Package ‘dm’

January 21, 2024

Title Relational Data Models

Version 1.0.10

Date 2024-01-21

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


BugReports https://github.com/cynkra/dm/issues

Depends R (>= 3.3)

Imports backports, cli (>= 2.2.0), dplyr (>= 1.1.0), glue, igraph, lifecycle (>= 1.0.3), memoise, methods, purrr (>= 1.0.0), rlang (>= 1.0.2), tibble (>= 3.0.0), tidyr (>= 1.0.0), tidyselect (>= 1.2.0), vctrs (>= 0.3.2)

Suggests brio, colourpicker, cover, crayon, DBI (>= 1.2.0), dbplyr (>= 2.3.4), DiagrammeR, DiagrammeRsvg, digest, duckdb (>= 0.4.0), fansi, forcats, htmltools, htmlwidgets, jsonlite, keyring, knitr, labelled (>= 2.12.0), magrittr, mocks, nycflights13, odbc, pillar, pixarfilms, pool, progress, reactable, RMariaDB (>= 1.3.0), rmarkdown, RPostgres, RSQLite (>= 2.2.8), rstudioapi, shiny, shinyAce, shinydashboard, testthat (>= 3.1.2), waldo, withr

Config/Needs/website brio, bslib, cynkra/cynkratemplate, htmltools, pagedown, purrr, rmarkdown, whisker, xml2

Config/Needs/check anthonynorth/roxyglobals

VignetteBuilder knitr

Config/autostyle/scope line_breaks

Config/autostyle/strict true

Config/testthat/edition 3
R topics documented:

```r
check_key .......................... 4
check_set_equality ...................... 5
check_subset .......................... 6
copy_dm_to .......................... 6
db_schema_create ...................... 8
db_schema_drop ......................... 9
db_schema_exists ...................... 10
db_schema_list ......................... 11
decompose_table ......................... 12
dm ........................................ 13
dm_add_fk ................................ 15
dm_add_pk ................................ 17
dm_add_uk ................................ 19
dm_deconstruct ......................... 20
dm_disambiguate_cols ................... 21
dm_draw ................................... 22
dm_enum_fk_candidates .................. 24
dm_examine_cardinalities ............. 25
dm_examine_constraints ................ 27
dm_filter ................................ 28
dm_financial .............................. 29
dm_flatten_to_tbl ...................... 30
dm_from_con .............................. 31
```
R topics documented:

- `dm_get_all_fks` ............................................. 33
- `dm_get_all_pks` ........................................... 34
- `dm_get_all_uk` ............................................ 35
- `dm_get_con` ................................................ 36
- `dm_get_tables` ............................................. 36
- `dm_gui` ..................................................... 37
- `dm_has_pk` .................................................. 38
- `dm_mutate_tbl` ............................................. 39
- `dm_nest_tbl` ............................................... 40
- `dm_nrow` ..................................................... 40
- `dm_nycflights13` .......................................... 41
- `dm_pack_tbl` ............................................... 42
- `dm_paste` .................................................... 43
- `dm_pixarfilms` ............................................. 44
- `dm_p_type` .................................................. 45
- `dm_rename` ................................................... 45
- `dm_rm_fk` .................................................... 46
- `dm_rm_pk` ..................................................... 47
- `dm_rm_uk` ..................................................... 48
- `dm_select` .................................................... 49
- `dm_select_tbl` ............................................. 50
- `dm_set_colors` ............................................. 51
- `dm_set_table_description` ............................... 52
- `dm_sql` ....................................................... 53
- `dm_unnest_tbl` ............................................. 54
- `dm_unpack_tbl` ............................................. 55
- `dm_unwrap_tbl` ............................................. 56
- `dm_validate` ................................................. 57
- `dm_wrap_tbl` ................................................ 58
- `dm_zoom_to` .................................................. 59
- `dplyr_join` .................................................. 61
- `dplyr_table_manipulation` ............................... 63
- `enum_pk_candidates` ....................................... 65
- `examine_cardinality` ...................................... 66
- `glimpse.dm` .................................................. 69
- `head.dm_zoomed` ........................................... 70
- `json_nest` .................................................... 71
- `json_nest_join` ............................................. 72
- `json_pack` .................................................... 73
- `json_pack_join` ............................................. 74
- `json_unnest` ................................................ 75
- `json_unpack` ................................................ 76
- `materialize` .................................................. 77
- `pack_join` .................................................... 78
- `pull_tbl` ....................................................... 79
- `reunite_parent_child` .................................... 80
- `rows.dm` ...................................................... 82
- `tidyr_table_manipulation` ............................... 84
check_key

**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(x, ..., .data = deprecated())

**Arguments**

- **x**
  - The data frame whose columns should be tested for key properties.

- **...**
  - The names of the columns to be checked, processed with dplyr::select(). If omitted, all columns will be checked.

- **.data**
  - Deprecated.

**Value**

Returns x, 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_key(data)
check_set_equality

Description

check_set_equality() is a wrapper of check_subset().

It tests if one table 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(
  x,
  y,
  ...,  
  x_select = NULL,
  y_select = NULL,
  by_position = NULL
)

Arguments

x, y A data frame or lazy table.
...
These dots are for future extensions and must be empty.
x_select, y_select Key columns to restrict the check, processed with dplyr::select().
by_position Set to TRUE to ignore column names and match by position instead. The default means matching by name, use x_select and/or y_select to align the names.

Value

Returns x, 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, data_2, x_select = a, y_select = 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, data_3, x_select = a, y_select = a)
# this is still failing:
try(check_set_equality(data_2, data_3))
check_subset | Check column values for subset

Description

check_subset() tests if x is a subset of y. For convenience, the x_select and y_select arguments allow restricting the check to a set of key columns without affecting the return value.

Usage

check_subset(x, y, ..., x_select = NULL, y_select = NULL, by_position = NULL)

Arguments

x, y  A data frame or lazy table.
...  These dots are for future extensions and must be empty.
x_select, y_select  Key columns to restrict the check, processed with dplyr::select().
by_position  Set to TRUE to ignore column names and match by position instead. The default means matching by name, use x_select and/or y_select to align the names.

Value

Returns x, 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, data_2, x_select = a, y_select = a)

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

copy_dm_to | Copy data model to data source

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. The default is to create temporary tables, set temporary = FALSE to create permanent tables. Unless set_key_constraints is FALSE, primary key constraints are set on all databases, and in addition foreign key constraints are set on MSSQL and Postgres databases.
Usage

copy_dm_to(  
dest,  
dm,  
...,  
set_key_constraints = TRUE,  
table_names = NULL,  
temporary = TRUE,  
schema = NULL,  
progress = NA,  
unique_table_names = NULL,  
copy_to = NULL  
)

Arguments

dest An object of class "src" or "DBIConnection".
dm A dm object.
... These dots are for future extensions and must be empty.
set_key_constraints If TRUE will mirror dm primary and foreign key constraints on a database and create indexes for foreign key constraints. Set to FALSE if your data model currently does not satisfy primary or foreign key constraints.
table_names Desired names for the tables on dest; the names within the dm remain unchanged. Can be NULL, a named character vector, or a vector of DBI::Id objects. If left NULL (default), the names will be determined automatically depending on the temporary argument:
1. temporary = TRUE (default): unique table names based on the names of the tables in the dm are created.
2. temporary = FALSE: the table names in the dm are used as names for the tables on dest.

If a function or one-sided formula, table_names is converted to a function using rlang::as_function(). This function is called with the unquoted table names of the dm object as the only argument. The output of this function is processed by DBI::dbQuoteIdentifier(), that result should be a vector of identifiers of the same length as the original table names.
Use a variant of table_names = ~ DBI::SQL(paste0("schema_name", ", ", ",x))) to specify the same schema for all tables. Use table_names = identity with temporary = TRUE to avoid giving temporary tables unique names.
If a named character vector, the names of this vector need to correspond to the table names in the dm, and its values are the desired names on dest. The value 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.
db_schema_create

Create a schema on a database

Description

[Experimental]
db_schema_create() creates a schema on the database.

Usage

db_schema_create(con, schema, ...)

temporary
If TRUE, only temporary tables will be created. These tables will vanish when disconnecting from the database.

schema
Name of schema to copy the dm to. If schema is provided, an error will be thrown if temporary = FALSE or table_names is not NULL. Not all DBMS are supported.

progress
Whether to display a progress bar, if NA (the default) hide in non-interactive mode, show in interactive mode. Requires the 'progress' package.

unique_table_names, copy_to
Must be NULL.

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)
)

dplyr::remote_name(persistent_dm$planes)

DBI::dbDisconnect(con)
Arguments

**con**
An object of class "src" or "DBIConnection".

**schema**
Class character or SQL (cf. Details), name of the schema

... Passed on to the individual methods.

Details

Methods are not available for all DBMS.
An error is thrown if a schema of that name already exists.
The argument schema (and dbname for MSSQL) can be provided as SQL objects. Keep in mind, that in this case it is assumed that they are already correctly quoted as identifiers using DBI::dbQuoteIdentifier().
Additional arguments are:

- **dbname**: supported for MSSQL. Create a schema in a different database on the connected MSSQL-server; default: database addressed by con.

Value

NULL invisibly.

See Also

Other schema handling functions: db_schema_drop(), db_schema_exists(), db_schema_list()
`db_schema_exists` Check for existence of a schema on a database

**Description**

[Experimental]  
`db_schema_exists()` checks, if a schema exists on the database.

**Usage**

`db_schema_exists(con, schema, ...)`

**Arguments**

- `con` An object of class "src" or "DBIConnection".
- `schema` Class character or SQL, name of the schema
- `...` Passed on to the individual methods.

**Details**

Methods are not available for all DBMS.

Additional arguments are:

- `dbname`: supported for MSSQL. Check if a schema exists on a different database on the connected MSSQL-server; default: database addressed by `con`.

**Value**

A boolean: TRUE if schema exists, FALSE otherwise.
**db_schema_list**

See Also

Other schema handling functions: db_schema_create(), db_schema_drop(), db_schema_list()

---

**db_schema_list**  
List schemas on a database

---

**Description**

[Experimental]

db_schema_list() lists the available schemas on the database.

**Usage**

db_schema_list(con, include_default = TRUE, ...)

**Arguments**

- **con**: An object of class "src" or "DBIConnection".
- **include_default**: Boolean, if TRUE (default), also the default schema on the database is included in the result
- **...**: Passed on to the individual methods.

**Details**

Methods are not available for all DBMS.

Additional arguments are:

- **dbname**: supported for MSSQL. List schemas on a different database on the connected MSSQL-server; default: database addressed by con.

**Value**

A tibble with the following columns:

- **schema_name**: the names of the schemas,
- **schema_owner**: the schema owner names.

**See Also**

Other schema handling functions: db_schema_create(), db_schema_drop(), db_schema_exists()
decompose_table  

Decompose a table into two linked tables

Description

[Experimental]
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.

de-compose_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

.data  
Data frame from which columns ... are to be extracted.

new_id_column  
Name of the identifier column (primary key column) for the parent table. A column of this name is also added in 'child table'.

...  
The columns to be extracted from the .data.
One or more unquoted expressions separated by commas. You can treat variable names as if they were positions, so you can use expressions like x:y 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 vignette("programming") for an introduction to those concepts.
See select helpers for more details, and the examples about tidyselect helpers, such as starts_with(), everything(), ...

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 "experimental" because it seems more useful when applied to a table in a dm object. Changing the interface later seems harmless because these functions are most likely used interactively.
dm

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 dm_validate() to double-check the returned object.

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"),
  .quiet = FALSE
)

new_dm(tables = list())

is_dm(x)
as_dm(x, ...)
Arguments

... Tables or existing dm objects to add to the dm object. Unnamed tables are auto-named, dm objects must not be named.

.name_repair, .quiet
Options for name repair. Forwarded as repair and quiet to \texttt{vctrs::vec\_as\_names()}.

tables A named list of the tables (tibble-objects, not names), to be included in the dm object.

x An object.

Value

For \texttt{dm()}, \texttt{new\_dm()}, \texttt{as\_dm()}: A dm object.

For \texttt{is\_dm()}: A scalar logical, TRUE if is this object is a dm.

See Also

- \texttt{dm\_from\_con()} for connecting to all tables in a database and importing the primary and foreign keys
- \texttt{dm\_get\_tables()} for returning a list of tables
- \texttt{dm\_add\_pk()} and \texttt{dm\_add\_fk()} for adding primary and foreign keys
- \texttt{copy\_dm\_to()} for DB interaction
- \texttt{dm\_draw()} for visualization
- \texttt{dm\_flatten\_to\_tbl()} for flattening
- \texttt{dm\_filter()} for filtering
- \texttt{dm\_select\_tbl()} for creating a dm with only a subset of the tables
- \texttt{dm\_nycflights13()} for creating an example dm object
- \texttt{decompose\_table()} for table surgery
- \texttt{check\_key()} and \texttt{check\_subset()} for checking for key properties
- \texttt{examine\_cardinality()} for checking the cardinality of the relation between two tables

Examples

\begin{verbatim}
dm(trees, mtcars)
new_dm(list(trees = trees, mtcars = mtcars))
as_dm(list(trees = trees, mtcars = mtcars))

is_dm(dm_nycflights13())
dm_nycflights13()$airports
dm_nycflights13()$airports
\end{verbatim}
\texttt{dm_add_fk()}

\begin{verbatim}
dm_nycflights13()[["airports"]]
dm_nycflights13() %>% names()

library(dm)
library(nycflights13)

# using `data.frame` objects
new_dm(tibble::lst(weather, airports))

# using `dm_keyed_tbl` objects
dm <- dm_nycflights13()
y1 <- dm$planes %>%
    mutate() %>%
    select(everything())
y2 <- dm$flights %>%
    left_join(dm$airlines, by = "carrier")

new_dm(list("tbl1" = y1, "tbl2" = y2))
\end{verbatim}

---

\textbf{dm_add_fk} \hspace{1cm} \textit{Add foreign keys}

\textbf{Description}

\texttt{dm_add_fk()} marks the specified columns as the foreign key of table \texttt{table} with respect to a key of table \texttt{ref_table}. Usually the referenced columns are a primary key in \texttt{ref_table}. However, it is also possible to specify other columns via the \texttt{ref_columns} argument. If \texttt{check == TRUE}, then it will first check if the values in columns are a subset of the values of the key in table \texttt{ref_table}.

\textbf{Usage}

\begin{verbatim}
dm_add_fk(
    dm,             # A \texttt{dm} object.
    table,         # A table in the \texttt{dm}.
    columns,       # The columns to be marked as foreign keys.
    ref_table,     # The table containing the key.
    ref_columns = NULL,    # A vector of columns to reference.
    ...,
    check = FALSE,   # If TRUE, check if the values are a subset.
    on_delete = c("no_action", "cascade")
)
\end{verbatim}

\textbf{Arguments}

- \texttt{dm} 
  - A \texttt{dm} object.
- \texttt{table} 
  - A table in the \texttt{dm}.
- \texttt{columns} 
  - The columns to be marked as foreign keys.
- \texttt{ref_table} 
  - The table containing the key.
- \texttt{ref_columns} (optional) 
  - A vector of columns to reference.
- \texttt{...} 
  - Further arguments.
- \texttt{check} (optional) 
  - If \texttt{TRUE}, check if the values are a subset.
- \texttt{on_delete} (optional) 
  - A vector of options for foreign key actions.
columns  The columns of table which are to become the foreign key columns that reference ref_table. To define a compound key, use c(col1, col2).

ref_table  The table which table will be referencing.

ref_columns  The column(s) of table which are to become the referenced column(s) in ref_table. By default, the primary key is used. To define a compound key, use c(col1, col2).

...  These dots are for future extensions and must be empty.

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

on_delete  [Experimental] Defines behavior if a row in the parent table is deleted. - "no_action", the default, means that no action is taken and the operation is aborted if child rows exist - "cascade" means that the child row is also deleted This setting is picked up by copy_dm_to() with set_key_constraints = TRUE, and by dm_sql(), and might be considered by dm_rows_delete() in a future version.

Details

It is possible that a foreign key (FK) is pointing to columns that are neither primary (PK) nor explicit unique keys (UK). This can happen

1. when a FK is added without a corresponding PK or UK being present in the parent table
2. when the PK or UK is removed (dm_rm_pk() / dm_rm_uk()) without first removing the associated Fks.

These columns are then a so-called "implicit unique key" of the referenced table and can be listed via dm_get_all_uks().

Value

An updated dm with an additional foreign key relation.

See Also

Other foreign key functions: dm_enum_fk_candidates(), dm_get_all_fks(), dm_rm_fk()

Examples

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

nycflights_dm %>%
  dm_draw()

# Create foreign keys:
**dm_add_pk**

```r
dm_add_pk(planes, tailnum) %>%
dm_add_fk(flights, tailnum, planes) %>%
dm_add_pk(weather, c(origin, time_hour)) %>%
dm_add_fk(flights, c(origin, time_hour), weather) %>%
dm_draw()
```

# Keys can be checked during creation:
```r
try(
  nycflights_dm %>%
  dm_add_pk(planes, tailnum) %>%
  dm_add_fk(flights, tailnum, planes, check = TRUE)
)
```

---

**dm_add_pk**  
*Add a primary key*

**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, without altering foreign keys previously pointing to that primary key.

**Usage**

dm_add_pk(
  dm,
  table,
  columns,
  ...,  
  autoincrement = FALSE,  
  check = FALSE,  
  force = FALSE
)

**Arguments**

- **dm**  
  A `dm` object.

- **table**  
  A table in the `dm`.

- **columns**  
  Table columns, unquoted. To define a compound key, use `c(col1, col2)`.

- **...**  
  These dots are for future extensions and must be empty.

- **autoincrement**  
  [Experimental] If TRUE, the column specified in `columns` will be populated automatically with a sequence of integers.

- **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.

Details

There can be only one primary key per table in a dm. It’s possible though to set an unlimited number of unique keys using dm_add_uk() or adding foreign keys pointing to columns other than the primary key columns with dm_add_fk().

Value

An updated dm with an additional primary key.

See Also

Other primary key functions: dm_add_uk(), dm_get_all_pks(), dm_get_all_uks(), dm_has_pk(), dm_rm_pk(), dm_rm_uk(), enum_pk_candidates()

Examples

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

nycflights_dm %>%
  dm_draw()

# Create primary keys:
try{
  nycflights_dm %>%
    dm_add_pk(planes, tailnum) %>%
    dm_add_pk(airports, faa, check = TRUE) %>%
    dm_add_pk(weather, c(origin, time_hour)) %>%
    dm_draw()
}
```
**dm_add_uk**

Add a unique key

**Description**

`dm_add_uk()` marks the specified columns as a unique 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.

**Usage**

```
dm_add_uk(dm, table, columns, ..., check = FALSE)
```

**Arguments**

- **dm**: A `dm` object.
- **table**: A table in the `dm`.
- **columns**: Table columns, unquoted. To define a compound key, use `c(col1, col2)`.
- **...**: These dots are for future extensions and must be empty.
- **check**: Boolean, if `TRUE`, a check is made if the combination of columns is a unique key of the table.

**Details**

The difference between a primary key (PK) and a unique key (UK) consists in the following:

- When a local `dm` is copied to a database (DB) with `copy_dm_to()`, a PK will be set on the DB by default, whereas a UK is being ignored.
- A PK can be set as an autoincrement key (also implemented on certain DBMS when the `dm` is transferred to the DB)
- There can be only one PK for each table, whereas there can be unlimited UKs
- A UK will be used, if the same table has an autoincrement PK in addition, to ensure that during delta load processes on the DB (cf. `dm_rows_append()`) the foreign keys are updated accordingly. If no UK is available, the insertion is done row-wise, which also ensures a correct matching, but can be much slower.
- A UK can generally enhance the data model by adding additional information
- There can also be implicit UKs, when the columns addressed by a foreign key are neither a PK nor a UK. These implicit UKs are also listed by `dm_get_all_uks()`

**Value**

An updated `dm` with an additional unique key.

**See Also**

Other primary key functions: `dm_add_pk()`, `dm_get_all_pks()`, `dm_get_all_uks()`, `dm_has_pk()`, `dm_rm_pk()`, `dm_rm_uk()`, `enum_pk_candidates()`
Examples

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

# Create unique keys:
nycflights_dm %>%
  dm_add_uk(planes, tailnum) %>%
  dm_add_uk(airports, faa, check = TRUE) %>%
  dm_add_uk(weather, c(origin, time_hour)) %>%
  dm_get_all_uks()

# Keys can be checked during creation:
try(
  nycflights_dm %>%
  dm_add_uk(planes, manufacturer, check = TRUE)
)
```

---

**dm_deconstruct**

Create code to deconstruct a dm object

**Description**

**[Experimental]**

Emits code that assigns each table in the dm to a variable, using `pull_tbl()` with `keyed = TRUE`. These tables retain information about primary and foreign keys, even after data transformations, and can be converted back to a dm object with `dm()`.

**Usage**

```r
dm_deconstruct(dm, dm_name = NULL)
```

**Arguments**

- **dm**: A dm object.
- **dm_name**: The code to use to access the dm object, by default the expression passed to this function.

**Value**

This function is called for its side effect of printing generated code.
Examples

```r
dm <- dm_nycflights13()
dm_deconstruct(dm)
airlines <- pull_tbl(dm, "airlines", keyed = TRUE)
airports <- pull_tbl(dm, "airports", keyed = TRUE)
flights <- pull_tbl(dm, "flights", keyed = TRUE)
planes <- pull_tbl(dm, "planes", keyed = TRUE)
weather <- pull_tbl(dm, "weather", keyed = TRUE)
by_origin <-
  flights %>%
  group_by(origin) %>%
  summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  ungroup()

by_origin
dm(airlines, airports, flights, planes, weather, by_origin) %>%
  dm_draw()
```

---

### dm_disambiguate_cols

**Resolve column name ambiguities**

**Description**

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

**Usage**

```r
dm_disambiguate_cols(
  dm,
  .sep = ".",
  ...,
  .quiet = FALSE,
  .position = c("suffix", "prefix")
)
```

**Arguments**

- **dm**
  A dm object.

- **.sep**
  The character variable that separates the names of the table and the names of the ambiguous columns.

- **...**
  These dots are for future extensions and must be empty.

- **.quiet**
  Boolean. By default, this function lists the renamed columns in a message, pass TRUE to suppress this message.

- **.position**
  [Experimental] By default, table names are appended to the column names to resolve conflicts. Prepending table names was the default for versions before 1.0.0, use "prefix" to achieve this behavior.
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

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

---

**dm_draw**

Draw a diagram of the data model

Description

`dm_draw()` draws a diagram, a visual representation of the data model.

Usage

```r
dm_draw(
  dm,
  rankdir = "LR",
  ...
)
```
Arguments

- **dm**: A `dm` object.
- **rankdir**: Graph attribute for direction (e.g., 'BT' = bottom –> top).
- **...**: These dots are for future extensions and must be empty.
- **col_attr**: Deprecated, use `column_types` instead.
- **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).
- **graphAttrs**: Additional graph attributes.
- **nodeAttrs**: Additional node attributes.
- **edgeAttrs**: Additional edge attributes.
- **focus**: A list of parameters for rendering (table filter).
- **graph_name**: The name of the graph.
- **column_types**: Set to TRUE to show column types.
- **backend**: Currently, only the default “DiagrammeR” is accepted. Pass this value explicitly if your code not only uses this function to display a data model but relies on the type of the return value.
- **font_size**: [Experimental]
  Font size for:
  - header, defaults to 16
  - column, defaults to 16
  - table_description, defaults to 8
  Can be set as a named integer vector, e.g. c(table_headers = 18L, table_description = 6L).

Details

Currently, **dm** uses **DiagrammeR** to draw diagrams. Use **DiagrammeR::export_svg()** to convert the diagram to an SVG file.

The backend for drawing the diagrams might change in the future. If you rely on DiagrammeR, pass an explicit value for the `backend` argument.

Value

An object with a `print()` method, which, when printed, produces the output seen in the viewer as a side effect. Currently, this is an object of class `grViz` (see also **DiagrammeR::grViz()**), but this is subject to change.

See Also

- **dm_set_colors()** for defining the table colors.
- **dm_set_table_description()** for adding details to one or more tables in the diagram
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_enum_fk_candidates  Foreign key candidates

Description

[Experimental]
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, ...)

dm_enum_fk_candidates(dm_zoomed, ref_table, ...)

Arguments

dm A dm object.
table The table whose columns should be tested for suitability as foreign keys.
ref_table A table with a primary key.
... These dots are for future extensions and must be empty.
dm_zoomed A dm 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:
dm_examine_cardinalities

- 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 this.

Life cycle

These functions are marked "experimental" 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_rm_fk()

Examples

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

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

---

dm_examine_cardinalities

Learn about your data model

Description

[Experimental]

This function returns a tibble with information about the cardinality of the FK constraints. The printing for this object is special, use as_tibble() to print as a regular tibble.
Usage

dm_examine_cardinalities(
  .dm,
  ..., 
  .progress = NA,
  dm = deprecated(),
  progress = deprecated()
)

Arguments

  .dm          A dm object.
  ...          These dots are for future extensions and must be empty.
  .progress    Whether to display a progress bar, if NA (the default) hide in non-interactive
               mode, show in interactive mode. Requires the 'progress' package.
  dm, progress [Deprecated]

Details

Uses examine_cardinality() on each foreign key that is defined in the dm.

Value

A tibble with the following columns:

  child_table  child table,
  child_fk_cols foreign key column(s) in child table as list of character vectors,
  parent_table parent table,
  parent_key_cols key column(s) in parent table as list of character vectors,
  cardinality  the nature of cardinality along the foreign key.

See Also

Other cardinality functions: examine_cardinality()

Examples

dm_nycflights13() %>%
  dm_examine_cardinalities()
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,
  ..., 
  .progress = NA,
  dm = deprecated(),
  progress = deprecated()
)

Arguments

  .dm A dm object.

  ... These dots are for future extensions and must be empty.

  .progress Whether to display a progress bar, if NA (the default) hide in non-interactive mode, show in interactive mode. Requires the 'progress' package.

  dm, progress [Deprecated]

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.
dm_filter

Examples

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

---

### Description

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()`. The filters work across related tables: The resulting dm object only contains rows that are related (directly or indirectly) to rows that remain after applying the filters on all tables.

### Usage

```r
dm_filter(.dm, ...)
```

### Arguments

- `.dm`  
  A dm object.

- `...`
  Named logical predicates. The names correspond to tables in the dm object. The predicates are defined in terms of the variables in the corresponding table, they are passed on to `dplyr::filter()`. Multiple conditions are combined with &. Only the rows where the condition evaluates to TRUE are kept.

### Details

As of dm 1.0.0, these conditions are no longer stored in the dm object, instead they are applied to all tables during the call to `dm_filter()`. Calling `dm_apply_filters()` or `dm_apply_filters_to_tbl()` is no longer necessary.

Use `dm_zoom_to()` and `dplyr::filter()` to filter rows without affecting related tables.

### Value

An updated dm object with filters executed across all tables.
Examples

```r
dm_nyc <- dm_nycflights13()
dm_nyc %>%
  dm_nrow()

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

dm_nyc_filtered %>%
  dm_nrow()

# 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_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_financial

Creates a dm object for the Financial data

### Description

`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

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

---

**dm_flatten_to_tbl**  
Flatten a part of a dm into a wide table

---

**Description**

`dm_flatten_to_tbl()` gathers all information of interest in one place in a wide table. It performs a disambiguation of column names and a cascade of joins.

**Usage**

```r
dm_flatten_to_tbl(dm, .start, ..., .recursive = FALSE, .join = left_join)
```

**Arguments**

- `dm`  
  A `dm` 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.
- `...`  
  [Experimental]  
  Unquoted names of the tables to be included in addition to the `.start` 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. `tidyselect` is supported, see `dplyr::select()` for details on the semantics.
- `recursive`  
  Logical, defaults to `FALSE`. Should not only parent tables be joined to `.start`, but also their ancestors?
- `join`  
  The type of join to be performed, see `dplyr::join()`.

**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.

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.

The difference between .recursive = FALSE and .recursive = TRUE is the following (see the examples):

- .recursive = FALSE allows only one level of hierarchy (i.e., direct neighbors to table .start), while
- .recursive = TRUE will go through all levels of hierarchy while joining.

Additionally, these functions differ from dm_wrap_tbl(), which always returns a dm object.

**Value**

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

**Examples**

```r
dm_financial() %>%
  dm_select_tbl(-loans) %>%
  dm_flatten_to_tbl(.start = cards)
```

```r
dm_financial() %>%
  dm_select_tbl(-loans) %>%
  dm_flatten_to_tbl(.start = cards, .recursive = TRUE)
```

---

**dm_from_con**  
**Load a dm from a remote data source**

**Description**

dm_from_con() 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**

```r
dm_from_con(
  con = NULL,
  table_names = NULL,
  learn_keys = NULL,
  .names = NULL,
  ...
)
```
Arguments

con  A DBI::DBIConnection or a Pool object.
table_names  A character vector of the names of the tables to include.
learn_keys  [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.
.names  [Experimental]
  A glue specification that describes how to name the tables within the output, currently only for MSSQL, Postgres and MySQL/MariaDB. This can use {.table} to stand for the table name, and {.schema} to stand for the name of the schema which the table lives within. The default (NULL) is equivalent to "{.table}" when a single schema is specified in schema, and "{.schema}.{.table}" for the case where multiple schemas are given, and may change in future versions.

Value

A dm object.

Examples

con <- dm_get_con(dm_financial())

# Avoid DBI::dbDisconnect() here, because we don't own the connection
### dm_get_all_fks

**Get foreign key constraints**

#### Description

Get a summary of all foreign key relations in a dm.

#### Usage

```r
dm_get_all_fks(dm, parent_table = NULL, ...)
```

#### Arguments

- `dm`: A dm object.
- `parent_table`: One or more table names, unquoted, to return foreign key information for. If given, foreign keys are returned in that order. The default NULL returns information for all tables.
- `...`: These dots are for future extensions and must be empty.

#### Value

A tibble with the following columns:

- `child_table`: Child table,
- `child_fk_cols`: Foreign key column(s) in child table as list of character vectors,
- `parent_table`: Parent table,
- `parent_key_cols`: Key column(s) in parent table as list of character vectors.
- `on_delete`: Behavior on deletion of rows in the parent table.

#### See Also

Other foreign key functions: `dm_add_fk()`, `dm_enum_fk_candidates()`, `dm_rm_fk()`

#### Examples

```r
dm_nycflights13() %>%
dm_get_all_fks()
```
**dm_get_all_pks**

Get all primary keys of a **dm** object

### Description

**dm_get_all_pks()** checks the **dm** object for primary keys and returns the tables and the respective primary key columns.

### Usage

```r
dm_get_all_pks(dm, table = NULL, ...)
```

### Arguments

- **dm**: A **dm** object.
- **table**: One or more table names, unquoted, to return primary key information for. If given, primary keys are returned in that order. The default `NULL` returns information for all tables.
- **...**: These dots are for future extensions and must be empty.

### Value

A tibble with the following columns:

- **table**: table name,
- **pk_col**: column name(s) of primary key, as list of character vectors.

### See Also

Other primary key functions: `dm_add_pk()`, `dm_add_uk()`, `dm_get_all_uks()`, `dm_has_pk()`, `dm_rm_pk()`, `dm_rm_uk()`, `enum_pk_candidates()`

### Examples

```r
dm_nycflights13() %>%
dm_get_all_pks()
```
**dm_get_all_uks**

*Get all unique keys of a dm object*

**Description**

`dm_get_all_uks()` checks the `dm` object for unique keys (primary keys, explicit and implicit unique keys) and returns the tables and the respective unique key columns.

**Usage**

```r
dm_get_all_uks(dm, table = NULL, ...)
```

**Arguments**

- `dm` A `dm` object.
- `table` One or more table names, unquoted, to return unique key information for. The default `NULL` returns information for all tables.
- `...` These dots are for future extensions and must be empty.

**Details**

There are 3 kinds of unique keys:

- PK: Primary key, set by `dm_add_pk()`
- explicit UK: Unique key, set by `dm_add_uk()`
- implicit UK: Unique key, not explicitly set, but referenced by a foreign key.

**Value**

A tibble with the following columns:

- `table` table name,
- `uk_col` column name(s) of primary key, as list of character vectors,
- `kind` kind of unique key, see details.

**See Also**

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

**Examples**

```r
dm_nycflights13() %>%
  dm_get_all_uks()
```
dm_get_con

Get connection

Description

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

Usage

dm_get_con(dm)

Arguments

- **dm**: A dm object.

Details

All lazy tables in a dm object must be stored on the same database server and accessed through the same connection, because a large part of the package’s functionality relies on efficient joins.

Value

The DBI::DBIConnection object for a dm object.

Examples

dm_financial() %>%
    dm_get_con()

---

dm_get_tables

Get tables

Description

dm_get_tables() returns a named list of dplyr tbl objects of a dm object.

Usage

dm_get_tables(x, ..., keyed = FALSE)
Arguments

x A dm object.

... These dots are for future extensions and must be empty.

keyed [Experimental] Set to TRUE to return objects of the internal class "dm_keyed_tbl" that will contain information on primary and foreign keys in the individual table objects. This allows using dplyr workflows on those tables and later reconstruct them into a dm object. See dm_deconstruct() for a function that generates corresponding code for an existing dm object, and vignette("tech-dm-keyed") for details.

Value

A named list with the tables (data frames or lazy tables) constituting the dm.

See Also

dm() and new_dm() for constructing a dm object from tables.

Examples

```r
dm_nycflights13() %>%
dm_get_tables()

dm_nycflights13() %>%
dm_get_tables(keyed = TRUE)

dm_nycflights13() %>%
dm_get_tables(keyed = TRUE) %>%
new_dm()
```

---

### dm_gui

**Shiny app for defining dm objects**

**Description**

[Experimental]

This function starts a Shiny application that allows to define dm objects from a database or from local data frames. The application generates R code that can be inserted or copy-pasted into an R script or function.

**Usage**

dm_gui(..., dm = NULL, select_tables = TRUE, debug = FALSE)
**Arguments**

- `...` These dots are for future extensions and must be empty.
- `dm` An initial dm object, currently required.
- `select_tables` Show selectize input to select tables?
- `debug` Set to TRUE to simplify debugging of the app.

**Details**

In a future release, the app will also allow composing dm objects directly from database connections or data frames.

The signature of this function is subject to change without notice. This should not pose too many problems, because it will usually be run interactively.

**Examples**

```r
## Not run:
dm <- dm_nycflights13(cycle = TRUE)
dm_gui(dm = dm)
## End(Not run)
```

---

**dm_has_pk**  
*Check for primary key*

**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.
- `...` These dots are for future extensions and must be empty.

**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_add_uk()`, `dm_get_all_pks()`, `dm_get_all_uks()`, `dm_rm_pk()`, `dm_rm_uk()`, `enum_pk_candidates()`
**dm_mutate_tbl**

Update tables in a `dm`

**Description**

[Experimental]

Updates one or more existing tables in a `dm`. For now, the column names must be identical. This restriction may be lifted optionally in the future.

**Usage**

```r
dm_mutate_tbl(dm, ...)
```

**Arguments**

- `dm` A `dm` object.
- `...` One or more tables to update in the `dm`. Must be named.

**See Also**

`dm()`, `dm_select_tbl()`

**Examples**

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

```r
dm_mutate_tbl(dm, ...)
```

```r
dm_nycflights13() %>%
  dm_mutate_tbl(flights = nycflights13::flights[1:3, ])
```
**dm_nest_tbl**  
**Nest a table inside its dm**

**Description**

[Experimental]

`dm_nest_tbl()` converts a child table to a nested column in its parent table. The child table should not have children itself (i.e. it needs to be a *terminal child table*).

**Usage**

```r
dm_nest_tbl(dm, child_table, into = NULL)
```

**Arguments**

- `dm`  
  A dm.
- `child_table`  
  A terminal table with one parent table.
- `into`  
  The table to nest `child_tables` into, optional as it can be guessed from the foreign keys unambiguously but useful to be explicit.

**See Also**

`dm_wrap_tbl()`, `dm_unwrap_tbl()`, `dm_pack_tbl()`

**Examples**

```r
nested_dm <-
dm_nycflights13() %>%
dm_select_tbl(airlines, flights) %>%
dm_nest_tbl(flights)

nested_dm

nested_dm$airlines
```

---

**dm_nrow**  
**Number of rows**

**Description**

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

**Usage**

```r
dm_nrow(dm)
```
Arguments

- **dm**: A `dm` object.

Value

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

Examples

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

---

**dm_nycflights13**  *Creates a dm object for the nycflights13 data*

---

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

```r
dm_nycflights13(
  ..., 
  cycle = FALSE,
  color = TRUE,
  subset = TRUE,
  compound = TRUE,
  table_description = FALSE
)
```

Arguments

- **...**: These dots are for future extensions and must be empty.
- **cycle**: Boolean. If `FALSE` (default), only one foreign key relation (from `flights$origin` to `airports$faa`) between the `flights` table and the `airports` table is established. If `TRUE`, a `dm` object with a double reference between those tables will be produced.
- **color**: Boolean, if `TRUE` (default), the resulting `dm` object will have colors assigned to different tables for visualization with `dm_draw()`.
- **subset**: Boolean, if `TRUE` (default), the `flights` table is reduced to flights with column `day` equal to 10.
compound

Boolean, if FALSE, no link will be established between tables flights and weather, because this requires compound keys.

table_description

Boolean, if TRUE, a description will be added for each table that will be displayed when drawing the table with `dm_draw()`.

Value

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

See Also

`vignette("howto-dm-df")`

Examples

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

---

**Description**

[Experimental]

`dm_pack_tbl()` converts a parent table to a packed column in its child table. The parent table should not have parent tables itself (i.e. it needs to be a terminal parent table).

**Usage**

```r
dm_pack_tbl(dm, parent_table, into = NULL)
```

**Arguments**

- `dm`  
  A `dm`.
- `parent_table`  
  A terminal table with one child table.
- `into`  
  The table to pack parent_tables into, optional as it can be guessed from the foreign keys unambiguously but useful to be explicit.

**See Also**

`dm_wrap_tbl()`, `dm_unwrap_tbl()`, `dm_nest_tbl()`.
Examples

```r
dm_packed <-
  dm_nycflights13() %>%
  dm_pack_tbl(planes)

dm_packed

dm_packed$flights

dm_packed$flights$planes
```

**dm_paste**  
*Create R code for a dm object*

Description

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

Usage

```r
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:
  - "tables": tibble() calls for empty table definitions derived from dm_ptype(), overrides "select".
  - "select": dm_select() statements for columns that are part of the dm.
  - "keys": dm_add_pk(), dm_add_fk() and dm_add_uk() statements for adding keys.
  - "color": dm_set_colors() statements to set color.
  - "all": All options above except "select"

Default NULL is equivalent to c("keys", "color")

- **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

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

dm_nycflights13() %>%
  dm_paste()

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

---

```
---

Value

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

Examples

```r
dm_pixarfilms()

dm_pixarfilms() %>%
  dm_draw()
```
**dm_ptype**

**Prototype for a dm object**

### Description

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

### Usage

```r
dm_ptype(dm)
```

### Arguments

- **dm**
  
  A dm object.

### Examples

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

**dm_rename**

**Rename columns**

### Description

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

### Usage

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

### Arguments

- **dm**
  
  A dm object.

- **table**
  
  A table in the dm.
One or more unquoted expressions separated by commas. You can treat variable names as if they were positions, and use expressions like x:y to select the ranges of variables.

Use named arguments, e.g. new_name = old_name, to rename the selected 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 vignette("programming", package = "dplyr") for an introduction to those concepts.

See select helpers for more details, and the examples about tidyselect helpers, such as starts_with(), everything(), ...

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

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

---

**dm_rm_fk**

Remove foreign keys

Description

`dm_rm_fk()` can remove either one reference between two tables, or multiple references at once (with a message). An error is thrown if no matching foreign key is found.

Usage

```r
dm_rm_fk(
  dm,
  table = NULL,
  columns = NULL,
  ref_table = NULL,
  ref_columns = NULL,
  ...
)
```
dm_rm_pk

Arguments

dm A dm object.
table A table in the dm. Pass NULL to remove all matching keys.
columns Table columns, unquoted. To refer to a compound key, use c(col1, col2). Pass NULL (the default) to remove all matching keys.
ref_table The table referenced by the table argument. Pass NULL to remove all matching keys.
ref_columns The columns of table that should no longer be referencing the primary key of ref_table. To refer to a compound key, use c(col1, col2).
... These dots are for future extensions and must be empty.

Value

An updated dm without the matching foreign key relation(s).

See Also

Other foreign key functions: dm_add_fk(), dm_enum_fk_candidates(), dm_get_all_fks()

Examples

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

---

dm_rm_pk Remove a primary key

Description

If a table name is provided, dm_rm_pk() removes the primary key from this table and leaves the dm object otherwise unaltered. If no table is given, the dm is stripped of all primary keys at once. An error is thrown if no primary key matches the selection criteria. If the selection criteria are ambiguous, a message with unambiguous replacement code is shown. Foreign keys are never removed.

Usage

dm_rm_pk(dm, table = NULL, columns = NULL, ..., fail_fk = NULL)
dm_rm_uk

Arguments

- **dm**: A `dm` object.
- **table**: A table in the `dm`. Pass `NULL` to remove all matching keys.
- **columns**: Table columns, unquoted. To refer to a compound key, use `c(col1, col2)`. Pass `NULL` (the default) to remove all matching keys.
- **...**: These dots are for future extensions and must be empty.
- **fail_fk**: [Deprecated]

Value

An updated `dm` without the indicated primary key(s).

See Also

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

Examples

```r
dm_nycflights13() %>%
  dm_rm_pk(airports) %>%
  dm_draw()
```

Description

`dm_rm_uk()` removes one or more unique keys from a table and leaves the `dm` object otherwise unaltered. An error is thrown if no unique key matches the selection criteria. If the selection criteria are ambiguous, a message with unambiguous replacement code is shown. Foreign keys are never removed.

Usage

```r
dm_rm_uk(dm, table = NULL, columns = NULL, ...)
```

Arguments

- **dm**: A `dm` object.
- **table**: A table in the `dm`. Pass `NULL` to remove all matching keys.
- **columns**: Table columns, unquoted. To refer to a compound key, use `c(col1, col2)`. Pass `NULL` (the default) to remove all matching keys.
- **...**: These dots are for future extensions and must be empty.
Value

An updated dm without the indicated unique key(s).

See Also

Other primary key functions: `dm_add_pk()`, `dm_add_uk()`, `dm_get_all_pks()`, `dm_get_all_uks()`, `dm_has_pk()`, `dm_rm_pk()`, `enum_pk_candidates()`
Examples

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

---

**dm_select_tbl**

*Select and rename tables*

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.

Examples

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

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

`dm_set_colors()` allows to define the colors that will be used to display the tables of the data model with `dm_draw()`. 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 [https://rpubs.com/krlmlr/colors](https://rpubs.com/krlmlr/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

```r
dm_set_colors(dm, ...)

dm_get_colors(dm)

dm_get_available_colors()
```

Arguments

- `dm`: A `dm` object.
- `...`: 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_set_colors()`: the updated data model.

For `dm_get_colors()`, a named character vector of table names with the colors in the names. This allows calling `dm_set_colors(!!!dm_get_colors(...))`. Use `tibble::enframe()` to convert this to a tibble.

For `dm_get_available_colors()`, a vector with the available colors.

Examples

```r
dm_nycflights13(color = FALSE) %>%
  dm_set_colors(
    darkblue = starts_with("air"),
    "#5986C4" = flights
  ) %>%
  dm_draw()
```
# Splicing is supported:

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

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

---

**dm_set_table_description**  
*Add info about a dm’s tables*

**Description**

When creating a diagram from a dm using `dm_draw()` the table descriptions set with `dm_set_table_description()` will be displayed.

**Usage**

```r
dm_set_table_description(dm, ...)  
dm_get_table_description(dm, table = NULL, ...)  
dm_reset_table_description(dm, table = NULL, ...)
```

**Arguments**

`dm`  
A `dm` object.

`...`  
For `dm_set_table_description()`: Descriptions for tables to set in the form `description = table`. `tidyselect` is supported, see `dplyr::select()` for details on the semantics.

For `dm_get_table_description()` and `dm_reset_table_description()`: These dots are for future extensions and must be empty.

`table`  
One or more table names, unquoted, for which to

1. get information about the current description(s) with `dm_get_table_description()`.
2. remove descriptions with `dm_reset_table_description()`.

In both cases the default applies to all tables in the dm.

**Details**

Multi-line descriptions can be achieved using the newline symbol `\n`. Descriptions are set with `dm_set_table_description()`. The currently set descriptions can be checked using `dm_get_table_description()`. Descriptions can be removed using `dm_reset_table_description()`.
dm_sql

Value

For dm_set_table_description(): A dm object containing descriptions for specified tables.
For dm_get_table_description: A named vector of tables, with the descriptions in the names.
For dm_reset_table_description(): A dm object without descriptions for specified tables.

Examples

desc_flights <- rlang::set_names(
  "flights",
  paste(
    "On-time data for all flights",
    "that departed NYC (i.e. JFK, LGA or EWR) in 2013.",
    sep = "\n"
  )
)
nyc_desc <- dm_nycflights13() %>%
  dm_set_table_description(
    !desc_flights,
    "Weather at the airport of\norigin at time of departure" = weather
  )
nyc_desc %>%
  dm_draw()

dm_get_table_description(nyc_desc)

reset_table_description(nyc_desc, flights) %>%
  dm_draw(font_size = c(header = 18L, table_description = 9L, column = 15L))
pull_tbl(nyc_desc, flights) %>%
  labelled::label_attribute()

---

dm_sql

Create DDL and DML scripts for a dm a and database connection

Description

Experimental

Generate SQL scripts to create tables, load data and set constraints, keys and indices. This function powers copy_dm_to() and is useful if you need more control over the process of copying a dm to a database.

Usage

dm_sql(dm, dest, table_names = NULL, temporary = TRUE)
dm_ddl_pre(dm, dest, table_names = NULL, temporary = TRUE)


\[
\begin{align*}
\text{dm_dml_load}(\text{dm}, \text{dest}, \text{table_names} = \text{NULL}, \text{temporary} = \text{TRUE}) \\
\text{dm_ddl_post}(\text{dm}, \text{dest}, \text{table_names} = \text{NULL}, \text{temporary} = \text{TRUE})
\end{align*}
\]

**Arguments**

- \text{dm} \quad \text{A \text{dm} object.}
- \text{dest} \quad \text{Connection to database.}
- \text{table_names} \quad \text{A named character vector or named vector of \text{DBI::Id}, \text{DBI::SQL} or \text{dbplyr} objects created with \text{dbplyr::ident()}, \text{dbplyr::in_schema()} or \text{dbplyr::in_catalog()}, with one unique element for each table in \text{dm}. The default, \text{NULL}, means to use the original table names.}
- \text{temporary} \quad \text{Should the tables be marked as \text{temporary}? Defaults to \text{TRUE}.}

**Details**

- \text{dm_ddl_pre()} \quad \text{generates \text{CREATE TABLE} statements (including \text{PRIMARY KEY} definition).}
- \text{dm_dml_load()} \quad \text{generates \text{INSERT INTO} statements.}
- \text{dm_ddl_post()} \quad \text{generates scripts for \text{FOREIGN KEY}, \text{UNIQUE KEY} and \text{INDEX}.}
- \text{dm_sql()} \quad \text{calls all three above and returns a complete set of scripts.}

**Value**

Nested list of SQL statements.

**Examples**

\begin{verbatim}
con <- DBI::dbConnect(RSQLite::SQLite())
dm <- dm_nycflights13()
s <- dm_sql(dm, con)
s
DBI::dbDisconnect(con)
\end{verbatim}

---

\textbf{dm_unnest_tbl} \quad \textit{Unnest columns from a wrapped table}

**Description**

[Experimental]

\text{dm_unnest_tbl()} \quad \text{target a specific column to unnest from the given table in a given \text{dm}. A \text{ptype} or a set of keys should be given, not both.}

**Usage**

\text{dm_unnest_tbl}(\text{dm, parent_table, col, ptype})
**Arguments**

- **dm**: A dm.
- **parent_table**: A table in the dm with nested columns.
- **col**: The column to unnest (unquoted).
- **ptype**: A dm, only used to query names of primary and foreign keys.

**Details**

`dm_nest_tbl()` is an inverse operation to `dm_unnest_tbl()` if differences in row and column order are ignored. The opposite is true if referential constraints between both tables are satisfied.

**Value**

A dm.

**See Also**

- `dm_unwrap_tbl()`, `dm_unpack_tbl()`, `dm_nest_tbl()`, `dm_pack_tbl()`, `dm_wrap_tbl()`, `dm_examine_constraints()`, `dm_examine_cardinalities()`, `dm_ptype()`.

**Examples**

```r
airlines_wrapped <-
  dm_nycflights13() %>%
  dm_wrap_tbl(airlines)

# The ptype is required for reconstruction.
# It can be an empty dm, only primary and foreign keys are considered.
ptype <- dm_ptype(dm_nycflights13())

airlines_wrapped %>%
  dm_unnest_tbl(airlines, flights, ptype)
```

---

**Description**

`# @description [Experimental]`

**Usage**

```r
dm_unpack_tbl(dm, child_table, col, ptype)
```
Arguments

- **dm**: A dm.
- **child_table**: A table in the dm with packed columns.
- **col**: The column to unpack (unquoted).
- **ptype**: A dm, only used to query names of primary and foreign keys.

Details

dm_unpack_tbl() targets a specific column to unpack from the given table in a given dm. A ptype or a set of keys should be given, not both.

dm_pack_tbl() is an inverse operation to dm_unpack_tbl() if differences in row and column order are ignored. The opposite is true if referential constraints between both tables are satisfied and if all rows in the parent table have at least one child row, i.e. if the relationship is of cardinality 1:n or 1:1.

See Also

dm_unwrap_tbl(), dm_unnest_tbl(), dm_nest_tbl(), dm_pack_tbl(), dm_wrap_tbl(), dm_examine_constraints(), dm_examine_cardinalities(), dm_ptype().

Examples

```r
flights_wrapped <-
  dm_nycflights13() %>%
  dm_wrap_tbl(flights)

# The ptype is required for reconstruction.
# It can be an empty dm, only primary and foreign keys are considered.
ptype <- dm_ptype(dm_nycflights13())

flights_wrapped %>%
  dm_unpack_tbl(flights, airlines, ptype)
```

Description

**Experimental**

dm_unwrap_tbl() unwraps all tables in a dm object so that the resulting dm matches a given ptype dm. It runs a sequence of dm_unnest_tbl() and dm_unpack_tbl() operations on the dm.

Usage

dm_unwrap_tbl(dm, ptype, progress = NA)
**dm_validate**

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dm</td>
<td>A dm.</td>
</tr>
<tr>
<td>ptype</td>
<td>A dm, only used to query names of primary and foreign keys.</td>
</tr>
<tr>
<td>progress</td>
<td>Whether to display a progress bar, if NA (the default) hide in non-interactive mode, show in interactive mode. Requires the 'progress' package.</td>
</tr>
</tbody>
</table>

**Value**

A dm.

**See Also**

`dm_wrap_tbl()`, `dm_unnest_tbl()`, `dm_examine_constraints()`, `dm_examine_cardinalities()`, `dm_ptype()`.

**Examples**

```r
roundtrip <-
  dm_nycflights13() %>%
  dm_wrap_tbl(root = flights) %>%
  dm_unwrap_tbl(ptype = dm_ptype(dm_nycflights13()))
roundtrip
# The roundtrip has the same structure but fewer rows:
  dm_nrow(dm_nycflights13())
  dm_nrow(roundtrip)
```

---

**dm_validate**

**Validator**

**Description**

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

**Usage**

`dm_validate(x)`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>An object.</td>
</tr>
</tbody>
</table>

**Details**

In theory, with the exception of `new_dm()`, all dm objects created or modified by functions in this package should be valid, and this function should not be needed. Please file an issue if any dm operation creates an invalid object.
Value

Returns the dm, invisibly, after finishing all checks.

Examples

dm_validate(dm())

bad_dm <- structure(list(bad = "dm"), class = "dm")
try(dm_validate(bad_dm))

---

**dm_wrap_tbl**

Wrap dm into a single tibble dm

---

Description

[Experimental]

dm_wrap_tbl() creates a single tibble dm containing the root table enhanced with all the data related to it through the relationships stored in the dm. It runs a sequence of dm_nest_tbl() and dm_pack_tbl() operations on the dm.

Usage

dm_wrap_tbl(dm, root, strict = TRUE, progress = NA)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dm</td>
<td>A cycle free dm object.</td>
</tr>
<tr>
<td>root</td>
<td>Table to wrap the dm into (unquoted).</td>
</tr>
<tr>
<td>strict</td>
<td>Whether to fail for cyclic dm's that cannot be wrapped into a single table, if FALSE a partially wrapped dm will be returned.</td>
</tr>
<tr>
<td>progress</td>
<td>Whether to display a progress bar, if NA (the default) hide in non-interactive mode, show in interactive mode. Requires the 'progress' package.</td>
</tr>
</tbody>
</table>

Details

dm_wrap_tbl() is an inverse to dm_unwrap_tbl(), i.e., wrapping after unwrapping returns the same information (disregarding row and column order). The opposite is not generally true: since dm_wrap_tbl() keeps only rows related directly or indirectly to rows in the root table. Even if all referential constraints are satisfied, unwrapping after wrapping loses rows in parent tables that don't have a corresponding row in the child table.

This function differs from dm_flatten_to_tbl() and dm_squash_to_tbl(), which always return a single table, and not a dm object.

Value

A dm object.
### 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("tech-db-zoom", package = "dm")` for a more detailed introduction.

**Usage**

```r
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**

- `dm` A `dm` object.
- `table` A table in the `dm`.
- `new_tbl_name` Name of the new table.
- `repair` Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", "universal", "unique_quiet", or "universal_quiet". 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.

The options "unique_quiet" and "universal_quiet" are here to help the user who calls this function indirectly, via another function which exposes repair but not quiet. Specifying repair = "unique_quiet" is like specifying repair = "unique", quiet = TRUE.

When the "*_quiet" options are used, any setting of quiet is silently overridden.

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.

Users can silence the name repair messages by setting the "rlib_name_repair_verbosity" global option to "quiet".

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 dm_zoomed: group_by(), summarise(), mutate(), transmute(), filter(), select(), rename() and ungroup(). You can use these functions just like you would with a normal table.

Calling filter() on a zoomed dm is different from calling dm_filter(): only with the latter, the filter expression is added to the list of table filters stored in the dm.

Furthermore, different join()-variants from dplyr are also supported, e.g. left_join() and semi_join(). (Support for nest_join() is planned.) The join-methods for dm_zoomed infer the columns to join by from the primary and foreign keys, and 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 dm_zoomed.

Value
For dm_zoom_to(): A dm_zoomed 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 %>%

### Description

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

### Usage

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

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

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

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

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

## S3 method for class 'dm_keyed_tbl'
full_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)
```
## S3 method for class 'dm_zoomed'
right_join(x, y, by = NULL, copy = NULL, suffix = NULL, select = NULL, ...)

## S3 method for class 'dm_keyed_tbl'
right_join(x, y, by = NULL, copy = NULL, suffix = NULL, ..., keep = FALSE)

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

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

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

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

## S3 method for class 'dm_zoomed'
nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...)

### Arguments

- **x, y**
  - tbs to join. x is the dm_zoomed 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 dm_zoomed. The table’s by column(s) are automatically added if missing in the selection.

- **...**
  - see dplyr::join

- **keep**
  - Should the new list-column contain join keys? The default will preserve the join keys for inequality joins.

- **name**
  - The name of the list-column created by the join. If NULL, the default, the name of y is used.

### Examples

```r
deflyr_join
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 `.dm_zoomed` suffix (see examples).

Usage

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

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

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

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

## S3 method for class 'dm_zoomed'
relocate(.data, ..., .before = NULL, .after = NULL)

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

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

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

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

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

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

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

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

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

## S3 method for class 'dm_zoomed'
count(
  x,
  ..., 
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .drop = group_by_drop_default(x)
)

## S3 method for class 'dm_zoomed'
tally(x, ...)

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

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

Arguments

.data object of class dm_zoomed

... see corresponding function in package dplyr or tidyr

.before, .after

<tidy-select> Destination of columns selected by .... Supplying neither will move columns to the left-hand side; specifying both is an error.

.keep_all For distinct.dm_zoomed(): see dplyr::distinct()

.keep_pk For slice.dm_zoomed: 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 slice.dm_zoomed() method.

x For ungroup.dm_zoomed: object of class dm_zoomed

wt <data-masking> Frequency weights. Can be NULL or a variable:

• If NULL (the default), counts the number of rows in each group.
• If a variable, computes \( \text{sum}(wt) \) for each group.
enum_pk_candidates

sort
   If TRUE, will show the largest groups at the top.

name
   The name of the new column in the output.
   If omitted, it will default to n. If there’s already a column called n, it will use
   nn. If there’s a column called n and nn, it’ll use nnn, and so on, adding ns until
   it gets a new name.

.drop
   Handling of factor levels that don’t appear in the data, passed on to group_by().
   For count(): if FALSE will include counts for empty groups (i.e. for levels of
   factors that don’t exist in the data).
   [Deprecated] For add_count(): deprecated since it can’t actually affect the
   output.

var
   A variable specified as:
      • a literal variable name
      • a positive integer, giving the position counting from the left
      • a negative integer, giving the position counting from the right.
   The default returns the last column (on the assumption that’s the column you’ve
   created most recently).
   This argument is taken by expression and supports quasiquotation (you can un-
   quote column names and column locations).

Examples

```r
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")
```

Description

[Experimental]
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, ...)

Primary key candidate
Arguments

- **table**: A table in the dm.
- **...**: These dots are for future extensions and must be empty.
- **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 "experimental" 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_add_uk(), dm_get_all_pks(), dm_get_all_uks(), dm_has_pk(), dm_rm_pk(), dm_rm_uk()

Examples

```r
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. Are all rows in x unique?
2. Are the rows in y a subset of the rows in x?
3. Does the relation between \( x \) and \( y \) meet the cardinality requirements? One row from \( x \) must correspond to the requested number of rows in \( y \), e.g. \( _0_1 \) means that there must be zero or one rows in \( y \) for each row in \( x \).

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

For convenience, the \( x \_select \) and \( y \_select \) arguments allow restricting the check to a set of key columns without affecting the return value.

**Usage**

```r
check_cardinality_0_n(
  x,
  y,
  ...,  
  x_select = NULL,
  y_select = NULL,
  by_position = NULL
)

check_cardinality_1_n(
  x,
  y,
  ...,  
  x_select = NULL,
  y_select = NULL,
  by_position = NULL
)

check_cardinality_1_1(
  x,
  y,
  ...,  
  x_select = NULL,
  y_select = NULL,
  by_position = NULL
)

check_cardinality_0_1(
  x,
  y,
  ...,  
  x_select = NULL,
  y_select = NULL,
  by_position = NULL
)

examine_cardinality(
  x,
```
Arguments

x
Parent table, data frame or lazy table.
y
Child table, data frame or lazy table.
...
These dots are for future extensions and must be empty.
x_select, y_select
Key columns to restrict the check, processed with `dplyr::select()`.
by_position
Set to TRUE to ignore column names and match by position instead. The default means matching by name, use `x_select` and/or `y_select` to align the names.

Details

All cardinality functions accept a parent and a child table (`x` and `y`). All rows in `x` must be unique, and all rows in `y` must be a subset of the rows in `x`. The `x_select` and `y_select` arguments allow restricting the check to a set of key columns without affecting the return value. If given, both arguments must refer to the same number of key columns.

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. "0", "1" and "n" refer to the occurrences of value combinations in `y` that correspond to each combination in the columns of the parent table. "n" means "more than one" in this context, with no upper limit.

"0_n": no restrictions, each row in `x` has at least 0 and at most `n` corresponding occurrences in `y`.

"1_n": each row in `x` has at least 1 and at most `n` corresponding occurrences in `y`. This means that there is a "surjective" mapping from the child table to the parent table, i.e. each parent table row exists at least once in the child table.

"0_1": each row in `x` has at least 0 and at most 1 corresponding occurrence in `y`. This means that there is a "injective" mapping from the child table to the parent table, i.e. no combination of values in the parent table columns is addressed multiple times. But not all parent table rows have to be referred to.

"1_1": each row in `x` occurs exactly once in `y`. This means that there is a "bijective" ("injective" AND "surjective") mapping between the child table and the parent table, i.e. the sets of rows are identical.

Finally, `examine_cardinality()` tests for and returns the nature of the relationship (injective, surjective, bijective, or none of these) between the two given sets of columns. If either `x` is not unique or there are rows in `y` that are missing from `x`, 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

check_cardinality(...) return x, invisibly, if the check is passed, to support pipes. Otherwise an error is thrown and the reason for it is explained.

examine_cardinality() returns a character variable specifying the type of relationship between the two columns.

See Also

Other cardinality functions: dm_examine_cardinalities()

Examples

d1 <- tibble::tibble(a = 1:5)
d2 <- tibble::tibble(a = c(1:4, 4L))
d3 <- tibble::tibble(c = c(1:5, 5L), d = 0)
# This does not pass, 'a' is not unique key of d2:
try(check_cardinality_0_n(d2, d1))

# Columns are matched by name by default:
try(check_cardinality_0_n(d1, d3))

# This passes, multiple values in d3$c are allowed:
check_cardinality_0_n(d1, d2)

# This does not pass, injectivity is violated:
try(check_cardinality_1_1(d1, d3, y_select = c(a = c)))
try(check_cardinality_0_1(d1, d3, x_select = c(c = a)))

# What kind of cardinality is it?
examine_cardinality(d1, d3, x_select = c(c = a))
examine_cardinality(d1, d2)

Description

glimpse() provides an overview (dimensions, column data types, primary keys, etc.) of all tables included in the dm object. It will additionally print details about outgoing foreign keys for the child table.

glimpse() is provided by the pillar package, and re-exported by dm. See pillar::glimpse() for more details.
Usage

## S3 method for class 'dm'
glimpse(x, width = NULL, ...)

## S3 method for class 'dm_zoomed'
glimpse(x, width = NULL, ...)

Arguments

x A `dm` object.

width Controls the maximum number of columns on a line used in printing. If `NULL`, `getOption("width")` will be consulted.

... Passed to `pillar::glimpse()`.

Examples

dm_nycflights13() %>% glimpse()

dm_nycflights13() %>%
  dm_zoom_to(flights) %>%
  glimpse()

head.dm_zoomed utils table manipulation methods for `dm_zoomed` objects

Description

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

Usage

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

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

Arguments

x object of class `dm_zoomed`
n

an integer vector of length up to \( \text{dim}(x) \) (or 1, for non-dimensioned objects). A logical is silently coerced to integer. 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 \( |n[i]| \), including all remaining indices. \( NA \) or non-specified values (when \( \text{length}(n) < \text{length(dim}(x)) \)) select all indices in that dimension. Must contain at least one non-missing value.

Details

see manual for the corresponding functions in \texttt{utils}.

Value

A \texttt{dm\_zoomed} object.

Examples

```r
zoomed <- \texttt{dm\_nycflights13()} %>%
  \texttt{dm\_zoom\_to(flights)} %>%
  \texttt{head(4)}
zoomed
\texttt{dm\_insert\_zoomed(zoomed, new\_tbl\_name = "head\_flights")}
```

json_nest

\textit{JSON nest}

Description

\textbf{[Experimental]}

A wrapper around \texttt{tidyr::nest()} which stores the nested data into JSON columns.

Usage

```
json_nest(.data, ..., .names_sep = NULL)
```

Arguments

\begin{itemize}
  \item \texttt{.data} A data frame, a data frame extension (e.g. a tibble), or a lazy data frame (e.g. from \texttt{dbplyr} or \texttt{dplyr}).
  \item \texttt{...} \texttt{<tidy-select>} Columns to pack, specified using name-variable pairs of the form \texttt{new\_col = c(col1, col2, col3)}. The right hand side can be any valid tidy select expression.
  \item \texttt{.names_sep} If \texttt{NULL}, the default, the names will be left as is.
\end{itemize}
See Also

`tidyr::nest()`, `json_nest_join()`

Examples

```r
df <- tibble::tibble(x = c(1, 1, 1, 2, 2, 3), y = 1:6, z = 6:1)
nested <- json_nest(df, data = c(y, z))
nested
```

Description

[Experimental]

A wrapper around `dplyr::nest_join()` which stores the joined data into a JSON column. `json_nest_join()` returns all rows and columns in `x` with a new JSON column that contains all nested matches from `y`.

Usage

`json_nest_join(x, y, by = NULL, ..., copy = FALSE, keep = FALSE, name = NULL)`

Arguments

- `x`, `y`:
  A pair of data frames or data frame extensions (e.g. a tibble).
- `by`:
  A join specification created with `join_by()`, or a character vector of variables to join by.
  If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.
  To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.
  To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.
  `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at `?join_by` for details on these types of joins.
  For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.
  To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
- `...`:
  Other parameters passed onto methods.
If \( x \) and \( y \) are not from the same data source, and `copy` is `TRUE`, then \( y \) will be copied into the same src as \( x \). This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

**keep**

Should the new list-column contain join keys? The default will preserve the join keys for inequality joins.

**name**

The name of the list-column created by the join. If `NULL`, the default, the name of \( y \) is used.

---

**Examples**

```r
df1 <- tibble::tibble(x = 1:3)
df2 <- tibble::tibble(x = c(1, 1, 2), y = c("first", "second", "third"))
df3 <- json_nest_join(df1, df2)
df3
df3$df2
```

---

**Description**

[A wrapper around `tidyr::pack()` which stores the packed data into JSON columns.]

---

**Usage**

```r
json_pack(.data, ..., .names_sep = NULL)
```

---

**Arguments**

- `.data` A data frame, a data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
- `...` `<tidy-select>` Columns to pack, specified using name-variable pairs of the form `new_col = c(col1, col2, col3)`. The right hand side can be any valid tidy select expression.
- `.names_sep` If `NULL`, the default, the names will be left as is.

---

**See Also**

`tidyr::pack()`, `json_pack_join()`
Examples

```r
df <- tibble::tibble(x1 = 1:3, x2 = 4:6, x3 = 7:9, y = 1:3)
packed <- json_pack(df, x = c(x1, x2, x3), y = y)
packed
```

---

**Description**

[Experimental]

A wrapper around `pack_join()` which stores the joined data into a JSON column. `json_pack_join()` returns all rows and columns in `x` with a new JSON column that contains all packed matches from `y`.

**Usage**

```r
json_pack_join(x, y, by = NULL, ..., copy = FALSE, keep = FALSE, name = NULL)
```

**Arguments**

- `x, y` A pair of data frames or data frame extensions (e.g. a tibble).
- `by` A join specification created with `join_by()`, or a character vector of variables to join by.
  - If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they’re correct; suppress the message by supplying `by` explicitly.
  - To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.
  - To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.
  - `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.
  - For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.
  - To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
- `...` Other parameters passed onto methods.
- `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
json_unnest

keep
Should the new list-column contain join keys? The default will preserve the join keys for inequality joins.

name
The name of the list-column created by the join. If NULL, the default, the name of y is used.

See Also
pack_join(), json_nest_join()

Examples

df1 <- tibble::tibble(x = 1:3)
df2 <- tibble::tibble(x = c(1, 1, 2), y = c("first", "second", "third"))
df3 <- json_pack_join(df1, df2)
df3
df3$df2

json_unnest Unnest a JSON column

Description
A wrapper around tidyr::unnest() that extracts its data from a JSON column. The inverse of json_nest().

Usage
json_unnest(data, cols, ..., names_sep = NULL, names_repair = "check_unique")

Arguments

data
A data frame, a data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).

cols
<List-columns> List-columns to unnest.
When selecting multiple columns, values from the same row will be recycled to their common size.

... Arguments passed to methods.

names_sep
If NULL, the default, the outer names will come from the inner names. If a string, the outer names will be formed by pasting together the outer and the inner column names, separated by names_sep.

names_repair Used to check that output data frame has valid names. Must be one of the following options:
• "minimal": no name repair or checks, beyond basic existence,
• "unique": make sure names are unique and not empty,
• "check_unique": (the default), no name repair, but check they are unique,
• "universal": make the names unique and syntactic
json_unpack

• a function: apply custom name repair.
• tidyr_legacy: use the name repair from tidyr 0.8.
• a formula: a purrr-style anonymous function (see rlang::as_function())

See vctrs::vec_as_names() for more details on these terms and the strategies used to enforce them.

Value

An object of the same type as `data`

Examples

tibble(a = 1, b = '[[ { "c": 2 }, { "c": 3 } ]]' %>%
  json_unnest(b)

---

Description

A wrapper around tidyr::unpack() that extracts its data from a JSON column. The inverse of json_pack().

Usage

`json_unpack(data, cols, ..., names_sep = NULL, names_repair = "check_unique")`

Arguments

data A data frame, a data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
cols <tidy-select> Columns to unpack.
... Arguments passed to methods.
names_sep If NULL, the default, the names will be left as is. In pack(), inner names will come from the former outer names; in unpack(), the new outer names will come from the inner names.

If a string, the inner and outer names will be used together. In unpack(), the names of the new outer columns will be formed by pasting together the outer and the inner column names, separated by `names_sep`. In pack(), the new inner names will have the outer names + `names_sep` automatically stripped. This makes `names_sep` roughly symmetric between packing and unpacking.
names_repair Used to check that output data frame has valid names. Must be one of the following options:
  • "minimal": no name repair or checks, beyond basic existence,
  • "unique": make sure names are unique and not empty,
• "check_unique": (the default), no name repair, but check they are unique,
  • "universal": make the names unique and syntactic
  • a function: apply custom name repair.
  • tidyr_legacy: use the name repair from tidyr 0.8.
  • a formula: a purrr-style anonymous function (see rlang::as_function())

See vctrs::vec_as_names() for more details on these terms and the strategies
used to enforce them.

Value

An object of the same type as data

Examples

tibble(a = 1, b = '{ "c": 2, "d": 3 }') %>%
  json_unpack(b)

---

Materialize

Description

compute() materializes all tables in a dm to new temporary tables on the database.
collect() downloads the tables in a dm object as local tibbles.

Usage

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

## S3 method for class 'dm'
collect(x, ..., progress = NA)

Arguments

x
  A dm object.

... Passed on to compute().

temporary Must remain TRUE.

progress Whether to display a progress bar, if NA (the default) hide in non-interactive
  mode, show in interactive mode. Requires the 'progress' package.

Details

Called on a dm object, these methods create a copy of all tables in the dm. Depending on the size
of your data this may take a long time.

To create permanent tables, first create the database schema using copy_dm_to() or dm_sql(),
and then use dm_rows_append().
Value

A dm object of the same structure as the input.

Examples

```r
financial <- dm_financial_sqlite()
financial %>%
  pull_tbl(districts) %>%
  dbplyr::remote_name()

# compute() copies the data to new tables:
financial %>%
  compute() %>%
  pull_tbl(districts) %>%
  dbplyr::remote_name()

# collect() returns a local dm:
financial %>%
  collect() %>%
  pull_tbl(districts) %>%
  class()
```

pack_join

**Pack Join**

Description

[Experimental]

pack_join() returns all rows and columns in x with a new packed column that contains all matches from y.

Usage

```r
pack_join(x, y, by = NULL, ..., copy = FALSE, keep = FALSE, name = NULL)
```

## S3 method for class 'dm_zoomed'

```r
pack_join(x, y, by = NULL, ..., copy = FALSE, keep = FALSE, name = NULL)
```

Arguments

- **x**, **y**  A pair of data frames or data frame extensions (e.g. a tibble).
- **by**  A join specification created with `join_by()`, or a character vector of variables to join by.
If NULL, the default, *_.join() will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly.

To join on different variables between x and y, use a *join_by() specification. For example, join_by(a == b) will match x$a to y$b.

To join by multiple variables, use a *join_by() specification with multiple expressions. For example, join_by(a == b, c == d) will match x$a to y$b and x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

*join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

... Other parameters passed onto methods.

*copy* If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

*keep* Should the new list-column contain join keys? The default will preserve the join keys for inequality joins.

*name* The name of the list-column created by the join. If NULL, the default, the name of y is used.

See Also

dplyr::nest_join(), tidyr::pack()

Examples

df1 <- tibble::tibble(x = 1:3)
df2 <- tibble::tibble(x = c(1, 1, 2), y = c("first", "second", "third"))
pack_join(df1, df2)
reunite_parent_child

Usage

```r
pull_tbl(dm, table, ..., keyed = FALSE)
```

Arguments

- `dm` A `dm` object.
- `table` One unquoted table name for `pull_tbl.dm()`, ignored for `pull_tbl.dm_zoomed()`.
- `...` These dots are for future extensions and must be empty.
- `keyed` [Experimental] Set to TRUE to return objects of the internal class "dm_keyed_tbl" that will contain information on primary and foreign keys in the individual table objects. This allows using dplyr workflows on those tables and later reconstruct them into a `dm` object. See `dm_deconstruct()` for a function that generates corresponding code for an existing `dm` object, and vignette("tech-dm-keyed") for details.

Value

The requested table.

See Also

- `dm_deconstruct()` to generate code of the form `pull_tbl(..., keyed = TRUE)` from an existing `dm` object.

Examples

```r
# 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()
```

Description

[Experimental]

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

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

```r
reunite_parent_child(child_table, parent_table, id_column)
reunite_parent_child_from_list(list_of_parent_child_tables, id_column)
```

Arguments

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

Value

A wide table produced by joining the two given tables.

Life cycle

These functions are marked "experimental" because they seem more useful when applied to a table in a `dm` object. Changing the interface later seems harmless because these functions are most likely used interactively.

See Also

Other table surgery functions: `decompose_table()`

Examples

```r
decomposed_table <- decompose_table(mtcars, new_id, am, gear, carb)
c <- decomposed_table$child_table
p <- decomposed_table$parent_table
reunite_parent_child(c, p, new_id)
reunite_parent_child_from_list(decomposed_table, new_id)
```
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()` with `conflict = "ignore"`. Duplicate records will be silently discarded. This operation requires primary keys on all tables, use `dm_rows_append()` to insert unconditionally.

`dm_rows_append()` adds new records via `rows_append()`. 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.

Usage

```r
dm_rows_insert(x, y, ..., in_place = NULL, progress = NA)
dm_rows_append(x, y, ..., in_place = NULL, progress = NA)
dm_rows_update(x, y, ..., in_place = NULL, progress = NA)
dm_rows_patch(x, y, ..., in_place = NULL, progress = NA)
dm_rows_upsert(x, y, ..., in_place = NULL, progress = NA)
dm_rows_delete(x, y, ..., in_place = NULL, progress = NA)
```
Arguments

- **x**: Target `dm` object.
- **y**: `dm` object with new data.
- **...**: These dots are for future extensions and must be empty.
- **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.
- **progress**: Whether to display a progress bar, if `NA` (the default) hide in non-interactive mode, show in interactive mode. Requires the `progress` package.

Value

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

Examples

```r
# 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_append(flights_sqlite, flights_jan_sqlite)
print(dm_nrow(flights_sqlite))

# Explicitly request persistence:
dm_rows_append(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_append(
  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_append(flights_sqlite, flights_feb_sqlite, in_place = TRUE)
print(dm_nrow(flights_sqlite))

DBI::dbDisconnect(sqlite)

---
tidyrlibrary

**tidy_r table manipulation methods for zoomed dm objects**

**Description**

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

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

## S3 method for class 'dm_keyed_tbl'
unite(data, ...)

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

## S3 method for class 'dm_keyed_tbl'
separate(data, ...)
```

Arguments

- `data` object of class `dm_zoomed`
- `col` For `unite.dm_zoomed`: see `tidyr::unite()
  For `separate.dm_zoomed`: see `tidyr::separate`
- `...` For `unite.dm_zoomed`: see `tidyr::unite()`
  For `separate.dm_zoomed`: see `tidyr::separate`
- `sep` For `unite.dm_zoomed`: see `tidyr::unite()
  For `separate.dm_zoomed`: see `tidyr::separate`
- `remove` For `unite.dm_zoomed`: see `tidyr::unite()
  For `separate.dm_zoomed`: see `tidyr::separate`
- `na.rm` see `tidyr::unite()
- `into` see `tidyr::separate`

Examples

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

- **DB interaction functions**
  - copy_dm_to, 6

- **cardinality functions**
  - dm_examine_cardinalities, 25
  - examine_cardinality, 66

- **flattening functions**
  - dm_flatten_to_tbl, 30

- **foreign key functions**
  - dm_add_fk, 15
  - dm_enum_fk_candidates, 24
  - dm_get_all_fks, 33
  - dm_rm_fk, 46

- **primary key functions**
  - dm_add_pk, 17
  - dm_add_uk, 19
  - dm_get_all_pks, 34
  - dm_get_all_uks, 35
  - dm_has_pk, 38
  - dm_rm_pk, 47
  - dm_rm_uk, 48
  - enum_pk_candidates, 65

- **schema handling functions**
  - db_schema_create, 8
  - db_schema_drop, 9
  - db_schema_exists, 10
  - db_schema_list, 11

- **table surgery functions**
  - decompose_table, 12
  - reunite_parent_child, 80

- **check functions**
  - check_cardinality_0_1 (examine_cardinality), 66
  - check_cardinality_0_n (examine_cardinality), 66
  - check_cardinality_1_1 (examine_cardinality), 66
  - check_cardinality_1_n (examine_cardinality), 66
  - check_key, 4
  - check_key(), 14
  - check_subset, 6
  - check_subset(), 5, 14
  - count.dm (materialize), 77
  - compute(), 77, 82
  - compute.dm (materialize), 77
  - db_schema_create, 8, 10, 11
  - db_schema_drop, 9, 9, 11
  - db_schema_exists, 9, 10, 10, 11
  - db_schema_list, 9–11, 11
  - DBI::DBIConnection, 6, 32, 36
  - DBI::dbQuoteIdentifier(), 7, 9
  - DBI::Id, 7, 54
  - DBI::SQL, 54
  - dbplyr::ident(), 54
  - dbplyr::in_catalog(), 54
  - dbplyr::in_schema(), 54
  - decompose_table, 12, 81
  - DiagrammeR::grViz(), 23
  - DiagrammeRsvg::export_svg(), 23
dm_unwrap_tbl, 56
dm_unwrap_tbl(), 40, 42, 55, 56, 59
dm_update_zoomed(dm_zoom_to), 59
dm_validate, 57
dm_validate(), 13
dm_wrap_tbl, 58
dm_wrap_tbl(), 40, 42, 55–57
dm_zoom_to, 59
dm_zoom_to(), 28
dm_zoomed_df(dm_zoom_to), 59
dplyr::distinct(), 64
dplyr::filter(), 28
dplyr::join, 62
dplyr::join(), 30
dplyr::nest_join(), 72, 73, 79
dplyr::select(), 4–6, 30, 50–52, 68
dplyr::src_dbi, 6
dplyr_join, 61
dplyr_table_manipulation, 63
enum_fk_candidates
d(dm_enum_fk_candidates), 24
enum_pk_candidates, 18, 19, 34, 35, 38, 48, 49, 65
examine_cardinality, 26, 66
examine_cardinality(), 14, 26
filter(), 60
filter.dm_zoomed
d(dmplyr_table_manipulation), 63
full_join.dm_keyed_tbl(dmplyr_join), 61
full_join.dm_zoomed(dmplyr_join), 61
glimpse.dm, 69
glimpse.dm_zoomed(glimpse.dm), 69
grDevices::colors(), 51
group_by(), 60, 65
group_by.dm_keyed_tbl
(dplyr_table_manipulation), 63
group_by.dm_zoomed
(dplyr_table_manipulation), 63
head.dm_zoomed, 70
inner_join.dm_keyed_tbl(dmplyr_join), 61
inner_join.dm_zoomed(dmplyr_join), 61
is.dm(dm), 13
join_by(), 72, 74, 78, 79
json_nest, 71
json_nest(), 75
json_nest_join, 72
json_nest_join(), 72, 75
json_pack, 73
json_pack(), 76
json_pack_join, 74
json_pack_join(), 73
json_unnest, 75
json_unpack, 76
left_join(), 60
left_join.dm_keyed_tbl(dmplyr_join), 61
left_join.dm_zoomed(dmplyr_join), 61
materialize, 77
mutate(), 60
mutate.dm_zoomed
(dplyr_table_manipulation), 63
nest_join(), 60
nest_join.dm_keyed(dmplyr_join), 61
new_dm(dm), 13
new_dm(), 37, 57
nycflights13::flights, 41
nycflights13::planes, 41
pack_join, 78
pack_join(), 74, 75
pillar::glimpse(), 69, 70
print(), 23
pull.dm_zoomed
(dplyr_table_manipulation), 63
pull_tbl, 79
pull_tbl(), 20
quasiquotation, 65
relocate.dm_zoomed
(dplyr_table_manipulation), 63
rename(), 60
rename.dm_zoomed
(dplyr_table_manipulation), 63
reunite_parent_child, 13, 80
reunite_parent_child_from_list
(reunite_parent_child), 80
right_join.dm_keyed(dmplyr_join), 61
right_join.dm_zoomed(dmplyr_join), 61
rlang::as_function(), 7, 76, 77
rows-dm, 82
rows_append(), 82
INDEX

rows_delete(), 82
rows_insert(), 82
rows_patch(), 82
rows_update(), 82
rows_upsert(), 82

select(), 60
select.dm_zoomed
dplyr_table_manipulation, 63

semi_join(), 60
semi_join.dm_keyed_tbl (dplyr_join), 61
semi_join.dm_zoomed (dplyr_join), 61

separate(), 60
separate.dm_keyed_tbl
tidyrr_table_manipulation, 84

slice.dm_zoomed
dplyr_table_manipulation, 63

src, 31

summarise(), 60
summarise.dm_keyed_tbl
dplyr_table_manipulation, 63
summarise.dm_zoomed
dplyr_table_manipulation, 63

tail.dm_zoomed (head.dm_zoomed), 70

tally.dm_zoomed
dplyr_table_manipulation, 63

tbl, 13, 36
tibble, 77
tibble(), 43
tibble::enframe(), 51
tidyrr::nest(), 71, 72
tidyrr::pack(), 73, 79
tidyrr::separate(), 85
tidyrr::unite(), 85
tidyrr::unnest(), 75
tidyrr::unpack(), 76
tidyrr_legacy, 76, 77
tidyrr_table_manipulation, 84

transmute(), 60
transmute.dm_zoomed
dplyr_table_manipulation, 63

ungroup(), 60
ungroup.dm_zoomed
dplyr_table_manipulation, 63

unite(), 60
unite.dm_keyed_tbl
tidyrr_table_manipulation, 84
unite.dm_zoomed
tidyrr_table_manipulation, 84

vctrs::vec_as_names(), 14, 76, 77

zoomed_df (dm_zoom_to), 59