Package 'dplyr'

July 4, 2019

Type Package

Title A Grammar of Data Manipulation

Version 0.8.3

Description A fast, consistent tool for working with data frame like objects, both in memory and out of memory.

License MIT + file LICENSE

URL http://dplyr.tidyverse.org, https://github.com/tidyverse/dplyr

BugReports https://github.com/tidyverse/dplyr/issues

Depends R (¿= 3.2.0)

Imports assertthat (¿= 0.2.0),
glue (¿= 1.3.0),
magrittr (¿= 1.5),
methods,
pkgconfig,
R6,
Rcpp (¿= 1.0.1),
rlang (¿= 0.4.0),
tibble (¿= 2.0.0),
tidyselect (¿= 0.2.5),
utils

Suggests bit64,
callr,
covr,
crayon (¿= 1.3.4),
DBI,
dbplyr,
dtplyr,
ggplot2,
hms,
knitr,
Lahman,
lubridate,
MASS,
mgcv,
microbenchmark,
nycflights13,
rmarkdown,

RMySQL,
RPostgreSQL,
RSQLite,
testthat,
withr,
broom,
purrr,
readr

LinkingTo BH,
plogr (¿= 0.2.0),
Rcpp (¿= 1.0.1)

VignetteBuilder knitr

Encoding UTF-8

LazyData yes

Roxygen list(markdown = TRUE, roclets = c(``rd'', ``namespace'', ``collate''))

RoxygenNote 6.1.1

R topics documented:

`dplyr-package` *dplyr: a grammar of data manipulation*

Description

dplyr provides a flexible grammar of data manipulation. It's the next iteration of plyr, focused on tools for working with data frames (hence the *d* in the name).

Details

It has three main goals:

- Identify the most important data manipulation verbs and make them easy to use from R.
- Provide blazing fast performance for in-memory data by writing key pieces in C++ (using Rcpp)
- Use the same interface to work with data no matter where it's stored, whether in a data frame, a data table or database.

To learn more about dplyr, start with the vignettes: `browseVignettes(package = "dplyr")`

Package options

`dplyr.show_progress` Should lengthy operations such as `do()` show a progress bar? Default: `TRUE`

Package configurations

These can be set on a package-by-package basis, or for the global environment. See `pkgconfig::set_config()` for usage.

`dplyr::na_matches` Should `NA` values be matched in data frame joins by default? Default: `"na"` (for compatibility with dplyr v0.5.0 and earlier, subject to change), alternative value: `"never"` (the default for database backends, see `join.tbl_df()`).

Author(s)

Maintainer: Hadley Wickham <hadley@rstudio.com> (0000-0003-4757-117X)

Authors:

- Romain François (0000-0002-2444-4226)
- Lionel Henry
- Kirill Müller (0000-0002-1416-3412)

Other contributors:

- RStudio [copyright holder, funder]

See Also

Useful links:

- http://dplyr.tidyverse.org
- https://github.com/tidyverse/dplyr
- Report bugs at https://github.com/tidyverse/dplyr/issues

`all_equal` *Flexible equality comparison for data frames*

Description

You can use `all_equal()` with any data frame, and dplyr also provides `tbl_df` methods for `all.equal()`.

Usage

```
all_equal(target, current, ignore_col_order = TRUE,
  ignore_row_order = TRUE, convert = FALSE, ...)

## S3 method for class 'tbl_df'
all.equal(target, current, ignore_col_order = TRUE,
  ignore_row_order = TRUE, convert = FALSE, ...)
```

Arguments

`target, current`	Two data frames to compare.
`ignore_col_order`	Should order of columns be ignored?
`ignore_row_order`	Should order of rows be ignored?
`convert`	Should similar classes be converted? Currently this will convert factor to character and integer to double.
`...`	Ignored. Needed for compatibility with `all.equal()`.

Value

`TRUE` if equal, otherwise a character vector describing the reasons why they're not equal. Use `isTRUE()` if using the result in an `if` expression.

Examples

```
scramble <- function(x) x[sample(nrow(x)), sample(ncol(x))]

# By default, ordering of rows and columns ignored
all_equal(mtcars, scramble(mtcars))

# But those can be overriden if desired
all_equal(mtcars, scramble(mtcars), ignore_col_order = FALSE)
all_equal(mtcars, scramble(mtcars), ignore_row_order = FALSE)

# By default all_equal is sensitive to variable differences
df1 <- data.frame(x = "a")
df2 <- data.frame(x = factor("a"))
all_equal(df1, df2)
# But you can request dplyr convert similar types
all_equal(df1, df2, convert = TRUE)
```

all_vars *Apply predicate to all variables*

Description

These quoting functions signal to scoped filtering verbs (e.g. `filter_if()` or `filter_all()`) that a predicate expression should be applied to all relevant variables. The `all_vars()` variant takes the intersection of the predicate expressions with `&` while the `any_vars()` variant takes the union with `|`.

Usage

```
all_vars(expr)

any_vars(expr)
```

Arguments

`expr`	A predicate expression. This variable supports unquoting and will be evaluated in the context of the data frame. It should return a logical vector. This argument is automatically quoted and later evaluated in the context of the data frame. It supports unquoting. See `vignette("programming")` for an introduction to these concepts.

See Also

`vars()` for other quoting functions that you can use with scoped verbs.

arrange *Arrange rows by variables*

Description

Order tbl rows by an expression involving its variables.

Usage

```
arrange(.data, ...)

## S3 method for class 'grouped_df'
arrange(.data, ..., .by_group = FALSE)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	Comma separated list of unquoted variable names, or expressions involving variable names. Use `desc()` to sort a variable in descending order.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

Value

An object of the same class as `.data`.

Locales

The sort order for character vectors will depend on the collating sequence of the locale in use: see `locales()`.

Missing values

Unlike base sorting with `sort()`, `NA` are:

- always sorted to the end for local data, even when wrapped with `desc()`.
- treated differently for remote data, depending on the backend.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with `tibble::rownames_to_column()`.

See Also

Other single table verbs: `filter`, `mutate`, `select`, `slice`, `summarise`

Examples

```
arrange(mtcars, cyl, disp)
arrange(mtcars, desc(disp))

# grouped arrange ignores groups
by_cyl <- mtcars %>% group_by(cyl)
by_cyl %>% arrange(desc(wt))
# Unless you specifically ask:
by_cyl %>% arrange(desc(wt), .by_group = TRUE)
```

---

arrange_all *Arrange rows by a selection of variables*

Description

These scoped variants of `arrange()` sort a data frame by a selection of variables. Like `arrange()`, you can modify the variables before ordering with the `.funs` argument.

Usage

```
arrange_all(.tbl, .funs = list(), ..., .by_group = FALSE)

arrange_at(.tbl, .vars, .funs = list(), ..., .by_group = FALSE)

arrange_if(.tbl, .predicate, .funs = list(), ..., .by_group = FALSE)
```

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.
.by_group	If TRUE, will sort first by grouping variable. Applies to grouped data frames only.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

The grouping variables that are part of the selection participate in the sorting of the data frame.

Examples

```
df <- as_tibble(mtcars)
df
arrange_all(df)

# You can supply a function that will be applied before taking the
# ordering of the variables. The variables of the sorted tibble
# keep their original values.
arrange_all(df, desc)
arrange_all(df, list(~desc(.)))
```

---

as.table.tbl_cube *Coerce a* tbl_cube *to other data structures*

Description

Supports conversion to tables, data frames, tibbles.

For a cube, the data frame returned by tibble::as_tibble() resulting data frame contains the dimensions as character values (and not as factors).

Usage

```
## S3 method for class 'tbl_cube'
as.table(x, ..., measure = 1L)

## S3 method for class 'tbl_cube'
as.data.frame(x, ...)

## S3 method for class 'tbl_cube'
as_tibble(x, ...)
```

Arguments

x	a tbl_cube
...	Passed on to individual methods; otherwise ignored.
measure	A measure name or index, default: the first measure

---

as.tbl_cube *Coerce an existing data structure into a* tbl_cube

---

Description

Coerce an existing data structure into a tbl_cube

Usage

```
as.tbl_cube(x, ...)

## S3 method for class 'array'
as.tbl_cube(x, dim_names = names(dimnames(x)),
  met_name = deparse(substitute(x)), ...)

## S3 method for class 'table'
as.tbl_cube(x, dim_names = names(dimnames(x)),
  met_name = "Freq", ...)

## S3 method for class 'matrix'
as.tbl_cube(x, dim_names = names(dimnames(x)),
  met_name = deparse(substitute(x)), ...)

## S3 method for class 'data.frame'
as.tbl_cube(x, dim_names = NULL,
  met_name = guess_met(x), ...)
```

Arguments

x	an object to convert. Built in methods will convert arrays, tables and data frames.
...	Passed on to individual methods; otherwise ignored.
dim_names	names of the dimensions. Defaults to the names of
met_name	a string to use as the name for the measure the dimnames().

auto_copy *Copy tables to same source, if necessary*

Description

Copy tables to same source, if necessary

Usage

```
auto_copy(x, y, copy = FALSE, ...)
```

Arguments

x, y	y will be copied to x, if necessary.
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.
...	Other arguments passed on to methods.

band_members *Band membership*

Description

These data sets describe band members of the Beatles and Rolling Stones. They are toy data sets that can be displayed in their entirety on a slide (e.g. to demonstrate a join).

Usage

```
band_members

band_instruments

band_instruments2
```

Format

Each is a tibble with two variables and three observations

Details

band_instruments and band_instruments2 contain the same data but use different column names for the first column of the data set. band_instruments uses name, which matches the name of the key column of band_members; band_instruments2 uses artist, which does not.

Examples

```
band_members
band_instruments
band_instruments2
```

`between` *Do values in a numeric vector fall in specified range?*

Description

This is a shortcut for `x >= left & x <= right`, implemented efficiently in C++ for local values, and translated to the appropriate SQL for remote tables.

Usage

```
between(x, left, right)
```

Arguments

`x`	A numeric vector of values
`left, right`	Boundary values

Examples

```
between(1:12, 7, 9)

x <- rnorm(1e2)
x[between(x, -1, 1)]
```

`bind` *Efficiently bind multiple data frames by row and column*

Description

This is an efficient implementation of the common pattern of `do.call(rbind,dfs)` or `do.call(cbind,dfs)` for binding many data frames into one.

Usage

```
bind_rows(..., .id = NULL)

bind_cols(...)
```

Arguments

`...`	Data frames to combine. Each argument can either be a data frame, a list that could be a data frame, or a list of data frames. When row-binding, columns are matched by name, and any missing columns will be filled with NA. When column-binding, rows are matched by position, so all data frames must have the same number of rows. To match by value, not position, see join.

`.id` Data frame identifier.

When `.id` is supplied, a new column of identifiers is created to link each row to its original data frame. The labels are taken from the named arguments to `bind_rows()`. When a list of data frames is supplied, the labels are taken from the names of the list. If no names are found a numeric sequence is used instead.

Details

The output of `bind_rows()` will contain a column if that column appears in any of the inputs.

Value

`bind_rows()` and `bind_cols()` return the same type as the first input, either a data frame, `tbl_df`, or `grouped_df`.

Deprecated functions

`rbind_list()` and `rbind_all()` have been deprecated. Instead use `bind_rows()`.

Examples

```
one <- mtcars[1:4, ]
two <- mtcars[11:14, ]

# You can supply data frames as arguments:
bind_rows(one, two)

# The contents of lists are spliced automatically:
bind_rows(list(one, two))
bind_rows(split(mtcars, mtcars$cyl))
bind_rows(list(one, two), list(two, one))


# In addition to data frames, you can supply vectors. In the rows
# direction, the vectors represent rows and should have inner
# names:
bind_rows(
  c(a = 1, b = 2),
  c(a = 3, b = 4)
)

# You can mix vectors and data frames:
bind_rows(
  c(a = 1, b = 2),
  tibble(a = 3:4, b = 5:6),
  c(a = 7, b = 8)
)


# Note that for historical reasons, lists containing vectors are
# always treated as data frames. Thus their vectors are treated as
# columns rather than rows, and their inner names are ignored:
ll <- list(
  a = c(A = 1, B = 2),
```

```
  b = c(A = 3, B = 4)
)
bind_rows(ll)

# You can circumvent that behaviour with explicit splicing:
bind_rows(!!!ll)


# When you supply a column name with the `.id` argument, a new
# column is created to link each row to its original data frame
bind_rows(list(one, two), .id = "id")
bind_rows(list(a = one, b = two), .id = "id")
bind_rows("group 1" = one, "group 2" = two, .id = "groups")

# Columns don't need to match when row-binding
bind_rows(data.frame(x = 1:3), data.frame(y = 1:4))
## Not run:
# Rows do need to match when column-binding
bind_cols(data.frame(x = 1), data.frame(y = 1:2))

## End(Not run)

bind_cols(one, two)
bind_cols(list(one, two))
```

`case_when` *A general vectorised if*

Description

This function allows you to vectorise multiple `if_else()` statements. It is an R equivalent of the SQL `CASE WHEN` statement. If no cases match, `NA` is returned.

Usage

```
case_when(...)
```

Arguments

...	A sequence of two-sided formulas. The left hand side (LHS) determines which values match this case. The right hand side (RHS) provides the replacement value. The LHS must evaluate to a logical vector. The RHS does not need to be logical, but all RHSs must evaluate to the same type of vector. Both LHS and RHS may have the same length of either 1 or `n`. The value of `n` must be consistent across all cases. The case of `n == 0` is treated as a variant of `n != 1`. `NULL` inputs are ignored. These dots support tidy dots features. In particular, if your patterns are stored in a list, you can splice that in with `!!!`.

Value

A vector of length 1 or n, matching the length of the logical input or output vectors, with the type (and attributes) of the first RHS. Inconsistent lengths or types will generate an error.

Examples

```
x <- 1:50
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)

# Like an if statement, the arguments are evaluated in order, so you must
# proceed from the most specific to the most general. This won't work:
case_when(
  TRUE ~ as.character(x),
  x %%  5 == 0 ~ "fizz",
  x %%  7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz"
)

# If none of the cases match, NA is used:
case_when(
  x %%  5 == 0 ~ "fizz",
  x %%  7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz"
)

# Note that NA values in the vector x do not get special treatment. If you want
# to explicitly handle NA values you can use the `is.na` function:
x[2:4] <- NA_real_
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "nope",
  TRUE ~ as.character(x)
)

# All RHS values need to be of the same type. Inconsistent types will throw an error.
# This applies also to NA values used in RHS: NA is logical, use
# typed values like NA_real_, NA_complex, NA_character_, NA_integer_ as appropriate.
case_when(
  x %% 35 == 0 ~ NA_character_,
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)
case_when(
  x %% 35 == 0 ~ 35,
  x %% 5 == 0 ~ 5,
  x %% 7 == 0 ~ 7,
  TRUE ~ NA_real_
```

```
)

# case_when() evaluates all RHS expressions, and then constructs its
# result by extracting the selected (via the LHS expressions) parts.
# In particular NaN are produced in this case:
y <- seq(-2, 2, by = .5)
case_when(
  y >= 0 ~ sqrt(y),
  TRUE   ~ y
)

# This throws an error as NA is logical not numeric
## Not run:
case_when(
  x %% 35 == 0 ~ 35,
  x %% 5 == 0 ~ 5,
  x %% 7 == 0 ~ 7,
  TRUE ~ NA
)

## End(Not run)

# case_when is particularly useful inside mutate when you want to
# create a new variable that relies on a complex combination of existing
# variables
starwars %>%
  select(name:mass, gender, species) %>%
  mutate(
    type = case_when(
      height > 200 | mass > 200 ~ "large",
      species == "Droid"        ~ "robot",
      TRUE                      ~ "other"
    )
  )


# `case_when()` is not a tidy eval function. If you'd like to reuse
# the same patterns, extract the `case_when()` call in a normal
# function:
case_character_type <- function(height, mass, species) {
  case_when(
    height > 200 | mass > 200 ~ "large",
    species == "Droid"        ~ "robot",
    TRUE                      ~ "other"
  )
}

case_character_type(150, 250, "Droid")
case_character_type(150, 150, "Droid")

# Such functions can be used inside `mutate()` as well:
starwars %>%
  mutate(type = case_character_type(height, mass, species)) %>%
  pull(type)

# `case_when()` ignores `NULL` inputs. This is useful when you'd
# like to use a pattern only under certain conditions. Here we'll
```

```
# take advantage of the fact that `if` returns `NULL` when there is
# no `else` clause:
case_character_type <- function(height, mass, species, robots = TRUE) {
  case_when(
    height > 200 | mass > 200      ~ "large",
    if (robots) species == "Droid" ~ "robot",
    TRUE                           ~ "other"
  )
}

starwars %>%
  mutate(type = case_character_type(height, mass, species, robots = FALSE)) %>%
  pull(type)
```

`coalesce` *Find first non-missing element*

Description

Given a set of vectors, `coalesce()` finds the first non-missing value at each position. This is inspired by the SQL `COALESCE` function which does the same thing for `NULL`s.

Usage

```
coalesce(...)
```

Arguments

`...`	Vectors. All inputs should either be length 1, or the same length as the first argument. These dots support tidy dots features.

Value

A vector the same length as the first `...` argument with missing values replaced by the first non-missing value.

See Also

`na_if()` to replace specified values with a `NA`. `tidyr::replace_na()` to replace `NA` with a value

Examples

```
# Use a single value to replace all missing values
x <- sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)

# Or match together a complete vector from missing pieces
y <- c(1, 2, NA, NA, 5)
z <- c(NA, NA, 3, 4, 5)
coalesce(y, z)

# Supply lists by splicing them into dots:
```

```
vecs <- list(
  c(1, 2, NA, NA, 5),
  c(NA, NA, 3, 4, 5)
)
coalesce(!!!vecs)
```

`combine` *Combine vectors*

Description

`combine()` acts like `c()` or `unlist()` but uses consistent dplyr coercion rules.

If `combine()` it is called with exactly one list argument, the list is simplified (similarly to `unlist(recursive = FALSE)`). `NULL` arguments are ignored. If the result is empty, `logical()` is returned. Use `vctrs::vec_c()` if you never want to unlist.

Usage

```
combine(...)
```

Arguments

`...`	Vectors to combine.

Details

Questioning

See Also

`bind_rows()` and `bind_cols()` in bind.

Examples

```
# combine applies the same coercion rules as bind_rows()
f1 <- factor("a")
f2 <- factor("b")
c(f1, f2)
unlist(list(f1, f2))

combine(f1, f2)
combine(list(f1, f2))
```

compute *Force computation of a database query*

Description

`compute()` stores results in a remote temporary table. `collect()` retrieves data into a local tibble. `collapse()` is slightly different: it doesn't force computation, but instead forces generation of the SQL query. This is sometimes needed to work around bugs in dplyr's SQL generation.

Usage

```
compute(x, name = random_table_name(), ...)

collect(x, ...)

collapse(x, ...)
```

Arguments

x	A tbl
name	Name of temporary table on database.
...	Other arguments passed on to methods

Details

All functions preserve grouping and ordering.

See Also

copy_to(), the opposite of `collect()`: it takes a local data frame and uploads it to the remote source.

Examples

```
if (require(dbplyr)) {
  mtcars2 <- src_memdb() %>%
    copy_to(mtcars, name = "mtcars2-cc", overwrite = TRUE)

  remote <- mtcars2 %>%
    filter(cyl == 8) %>%
    select(mpg:drat)

  # Compute query and save in remote table
  compute(remote)

  # Compute query bring back to this session
  collect(remote)

  # Creates a fresh query based on the generated SQL
  collapse(remote)
}
```

copy_to *Copy a local data frame to a remote src*

Description

This function uploads a local data frame into a remote data source, creating the table definition as needed. Wherever possible, the new object will be temporary, limited to the current connection to the source.

Usage

```
copy_to(dest, df, name = deparse(substitute(df)), overwrite = FALSE,
  ...)
```

Arguments

dest	remote data source
df	local data frame
name	name for new remote table.
overwrite	If TRUE, will overwrite an existing table with name name. If FALSE, will throw an error if name already exists.
...	other parameters passed to methods.

Value

a tbl object in the remote source

See Also

collect() for the opposite action; downloading remote data into a local dbl.

Examples

```
## Not run:
iris2 <- dbplyr::src_memdb() %>% copy_to(iris, overwrite = TRUE)
iris2

## End(Not run)
```

cumall *Cumulativate versions of any, all, and mean*

Description

dplyr provides cumall(), cumany(), and cummean() to complete R's set of cumulative functions.

Usage

```
cumall(x)

cumany(x)

cummean(x)
```

Arguments

x	For `cumall()` and `cumany()`, a logical vector; for `cummean()` an integer or numeric vector.

Value

A vector the same length as `x`.

Cumulative logical functions

These are particularly useful in conjunction with `filter()`:

- `cumall(x)`: all cases until the first `FALSE`.
- `cumall(!x)`: all cases until the first `TRUE`.
- `cumany(x)`: all cases after the first `TRUE`.
- `cumany(!x)`: all cases after the first `FALSE`.

Examples

```
# `cummean()` returns a numeric/integer vector of the same length
# as the input vector.
x <- c(1, 3, 5, 2, 2)
cummean(x)
cumsum(x) / seq_along(x)

# `cumall()` and `cumany()` return logicals
cumall(x < 5)
cumany(x == 3)

# `cumall()` vs. `cumany()`
df <- data.frame(
  date = as.Date("2020-01-01") + 0:6,
  balance = c(100, 50, 25, -25, -50, 30, 120)
)
# all rows after first overdraft
df %>% filter(cumany(balance < 0))
# all rows until first overdraft
df %>% filter(cumall(!(balance < 0)))
```

desc *Descending order*

Description

Transform a vector into a format that will be sorted in descending order. This is useful within `arrange()`.

Usage

```
desc(x)
```

Arguments

x	vector to transform

Examples

```
desc(1:10)
desc(factor(letters))

first_day <- seq(as.Date("1910/1/1"), as.Date("1920/1/1"), "years")
desc(first_day)

starwars %>% arrange(desc(mass))
```

distinct *Select distinct/unique rows*

Description

Retain only unique/distinct rows from an input tbl. This is similar to `unique.data.frame()`, but considerably faster.

Usage

```
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

.data	a tbl
...	Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables.
.keep_all	If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

Details

Comparing list columns is not fully supported. Elements in list columns are compared by reference. A warning will be given when trying to include list columns in the computation. This behavior is kept for compatibility reasons and may change in a future version. See examples.

Examples

```
df <- tibble(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE)
)
nrow(df)
nrow(distinct(df))
nrow(distinct(df, x, y))

distinct(df, x)
distinct(df, y)

# Can choose to keep all other variables as well
distinct(df, x, .keep_all = TRUE)
distinct(df, y, .keep_all = TRUE)

# You can also use distinct on computed variables
distinct(df, diff = abs(x - y))

# The same behaviour applies for grouped data frames
# except that the grouping variables are always included
df <- tibble(
  g = c(1, 1, 2, 2),
  x = c(1, 1, 2, 1)
) %>% group_by(g)
df %>% distinct()
df %>% distinct(x)

# Values in list columns are compared by reference, this can lead to
# surprising results
tibble(a = as.list(c(1, 1, 2))) %>% glimpse() %>% distinct()
tibble(a = as.list(1:2)[c(1, 1, 2)]) %>% glimpse() %>% distinct()
```

---

distinct_all *Select distinct rows by a selection of variables*

Description

These scoped variants of `distinct()` extract distinct rows by a selection of variables. Like `distinct()`, you can modify the variables before ordering with the `.funs` argument.

Usage

```
distinct_all(.tbl, .funs = list(), ..., .keep_all = FALSE)

distinct_at(.tbl, .vars, .funs = list(), ..., .keep_all = FALSE)

distinct_if(.tbl, .predicate, .funs = list(), ..., .keep_all = FALSE)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.
`.keep_all`	If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

The grouping variables that are part of the selection are taken into account to determine distinct rows.

Examples

```
df <- tibble(x = rep(2:5, each = 2) / 2, y = rep(2:3, each = 4) / 2)
df
distinct_all(df)
distinct_at(df, vars(x,y))
distinct_if(df, is.numeric)

# You can supply a function that will be applied before extracting the distinct values
# The variables of the sorted tibble keep their original values.
distinct_all(df, round)
arrange_all(df, list(~round(.)))
```

---

do *Do anything*

Description

This is a general purpose complement to the specialised manipulation functions `filter()`, `select()`, `mutate()`, `summarise()` and `arrange()`. You can use `do()` to perform arbitrary computation, returning either a data frame or arbitrary objects which will be stored in a list. This is particularly useful when working with models: you can fit models per group with `do()` and then flexibly extract components with either another `do()` or `summarise()`.

For an empty data frame, the expressions will be evaluated once, even in the presence of a grouping. This makes sure that the format of the resulting data frame is the same for both empty and non-empty input.

Usage

```
do(.data, ...)
```

Arguments

.data	a tbl
...	Expressions to apply to each group. If named, results will be stored in a new column. If unnamed, should return a data frame. You can use . to refer to the current group. You can not mix named and unnamed arguments.

Details

Questioning

Value

do() always returns a data frame. The first columns in the data frame will be the labels, the others will be computed from .... Named arguments become list-columns, with one element for each group; unnamed elements must be data frames and labels will be duplicated accordingly.

Groups are preserved for a single unnamed input. This is different to summarise() because do() generally does not reduce the complexity of the data, it just expresses it in a special way. For multiple named inputs, the output is grouped by row with rowwise(). This allows other verbs to work in an intuitive way.

Alternative

do() is marked as questioning as of dplyr 0.8.0, and may be advantageously replaced by group_map().

Connection to plyr

If you're familiar with plyr, do() with named arguments is basically equivalent to plyr::dlply(), and do() with a single unnamed argument is basically equivalent to plyr::ldply(). However, instead of storing labels in a separate attribute, the result is always a data frame. This means that summarise() applied to the result of do() can act like ldply().

Examples

```
by_cyl <- group_by(mtcars, cyl)
do(by_cyl, head(., 2))

models <- by_cyl %>% do(mod = lm(mpg ~ disp, data = .))
models

summarise(models, rsq = summary(mod)$r.squared)
models %>% do(data.frame(coef = coef(.$mod)))
models %>% do(data.frame(
  var = names(coef(.$mod)),
  coef(summary(.$mod)))
)

models <- by_cyl %>% do(
  mod_linear = lm(mpg ~ disp, data = .),
  mod_quad = lm(mpg ~ poly(disp, 2), data = .)
)
models
```

```
compare <- models %>% do(aov = anova(.$mod_linear, .$mod_quad))
# compare %>% summarise(p.value = aov$`Pr(>F)`)

if (require("nycflights13")) {
# You can use it to do any arbitrary computation, like fitting a linear
# model. Let's explore how carrier departure delays vary over the time
carriers <- group_by(flights, carrier)
group_size(carriers)

mods <- do(carriers, mod = lm(arr_delay ~ dep_time, data = .))
mods %>% do(as.data.frame(coef(.$mod)))
mods %>% summarise(rsq = summary(mod)$r.squared)

## Not run:
# This longer example shows the progress bar in action
by_dest <- flights %>% group_by(dest) %>% filter(n() > 100)
library(mgcv)
by_dest %>% do(smooth = gam(arr_delay ~ s(dep_time) + month, data = .))

## End(Not run)
}
```

`dr_dplyr` *Dr Dplyr checks your installation for common problems.*

Description

Only run this if you are seeing problems, like random crashes. It's possible for `dr_dplyr` to return false positives, so there's no need to run if all is ok.

Usage

```
dr_dplyr()
```

Examples

```
## Not run:
dr_dplyr()

## End(Not run)
```

`explain` *Explain details of a tbl*

Description

This is a generic function which gives more details about an object than `print()`, and is more focused on human readable output than `str()`.

Usage

```
explain(x, ...)

show_query(x, ...)
```

Arguments

x	An object to explain
...	Other parameters possibly used by generic

Value

The first argument, invisibly.

Databases

Explaining a `tbl_sql` will run the SQL `EXPLAIN` command which will describe the query plan. This requires a little bit of knowledge about how `EXPLAIN` works for your database, but is very useful for diagnosing performance problems.

Examples

```
if (require("dbplyr")) {

lahman_s <- lahman_sqlite()
batting <- tbl(lahman_s, "Batting")
batting %>% show_query()
batting %>% explain()

# The batting database has indices on all ID variables:
# SQLite automatically picks the most restrictive index
batting %>% filter(lgID == "NL" & yearID == 2000L) %>% explain()

# OR's will use multiple indexes
batting %>% filter(lgID == "NL" | yearID == 2000) %>% explain()

# Joins will use indexes in both tables
teams <- tbl(lahman_s, "Teams")
batting %>% left_join(teams, c("yearID", "teamID")) %>% explain()
}
```

`filter` *Return rows with matching conditions*

Description

Use `filter()` to choose rows/cases where conditions are true. Unlike base subsetting with `[`, rows where the condition evaluates to `NA` are dropped.

Usage

```
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	Logical predicates defined in terms of the variables in `.data`. Multiple conditions are combined with `&`. Only 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")` for an introduction to these concepts.
`.preserve`	when `FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.

Details

Note that dplyr is not yet smart enough to optimise filtering optimisation on grouped datasets that don't need grouped calculations. For this reason, filtering is often considerably faster on `ungroup()`ed data.

Value

An object of the same class as `.data`.

Useful filter functions

- `==`, `>`, `>=` etc
- `&`, `|`, `!`, `xor()`
- `is.na()`
- `between()`, `near()`

Grouped tibbles

Because filtering expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped filtering:

```
starwars %>% filter(mass > mean(mass, na.rm = TRUE))
```

With the grouped equivalent:

```
starwars %>% group_by(gender) %>% filter(mass > mean(mass, na.rm = TRUE))
```

The former keeps rows with `mass` greater than the global average whereas the latter keeps rows with `mass` greater than the gender average.

It is valid to use grouping variables in filter expressions.

When applied on a grouped tibble, `filter()` automatically rearranges the tibble by groups for performance reasons.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with `tibble::rownames_to_column()`.

Scoped filtering

The three scoped variants (`filter_all()`, `filter_if()` and `filter_at()`) make it easy to apply a filtering condition to a selection of variables.

See Also

`filter_all()`, `filter_if()` and `filter_at()`.

Other single table verbs: `arrange`, `mutate`, `select`, `slice`, `summarise`

Examples

```
filter(starwars, species == "Human")
filter(starwars, mass > 1000)

# Multiple criteria
filter(starwars, hair_color == "none" & eye_color == "black")
filter(starwars, hair_color == "none" | eye_color == "black")

# Multiple arguments are equivalent to and
filter(starwars, hair_color == "none", eye_color == "black")


# The filtering operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
#
# The following filters rows where `mass` is greater than the
# global average:
starwars %>% filter(mass > mean(mass, na.rm = TRUE))

# Whereas this keeps rows with `mass` greater than the gender
# average:
starwars %>% group_by(gender) %>% filter(mass > mean(mass, na.rm = TRUE))


# Refer to column names stored as strings with the `.data` pronoun:
vars <- c("mass", "height")
cond <- c(80, 150)
starwars %>%
  filter(
    .data[[vars[[1]]]] > cond[[1]],
    .data[[vars[[2]]]] > cond[[2]]
  )

# For more complex cases, knowledge of tidy evaluation and the
# unquote operator `!!` is required. See https://tidyeval.tidyverse.org/
#
# One useful and simple tidy eval technique is to use `!!` to bypass
# the data frame and its columns. Here is how to filter the columns
# `mass` and `height` relative to objects of the same names:
mass <- 80
height <- 150
filter(starwars, mass > !!mass, height > !!height)
```

`filter_all` *Filter within a selection of variables*

Description

These scoped filtering verbs apply a predicate expression to a selection of variables. The predicate expression should be quoted with `all_vars()` or `any_vars()` and should mention the pronoun `.` to refer to variables.

Usage

```
filter_all(.tbl, .vars_predicate, .preserve = FALSE)

filter_if(.tbl, .predicate, .vars_predicate, .preserve = FALSE)

filter_at(.tbl, .vars, .vars_predicate, .preserve = FALSE)
```

Arguments

`.tbl`	A `tbl` object.
`.vars_predicate`	A quoted predicate expression as returned by `all_vars()` or `any_vars()`. Can also be a function or purrr-like formula. In this case, the intersection of the results is taken by default and there's currently no way to request the union.
`.preserve`	when `FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.

Grouping variables

The grouping variables that are part of the selection are taken into account to determine filtered rows.

Examples

```
# While filter() accepts expressions with specific variables, the
# scoped filter verbs take an expression with the pronoun `.` and
# replicate it over all variables. This expression should be quoted
# with all_vars() or any_vars():
all_vars(is.na(.))
any_vars(is.na(.))


# You can take the intersection of the replicated expressions:
filter_all(mtcars, all_vars(. > 150))
```

```
# Or the union:
filter_all(mtcars, any_vars(. > 150))


# You can vary the selection of columns on which to apply the
# predicate. filter_at() takes a vars() specification:
filter_at(mtcars, vars(starts_with("d")), any_vars((. %% 2) == 0))

# And filter_if() selects variables with a predicate function:
filter_if(mtcars, ~ all(floor(.) == .), all_vars(. != 0))


# We're working on a new syntax to allow functions instead,
# including purrr-like lambda functions. This is already
# operational, but there's currently no way to specify the union of
# the predicate results:
mtcars %>% filter_at(vars(hp, vs), ~ . %% 2 == 0)
```

`funs` *Create a list of functions calls.*

Description

`funs()` provides a flexible way to generate a named list of functions for input to other functions like `summarise_at()`.

Usage

```
funs(..., .args = list())
```

Arguments

`...`	A list of functions specified by: • Their name, `"mean"` • The function itself, `mean` • A call to the function with `.` as a dummy argument, `mean(.,na.rm = TRUE)` These arguments are automatically quoted. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts. The following notations are **not** supported, see examples: • An anonymous function, `function(x) mean(x,na.rm = TRUE)` • An anonymous function in **purrr** notation, `~mean(.,na.rm = TRUE)`
`.args, args`	A named list of additional arguments to be added to all function calls. As `funs()` is being deprecated, use other methods to supply arguments: `...` argument in scoped verbs or make own functions with `purrr::partial()`.

Details

Soft-deprecated

Examples

```
funs(mean, "mean", mean(., na.rm = TRUE))

# Override default names
funs(m1 = mean, m2 = "mean", m3 = mean(., na.rm = TRUE))

# If you have function names in a vector, use funs_
fs <- c("min", "max")
funs_(fs)

# Not supported
## Not run:
funs(function(x) mean(x, na.rm = TRUE))
funs(~mean(x, na.rm = TRUE))
## End(Not run)
```

---

groups *Return grouping variables*

Description

`group_vars()` returns a character vector; `groups()` returns a list of symbols.

Usage

```
groups(x)

group_vars(x)
```

Arguments

x	A `tbl()`

See Also

`group_cols()` for matching grouping variables in selection contexts.

Other grouping functions: `group_by_all`, `group_by`, `group_indices`, `group_keys`, `group_map`, `group_nest`, `group_rows`, `group_size`, `group_trim`

Examples

```
df <- tibble(x = 1, y = 2) %>% group_by(x, y)
group_vars(df)
groups(df)
```

group_by *Group by one or more variables*

Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

Usage

```
group_by(.data, ..., add = FALSE, .drop = group_by_drop_default(.data))

ungroup(x, ...)
```

Arguments

`.data`	a tbl
`...`	Variables to group by. All tbls accept variable names. Some tbls will accept functions of variables. Duplicated groups will be silently dropped.
`add`	When `add = FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `add = TRUE`.
`.drop`	When `.drop = TRUE`, empty groups are dropped. See `group_by_drop_default()` for what the default value is for this argument.
`x`	A `tbl()`

Value

A grouped data frame, unless the combination of `...` and `add` yields a non empty set of grouping columns, a regular (ungrouped) data frame otherwise.

Tbl types

`group_by()` is an S3 generic with methods for the three built-in tbls. See the help for the corresponding classes and their manip methods for more details:

- data.frame: grouped_df
- data.table: dtplyr::grouped_dt
- SQLite: `src_sqlite()`
- PostgreSQL: `src_postgres()`
- MySQL: `src_mysql()`

Scoped grouping

The three scoped variants (`group_by_all()`, `group_by_if()` and `group_by_at()`) make it easy to group a dataset by a selection of variables.

See Also

Other grouping functions: `group_by_all`, `group_indices`, `group_keys`, `group_map`, `group_nest`, `group_rows`, `group_size`, `group_trim`, `groups`

Examples

```
by_cyl <- mtcars %>% group_by(cyl)

# grouping doesn't change how the data looks (apart from listing
# how it's grouped):
by_cyl

# It changes how it acts with the other dplyr verbs:
by_cyl %>% summarise(
  disp = mean(disp),
  hp = mean(hp)
)
by_cyl %>% filter(disp == max(disp))

# Each call to summarise() removes a layer of grouping
by_vs_am <- mtcars %>% group_by(vs, am)
by_vs <- by_vs_am %>% summarise(n = n())
by_vs
by_vs %>% summarise(n = sum(n))

# To removing grouping, use ungroup
by_vs %>%
  ungroup() %>%
  summarise(n = sum(n))

# You can group by expressions: this is just short-hand for
# a mutate/rename followed by a simple group_by
mtcars %>% group_by(vsam = vs + am)

# By default, group_by overrides existing grouping
by_cyl %>%
  group_by(vs, am) %>%
  group_vars()

# Use add = TRUE to instead append
by_cyl %>%
  group_by(vs, am, add = TRUE) %>%
  group_vars()

# when factors are involved, groups can be empty
tbl <- tibble(
  x = 1:10,
  y = factor(rep(c("a", "c"), each  = 5), levels = c("a", "b", "c"))
)
tbl %>%
  group_by(y) %>%
  group_rows()
```

---

group_by_all *Group by a selection of variables*

Description

These scoped variants of group_by() group a data frame by a selection of variables. Like group_by(), they have optional mutate semantics.

Usage

```
group_by_all(.tbl, .funs = list(), ..., .add = FALSE,
  .drop = group_by_drop_default(.tbl))

group_by_at(.tbl, .vars, .funs = list(), ..., .add = FALSE,
  .drop = group_by_drop_default(.tbl))

group_by_if(.tbl, .predicate, .funs = list(), ..., .add = FALSE,
  .drop = group_by_drop_default(.tbl))
```

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.
.add	See group_by()
.drop	When .drop = TRUE, empty groups are dropped. See group_by_drop_default() for what the default value is for this argument.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

Existing grouping variables are maintained, even if not included in the selection.

See Also

Other grouping functions: group_by, group_indices, group_keys, group_map, group_nest, group_rows, group_size, group_trim, groups

Examples

```
# Group a data frame by all variables:
group_by_all(mtcars)

# Group by variables selected with a predicate:
group_by_if(iris, is.factor)

# Group by variables selected by name:
group_by_at(mtcars, vars(vs, am))

# Like group_by(), the scoped variants have optional mutate
```

```
# semantics. This provide a shortcut for group_by() + mutate():
d <- tibble(x=c(1,1,2,2), y=c(1,2,1,2))
group_by_all(d, as.factor)
group_by_if(iris, is.factor, as.character)
```

`group_by_drop_default` *Default value for .drop argument of group_by*

Description

Default value for .drop argument of group_by

Usage

```
group_by_drop_default(.tbl)
```

Arguments

`.tbl`	A data frame

Value

`TRUE` unless `.tbl` is a grouped data frame that was previously obtained by `group_by(.drop = FALSE)`

Examples

```
group_by_drop_default(iris)

iris %>%
  group_by(Species) %>%
  group_by_drop_default()

iris %>%
  group_by(Species, .drop = FALSE) %>%
  group_by_drop_default()
```

`group_cols` *Select grouping variables*

Description

This selection helpers matches grouping variables. It can be used in `select()` or vars() selections.

Usage

```
group_cols(vars = peek_vars())
```

Arguments

vars	A character vector of variable names. When called from inside selecting functions like dplyr::select() these are automatically set to the names of the table.

See Also

groups() and group_vars() for retrieving the grouping variables outside selection contexts.

Examples

```
gdf <- iris %>% group_by(Species)

# Select the grouping variables:
gdf %>% select(group_cols())

# Remove the grouping variables from mutate selections:
gdf %>% mutate_at(vars(-group_cols()), `/`, 100)
```

---

group_keys *Split data frame by groups*

Description

Split data frame by groups

Usage

```
group_keys(.tbl, ...)

group_split(.tbl, ..., keep = TRUE)
```

Arguments

.tbl	A tbl
...	Grouping specification, forwarded to group_by()
keep	Should the grouping columns be kept

Details

Experimental

group_split() works like base::split() but

- it uses the grouping structure from group_by() and therefore is subject to the data mask
- it does not name the elements of the list based on the grouping as this typically loses information and is confusing.

group_keys() explains the grouping structure, by returning a data frame that has one row per group and one column per grouping variable.

Value

- group_split() returns a list of tibbles. Each tibble contains the rows of .tbl for the associated group and all the columns, including the grouping variables.
- group_keys() returns a tibble with one row per group, and one column per grouping variable

Grouped data frames

The primary use case for group_split() is with already grouped data frames, typically a result of group_by(). In this case group_split() only uses the first argument, the grouped tibble, and warns when ... is used.

Because some of these groups may be empty, it is best paried with group_keys() which identifies the representatives of each grouping variable for the group.

Ungrouped data frames

When used on ungrouped data frames, group_split() and group_keys() forwards the ... to group_by() before the split, therefore the ... are subject to the data mask.

Using these functions on an ungrouped data frame only makes sense if you need only one or the other, because otherwise the grouping algorithm is performed each time.

Rowwise data frames

group_split() returns a list of one-row tibbles is returned, and the ... are ignored and warned against

See Also

Other grouping functions: group_by_all, group_by, group_indices, group_map, group_nest, group_rows, group_size, group_trim, groups

Examples

```
# ----- use case 1 : on an already grouped tibble
ir <- iris %>%
  group_by(Species)

group_split(ir)
group_keys(ir)

# this can be useful if the grouped data has been altered before the split
ir <- iris %>%
  group_by(Species) %>%
  filter(Sepal.Length > mean(Sepal.Length))

group_split(ir)
group_keys(ir)

# ----- use case 2: using a group_by() grouping specification

# both group_split() and group_keys() have to perform the grouping
# so it only makes sense to do this if you only need one or the other
iris %>%
```

```
  group_split(Species)

iris %>%
  group_keys(Species)
```

`group_map` *Apply a function to each group*

Description

`group_map()`, `group_modify()` and `group_walk()` are purrr-style functions that can be used to iterate on grouped tibbles.

Usage

```
group_map(.tbl, .f, ..., keep = FALSE)

group_modify(.tbl, .f, ..., keep = FALSE)

group_walk(.tbl, .f, ...)
```

Arguments

`.tbl`	A grouped tibble
`.f`	A function or formula to apply to each group. It must return a data frame. If a **function**, it is used as is. It should have at least 2 formal arguments. If a **formula**, e.g. `~ head(.x)`, it is converted to a function. In the formula, you can use • `.` or `.x` to refer to the subset of rows of `.tbl` for the given group • `.y` to refer to the key, a one row tibble with one column per grouping variable that identifies the group
`...`	Additional arguments passed on to `.f`
`keep`	are the grouping variables kept in `.x`

Details

Experimental

Use `group_modify()` when `summarize()` is too limited, in terms of what you need to do and return for each group. `group_modify()` is good for "data frame in, data frame out". If that is too limited, you need to use a nested or split workflow. `group_modify()` is an evolution of `do()`, if you have used that before.

Each conceptual group of the data frame is exposed to the function `.f` with two pieces of information:

- The subset of the data for the group, exposed as `.x`.
- The key, a tibble with exactly one row and columns for each grouping variable, exposed as `.y`.

For completeness, `group_modify()`, `group_map` and `group_walk()` also work on ungrouped data frames, in that case the function is applied to the entire data frame (exposed as `.x`), and `.y` is a one row tibble with no column, consistently with `group_keys()`.

Value

- `group_modify()` returns a grouped tibble. In that case `.f` must return a data frame.
- `group_map()` returns a list of results from calling `.f` on each group
- `group_walk()` calls `.f` for side effects and returns the input `.tbl`, invisibly

See Also

Other grouping functions: group_by_all, group_by, group_indices, group_keys, group_nest, group_rows, group_size, group_trim, groups

Examples

```
# return a list
mtcars %>%
  group_by(cyl) %>%
  group_map(~ head(.x, 2L))

# return a tibble grouped by `cyl` with 2 rows per group
# the grouping data is recalculated
mtcars %>%
  group_by(cyl) %>%
  group_modify(~ head(.x, 2L))

if (requireNamespace("broom", quietly = TRUE)) {
  # a list of tibbles
  iris %>%
    group_by(Species) %>%
    group_map(~ broom::tidy(lm(Petal.Length ~ Sepal.Length, data = .x)))

  # a restructured grouped tibble
  iris %>%
    group_by(Species) %>%
    group_modify(~ broom::tidy(lm(Petal.Length ~ Sepal.Length, data = .x)))
}

# a list of vectors
iris %>%
  group_by(Species) %>%
  group_map(~ quantile(.x$Petal.Length, probs = c(0.25, 0.5, 0.75)))

# to use group_modify() the lambda must return a data frame
iris %>%
  group_by(Species) %>%
  group_modify(~ {
     quantile(.x$Petal.Length, probs = c(0.25, 0.5, 0.75)) %>%
     tibble::enframe(name = "prob", value = "quantile")
  })

iris %>%
  group_by(Species) %>%
  group_modify(~ {
    .x %>%
      purrr::map_dfc(fivenum) %>%
      mutate(nms = c("min", "Q1", "median", "Q3", "max"))
  })
```

```
# group_walk() is for side effects
dir.create(temp <- tempfile())
iris %>%
  group_by(Species) %>%
  group_walk(~ write.csv(.x, file = file.path(temp, paste0(.y$Species, ".csv"))))
list.files(temp, pattern = "csv$")
unlink(temp, recursive = TRUE)

# group_modify() and ungrouped data frames
mtcars %>%
  group_modify(~ head(.x, 2L))
```

---

group_rows *Grouping data*

Description

Grouping data

Usage

```
group_rows(.data)

group_data(.data)
```

Arguments

.data	a tibble

Value

`group_data()` return a tibble with one row per group. The last column, always called `.rows` is a list of integer vectors indicating the rows for each group. If `.data` is a grouped data frame the first columns are the grouping variables. `group_rows()` just returns the list of indices.

See Also

Other grouping functions: group_by_all, group_by, group_indices, group_keys, group_map, group_nest, group_size, group_trim, groups

Examples

```
df <- tibble(x = c(1,1,2,2))

# one row
group_data(df)
group_rows(df)

# 2 rows, one for each group
group_by(df,x) %>% group_data()
group_by(df,x) %>% group_rows()
```

group_trim *Trim grouping structure*

Description

Drop unused levels of all factors that are used as grouping variables, then recalculates the grouping structure.

`group_trim()` is particularly useful after a `filter()` that is intended to select a subset of groups.

Usage

```
group_trim(.tbl, .drop = group_by_drop_default(.tbl))
```

Arguments

`.tbl`	A grouped data frame
`.drop`	See `group_by()`

Details

Experimental

Value

A grouped data frame

See Also

Other grouping functions: `group_by_all`, `group_by`, `group_indices`, `group_keys`, `group_map`, `group_nest`, `group_rows`, `group_size`, `groups`

Examples

```
iris %>%
  group_by(Species) %>%
  filter(Species == "setosa", .preserve = TRUE) %>%
  group_trim()
```

hybrid_call *Inspect how dplyr evaluates an expression*

Description

Inspect how dplyr evaluates an expression

Usage

```
hybrid_call(.data, expr)
```

Arguments

.data	a tibble
expr	an expression

Examples

```
# hybrid evaulation
hybrid_call(iris, n())

# standard evaluation
hybrid_call(iris, n() + 1L)
```

---

ident *Flag a character vector as SQL identifiers*

Description

`ident()` takes unquoted strings and flags them as identifiers. `ident_q()` assumes its input has already been quoted, and ensures it does not get quoted again. This is currently used only for for `schema.table`.

Usage

```
ident(...)
```

Arguments

...	A character vector, or name-value pairs

Examples

```
# Identifiers are escaped with "
if (requireNamespace("dbplyr", quietly = TRUE)) {
  ident("x")
}
```

---

if_else *Vectorised if*

Description

Compared to the base `ifelse()`, this function is more strict. It checks that `true` and `false` are the same type. This strictness makes the output type more predictable, and makes it somewhat faster.

Usage

```
if_else(condition, true, false, missing = NULL)
```

Arguments

`condition`	Logical vector
`true, false`	Values to use for `TRUE` and `FALSE` values of `condition`. They must be either the same length as `condition`, or length 1. They must also be the same type: `if_else()` checks that they have the same type and same class. All other attributes are taken from `true`.
`missing`	If not `NULL`, will be used to replace missing values.

Value

Where `condition` is `TRUE`, the matching value from `true`, where it's `FALSE`, the matching value from `false`, otherwise `NA`.

Examples

```
x <- c(-5:5, NA)
if_else(x < 0, NA_integer_, x)
if_else(x < 0, "negative", "positive", "missing")

# Unlike ifelse, if_else preserves types
x <- factor(sample(letters[1:5], 10, replace = TRUE))
ifelse(x %in% c("a", "b", "c"), x, factor(NA))
if_else(x %in% c("a", "b", "c"), x, factor(NA))
# Attributes are taken from the `true` vector,
```

---

join *Join two tbls together*

Description

These are generic functions that dispatch to individual tbl methods - see the method documentation for details of individual data sources. `x` and `y` should usually be from the same data source, but if `copy` is `TRUE`, `y` will automatically be copied to the same source as `x`.

Usage

```
inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"),
  ...)

left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"),
  ...)

full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

semi_join(x, y, by = NULL, copy = FALSE, ...)

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

anti_join(x, y, by = NULL, copy = FALSE, ...)
```

Arguments

x, y	tbls to join
by	a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, by = c("a" = "b") will match x.a to y.b.
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.
suffix	If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
...	other parameters passed onto methods, for instance, na_matches to control how NA values are matched. See join.tbl_df for more.
keep	If TRUE the by columns are kept in the nesting joins.
name	the name of the list column nesting joins create. If NULL the name of y is used.

Join types

Currently dplyr supports four types of mutating joins, two types of filtering joins, and a nesting join.

Mutating joins combine variables from the two data.frames:

inner_join() return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

left_join() return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

right_join() return all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

full_join() return all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.

Filtering joins keep cases from the left-hand data.frame:

semi_join() return all rows from x where there are matching values in y, keeping just columns from x.

A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x.

anti_join() return all rows from x where there are not matching values in y, keeping just columns from x.

Nesting joins create a list column of data.frames:

`nest_join()` return all rows and all columns from `x`. Adds a list column of tibbles. Each tibble contains all the rows from `y` that match that row of `x`. When there is no match, the list column is a 0-row tibble with the same column names and types as `y`.

`nest_join()` is the most fundamental join since you can recreate the other joins from it. An `inner_join()` is a `nest_join()` plus an `tidyr::unnest()`, and `left_join()` is a `nest_join()` plus an `unnest(.drop = FALSE)`. A `semi_join()` is a `nest_join()` plus a `filter()` where you check that every element of data has at least one row, and an `anti_join()` is a `nest_join()` plus a `filter()` where you check every element has zero rows.

Grouping

Groups are ignored for the purpose of joining, but the result preserves the grouping of `x`.

Examples

```
# "Mutating" joins combine variables from the LHS and RHS
band_members %>% inner_join(band_instruments)
band_members %>% left_join(band_instruments)
band_members %>% right_join(band_instruments)
band_members %>% full_join(band_instruments)

# "Filtering" joins keep cases from the LHS
band_members %>% semi_join(band_instruments)
band_members %>% anti_join(band_instruments)

# "Nesting" joins keep cases from the LHS and nests the RHS
band_members %>% nest_join(band_instruments)

# To suppress the message, supply by
band_members %>% inner_join(band_instruments, by = "name")
# This is good practice in production code

# Use a named `by` if the join variables have different names
band_members %>% full_join(band_instruments2, by = c("name" = "artist"))
# Note that only the key from the LHS is kept
```

---

lead-lag *Lead and lag.*

Description

Find the "next" or "previous" values in a vector. Useful for comparing values ahead of or behind the current values.

Usage

```
lead(x, n = 1L, default = NA, order_by = NULL, ...)

lag(x, n = 1L, default = NA, order_by = NULL, ...)
```

Arguments

x	a vector of values
n	a positive integer of length 1, giving the number of positions to lead or lag by
default	value used for non-existent rows. Defaults to NA.
order_by	override the default ordering to use another vector
...	Needed for compatibility with lag generic.

Examples

```
lead(1:10, 1)
lead(1:10, 2)

lag(1:10, 1)
lead(1:10, 1)

x <- runif(5)
cbind(ahead = lead(x), x, behind = lag(x))

# Use order_by if data not already ordered
df <- data.frame(year = 2000:2005, value = (0:5) ^ 2)
scrambled <- df[sample(nrow(df)), ]

wrong <- mutate(scrambled, prev = lag(value))
arrange(wrong, year)

right <- mutate(scrambled, prev = lag(value, order_by = year))
arrange(right, year)
```

---

mutate *Create or transform variables*

Description

`mutate()` adds new variables and preserves existing ones; `transmute()` adds new variables and drops existing ones. Both functions preserve the number of rows of the input. New variables overwrite existing variables of the same name.

Usage

```
mutate(.data, ...)

transmute(.data, ...)
```

Arguments

.data	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.

`...` Name-value pairs of expressions, each with length 1 or the same length as the number of rows in the group (if using `group_by()`) or in the entire input (if not using groups). The name of each argument will be the name of a new variable, and the value will be its corresponding value. Use a `NULL` value in `mutate` to drop a variable. New variables overwrite existing variables of the same name.

The arguments in `...` are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts.

Value

An object of the same class as `.data`.

Useful functions available in calculations of variables

- `+`, `-`, `log()`, etc., for their usual mathematical meanings
- `lead()`, `lag()`
- `dense_rank()`, `min_rank()`, `percent_rank()`, `row_number()`, `cume_dist()`, `ntile()`
- `cumsum()`, `cummean()`, `cummin()`, `cummax()`, `cumany()`, `cumall()`
- `na_if()`, `coalesce()`
- `if_else()`, `recode()`, `case_when()`

Grouped tibbles

Because mutating expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped mutate:

```
starwars %>%
  mutate(mass / mean(mass, na.rm = TRUE)) %>%
  pull()
```

With the grouped equivalent:

```
starwars %>%
  group_by(gender) %>%
  mutate(mass / mean(mass, na.rm = TRUE)) %>%
  pull()
```

The former normalises `mass` by the global average whereas the latter normalises by the averages within gender levels.

Note that you can't overwrite a grouping variable within `mutate()`.

`mutate()` does not evaluate the expressions when the group is empty.

Scoped mutation and transmutation

The three scoped variants of `mutate()` (`mutate_all()`, `mutate_if()` and `mutate_at()`) and the three variants of `transmute()` (`transmute_all()`, `transmute_if()`, `transmute_at()`) make it easy to apply a transformation to a selection of variables.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with tibble::rownames_to_column().

See Also

Other single table verbs: arrange, filter, select, slice, summarise

Examples

```
# Newly created variables are available immediately
mtcars %>% as_tibble() %>% mutate(
  cyl2 = cyl * 2,
  cyl4 = cyl2 * 2
)

# You can also use mutate() to remove variables and
# modify existing variables
mtcars %>% as_tibble() %>% mutate(
  mpg = NULL,
  disp = disp * 0.0163871 # convert to litres
)


# window functions are useful for grouped mutates
mtcars %>%
 group_by(cyl) %>%
 mutate(rank = min_rank(desc(mpg)))
# see `vignette("window-functions")` for more details

# You can drop variables by setting them to NULL
mtcars %>% mutate(cyl = NULL)

# mutate() vs transmute --------------------------
# mutate() keeps all existing variables
mtcars %>%
  mutate(displ_l = disp / 61.0237)

# transmute keeps only the variables you create
mtcars %>%
  transmute(displ_l = disp / 61.0237)


# The mutate operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
# The following normalises `mass` by the global average:
starwars %>%
  mutate(mass / mean(mass, na.rm = TRUE)) %>%
  pull()

# Whereas this normalises `mass` by the averages within gender
# levels:
starwars %>%
  group_by(gender) %>%
  mutate(mass / mean(mass, na.rm = TRUE)) %>%
  pull()
```

```
# Note that you can't overwrite grouping variables:
gdf <- mtcars %>% group_by(cyl)
try(mutate(gdf, cyl = cyl * 100))


# Refer to column names stored as strings with the `.data` pronoun:
vars <- c("mass", "height")
mutate(starwars, prod = .data[[vars[[1]]]] * .data[[vars[[2]]]])

# For more complex cases, knowledge of tidy evaluation and the
# unquote operator `!!` is required. See https://tidyeval.tidyverse.org/
#
# One useful and simple tidy eval technique is to use `!!` to
# bypass the data frame and its columns. Here is how to divide the
# column `mass` by an object of the same name:
mass <- 100
mutate(starwars, mass = mass / !!mass)
```

`mutate_all` *Mutate multiple columns*

Description

The scoped variants of `mutate()` and `transmute()` make it easy to apply the same transformation to multiple variables. There are three variants:

- _all affects every variable
- _at affects variables selected with a character vector or vars()
- _if affects variables selected with a predicate function:

Usage

```
mutate_all(.tbl, .funs, ...)

mutate_if(.tbl, .predicate, .funs, ...)

mutate_at(.tbl, .vars, .funs, ..., .cols = NULL)

transmute_all(.tbl, .funs, ...)

transmute_if(.tbl, .predicate, .funs, ...)

transmute_at(.tbl, .vars, .funs, ..., .cols = NULL)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.

`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.cols`	This argument has been renamed to `.vars` to fit dplyr's terminology and is deprecated.

Value

A data frame. By default, the newly created columns have the shortest names needed to uniquely identify the output. To force inclusion of a name, even when not needed, name the input (see examples for details).

Grouping variables

If applied on a grouped tibble, these operations are *not* applied to the grouping variables. The behaviour depends on whether the selection is **implicit** (`all` and `if` selections) or **explicit** (`at` selections).

- Grouping variables covered by explicit selections in `mutate_at()` and `transmute_at()` are always an error. Add `-group_cols()` to the `vars()` selection to avoid this:

  ```
  data %>% mutate_at(vars(-group_cols(), ...), myoperation)
  ```

  Or remove `group_vars()` from the character vector of column names:

  ```
  nms <- setdiff(nms, group_vars(data))
  data %>% mutate_at(vars, myoperation)
  ```

- Grouping variables covered by implicit selections are ignored by `mutate_all()`, `transmute_all()`, `mutate_if()`, and `transmute_if()`.

Naming

The names of the created columns is derived from the names of the input variables and the names of the functions.

- if there is only one unnamed function, the names of the input variables are used to name the created columns
- if there is only one unnamed variable, the names of the functions are used to name the created columns.
- otherwise in the most general case, the created names are created by concatenating the names of the input variables and the names of the functions.

The names of the functions here means the names of the list of functions that is supplied. When needed and not supplied, the name of a function is the prefix "fn" followed by the index of this function within the unnamed functions in the list. Ultimately, names are made unique.

See Also

The other scoped verbs, `vars()`

Examples

```
iris <- as_tibble(iris)

# All variants can be passed functions and additional arguments,
# purrr-style. The _at() variants directly support strings. Here
# we'll scale the variables `height` and `mass`:
scale2 <- function(x, na.rm = FALSE) (x - mean(x, na.rm = na.rm)) / sd(x, na.rm)
starwars %>% mutate_at(c("height", "mass"), scale2)

# You can pass additional arguments to the function:
starwars %>% mutate_at(c("height", "mass"), scale2, na.rm = TRUE)

# You can also pass formulas to create functions on the spot, purrr-style:
starwars %>% mutate_at(c("height", "mass"), ~scale2(., na.rm = TRUE))

# You can also supply selection helpers to _at() functions but you have
# to quote them with vars():
iris %>% mutate_at(vars(matches("Sepal")), log)

# The _if() variants apply a predicate function (a function that
# returns TRUE or FALSE) to determine the relevant subset of
# columns. Here we divide all the numeric columns by 100:
starwars %>% mutate_if(is.numeric, scale2, na.rm = TRUE)

# mutate_if() is particularly useful for transforming variables from
# one type to another
iris %>% mutate_if(is.factor, as.character)
iris %>% mutate_if(is.double, as.integer)


# Multiple transformations ----------------------------------------

# If you want to apply multiple transformations, pass a list of
# functions. When there are multiple functions, they create new
# variables instead of modifying the variables in place:
iris %>% mutate_if(is.numeric, list(scale2, log))

# The list can contain purrr-style formulas:
iris %>% mutate_if(is.numeric, list(~scale2(.), ~log(.)))

# Note how the new variables include the function name, in order to
# keep things distinct. The default names are not always helpful
# but you can also supply explicit names:
iris %>% mutate_if(is.numeric, list(scale = scale2, log = log))

# When there's only one function in the list, it modifies existing
# variables in place. Give it a name to instead create new variables:
iris %>% mutate_if(is.numeric, list(scale2))
iris %>% mutate_if(is.numeric, list(scale = scale2))
```

---

n *The number of observations in the current group.*

Description

This function is implemented specifically for each data source and can only be used from within summarise(), mutate() and filter().

Usage

```
n()
```

Examples

```
if (require("nycflights13")) {
carriers <- group_by(flights, carrier)
summarise(carriers, n())
mutate(carriers, n = n())
filter(carriers, n() < 100)
}
```

---

nasa *NASA spatio-temporal data*

Description

This data comes from the ASA 2007 data expo, http://stat-computing.org/dataexpo/2006/. The data are geographic and atmospheric measures on a very coarse 24 by 24 grid covering Central America. The variables are: temperature (surface and air), ozone, air pressure, and cloud cover (low, mid, and high). All variables are monthly averages, with observations for Jan 1995 to Dec 2000. These data were obtained from the NASA Langley Research Center Atmospheric Sciences Data Center (with permission; see important copyright terms below).

Usage

```
nasa
```

Format

A tbl_cube with 41,472 observations.

Dimensions

- `lat`, `long`: latitude and longitude
- `year`, `month`: month and year

Measures

- `cloudlow`, `cloudmed`, `cloudhigh`: cloud cover at three heights
- `ozone`
- `surftemp` and `temperature`
- `pressure`

Examples

```
nasa
```

na_if *Convert values to NA*

Description

This is a translation of the SQL command NULL_IF. It is useful if you want to convert an annoying value to NA.

Usage

```
na_if(x, y)
```

Arguments

x	Vector to modify
y	Value to replace with NA

Value

A modified version of x that replaces any values that are equal to y with NA.

See Also

coalesce() to replace missing values with a specified value.

tidyr::replace_na() to replace NA with a value.

recode() to more generally replace values.

Examples

```
na_if(1:5, 5:1)

x <- c(1, -1, 0, 10)
100 / x
100 / na_if(x, 0)

y <- c("abc", "def", "", "ghi")
na_if(y, "")

# na_if is particularly useful inside mutate,
# and is meant for use with vectors rather than entire data frames
starwars %>%
  select(name, eye_color) %>%
  mutate(eye_color = na_if(eye_color, "unknown"))

# na_if can also be used with scoped variants of mutate
# like mutate_if to mutate multiple columns
starwars %>%
  mutate_if(is.character, list(~na_if(., "unknown")))
```

near *Compare two numeric vectors*

Description

This is a safe way of comparing if two vectors of floating point numbers are (pairwise) equal. This is safer than using ==, because it has a built in tolerance

Usage

```
near(x, y, tol = .Machine$double.eps^0.5)
```

Arguments

x, y	Numeric vectors to compare
tol	Tolerance of comparison.

Examples

```
sqrt(2) ^ 2 == 2
near(sqrt(2) ^ 2, 2)
```

nest_join.data.frame *Join data frame tbls*

Description

See join for a description of the general purpose of the functions.

Usage

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

## S3 method for class 'tbl_df'
inner_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))

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

## S3 method for class 'tbl_df'
left_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))

## S3 method for class 'tbl_df'
```

```
right_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))

## S3 method for class 'tbl_df'
full_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))

## S3 method for class 'tbl_df'
semi_join(x, y, by = NULL, copy = FALSE, ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))

## S3 method for class 'tbl_df'
anti_join(x, y, by = NULL, copy = FALSE, ...,
  na_matches = pkgconfig::get_config("dplyr::na_matches"))
```

Arguments

x	tbls to join
y	tbls to join
by	a character vector of variables to join by. If `NULL`, the default, `*_join()` will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, `by = c("a" = "b")` will match `x.a` to `y.b`.
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	If `TRUE` the by columns are kept in the nesting joins.
name	the name of the list column nesting joins create. If `NULL` the name of `y` is used.
...	included for compatibility with the generic; otherwise ignored.
suffix	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
na_matches	Use `"never"` to always treat two `NA` or `NaN` values as different, like joins for database sources, similarly to `merge(incomparables = FALSE)`. The default, `"na"`, always treats two `NA` or `NaN` values as equal, like `merge()`. Users and package authors can change the default behavior by calling `pkgconfig::set_config("dplyr::na_matches" = "never")`.

Examples

```
if (require("Lahman")) {
batting_df <- tbl_df(Batting)
person_df <- tbl_df(Master)

uperson_df <- tbl_df(Master[!duplicated(Master$playerID), ])
```

```
# Inner join: match batting and person data
inner_join(batting_df, person_df)
inner_join(batting_df, uperson_df)

# Left join: match, but preserve batting data
left_join(batting_df, uperson_df)

# Anti join: find batters without person data
anti_join(batting_df, person_df)
# or people who didn't bat
anti_join(person_df, batting_df)
}
```

`nth` *Extract the first, last or nth value from a vector*

Description

These are straightforward wrappers around `[[`. The main advantage is that you can provide an optional secondary vector that defines the ordering, and provide a default value to use when the input is shorter than expected.

Usage

```
nth(x, n, order_by = NULL, default = default_missing(x))

first(x, order_by = NULL, default = default_missing(x))

last(x, order_by = NULL, default = default_missing(x))
```

Arguments

`x`	A vector
`n`	For `nth_value()`, a single integer specifying the position. Negative integers index from the end (i.e. `-1L` will return the last value in the vector). If a double is supplied, it will be silently truncated.
`order_by`	An optional vector used to determine the order
`default`	A default value to use if the position does not exist in the input. This is guessed by default for base vectors, where a missing value of the appropriate type is returned, and for lists, where a `NULL` is return. For more complicated objects, you'll need to supply this value. Make sure it is the same type as `x`.

Value

A single value. `[[` is used to do the subsetting.

Examples

```
x <- 1:10
y <- 10:1

first(x)
last(y)

nth(x, 1)
nth(x, 5)
nth(x, -2)
nth(x, 11)

last(x)
# Second argument provides optional ordering
last(x, y)

# These functions always return a single value
first(integer())
```

---

n_distinct *Efficiently count the number of unique values in a set of vector*

Description

This is a faster and more concise equivalent of `length(unique(x))`

Usage

```
n_distinct(..., na.rm = FALSE)
```

Arguments

`...`	vectors of values
`na.rm`	if `TRUE` missing values don't count

Examples

```
x <- sample(1:10, 1e5, rep = TRUE)
length(unique(x))
n_distinct(x)
```

---

order_by *A helper function for ordering window function output*

Description

This function makes it possible to control the ordering of window functions in R that don't have a specific ordering parameter. When translated to SQL it will modify the order clause of the OVER function.

Usage

```
order_by(order_by, call)
```

Arguments

`order_by`	a vector to order_by
`call`	a function call to a window function, where the first argument is the vector being operated on

Details

This function works by changing the `call` to instead call `with_order()` with the appropriate arguments.

Examples

```
order_by(10:1, cumsum(1:10))
x <- 10:1
y <- 1:10
order_by(x, cumsum(y))

df <- data.frame(year = 2000:2005, value = (0:5) ^ 2)
scrambled <- df[sample(nrow(df)), ]

wrong <- mutate(scrambled, running = cumsum(value))
arrange(wrong, year)

right <- mutate(scrambled, running = order_by(year, cumsum(value)))
arrange(right, year)
```

`pull` *Pull out a single variable*

Description

This works like `[[` for local data frames, and automatically collects before indexing for remote data tables.

Usage

```
pull(.data, var = -1)
```

Arguments

`.data`	A table of data
`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 unquote column names and column positions).

Examples

```
mtcars %>% pull(-1)
mtcars %>% pull(1)
mtcars %>% pull(cyl)

# Also works for remote sources
if (requireNamespace("dbplyr", quietly = TRUE)) {
df <- dbplyr::memdb_frame(x = 1:10, y = 10:1, .name = "pull-ex")
df %>%
  mutate(z = x * y) %>%
  pull()
}
```

---

ranking *Windowed rank functions.*

Description

Six variations on ranking functions, mimicking the ranking functions described in SQL2003. They are currently implemented using the built in `rank` function, and are provided mainly as a convenience when converting between R and SQL. All ranking functions map smallest inputs to smallest outputs. Use `desc()` to reverse the direction.

Usage

```
row_number(x)

ntile(x = row_number(), n)

min_rank(x)

dense_rank(x)

percent_rank(x)

cume_dist(x)
```

Arguments

`x`	a vector of values to rank. Missing values are left as is. If you want to treat them as the smallest or largest values, replace with Inf or -Inf before ranking.
`n`	number of groups to split up into.

Details

- `row_number()`: equivalent to `rank(ties.method = "first")`
- `min_rank()`: equivalent to `rank(ties.method = "min")`
- `dense_rank()`: like `min_rank()`, but with no gaps between ranks
- `percent_rank()`: a number between 0 and 1 computed by rescaling `min_rank` to `[0,1]`

- `cume_dist()`: a cumulative distribution function. Proportion of all values less than or equal to the current rank.
- `ntile()`: a rough rank, which breaks the input vector into `n` buckets.

Examples

```
x <- c(5, 1, 3, 2, 2, NA)
row_number(x)
min_rank(x)
dense_rank(x)
percent_rank(x)
cume_dist(x)

ntile(x, 2)
ntile(runif(100), 10)

# row_number can be used with single table verbs without specifying x
# (for data frames and databases that support windowing)
mutate(mtcars, row_number() == 1L)
mtcars %>% filter(between(row_number(), 1, 10))
```

---

recode *Recode values*

Description

This is a vectorised version of `switch()`: you can replace numeric values based on their position or their name, and character or factor values only by their name. This is an S3 generic: dplyr provides methods for numeric, character, and factors. For logical vectors, use `if_else()`. For more complicated criteria, use `case_when()`.

Usage

```
recode(.x, ..., .default = NULL, .missing = NULL)

recode_factor(.x, ..., .default = NULL, .missing = NULL,
  .ordered = FALSE)
```

Arguments

`.x`	A vector to modify
`...`	Replacements. For character and factor `.x`, these should be named and replacement is based only on their name. For numeric `.x`, these can be named or not. If not named, the replacement is done based on position i.e. `.x` represents positions to look for in replacements. See examples. When named, the argument names should be the current values to be replaced, and the argument values should be the new (replacement) values. All replacements must be the same type, and must have either length one or the same length as `.x`. These dots support tidy dots features.

`.default`	If supplied, all values not otherwise matched will be given this value. If not supplied and if the replacements are the same type as the original values in `.x`, unmatched values are not changed. If not supplied and if the replacements are not compatible, unmatched values are replaced with `NA`. `.default` must be either length 1 or the same length as `.x`.
`.missing`	If supplied, any missing values in `.x` will be replaced by this value. Must be either length 1 or the same length as `.x`.
`.ordered`	If `TRUE`, `recode_factor()` creates an ordered factor.

Details

You can use `recode()` directly with factors; it will preserve the existing order of levels while changing the values. Alternatively, you can use `recode_factor()`, which will change the order of levels to match the order of replacements. See the forcats package for more tools for working with factors and their levels.

Value

A vector the same length as `.x`, and the same type as the first of `...`, `.default`, or `.missing`. `recode_factor()` returns a factor whose levels are in the same order as in `...`. The levels in `.default` and `.missing` come last.

See Also

`na_if()` to replace specified values with a `NA`.

`coalesce()` to replace missing values with a specified value.

`tidyr::replace_na()` to replace `NA` with a value.

Examples

```
# For character values, recode values with named arguments only. Unmatched
# values are unchanged.
char_vec <- sample(c("a", "b", "c"), 10, replace = TRUE)
recode(char_vec, a = "Apple")
recode(char_vec, a = "Apple", b = "Banana")

# Use .default as replacement for unmatched values
recode(char_vec, a = "Apple", b = "Banana", .default = NA_character_)

# Use a named character vector for unquote splicing with !!!
level_key <- c(a = "apple", b = "banana", c = "carrot")
recode(char_vec, !!!level_key)

# For numeric values, named arguments can also be used
num_vec <- c(1:4, NA)
recode(num_vec, `2` = 20L, `4` = 40L)

# Or if you don't name the arguments, recode() matches by position.
# (Only works for numeric vector)
recode(num_vec, "a", "b", "c", "d")
# .x (position given) looks in (...), then grabs (... value at position)
# so if nothing at position (here 5), it uses .default or NA.
recode(c(1,5,3), "a", "b", "c", "d", .default = "nothing")
```

```
# Note that if the replacements are not compatible with .x,
# unmatched values are replaced by NA and a warning is issued.
recode(num_vec, `2` = "b", `4` = "d")
# use .default to change the replacement value
recode(num_vec, "a", "b", "c", .default = "other")
# use .missing to replace missing values in .x
recode(num_vec, "a", "b", "c", .default = "other", .missing = "missing")

# For factor values, use only named replacements
# and supply default with levels()
factor_vec <- factor(c("a", "b", "c"))
recode(factor_vec, a = "Apple", .default = levels(factor_vec))

# Use recode_factor() to create factors with levels ordered as they
# appear in the recode call. The levels in .default and .missing
# come last.
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x")
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x",
              .default = "D")
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x",
              .default = "D", .missing = "M")

# When the input vector is a compatible vector (character vector or
# factor), it is reused as default.
recode_factor(letters[1:3], b = "z", c = "y")
recode_factor(factor(letters[1:3]), b = "z", c = "y")

# Use a named character vector to recode factors with unquote splicing.
level_key <- c(a = "apple", b = "banana", c = "carrot")
recode_factor(char_vec, !!!level_key)
```

`rowwise` *Group input by rows*

Description

Questioning

Usage

```
rowwise(data)
```

Arguments

`data`	Input data frame.

Details

See this repository for alternative ways to perform row-wise operations

`rowwise()` is used for the results of `do()` when you create list-variables. It is also useful to support arbitrary complex operations that need to be applied to each row.

Currently, rowwise grouping only works with data frames. Its main impact is to allow you to work with list-variables in `summarise()` and `mutate()` without having to use `[[1]]`. This makes `summarise()` on a rowwise tbl effectively equivalent to `plyr::ldply()`.

Examples

```
df <- expand.grid(x = 1:3, y = 3:1)
df_done <- df %>% rowwise() %>% do(i = seq(.$x, .$y))
df_done
df_done %>% summarise(n = length(i))
```

`sample` *Sample n rows from a table*

Description

This is a wrapper around `sample.int()` to make it easy to select random rows from a table. It currently only works for local tbls.

Usage

```
sample_n(tbl, size, replace = FALSE, weight = NULL, .env = NULL, ...)

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL,
  .env = NULL, ...)
```

Arguments

`tbl`	tbl of data.
`size`	For `sample_n()`, the number of rows to select. For `sample_frac()`, the fraction of rows to select. If `tbl` is grouped, `size` applies to each group.
`replace`	Sample with or without replacement?
`weight`	Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. This argument is automatically quoted and later evaluated in the context of the data frame. It supports unquoting. See `vignette("programming")` for an introduction to these concepts.
`.env`	This variable is deprecated and no longer has any effect. To evaluate `weight` in a particular context, you can now unquote a quosure.
`...`	ignored

Examples

```
by_cyl <- mtcars %>% group_by(cyl)

# Sample fixed number per group
sample_n(mtcars, 10)
sample_n(mtcars, 50, replace = TRUE)
sample_n(mtcars, 10, weight = mpg)

sample_n(by_cyl, 3)
sample_n(by_cyl, 10, replace = TRUE)
sample_n(by_cyl, 3, weight = mpg / mean(mpg))

# Sample fixed fraction per group
```

```
# Default is to sample all data = randomly resample rows
sample_frac(mtcars)

sample_frac(mtcars, 0.1)
sample_frac(mtcars, 1.5, replace = TRUE)
sample_frac(mtcars, 0.1, weight = 1 / mpg)

sample_frac(by_cyl, 0.2)
sample_frac(by_cyl, 1, replace = TRUE)
```

---

scoped *Operate on a selection of variables*

Description

The variants suffixed with _if, _at or _all apply an expression (sometimes several) to all variables within a specified subset. This subset can contain all variables (_all variants), a vars() selection (_at variants), or variables selected with a predicate (_if variants).

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.

Details

The verbs with scoped variants are:

- mutate(), transmute() and summarise(). See summarise_all().
- filter(). See filter_all().
- group_by(). See group_by_all().
- rename() and select(). See select_all().
- arrange(). See arrange_all()

There are three kinds of scoped variants. They differ in the scope of the variable selection on which operations are applied:

- Verbs suffixed with _all() apply an operation on all variables.
- Verbs suffixed with _at() apply an operation on a subset of variables specified with the quoting function vars(). This quoting function accepts tidyselect::vars_select() helpers like starts_with(). Instead of a vars() selection, you can also supply an integerish vector of column positions or a character vector of column names.
- Verbs suffixed with _if() apply an operation on the subset of variables for which a predicate function returns TRUE. Instead of a predicate function, you can also supply a logical vector.

Grouping variables

Most of these operations also apply on the grouping variables when they are part of the selection. This includes:

- `arrange_all()`, `arrange_at()`, and `arrange_if()`
- `distinct_all()`, `distinct_at()`, and `distinct_if()`
- `filter_all()`, `filter_at()`, and `filter_if()`
- `group_by_all()`, `group_by_at()`, and `group_by_if()`
- `select_all()`, `select_at()`, and `select_if()`

This is not the case for summarising and mutating variants where operations are *not* applied on grouping variables. The behaviour depends on whether the selection is **implicit** (`all` and `if` selections) or **explicit** (`at` selections). Grouping variables covered by explicit selections (with `summarise_at()`, `mutate_at()`, and `transmute_at()`) are always an error. For implicit selections, the grouping variables are always ignored. In this case, the level of verbosity depends on the kind of operation:

- Summarising operations (`summarise_all()` and `summarise_if()`) ignore grouping variables silently because it is obvious that operations are not applied on grouping variables.
- On the other hand it isn't as obvious in the case of mutating operations (`mutate_all()`, `mutate_if()`, `transmute_all()`, and `transmute_if()`). For this reason, they issue a message indicating which grouping variables are ignored.

`select` *Select/rename variables by name*

Description

Choose or rename variables from a tbl. `select()` keeps only the variables you mention; `rename()` keeps all variables.

Usage

```
select(.data, ...)

rename(.data, ...)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	One or more unquoted expressions separated by commas. You can treat variable names like they are positions, so you can use expressions like `x:y` to select ranges of variables. Positive values select variables; negative values drop variables. If the first expression is negative, `select()` will automatically start with all variables. Use named arguments, e.g. `new_name = old_name`, to rename 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")` for an introduction to these concepts.

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

Details

These functions work by column index, not value; thus, an expression like `select(data.frame(x = 1:5,y = 10),z = x+1)` does not create a variable with values `2:6`. (In the current implementation, the expression `z = x+1` wouldn't do anything useful.) To calculate using column values, see `mutate()`/`transmute()`.

Value

An object of the same class as `.data`.

Useful functions

As well as using existing functions like `:` and `c()`, there are a number of special functions that only work inside `select()`:

- `starts_with()`, `ends_with()`, `contains()`
- `matches()`
- `num_range()`
- `one_of()`
- `everything()`
- `group_cols()`

To drop variables, use `-`.

Note that except for `:`, `-` and `c()`, all complex expressions are evaluated outside the data frame context. This is to prevent accidental matching of data frame variables when you refer to variables from the calling context.

Scoped selection and renaming

The three scoped variants of `select()` (`select_all()`, `select_if()` and `select_at()`) and the three variants of `rename()` (`rename_all()`, `rename_if()`, `rename_at()`) make it easy to apply a renaming function to a selection of variables.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with `tibble::rownames_to_column()`.

See Also

Other single table verbs: `arrange`, `filter`, `mutate`, `slice`, `summarise`

Examples

```
iris <- as_tibble(iris) # so it prints a little nicer
select(iris, starts_with("Petal"))
select(iris, ends_with("Width"))

# Move Species variable to the front
select(iris, Species, everything())

# Move Sepal.Length variable to back
# first select all variables except Sepal.Length, then re select Sepal.Length
select(iris, -Sepal.Length, Sepal.Length)

df <- as.data.frame(matrix(runif(100), nrow = 10))
df <- tbl_df(df[c(3, 4, 7, 1, 9, 8, 5, 2, 6, 10)])
select(df, V4:V6)
select(df, num_range("V", 4:6))

# Drop variables with -
select(iris, -starts_with("Petal"))

# Select the grouping variables:
starwars %>% group_by(gender) %>% select(group_cols())


# The .data pronoun is available:
select(mtcars, .data$cyl)
select(mtcars, .data$mpg : .data$disp)

# However it isn't available within calls since those are evaluated
# outside of the data context. This would fail if run:
# select(mtcars, identical(.data$cyl))


# Renaming ----------------------------------------
# * select() keeps only the variables you specify
select(iris, petal_length = Petal.Length)

# * rename() keeps all variables
rename(iris, petal_length = Petal.Length)

# * select() can rename variables in a group
select(iris, obs = starts_with('S'))

# Unquoting ----------------------------------------

# Like all dplyr verbs, select() supports unquoting of symbols:
vars <- list(
  var1 = sym("cyl"),
  var2 = sym("am")
)
select(mtcars, !!!vars)

# For convenience it also supports strings and character
# vectors. This is unlike other verbs where strings would be
# ambiguous.
vars <- c(var1 = "cyl", var2 ="am")
```

```
select(mtcars, !!vars)
rename(mtcars, !!vars)
```

`select_all` *Select and rename a selection of variables*

Description

These scoped variants of `select()` and `rename()` operate on a selection of variables. The semantics of these verbs have subtle but important differences:

- Selection drops variables that are not in the selection while renaming retains them.
- The renaming function is optional for selection but not for renaming.

The `_if` and `_at` variants always retain grouping variables for grouped data frames.

Usage

```
select_all(.tbl, .funs = list(), ...)

rename_all(.tbl, .funs = list(), ...)

select_if(.tbl, .predicate, .funs = list(), ...)

rename_if(.tbl, .predicate, .funs = list(), ...)

select_at(.tbl, .vars, .funs = list(), ...)

rename_at(.tbl, .vars, .funs = list(), ...)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a purrr style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.

Grouping variables

Existing grouping variables are always kept in the data frame, even if not included in the selection.

Examples

```
# Supply a renaming function:
select_all(mtcars, toupper)
select_all(mtcars, "toupper")
select_all(mtcars, list(~toupper(.)))

# Selection drops unselected variables:
is_whole <- function(x) all(floor(x) == x)
select_if(mtcars, is_whole, toupper)
select_at(mtcars, vars(-contains("ar"), starts_with("c")), toupper)

# But renaming retains them:
rename_if(mtcars, is_whole, toupper)
rename_at(mtcars, vars(-(1:3)), toupper)
rename_all(mtcars, toupper)

# The renaming function is optional for selection:
select_if(mtcars, is_whole)
select_at(mtcars, vars(-everything()))
select_all(mtcars)
```

---

select_vars *Select variables*

Description

Retired: These functions now live in the tidyselect package as `tidyselect::vars_select()`, `tidyselect::vars_rename()` and `tidyselect::vars_pull()`. These dplyr aliases are soft-deprecated and will be deprecated sometimes in the future.

Usage

```
select_vars(vars = chr(), ..., include = chr(), exclude = chr())

rename_vars(vars = chr(), ..., strict = TRUE)

select_var(vars, var = -1)

current_vars(...)
```

Arguments

`vars`	A character vector of existing column names.
`...`	Expressions to compute.
`include, exclude`	Character vector of column names to always include/exclude.
`strict`	If `TRUE`, will throw an error if you attempt to rename a variable that doesn't exist.
`var`	A variable specified as in the same argument of `tidyselect::vars_pull()`.

Details

Deprecated

`setops` *Set operations*

Description

These functions override the set functions provided in base to make them generic so that efficient versions for data frames and other tables can be provided. The default methods call the base versions. Beware that `intersect()`, `union()` and `setdiff()` remove duplicates.

Usage

```
intersect(x, y, ...)

union(x, y, ...)

union_all(x, y, ...)

setdiff(x, y, ...)

setequal(x, y, ...)
```

Arguments

`x, y`	objects to perform set function on (ignoring order)
`...`	other arguments passed on to methods

Examples

```
mtcars$model <- rownames(mtcars)
first <- mtcars[1:20, ]
second <- mtcars[10:32, ]

intersect(first, second)
union(first, second)
setdiff(first, second)
setdiff(second, first)

union_all(first, second)
setequal(mtcars, mtcars[32:1, ])

# Handling of duplicates:
a <- data.frame(column = c(1:10, 10))
b <- data.frame(column = c(1:5, 5))

# intersection is 1 to 5, duplicates removed (5)
intersect(a, b)

# union is 1 to 10, duplicates removed (5 and 10)
union(a, b)
```

```
# set difference, duplicates removed (10)
setdiff(a, b)

# union all does not remove duplicates
union_all(a, b)
```

`slice` *Choose rows by position*

Description

Choose rows by their ordinal position in the tbl. Grouped tbls use the ordinal position within the group.

Usage

```
slice(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A tbl.
`...`	Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. The arguments in `...` are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts.
`.preserve`	when `FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.

Details

Slice does not work with relational databases because they have no intrinsic notion of row order. If you want to perform the equivalent operation, use `filter()` and `row_number()`.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with `tibble::rownames_to_column()`.

See Also

Other single table verbs: `arrange`, `filter`, `mutate`, `select`, `summarise`

Examples

```
slice(mtcars, 1L)
# Similar to tail(mtcars, 1):
slice(mtcars, n())
slice(mtcars, 5:n())
# Rows can be dropped with negative indices:
slice(mtcars, -5:-n())
# In this case, the result will be equivalent to:
slice(mtcars, 1:4)

by_cyl <- group_by(mtcars, cyl)
slice(by_cyl, 1:2)

# Equivalent code using filter that will also work with databases,
# but won't be as fast for in-memory data. For many databases, you'll
# need to supply an explicit variable to use to compute the row number.
filter(mtcars, row_number() == 1L)
filter(mtcars, row_number() == n())
filter(mtcars, between(row_number(), 5, n()))
```

---

sql *SQL escaping.*

Description

These functions are critical when writing functions that translate R functions to sql functions. Typically a conversion function should escape all its inputs and return an sql object.

Usage

```
sql(...)
```

Arguments

...	Character vectors that will be combined into a single SQL expression.

---

src_dbi *Source for database backends*

Description

For backward compatibility dplyr provides three srcs for popular open source databases:

- `src_mysql()` connects to a MySQL or MariaDB database using `RMySQL::MySQL()`.
- `src_postgres()` connects to PostgreSQL using `RPostgreSQL::PostgreSQL()`
- `src_sqlite()` to connect to a SQLite database using `RSQLite::SQLite()`.

However, modern best practice is to use `tbl()` directly on an `DBIConnection`.

Usage

```
src_mysql(dbname, host = NULL, port = 0L, username = "root",
  password = "", ...)

src_postgres(dbname = NULL, host = NULL, port = NULL, user = NULL,
  password = NULL, ...)

src_sqlite(path, create = FALSE)
```

Arguments

dbname	Database name
host, port	Host name and port number of database
...	for the src, other arguments passed on to the underlying database connector, DBI::dbConnect(). For the tbl, included for compatibility with the generic, but otherwise ignored.
user, username, password	User name and password. Generally, you should avoid saving username and password in your scripts as it is easy to accidentally expose valuable credentials. Instead, retrieve them from environment variables, or use database specific credential scores. For example, with MySQL you can set up my.cnf as described in RMySQL::MySQL().
path	Path to SQLite database. You can use the special path ":memory:" to create a temporary in memory database.
create	if FALSE, path must already exist. If TRUE, will create a new SQLite3 database at path if path does not exist and connect to the existing database if path does exist.

Details

All data manipulation on SQL tbls are lazy: they will not actually run the query or retrieve the data unless you ask for it: they all return a new tbl_dbi object. Use compute() to run the query and save the results in a temporary in the database, or use collect() to retrieve the results to R. You can see the query with show_query().

For best performance, the database should have an index on the variables that you are grouping by. Use explain() to check that the database is using the indexes that you expect.

There is one exception: do() is not lazy since it must pull the data into R.

Value

An S3 object with class src_dbi, src_sql, src.

Examples

```
# Basic connection using DBI ----------------------------------------------
if (require(dbplyr, quietly = TRUE)) {

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
copy_to(con, mtcars)
```

```
DBI::dbListTables(con)

# To retrieve a single table from a source, use `tbl()`
con %>% tbl("mtcars")

# You can also use pass raw SQL if you want a more sophisticated query
con %>% tbl(sql("SELECT * FROM mtcars WHERE cyl == 8"))

# To show off the full features of dplyr's database integration,
# we'll use the Lahman database. lahman_sqlite() takes care of
# creating the database.
lahman_p <- lahman_sqlite()
batting <- lahman_p %>% tbl("Batting")
batting

# Basic data manipulation verbs work in the same way as with a tibble
batting %>% filter(yearID > 2005, G > 130)
batting %>% select(playerID:lgID)
batting %>% arrange(playerID, desc(yearID))
batting %>% summarise(G = mean(G), n = n())

# There are a few exceptions. For example, databases give integer results
# when dividing one integer by another. Multiply by 1 to fix the problem
batting %>%
  select(playerID:lgID, AB, R, G) %>%
  mutate(
   R_per_game1 = R / G,
   R_per_game2 = R * 1.0 / G
 )

# All operations are lazy: they don't do anything until you request the
# data, either by `print()`ing it (which shows the first ten rows),
# or by `collect()`ing the results locally.
system.time(recent <- filter(batting, yearID > 2010))
system.time(collect(recent))

# You can see the query that dplyr creates with show_query()
batting %>%
  filter(G > 0) %>%
  group_by(playerID) %>%
  summarise(n = n()) %>%
  show_query()
}
```

`starwars` *Starwars characters*

Description

This data comes from SWAPI, the Star Wars API, http://swapi.co/

Usage

```
starwars
```

Format

A tibble with 87 rows and 13 variables:

name Name of the character

height Height (cm)

mass Weight (kg)

hair_color,skin_color,eye_color Hair, skin, and eye colors

birth_year Year born (BBY = Before Battle of Yavin)

gender male, female, hermaphrodite, or none.

homeworld Name of homeworld

species Name of species

films List of films the character appeared in

vehicles List of vehicles the character has piloted

starships List of starships the character has piloted

Examples

```
starwars
```

`storms` *Storm tracks data*

Description

This data is a subset of the NOAA Atlantic hurricane database best track data, http://www.nhc.noaa.gov/data/#hurdat. The data includes the positions and attributes of 198 tropical storms, measured every six hours during the lifetime of a storm.

Usage

```
storms
```

Format

A tibble with 10,010 observations and 13 variables:

name Storm Name

year,month,day Date of report

hour Hour of report (in UTC)

lat,long Location of storm center

status Storm classification (Tropical Depression, Tropical Storm, or Hurricane)

category Saffir-Simpson storm category (estimated from wind speed. -1 = Tropical Depression, 0 = Tropical Storm)

wind storm's maximum sustained wind speed (in knots)

pressure Air pressure at the storm's center (in millibars)

ts_diameter Diameter of the area experiencing tropical storm strength winds (34 knots or above)

hu_diameter Diameter of the area experiencing hurricane strength winds (64 knots or above)

See Also

The script to create the storms data set: https://github.com/tidyverse/dplyr/blob/master/data-raw/storms.R

Examples

```
storms
```

---

summarise *Reduce multiple values down to a single value*

Description

Create one or more scalar variables summarizing the variables of an existing tbl. Tbls with groups created by `group_by()` will result in one row in the output for each group. Tbls with no groups will result in one row.

Usage

```
summarise(.data, ...)

summarize(.data, ...)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	Name-value pairs of summary functions. The name will be the name of the variable in the result. The value should be an expression that returns a single value like `min(x)`, `n()`, or `sum(is.na(y))`. The arguments in `...` are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts.

Details

`summarise()` and `summarize()` are synonyms.

Value

An object of the same class as `.data`. One grouping level will be dropped.

Useful functions

- Center: `mean()`, `median()`
- Spread: `sd()`, `IQR()`, `mad()`
- Range: `min()`, `max()`, `quantile()`
- Position: `first()`, `last()`, `nth()`,
- Count: `n()`, `n_distinct()`
- Logical: `any()`, `all()`

Backend variations

The data frame backend supports creating a variable and using it in the same summary. This means that previously created summary variables can be further transformed or combined within the summary, as in mutate(). However, it also means that summary variables with the same names as previous variables overwrite them, making those variables unavailable to later summary variables.

This behaviour may not be supported in other backends. To avoid unexpected results, consider using new names for your summary variables, especially when creating multiple summaries.

Tidy data

When applied to a data frame, row names are silently dropped. To preserve, convert to an explicit variable with tibble::rownames_to_column().

See Also

Other single table verbs: arrange, filter, mutate, select, slice

Examples

```
# A summary applied to ungrouped tbl returns a single row
mtcars %>%
  summarise(mean = mean(disp), n = n())

# Usually, you'll want to group first
mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(disp), n = n())

# Each summary call removes one grouping level (since that group
# is now just a single row)
mtcars %>%
  group_by(cyl, vs) %>%
  summarise(cyl_n = n()) %>%
  group_vars()


# Reusing variable names when summarising may lead to unexpected results
mtcars %>%
  group_by(cyl) %>%
  summarise(disp = mean(disp), sd = sd(disp), double_disp = disp * 2)


# Refer to column names stored as strings with the `.data` pronoun:
var <- "mass"
summarise(starwars, avg = mean(.data[[var]], na.rm = TRUE))

# For more complex cases, knowledge of tidy evaluation and the
# unquote operator `!!` is required. See https://tidyeval.tidyverse.org/
#
# One useful and simple tidy eval technique is to use `!!` to
# bypass the data frame and its columns. Here is how to divide the
# column `mass` by an object of the same name:
mass <- 100
summarise(starwars, avg = mean(mass / !!mass, na.rm = TRUE))
```

summarise_all *Summarise multiple columns*

Description

The scoped variants of `summarise()` make it easy to apply the same transformation to multiple variables. There are three variants.

- `summarise_all()` affects every variable
- `summarise_at()` affects variables selected with a character vector or vars()
- `summarise_if()` affects variables selected with a predicate function

Usage

```
summarise_all(.tbl, .funs, ...)

summarise_if(.tbl, .predicate, .funs, ...)

summarise_at(.tbl, .vars, .funs, ..., .cols = NULL)

summarize_all(.tbl, .funs, ...)

summarize_if(.tbl, .predicate, .funs, ...)

summarize_at(.tbl, .vars, .funs, ..., .cols = NULL)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.cols`	This argument has been renamed to `.vars` to fit dplyr's terminology and is deprecated.

Value

A data frame. By default, the newly created columns have the shortest names needed to uniquely identify the output. To force inclusion of a name, even when not needed, name the input (see examples for details).

Grouping variables

If applied on a grouped tibble, these operations are *not* applied to the grouping variables. The behaviour depends on whether the selection is **implicit** (`all` and `if` selections) or **explicit** (`at` selections).

- Grouping variables covered by explicit selections in `summarise_at()` are always an error. Add `-group_cols()` to the `vars()` selection to avoid this:

  ```
  data %>%
    summarise_at(vars(-group_cols(), ...), myoperation)
  ```

  Or remove `group_vars()` from the character vector of column names:

  ```
  nms <- setdiff(nms, group_vars(data))
  data %>% summarise_at(vars, myoperation)
  ```

- Grouping variables covered by implicit selections are silently ignored by `summarise_all()` and `summarise_if()`.

Naming

The names of the created columns is derived from the names of the input variables and the names of the functions.

- if there is only one unnamed function, the names of the input variables are used to name the created columns
- if there is only one unnamed variable, the names of the functions are used to name the created columns.
- otherwise in the most general case, the created names are created by concatenating the names of the input variables and the names of the functions.

The names of the functions here means the names of the list of functions that is supplied. When needed and not supplied, the name of a function is the prefix "fn" followed by the index of this function within the unnamed functions in the list. Ultimately, names are made unique.

See Also

The other scoped verbs, `vars()`

Examples

```
by_species <- iris %>%
  group_by(Species)


# The _at() variants directly support strings:
starwars %>%
  summarise_at(c("height", "mass"), mean, na.rm = TRUE)

# You can also supply selection helpers to _at() functions but you have
# to quote them with vars():
starwars %>%
  summarise_at(vars(height:mass), mean, na.rm = TRUE)

# The _if() variants apply a predicate function (a function that
# returns TRUE or FALSE) to determine the relevant subset of
```

```
# columns. Here we apply mean() to the numeric columns:
starwars %>%
  summarise_if(is.numeric, mean, na.rm = TRUE)

# If you want to apply multiple transformations, pass a list of
# functions. When there are multiple functions, they create new
# variables instead of modifying the variables in place:
by_species %>%
  summarise_all(list(min, max))

# Note how the new variables include the function name, in order to
# keep things distinct. Passing purrr-style lambdas often creates
# better default names:
by_species %>%
  summarise_all(list(~min(.), ~max(.)))

# When that's not good enough, you can also supply the names explicitly:
by_species %>%
  summarise_all(list(min = min, max = max))

# When there's only one function in the list, it modifies existing
# variables in place. Give it a name to create new variables instead:
by_species %>% summarise_all(list(med = median))
by_species %>% summarise_all(list(Q3 = quantile), probs = 0.75)
```

`tally` *Count/tally observations by group*

Description

`tally()` is a convenient wrapper for summarise that will either call `n()` or `sum(n)` depending on whether you're tallying for the first time, or re-tallying. `count()` is similar but calls `group_by()` before and `ungroup()` after. If the data is already grouped, `count()` adds an additional group that is removed afterwards.

`add_tally()` adds a column `n` to a table based on the number of items within each existing group, while `add_count()` is a shortcut that does the grouping as well. These functions are to `tally()` and `count()` as `mutate()` is to `summarise()`: they add an additional column rather than collapsing each group.

Usage

```
tally(x, wt = NULL, sort = FALSE, name = "n")

count(x, ..., wt = NULL, sort = FALSE, name = "n",
  .drop = group_by_drop_default(x))

add_tally(x, wt, sort = FALSE, name = "n")

add_count(x, ..., wt = NULL, sort = FALSE, name = "n")
```

Arguments

x	a tbl() to tally/count.
wt	(Optional) If omitted (and no variable named n exists in the data), will count the number of rows. If specified, will perform a "weighted" tally by summing the (non-missing) values of variable wt. A column named n (but not nn or nnn) will be used as weighting variable by default in tally(), but not in count(). This argument is automatically quoted and later evaluated in the context of the data frame. It supports unquoting. See vignette("programming") for an introduction to these concepts.
sort	if TRUE will sort output in descending order of n
name	The output column name. If omitted, it will be n.
...	Variables to group by.
.drop	see group_by()

Value

A tbl, grouped the same way as x.

Note

The column name in the returned data is given by the name argument, set to "n" by default.

If the data already has a column by that name, the output column will be prefixed by an extra "n" as many times as necessary.

Examples

```
# tally() is short-hand for summarise()
mtcars %>% tally()
mtcars %>% group_by(cyl) %>% tally()
# count() is a short-hand for group_by() + tally()
mtcars %>% count(cyl)
# Note that if the data is already grouped, count() adds
# an additional group that is removed afterwards
mtcars %>% group_by(gear) %>% count(carb)

# add_tally() is short-hand for mutate()
mtcars %>% add_tally()
# add_count() is a short-hand for group_by() + add_tally()
mtcars %>% add_count(cyl)

# count() and tally() are designed so that you can call
# them repeatedly, each time rolling up a level of detail
species <-
 starwars %>%
 count(species, homeworld, sort = TRUE)
species
species %>% count(species, sort = TRUE)

# Change the name of the newly created column:
species <-
 starwars %>%
 count(species, homeworld, sort = TRUE, name = "n_species_by_homeworld")
species
```

```
species %>%
 count(species, sort = TRUE, name = "n_species")

# add_count() is useful for groupwise filtering
# e.g.: show details for species that have a single member
starwars %>%
  add_count(species) %>%
  filter(n == 1)
```

`tbl` *Create a table from a data source*

Description

This is a generic method that dispatches based on the first argument.

Usage

```
tbl(src, ...)

is.tbl(x)

as.tbl(x, ...)
```

Arguments

`src`	A data source
`...`	Other arguments passed on to the individual methods
`x`	an object to coerce to a `tbl`

`tbl_cube` *A data cube tbl*

Description

A cube tbl stores data in a compact array format where dimension names are not needlessly repeated. They are particularly appropriate for experimental data where all combinations of factors are tried (e.g. complete factorial designs), or for storing the result of aggregations. Compared to data frames, they will occupy much less memory when variables are crossed, not nested.

Usage

```
tbl_cube(dimensions, measures)
```

Arguments

`dimensions`	A named list of vectors. A dimension is a variable whose values are known before the experiment is conducted; they are fixed by design (in **reshape2** they are known as id variables). `tbl_cubes` are dense which means that almost every combination of the dimensions should have associated measurements: missing values require an explicit NA, so if the variables are nested, not crossed, the majority of the data structure will be empty. Dimensions are typically, but not always, categorical variables.
`measures`	A named list of arrays. A measure is something that is actually measured, and is not known in advance. The dimension of each array should be the same as the length of the dimensions. Measures are typically, but not always, continuous values.

Details

`tbl_cube` support is currently experimental and little performance optimisation has been done, but you may find them useful if your data already comes in this form, or you struggle with the memory overhead of the sparse/crossed of data frames. There is no support for hierarchical indices (although I think that would be a relatively straightforward extension to storing data frames for indices rather than vectors).

Implementation

Manipulation functions:

- `select()` (M)
- `summarise()` (M), corresponds to roll-up, but rather more limited since there are no hierarchies.
- `filter()` (D), corresponds to slice/dice.
- `mutate()` (M) is not implemented, but should be relatively straightforward given the implementation of `summarise`.
- `arrange()` (D?) Not implemented: not obvious how much sense it would make

Joins: not implemented. See `vignettes/joins.graffle` for ideas. Probably straightforward if you get the indexes right, and that's probably some straightforward array/tensor operation.

See Also

`as.tbl_cube()` for ways of coercing existing data structures into a `tbl_cube`.

Examples

```
# The built in nasa dataset records meterological data (temperature,
# cloud cover, ozone etc) for a 4d spatio-temporal dataset (lat, long,
# month and year)
nasa
head(as.data.frame(nasa))

titanic <- as.tbl_cube(Titanic)
head(as.data.frame(titanic))

admit <- as.tbl_cube(UCBAdmissions)
```

```
head(as.data.frame(admit))

as.tbl_cube(esoph, dim_names = 1:3)

# Some manipulation examples with the NASA dataset -------------------------

# select() operates only on measures: it doesn't affect dimensions in any way
select(nasa, cloudhigh:cloudmid)
select(nasa, matches("temp"))

# filter() operates only on dimensions
filter(nasa, lat > 0, year == 2000)
# Each component can only refer to one dimensions, ensuring that you always
# create a rectangular subset
## Not run: filter(nasa, lat > long)

# Arrange is meaningless for tbl_cubes

by_loc <- group_by(nasa, lat, long)
summarise(by_loc, pressure = max(pressure), temp = mean(temperature))
```

`top_n` *Select top (or bottom) n rows (by value)*

Description

This is a convenient wrapper that uses `filter()` and `min_rank()` to select the top or bottom entries in each group, ordered by `wt`.

Usage

```
top_n(x, n, wt)

top_frac(x, n, wt)
```

Arguments

`x`	a `tbl()` to filter
`n`	number of rows to return for `top_n()`, fraction of rows to return for `top_frac()`. If `x` is grouped, this is the number (or fraction) of rows per group. Will include more rows if there are ties. If `n` is positive, selects the top rows. If negative, selects the bottom rows.
`wt`	(Optional). The variable to use for ordering. If not specified, defaults to the last variable in the tbl.

Details

Both `n` and `wt` are automatically quoted and later evaluated in the context of the data frame. It supports unquoting.

Examples

```
df <- data.frame(x = c(10, 4, 1, 6, 3, 1, 1))
df %>% top_n(2)

# half the rows
df %>% top_n(n() * .5)
df %>% top_frac(.5)

# Negative values select bottom from group. Note that we get more
# than 2 values here because there's a tie: top_n() either takes
# all rows with a value, or none.
df %>% top_n(-2)

if (require("Lahman")) {
# Find 10 players with most games
tbl_df(Batting) %>%
  group_by(playerID) %>%
  tally(G) %>%
  top_n(10)

# Find year with most games for each player
## Not run:
tbl_df(Batting) %>%
  group_by(playerID) %>%
  top_n(1, G)

## End(Not run)
}
```

---

vars *Select variables*

Description

This helper is intended to provide equivalent semantics to `select()`. It is used for instance in scoped summarising and mutating verbs (`mutate_at()` and `summarise_at()`).

Usage

```
vars(...)
```

Arguments

`...`	Variables to include/exclude in mutate/summarise. You can use same specifications as in `select()`. If missing, defaults to all non-grouping variables. These arguments are automatically quoted and later evaluated in the context of the data frame. They support unquoting. See `vignette("programming")` for an introduction to these concepts.

Details

Note that verbs accepting a `vars()` specification also accept a numeric vector of positions or a character vector of column names.

See Also

`all_vars()` and `any_vars()` for other quoting functions that you can use with scoped verbs.

Index