Package 'dm'

June 20, 2021

Title Relational Data Models

Version 0.2.3

Date 2021-06-20

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

License MIT + file LICENSE

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

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

Depends R (>= 3.3)

Imports backports, cli (>= 2.2.0), DBI, dplyr (>= 1.0.3), ellipsis,
glue, igraph, lifecycle, magrittr, memoise, methods, pillar,
purrr, rlang (>= 0.4.10), tibble, tidyr (>= 1.0.0), tidyselect
(>= 1.0.1), vctrs (>= 0.3.2)

Suggests brio, covr, crayon, dbplyr, DiagrammeR, DiagrammeRsvg,
digest, duckdb, fansi, keyring, knitr, mockr, nycflights13,
odbc, progress, RMariaDB (>= 1.0.10), rmarkdown, RPostgres,
RSQLite, testthat (>= 3.0.2), tidyverse, withr

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.1.1.9001

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first zzx-deprecated, flatten, dplyr, filter-dm,
draw-dm, bind, rows-dm

NeedsCompilation no

Author Tobias Schieferdecker [aut],
Kirill Müller [aut, cre] (<<https://orcid.org/0000-0002-1416-3412>>),
Darko Bergant [aut],

Katharina Brunner [ctb],
James Wondrasek [ctb],
energie360° AG [fnd],
cynkra GmbH [fnd, cph]

R **topics documented:**

---

check_key *Check if column(s) can be used as keys*

Description

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

Usage

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

Arguments

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

...   The names of the columns to be checked.

               One or more unquoted expressions separated by commas. Variable names can
               be treated 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 these concepts.

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

Value

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

Examples

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

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

---

check_set_equality   *Check column values for set equality*

Description

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

Usage

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

Arguments

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

Value

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

Examples

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

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

---

check_subset   *Check column values for subset*

Description

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

Usage

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

Arguments

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

Value

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

Examples

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

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

---

copy_dm_to   *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,
  ...,
  types = NULL,
  overwrite = NULL,
  indexes = NULL,
  unique_indexes = NULL,
  set_key_constraints = TRUE,
  unique_table_names = NULL,
  table_names = NULL,
  temporary = TRUE,
  schema = NULL,
  progress = NA,
  copy_to = NULL
)
```

Arguments

dest	An object of class "src" or "DBIConnection".
dm	A dm object.
...	Passed on to [dplyr::copy_to()](#) or to the function specified by the copy_to argument.

overwrite, types, indexes, unique_indexes

                Must remain NULL.

set_key_constraints

                If TRUE will mirror dm primary and foreign key constraints on a database and create unique indexes. Set to FALSE if your data model currently does not satisfy primary or foreign key constraints.

unique_table_names

                Deprecated.

table_names	Desired names for the tables on dest; the names within the dm remain unchanged. Can be NULL, a named character vector, a function or a one-sided formula.

                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.

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.
copy_to	By default, dplyr::copy_to() is called to upload the individual tables to the target data source. This argument allows overriding the standard behavior in cases when the default does not work as expected, such as spatial data frames or other tables with special data types. If not NULL, this argument is processed with rlang::as_function().

Details

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

Value

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

Examples

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

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

# Persist, explicitly specify table names:
persistent_dm <- copy_dm_to(
  con,
  dm_nycflights13(),
  temporary = FALSE,
```

```
    table_names = ~ paste0("flights_", .x)
  )
  dbplyr::remote_name(persistent_dm$planes)

  DBI::dbDisconnect(con)
```

---

decompose_table   *Decompose a table into two linked tables*

Description

[Questioning]

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

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

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

Usage

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

Arguments

.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 "questioning" because it feels more useful when applied to a table in a dm object.

See Also

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

Examples

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

---

dm   *Data model class*

Description

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

dm() creates a dm object from [tbl](#) objects (tibbles or lazy data objects).

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

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

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.

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

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

as_dm() coerces objects to the dm class

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

Usage

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

new_dm(tables = list())

dm_get_con(x)
```

```
dm_get_tables(x)

is_dm(x)

as_dm(x)

validate_dm(x)
```

Arguments

`...`	Tables to add to the dm object. If no names are provided, the tables are auto-named.
`.name_repair`	Options for name repair. Forwarded as `repair` to [`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.

Details

All lazy tables in a dm object must be stored on the same database server and accessed through the same connection.

Value

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

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

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

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

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

See Also

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

Examples

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


dm_nycflights13()$airports
dm_nycflights13() %>% names()

copy_dm_to(
  dbplyr::src_memdb(),
  dm_nycflights13()
) %>%
  dm_get_con()

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

dm_add_fk	*Add foreign keys*

Description

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

Usage

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

Arguments

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

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:
nycflights_dm %>%
  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:
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, ..., 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.
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.

Value

An updated dm with an additional primary key.

See Also

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

Examples

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

nycflights_dm %>%
  dm_draw()

# Create primary keys:
```

```
nycflights_dm %>%
  dm_add_pk(planes, tailnum) %>%
  dm_add_pk(airports, faa, check = TRUE) %>%
  dm_add_pk(weather, c(origin, time_hour)) %>%
  dm_draw()

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

---

dm_add_tbl   *Add tables to a* [dm](#)

Description

Adds one or more new tables to a [dm](#). Existing tables are not overwritten.

Usage

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

Arguments

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

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

The "check_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria. |
| quiet | By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set quiet to TRUE to silence the messages. |

Value

The initial dm with the additional table(s).

See Also

dm_mutate_tbl(), dm_rm_tbl()

Examples

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

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

---

dm_bind   *Merge several* dm

Description

Create a single dm from two or more dm objects.

Usage

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

Arguments

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

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

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

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

Details

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

Value

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

Examples

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

---

dm_disambiguate_cols   *Resolve column name ambiguities*

Description

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

Usage

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

Arguments

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

Details

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

Value

A dm whose column names are unambiguous.

Examples

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

---

dm_draw   *Draw a diagram of the data model*

Description

dm_draw() uses **DiagrammeR** to draw diagrams. Use `DiagrammeRsvg::export_svg()` to convert the diagram to an SVG file.

Usage

```
dm_draw(
  dm,
  rankdir = "LR",
  col_attr = NULL,
  view_type = c("keys_only", "all", "title_only"),
  columnArrows = TRUE,
  graph_attrs = "",
  node_attrs = "",
  edge_attrs = "",
  focus = NULL,
  graph_name = "Data Model",
  ...,
  column_types = NULL
)
```

Arguments

dm	A dm object.
rankdir	Graph attribute for direction (e.g., 'BT' = bottom –> top).
col_attr	Deprecated, use `colummn_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`).
graph_attrs	Additional graph attributes.
node_attrs	Additional node attributes.
edge_attrs	Additional edge attributes.
focus	A list of parameters for rendering (table filter).
graph_name	The name of the graph.
...	These dots are for future extensions and must be empty.
column_types	Set to `TRUE` to show column types.

Value

An object of class grViz (see also `DiagrammeR::grViz()`), which, when printed, produces the output seen in the viewer as a side effect.

See Also

[dm_set_colors()](#) for defining the table colors.

Examples

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

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

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

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

---

dm_enum_fk_candidates   *Foreign key candidates*

Description

[Questioning]

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

Usage

```
dm_enum_fk_candidates(dm, table, ref_table, ...)

enum_fk_candidates(zoomed_dm, ref_table, ...)
```

Arguments

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.
zoomed_dm	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:

- an empty value for a column of table that is a suitable foreign key candidate

- the count and percentage of missing matches for a column that is not suitable

- the error message triggered for unsuitable candidates that may include the types of mismatched columns

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

Value

A tibble with the following columns:

columns columns of table,

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

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

Life cycle

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

See Also

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

Examples

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

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

---

dm_examine_constraints

Validate your data model

Description

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

Usage

```
dm_examine_constraints(dm, progress = NA)
```

Arguments

dm   A dm object.

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

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

Value

A tibble with the following columns:

table the table in the dm,

kind "PK" or "FK",

columns the table columns that define the key,

ref_table for foreign keys, the referenced table,

is_key logical,

problem if is_key = FALSE, the reason for that.

Examples

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

---

dm_filter   *Filtering*

Description

[Questioning]

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

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

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

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

Usage

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

dm_apply_filters(dm)

dm_apply_filters_to_tbl(dm, table)
```

Arguments

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

Details

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

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

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

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

Value

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

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

For dm_apply_filters_to_tbl(), a table.

Life cycle

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

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

Examples

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

dm_apply_filters_to_tbl(dm_nyc_filtered, flights)

dm_nyc_filtered %>%
  dm_apply_filters()

# If you want to keep only those rows in the parent tables
# whose primary key values appear as foreign key values in
# `flights`, you can set a `TRUE` filter in `flights`:
dm_nyc %>%
  dm_filter(flights, 1 == 1) %>%
  dm_apply_filters() %>%
  dm_nrow()
# note that in this example, the only affected table is
# `airports` because the departure airports in `flights` are
# only the three New York airports.


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

```
compute()
```

---

dm_financial   *Creates a dm object for the Financial data*

Description

[Experimental]

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

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

Usage

```
dm_financial()

dm_financial_sqlite()
```

Value

A dm object.

Examples

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

---

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

Description

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

Usage

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

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

Arguments

dm	A [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.
...	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. If this includes tables that are not direct neighbors of start, it will only work with dm_squash_to_tbl() (given one of the allowed join-methods). tidyselect is supported, see [dplyr::select()](#) for details on the semantics.
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.

How does filtering affect the result?

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

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

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

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

Value

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

See Also

Other flattening functions: `dm_join_to_tbl`()

Examples

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

---

dm_from_src	*Load a dm from a remote data source*

Description

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

Usage

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

Arguments

`src`	A **dplyr** table source object or a `DBI::DBIConnection` object is accepted.
`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.
`...`	**[Experimental]** Additional parameters for the schema learning query.

- schema: supported for MSSQL (default: `"dbo"`), Postgres (default: `"public"`), and MariaDB/MySQL (default: current database). Learn the tables in a specific schema (or database for MariaDB/MySQL).
- dbname: supported for MSSQL. Access different databases on the connected MSSQL-server; default: active database.
- table_type: supported for Postgres (default: `"BASE TABLE"`). Specify the table type. Options are:
  1. `"BASE TABLE"` for a persistent table (normal table type)
  2. `"VIEW"` for a view
  3. `"FOREIGN TABLE"` for a foreign table
  4. `"LOCAL TEMPORARY"` for a temporary table

Value

A dm object.

Examples

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

dm_from_src(con)

DBI::dbDisconnect(con)
```

---

dm_get_all_fks   *Get foreign key constraints*

Description

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

Usage

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

Arguments

dm	A dm object.
parent_table	One or more table names, as character vector, to return foreign key information for. 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.

See Also

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

Examples

```
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 set primary keys and returns the tables, the respective primary key columns and their classes.

Usage

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

Arguments

dm	A dm object.
table	One or more table names, as character vector, to return primary key information for. 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_cols column name(s) of primary key, as list of character vectors.

See Also

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

Examples

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

---

dm_get_filters   *Get filter expressions*

Description

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

Usage

```
dm_get_filters(x)
```

Arguments

x   An object.

Value

A tibble with the following columns:

table table that was filtered,

filter the filter expression,

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

---

dm_get_referencing_tables
                        Get the names of referencing tables

Description

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

Usage

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

Arguments

dm   A dm object.

table   A table in the dm.

Value

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

See Also

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

Examples

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

---

dm_has_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_get_all_pks()`, `dm_rm_pk()`, `enum_pk_candidates()`

Examples

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

---

dm_is_referenced   *Check foreign key reference*

Description

Is a table of a dm referenced by another table?

Usage

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

Arguments

dm   A dm object.

table   A table in the dm.

Value

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

See Also

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

Examples

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

---

dm_join_to_tbl   *Join two tables*

Description

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

Usage

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

Arguments

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

Value

The resulting table of the join.

See Also

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

Examples

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

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

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

---

dm_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 levied optionally in the future.

Usage

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

Arguments

dm	A [dm](#) object.
...	One or more tables to update in the dm. Must be named.

See Also

dm_add_tbl(), dm_rm_tbl()

Examples

```
dm_nycflights13() %>%
  dm_mutate_tbl(flights = nycflights13::flights[1:3, ])
```

---

dm_nrow   *Number of rows*

Description

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

Usage

```
dm_nrow(dm)
```

Arguments

dm   A dm object.

Value

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

Examples

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

---

dm_nycflights13   *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

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

Arguments

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.

Value

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

Examples

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

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

```
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() and dm_add_fk() 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

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


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

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

---

dm_ptype   *Prototype for a dm object*

Description

[Experimental]

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

Usage

```
dm_ptype(dm)
```

Arguments

dm   A dm object.

Examples

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

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

---

dm_rename   *Rename columns*

Description

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

Usage

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

Arguments

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

```
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

```
dm_rm_fk(
  dm,
  table = NULL,
  columns = NULL,
  ref_table = NULL,
  ref_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.
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

dm_rm_pk() removes one or more primary keys from a table and leaves the [dm](#) object otherwise unaltered. An error is thrown if no private 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 = TRUE)
```

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	Boolean: if TRUE (default), will throw an error if there are foreign keys addressing the primary key that is to be removed.

Value

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

See Also

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

Examples

```
dm_nycflights13() %>%
  dm_rm_pk(airports, fail_fk = FALSE) %>%
  dm_draw()
```

---

dm_rm_tbl   *Remove tables*

Description

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

Usage

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

Arguments

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

Value

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

See Also

[dm_add_tbl()](), [dm_select_tbl()]()

Examples

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

---

dm_select   *Select columns*

Description

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

Usage

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

Arguments

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. If key columns are removed, then all related relations are dropped as well.

Value

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

Examples

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

---

dm_select_tbl   *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 sup-
                ported, see dplyr::select() for details on the semantics.

Value

The input dm with tables renamed or removed.

See Also

dm_rm_tbl()

Examples

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


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

---

dm_set_colors   *Color in database diagrams*

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 <http://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

```
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 two-column tibble with one row per table.

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

Examples

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

# Splicing is supported:
```

```
nyc_cols <-
  dm_nycflights13() %>%
  dm_get_colors()
nyc_cols

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

---

dm_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

```
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", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown.

- Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.

- Unique names are unique. A suffix is appended to duplicate names to make them unique.

- Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error.

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

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

Details

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

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

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 zoomed_dm 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 zoomed_dm.

Value

For dm_zoom_to(): A zoomed_dm object.

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

Examples

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

flights_zoomed

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

flights_zoomed_transformed
```

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

# insert the zoomed table as a new table
flights_zoomed_transformed %>%
  dm_insert_zoomed("extended_flights") %>%
  dm_draw()

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

---

dplyr_join   **dplyr** *join methods for zoomed dm objects*

Description

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

Usage

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

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

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

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

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

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

Arguments

x, y   tbls to join. x is the zoomed_dm and y is another table in the dm.

by   If left NULL (default), the join will be performed by via the foreign key relation
                that exists between the originally zoomed table (now x) and the other table (y).
                If you provide a value (for the syntax see dplyr::join), you can also join tables
                that are not connected in the dm.

copy	Disabled, since all tables in a dm are by definition on the same src.
suffix	Disabled, since columns are disambiguated automatically if necessary, changing the column names to table_name.column_name.
select	Select a subset of the **RHS-table**'s columns, the syntax being select = c(col_1,col_2,col_3) (unquoted or quoted). This argument is specific for the join-methods for zoomed_dm. The table's by column(s) are automatically added if missing in the selection.
...	see dplyr::join

Examples

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

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

---

dplyr_table_manipulation

dplyr *table manipulation methods for zoomed dm objects*

Description

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

Usage

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Arguments

`.data`	object of class zoomed_dm
`...`	see corresponding function in package **dplyr** or **tidyr**
`.keep_all`	For distinct.zoomed_dm(): see [dplyr::distinct](#)
`.keep_pk`	For slice.zoomed_dm: 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.zoomed_dm() method.
`x`	For ungroup.zoomed_dm: object of class zoomed_dm
`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 sum(wt) for each group.

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 error, and require you to specify the name.
.drop	For count(): if FALSE will include counts for empty groups (i.e. for levels of factors that don't exist in the data). Deprecated in add_count() since it didn'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

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

---

enum_pk_candidates *Primary key candidate*

Description

[Questioning]

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

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

Usage

```
enum_pk_candidates(table, ...)

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

Arguments

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

See Also

Other primary key functions: [dm_add_pk](), [dm_get_all_pks](), [dm_has_pk](), [dm_rm_pk]()

Examples

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



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

---

examine_cardinality   *Check table relations*

Description

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

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

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

Usage

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

check_cardinality_1_n(parent_table, pk_column, child_table, fk_column)

check_cardinality_1_1(parent_table, pk_column, child_table, fk_column)

check_cardinality_0_1(parent_table, pk_column, child_table, fk_column)

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

Arguments

parent_table   Data frame.

pk_column   Column of `parent_table` that has to be one of its unique keys.

child_table   Data frame.

fk_column   Column of `child_table` that has to be a foreign key to `pk_column` in `parent_table`.

Details

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

The cardinality specifications 0_n, 1_n, 0_1, 1_1 refer to the expected relation that the child table has with the parent table. The numbers 0, 1 and n refer to the number of values in the column of the child table that correspond to each value of the column of the parent table. n means "more than one" in this context, with no upper limit.
0_n means, that each value of the pk_column has at least 0 and at most n corresponding values in the column of the child table (which translates to no further restrictions).
1_n means, that each value of the pk_column has at least 1 and at most n corresponding values in the column of the child table. This means that there is a "surjective" mapping from the child table to the parent table w.r.t. the specified columns, i.e. for each parent table column value there exists at least one equal child table column value.
0_1 means, that each value of the pk_column has at least 0 and at most 1 corresponding values in the column of the child table. This means that there is a "injective" mapping from the child table to the parent table w.r.t. the specified columns, i.e. no parent table column value is addressed multiple times. But not all of the parent table column values have to be referred to.
1_1 means, that each value of the pk_column has exactly 1 corresponding value in the column of the child table. This means that there is a "bijective" ("injective" AND "surjective") mapping between the child table and the parent table w.r.t. the specified columns, i.e. the sets of values of the two columns are equal and there are no duplicates in either of them.

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

Value

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

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

Examples

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

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

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

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

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

---

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

Description

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

Usage

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

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

Arguments

x   object of class zoomed_dm

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

Details

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

Value

A zoomed_dm object.

Examples

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

---

materialize	*Materialize*

Description

compute() materializes all tables in a dm to new (temporary or permanent) tables on the database.

collect() downloads the tables in a dm object as local [tibbles](#).

Usage

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

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

Arguments

x	A dm.
...	Passed on to [compute()](#).
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.

Value

A dm object of the same structure as the input.

Examples

```
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()
```

---

pull_tbl   *Retrieve a table*

Description

This function has methods for both dm classes:

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

Usage

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

Arguments

dm	A dm object.
table	One unquoted table name for pull_tbl.dm(), ignored for pull_tbl.zoomed_dm().

Value

The requested table

Examples

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

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

---

reunite_parent_child   *Merge two tables that are linked by a foreign key relation*

Description

[Questioning]

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

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

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

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

Usage

```
reunite_parent_child(child_table, parent_table, id_column)

reunite_parent_child_from_list(list_of_parent_child_tables, id_column)
```

Arguments

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

See Also

Other table surgery functions: decompose_table()

Examples

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

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

---

rows-db   *Updating database tables*

Description

[Experimental]

These methods provide a framework for manipulating individual rows in existing tables. All oper-
ations expect that both existing and new data are presented in two compatible [tbl](tbl) objects.

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

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

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

Usage

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

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

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

sql_rows_insert(x, y, ...)

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

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

Arguments

x	A pair of data frames or data frame extensions (e.g. a tibble). y must have the same columns of x or a subset.
y	A pair of data frames or data frame extensions (e.g. a tibble). y must have the same columns of x or a subset.
by	An unnamed character vector giving the key columns. The key values must uniquely identify each row (i.e. each combination of key values occurs at most once), and the key columns must exist in both x and y.
	By default, we use the first column in y, since the first column is a reasonable place to put an identifier variable.
...	Other parameters passed onto methods.
in_place	Should x be modified in place? This argument is only relevant for mutable backends (e.g. databases, data.tables).
	When TRUE, a modified version of x is returned invisibly; when FALSE, a new object representing the resulting changes is returned.
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.
check	Set to TRUE to always check keys, or FALSE to never check. The default is to check only if in_place is TRUE or NULL.
	Currently these checks are no-ops and need yet to be implemented.

Value

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

Examples

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

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

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

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

---

rows-dm   *Modifying rows for multiple tables*

Description

[Experimental]

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

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

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

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

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

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

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

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

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

Usage

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

dm_rows_truncate(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_ptype()](#) as x. If in_place = TRUE, the underlying data is updated as a side effect, and x is returned, invisibly.

Examples

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

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

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

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

# Copy to temporary tables on the target database:
```

```
flights_jan_sqlite <- copy_dm_to(sqlite, flights_jan)

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

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

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

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

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

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

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

DBI::dbDisconnect(sqlite)
```

---

rows_truncate   *Truncate all rows*

Description

rows_truncate() removes all rows. This operation corresponds to TRUNCATE in SQL. ... is ig-
nored.

Usage

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

sql_rows_truncate(x, ...)
```

Arguments

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

---

sql_schema_create   *Create a schema on a database*

Description

sql_schema_create() creates a schema on the database.

Usage

```
sql_schema_create(dest, schema, ...)
```

Arguments

dest	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 dest.

Value

NULL invisibly.

See Also

Other schema handling functions: sql_schema_drop(), sql_schema_exists(), sql_schema_list()

---

sql_schema_drop   *Remove a schema from a database*

Description

sql_schema_drop() deletes a schema from the database. For certain DBMS it is possible to force
the removal of a non-empty schema, see below.

Usage

```
sql_schema_drop(dest, schema, force = FALSE, ...)
```

Arguments

dest	An object of class "src" or "DBIConnection".
schema	Class character or SQL (cf. Details), name of the schema
force	Boolean, default FALSE. Set to TRUE to drop a schema and all objects it contains at once. Currently only supported for Postgres.
...	Passed on to the individual methods.

Details

Methods are not available for all DBMS.

An error is thrown if no schema of that name 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.

Additional arguments are:

- dbname: supported for MSSQL. Remove a schema from a different database on the connected
  MSSQL-server; default: database addressed by dest.

Value

NULL invisibly.

See Also

Other schema handling functions: sql_schema_create(), sql_schema_exists(), sql_schema_list()

---

sql_schema_exists *Check for existence of a schema on a database*

Description

sql_schema_exists() checks, if a schema exists on the database.

Usage

```
sql_schema_exists(dest, schema, ...)
```

Arguments

dest	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 dest.

Value

A boolean: TRUE if schema exists, FALSE otherwise.

See Also

Other schema handling functions: [sql_schema_create](), [sql_schema_drop](), [sql_schema_list]()

---

sql_schema_list *List schemas on a database*

Description

sql_schema_list() lists the available schemas on the database.

Usage

```
sql_schema_list(dest, include_default = TRUE, ...)
```

Arguments

dest   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 dest.

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: [sql_schema_create](), [sql_schema_drop](), [sql_schema_exists]()

---

tidyr_table_manipulation

                    tidyr *table manipulation methods for zoomed dm objects*

Description

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

Usage

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

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

Arguments

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

Examples

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

Index