x</code> be modified in place? This argument is only relevant for mutable backends (e.g. databases, <code>data.tables</code>). When <code>TRUE</code> , a modified version of <code>x</code> is returned invisibly; when <code>FALSE</code> , a new object representing the resulting changes is returned.

Value

A dm object of the same `dm_ptype()` as `x`. If `in_place = TRUE`, the underlying data is updated as a side effect, and `x` is returned, invisibly.

Examples

```
# Establish database connection:
sqlite <- DBI::dbConnect(RSQLite::SQLite())

# Entire dataset with all dimension tables populated
# with flights and weather data truncated:
flights_init <-
  dm_nycflights13() %>%
```

```
dm_zoom_to(flights) %>%
filter(FALSE) %>%
dm_update_zoomed() %>%
dm_zoom_to(weather) %>%
filter(FALSE) %>%
dm_update_zoomed()

# Target database:
flights_sqlite <- copy_dm_to(sqlite, flights_init, temporary = FALSE)
print(dm_nrow(flights_sqlite))

# First update:
flights_jan <-
  dm_nycflights13() %>%
  dm_select_tbl(flights, weather) %>%
  dm_zoom_to(flights) %>%
  filter(month == 1) %>%
  dm_update_zoomed() %>%
  dm_zoom_to(weather) %>%
  filter(month == 1) %>%
  dm_update_zoomed()
print(dm_nrow(flights_jan))

# Copy to temporary tables on the target database:
flights_jan_sqlite <- copy_dm_to(sqlite, flights_jan)

# Dry run by default:
dm_rows_insert(flights_sqlite, flights_jan_sqlite)
print(dm_nrow(flights_sqlite))

# Explicitly request persistence:
dm_rows_insert(flights_sqlite, flights_jan_sqlite, in_place = TRUE)
print(dm_nrow(flights_sqlite))

# Second update:
flights_feb <-
  dm_nycflights13() %>%
  dm_select_tbl(flights, weather) %>%
  dm_zoom_to(flights) %>%
  filter(month == 2) %>%
  dm_update_zoomed() %>%
  dm_zoom_to(weather) %>%
  filter(month == 2) %>%
  dm_update_zoomed()

# Copy to temporary tables on the target database:
flights_feb_sqlite <- copy_dm_to(sqlite, flights_feb)

# Explicit dry run:
flights_new <- dm_rows_insert(
  flights_sqlite,
  flights_feb_sqlite,
  in_place = FALSE
```



```

)
print(dm_nrow(flights_new))
print(dm_nrow(flights_sqlite))

# Check for consistency before applying:
flights_new %>%
  dm_examine_constraints()

# Apply:
dm_rows_insert(flights_sqlite, flights_feb_sqlite, in_place = TRUE)
print(dm_nrow(flights_sqlite))

DBI::dbDisconnect(sqlite)

```

rows_truncate	<i>Truncate all rows</i>
---------------	--------------------------

Description

rows_truncate() removes all rows. This operation corresponds to TRUNCATE in SQL. ... is ignored.

Usage

```

rows_truncate(x, ..., in_place = FALSE)

sql_rows_truncate(x, ...)

```

Arguments

x	A data frame or data frame extension (e.g. a tibble).
...	Other parameters passed onto methods.
in_place	Should x be modified in place? This argument is only relevant for mutable backends (e.g. databases, data.tables). When TRUE, a modified version of x is returned invisibly; when FALSE, a new object representing the resulting changes is returned.

tbl.dm

dplyr table retrieval, table info and DB interaction methods**Description**

Use these methods without the '.dm' or '.zoomed_dm' suffix (see examples).

Usage

```
## S3 method for class 'dm'
tbl(src, from, ...)

## S3 method for class 'dm'
compute(x, ...)

## S3 method for class 'zoomed_dm'
compute(x, ...)

## S3 method for class 'dm'
src_tbls(x, ...)

## S3 method for class 'dm'
copy_to(
  dest,
  df,
  name = deparse(substitute(df)),
  overwrite = FALSE,
  temporary = TRUE,
  repair = "unique",
  quiet = FALSE,
  ...
)

## S3 method for class 'dm'
collect(x, ...)
```

Arguments

src	A dm object.
from	A length one character variable containing the name of the requested table
...	See original function documentation
x	Either a dm or a zoomed_dm; the latter leads to an error for src_tbls.dm()
dest	For copy_to.dm(): The dm object to which a table should be copied.
df	For copy_to.dm(): A table (can be on a different src)
name	For copy_to.dm(): See dplyr::copy_to

overwrite	For <code>copy_to.dm()</code> : See <code>dplyr::copy_to</code> ; TRUE leads to an error
temporary	For <code>copy_to.dm()</code> : If the dm is on a DB, the copied version of df will only be written temporarily to the DB. After the connection is reset it will no longer be available.
repair, quiet	Name repair options; cf. <code>vctrs::vec_as_names</code>

tidyr_table_manipulation

tidyr table manipulation methods for zoomed dm objects

Description

Use these methods without the `'.zoomed_dm'` suffix (see examples).

Usage

```
## S3 method for class 'zoomed_dm'
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)

## S3 method for class 'zoomed_dm'
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, ...)
```

Arguments

data	object of class <code>zoomed_dm</code>
col	For <code>unite.zoomed_dm</code> : see <code>tidyr::unite</code> For <code>separate.zoomed_dm</code> : see <code>tidyr::separate</code>
...	For <code>unite.zoomed_dm</code> : see <code>tidyr::unite</code> For <code>separate.zoomed_dm</code> : see <code>tidyr::separate</code>
sep	For <code>unite.zoomed_dm</code> : see <code>tidyr::unite</code> For <code>separate.zoomed_dm</code> : see <code>tidyr::separate</code>
remove	For <code>unite.zoomed_dm</code> : see <code>tidyr::unite</code> For <code>separate.zoomed_dm</code> : see <code>tidyr::separate</code>
na.rm	see <code>tidyr::unite</code>
into	see <code>tidyr::separate</code>

Examples

```
zoom_united <- dm_nycflights13() %>%
  dm_zoom_to(flights) %>%
  select(year, month, day) %>%
  unite("month_day", month, day)
zoom_united
zoom_united %>%
  separate(month_day, c("month", "day"))
```

Index

- * **DB interaction functions**
 - copy_dm_to, 5
- * **flattening functions**
 - dm_flatten_to_tbl, 24
 - dm_join_to_tbl, 34
- * **foreign key functions**
 - dm_add_fk, 11
 - dm_enum_fk_candidates, 19
 - dm_get_all_fks, 27
 - dm_get_fk, 29
 - dm_has_fk, 32
- * **functions utilizing foreign key relations**
 - dm_get_referencing_tables, 31
 - dm_is_referenced, 33
- * **primary key functions**
 - dm_add_pk, 12
 - dm_get_all_pks, 28
 - dm_get_pk, 30
 - dm_has_pk, 33
 - enum_pk_candidates, 47
- * **table surgery functions**
 - decompose_table, 7
 - reunite_parent_child, 51
- anti_join.zoomed_dm (dplyr_join), 44
- arrange.zoomed_dm
 - (dplyr_table_manipulation), 45
- as_dm (dm), 9
- as_tibble(), 21
- check_cardinality_0_1
 - (examine_cardinality), 48
- check_cardinality_0_n
 - (examine_cardinality), 48
- check_cardinality_1_1
 - (examine_cardinality), 48
- check_cardinality_1_n
 - (examine_cardinality), 48
- check_key, 3
- check_key(), 10
- check_set_equality, 4
- check_subset, 5
- check_subset(), 10
- collect_dm (tbl_dm), 58
- compute(), 53, 54
- compute_dm (tbl_dm), 58
- compute_zoomed_dm (tbl_dm), 58
- copy_dm_to, 5
- copy_dm_to(), 10, 54
- copy_to(), 53, 54
- copy_to_dm (tbl_dm), 58
- DBI::DBIConnection, 5, 9, 10, 26
- DBI::dbQuoteIdentifier(), 6, 7
- decompose_table, 7, 52
- decompose_table(), 10
- DiagrammeR::grViz(), 18
- distinct.zoomed_dm
 - (dplyr_table_manipulation), 45
- dm, 5, 9, 13, 14, 18, 19, 22, 24–30, 33–36, 38–42, 47, 54
- dm_add_fk, 11, 20, 27, 30, 32
- dm_add_fk(), 10, 37
- dm_add_pk, 12, 28, 31, 33, 47
- dm_add_pk(), 10, 37
- dm_add_tbl, 14
- dm_add_tbl(), 40
- dm_apply_filters (dm_filter), 22
- dm_apply_filters_to_tbl (dm_filter), 22
- dm_bind, 15
- dm_disambiguate_cols, 16
- dm_discard_zoomed (dm_zoom_to), 42
- dm_draw, 17
- dm_draw(), 10
- dm_enum_fk_candidates, 12, 19, 27, 30, 32
- dm_enum_pk_candidates
 - (enum_pk_candidates), 47
- dm_examine_constraints, 21
- dm_examine_constraints(), 54
- dm_filter, 22

- dm_filter(), 10
- dm_financial, 24
- dm_financial_sqlite(dm_financial), 24
- dm_flatten_to_tbl, 24, 35
- dm_flatten_to_tbl(), 34
- dm_from_src, 26
- dm_from_src(), 10
- dm_get_all_fks, 12, 20, 27, 30, 32
- dm_get_all_pks, 13, 28, 31, 33, 47
- dm_get_available_colors(dm_draw), 17
- dm_get_colors(dm_draw), 17
- dm_get_con(dm), 9
- dm_get_filters, 29
- dm_get_fk, 12, 20, 27, 29, 32
- dm_get_pk, 13, 28, 30, 33, 47
- dm_get_referencing_tables, 31, 34
- dm_get_src(dm), 9
- dm_get_tables(dm), 9
- dm_has_fk, 12, 20, 27, 30, 32
- dm_has_pk, 13, 28, 31, 33, 47
- dm_insert_zoomed(dm_zoom_to), 42
- dm_is_referenced, 31, 33
- dm_join_to_tbl, 26, 34
- dm_join_to_tbl(), 10
- dm_nrow, 35
- dm_nycflights13, 36
- dm_nycflights13(), 10
- dm_paste, 36
- dm_ptype, 38
- dm_ptype(), 37, 55
- dm_rename, 38
- dm_rename_tbl(dm_select_tbl), 41
- dm_rm_fk(dm_add_fk), 11
- dm_rm_pk(dm_add_pk), 12
- dm_rm_tbl, 39
- dm_rm_tbl(), 15, 41
- dm_rows_delete(rows-dm), 54
- dm_rows_insert(rows-dm), 54
- dm_rows_patch(rows-dm), 54
- dm_rows_truncate(rows-dm), 54
- dm_rows_update(rows-dm), 54
- dm_rows_upsert(rows-dm), 54
- dm_select, 40
- dm_select(), 37
- dm_select_tbl, 41
- dm_select_tbl(), 10, 40
- dm_set_colors(dm_draw), 17
- dm_set_colors(), 37
- dm_squash_to_tbl(dm_flatten_to_tbl), 24
- dm_update_zoomed(dm_zoom_to), 42
- dm_zoom_to, 42
- dm_zoom_to(), 23
- dplyr::copy_to, 58, 59
- dplyr::copy_to(), 6
- dplyr::distinct, 46
- dplyr::filter(), 22, 23
- dplyr::join, 44, 45
- dplyr::join(), 25, 34
- dplyr::left_join(), 53
- dplyr::select(), 18, 25, 39, 41
- dplyr::semi_join(), 23
- dplyr::src_dbi, 5
- dplyr_join, 44
- dplyr_table_manipulation, 45
- enum_fk_candidates
 - (dm_enum_fk_candidates), 19
- enum_pk_candidates, 13, 28, 31, 33, 47
- examine_cardinality, 48
- examine_cardinality(), 10
- filter.zoomed_dm
 - (dplyr_table_manipulation), 45
- full_join.zoomed_dm(dplyr_join), 44
- group_by.zoomed_dm
 - (dplyr_table_manipulation), 45
- head.zoomed_dm, 50
- inner_join.zoomed_dm(dplyr_join), 44
- is_dm(dm), 9
- left_join.zoomed_dm(dplyr_join), 44
- mutate(), 14
- mutate.zoomed_dm
 - (dplyr_table_manipulation), 45
- new_dm(dm), 9
- nycflights13::flights, 36
- nycflights13::planes, 36
- pull.zoomed_dm
 - (dplyr_table_manipulation), 45
- pull_tbl, 51
- quasiquote, 46

rename.zoomed_dm
 (dplyr_table_manipulation), 45

reunite_parent_child, 8, 51

reunite_parent_child_from_list
 (reunite_parent_child), 51

right_join.zoomed_dm (dplyr_join), 44

rlang::as_function(), 6

rows-db, 53

rows-dm, 54

rows_delete(), 55

rows_insert(), 54

rows_insert.tbl_dbi (rows-db), 53

rows_patch(), 55

rows_truncate, 57

rows_truncate(), 55

rows_update(), 54

rows_update.tbl_dbi (rows-db), 53

rows_upsert(), 55

select.zoomed_dm
 (dplyr_table_manipulation), 45

semi_join.zoomed_dm (dplyr_join), 44

separate.zoomed_dm
 (tidyr_table_manipulation), 59

slice.zoomed_dm
 (dplyr_table_manipulation), 45

sql_rows_insert (rows-db), 53

sql_rows_truncate (rows_truncate), 57

sql_rows_update (rows-db), 53

src, 26

src_tbls.dm (tbl.dm), 58

summarise.zoomed_dm
 (dplyr_table_manipulation), 45

tail.zoomed_dm (head.zoomed_dm), 50

tbl, 9, 53

tbl.dm, 58

tibble(), 37

tidyr::separate, 59

tidyr::unite, 59

tidyr_table_manipulation, 59

transmute.zoomed_dm
 (dplyr_table_manipulation), 45

ungroup.zoomed_dm
 (dplyr_table_manipulation), 45

unite.zoomed_dm
 (tidyr_table_manipulation), 59

validate_dm (dm), 9

vctrs::vec_as_names, 59

vctrs::vec_as_names(), 10