Package 'dplyr'

June 22, 2017

Type Package

Version 0.7.1

Title A Grammar of Data Manipulation

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

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

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

Encoding UTF-8

Depends R (>= 3.1.2)

Imports assertthat, bindrcpp (>= 0.2), glue (>= 1.1.0), magrittr, methods, pkgconfig, rlang (>= 0.1), R6, Rcpp (>= 0.12.6), tibble (>= 1.3.1), utils

Suggests bit64, covr, dbplyr, dtplyr, DBI, ggplot2, hms, knitr, Lahman (>= 3.0-1), mgcv, microbenchmark, nycflights13, rmarkdown, RMySQL, RPostgreSQL, RSQLite, testthat, withr

VignetteBuilder knitr

LinkingTo Rcpp (>= 0.12.0), BH (>= 1.58.0-1), bindrcpp, plogr

LazyData yes

License MIT + file LICENSE

RoxygenNote 6.0.1

NeedsCompilation yes

Author Hadley Wickham [aut, cre], Romain Francois [aut], Lionel Henry [aut], Kirill Müller [aut], RStudio [cph, fnd]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2017-06-22 13:31:04 UTC

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>

Authors:

- Romain Francois <romain@r-enthusiasts.com>
- Lionel Henry
- Kirill Müller

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

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

---

arrange *Arrange rows by variables*

Description

Use `desc()` to sort a variable in descending order.

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. 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()`.

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 funs().

Usage

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

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

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

Arguments

.tbl	A tbl object.
.funs	List of function calls generated by funs(), or a character vector of function names, or simply a function. Bare formulas are passed to rlang::as_function() to create purrr-style lambda functions. Note that these lambda prevent hybrid evaluation from happening and it is thus more efficient to supply functions like mean() directly rather than in a lambda-formula.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with explicit splicing.
.vars	A list of columns generated by vars(), or a character vector of column names, or a numeric vector of column positions.
.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.

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, funs(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_data_frame()` 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_data_frame(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 dimesions. 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 neccessary.
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

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

Usage

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

bind_cols(...)

combine(...)
```

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 with 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 is automatically spliced:
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),
  data_frame(a = 3:4, b = 5:6),
  c(a = 7, b = 8)
)


# Note that for historical reasons, lists containg 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))

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

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

---

case_when *A general vectorised if*

Description

This function allows you to vectorise multiple `if` and `else if` statements. It is an R equivalent of the SQL `CASE WHEN` statement.

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. Each logical vector can either have length 1 or a common length. All RHSs must evaluate to the same type of vector. These dots are evaluated with explicit splicing.

Value

A vector as long as the longest LHS, 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"
)

# 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"
```

```
    )
  )

# Dots support splicing:
patterns <- list(
  TRUE ~ as.character(x),
  x %%  5 == 0 ~ "fizz",
  x %%  7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz"
)
case_when(!!! patterns)
```

`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 are evaluated with explicit splicing.

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`.

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)
```

`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 <- src_memdb() %>% copy_to(iris, overwrite = TRUE)
iris2

## End(Not run)
```

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

Description

dplyr adds `cumall()`, `cumany()`, and `cummean()` to complete R's set of cumulate functions to match the aggregation functions available in most databases

Usage

```
cumall(x)

cumany(x)

cummean(x)
```

Arguments

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

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.

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)
```

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()`.

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

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.

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.

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)
}
```

`explain` *Explain details of a tbl*

Description

This is a generic function which gives more details about an object than `print()`, and is more focussed 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()` find rows/cases where conditions are true. Unlike base subsetting, rows where the condition evaluates to `NA` are dropped.

Usage

```
filter(.data, ...)
```

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 conditon evalutes to `TRUE` are kept. These arguments 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

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()`

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")
```

---

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)

filter_if(.tbl, .predicate, .vars_predicate)

filter_at(.tbl, .vars, .vars_predicate)
```

Arguments

`.tbl`	A `tbl` object.
`.vars_predicate`	A quoted predicate expression as returned by `all_vars()` or `any_vars()`.
`.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()`, or a character vector of column names, or a numeric vector of column positions.

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))
```

`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.

`.args, args` A named list of additional arguments to be added to all function calls.

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)
```

---

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()`

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)

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.
x	A tbl()

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.

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()
```

`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(), ...)

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

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

Arguments

`.tbl`	A `tbl` object.
`.funs`	List of function calls generated by `funs()`, or a character vector of function names, or simply a function. Bare formulas are passed to `rlang::as_function()` to create purrr-style lambda functions. Note that these lambda prevent hybrid evaluation from happening and it is thus more efficient to supply functions like `mean()` directly rather than in a lambda-formula.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with explicit splicing.
`.vars`	A list of columns generated by `vars()`, or a character vector of column names, or a numeric vector of column positions.
`.add`	Passed to the `add` argument of `group_by()`.

.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.

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():
group_by_all(mtcars, as.factor)
group_by_if(iris, is.factor, as.character)
```

---

ident *Flag a character vector as SQL identifiers*

Description

ident() takes unquoted strings and quotes them for you; ident_q() assumes its input has already been quoted.

Usage

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

Arguments

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

Details

These two ident clsases are used during SQL generation to make sure the values will be quoted as, not as strings.

`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, ...)

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 diambiguate them. Should be a character vector of length 2.
...	other parameters passed onto methods

Join types

Currently dplyr supports four join types:

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.

`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`.

`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.

Grouping

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

Examples

```
# "Mutating" joins add variables to the LHS
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)

# 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
```

---

join.tbl_df *Join data frame tbls*

Description

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

Usage

```
## 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'
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.
suffix	If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to diambiguate them. Should be a character vector of length 2.
...	included for compatibility with the generic; otherwise ignored.
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" =

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)
}
```

`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 postive integer of length 1, giving the number of positions to lead or lag by
`default`	value used for non-existant 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 *Add new variables*

Description

`mutate()` adds new variables and preserves existing; `transmute()` drops existing variables.

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. Use `NULL` to drop a variable. These arguments 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

- +, - etc
- log()
- 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()

Scoped mutation and transmuation

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)


# mutate() supports quasiquotation. You can unquote quosures, which
# can refer to both contextual variables and variable names:
var <- 100
as_tibble(mtcars) %>% mutate(cyl = !! quo(cyl * var))
```

---

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.

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, "")
```

`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)
```

`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	id 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, mimicing 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, 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, and character values 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. These should be named for character and factor `.x`, and can be named for numeric `.x`. 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 are evaluated with explicit splicing.
`.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 `...`.

Examples

```
# Recode values with named arguments
x <- sample(c("a", "b", "c"), 10, replace = TRUE)
recode(x, a = "Apple")
recode(x, a = "Apple", .default = NA_character_)

# Named arguments also work with numeric values
x <- c(1:5, NA)
recode(x, `2` = 20L, `4` = 40L)

# Note that if the replacements are not compatible with .x,
# unmatched values are replaced by NA and a warning is issued.
recode(x, `2` = "b", `4` = "d")

# If you don't name the arguments, recode() matches by position
recode(x, "a", "b", "c")
recode(x, "a", "b", "c", .default = "other")
recode(x, "a", "b", "c", .default = "other", .missing = "missing")

# Supply default with levels() for factors
x <- factor(c("a", "b", "c"))
recode(x, a = "Apple", .default = levels(x))

# Use recode_factor() to create factors with levels ordered as they
# appear in the recode call. The levels in .default and .missing
# come last.
x <- c(1:4, NA)
recode_factor(x, `1` = "z", `2` = "y", `3` = "x")
recode_factor(x, `1` = "z", `2` = "y", .default = "D")
recode_factor(x, `1` = "z", `2` = "y", .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")
```

rowwise *Group input by rows*

Description

`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.

Usage

```
rowwise(data)
```

Arguments

`data`	Input data frame.

Details

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 %>% rowwise() %>% do(i = seq(.$x, .$y))
.Last.value %>% 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.

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	List of function calls generated by funs(), or a character vector of function names, or simply a function. Bare formulas are passed to rlang::as_function() to create purrr-style lambda functions. Note that these lambda prevent hybrid evaluation from happening and it is thus more efficient to supply functions like mean() directly rather than in a lambda-formula.
.vars	A list of columns generated by vars(), or a character vector of column names, or a numeric vector of column positions.
.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 explicit splicing.

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 select_vars() 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.

`select` *Select/rename variables by name*

Description

`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. Positive values select variables; negative values to drop variables. If the first expression is negative, `select()` will automatically start with all variables. Use named arguments to rename selected variables. These arguments are automatically quoted and evaluated in a context where column names represent column positions. 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

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()`

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())

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"))


# 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_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 simple 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.

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 single expression quoted with `funs()` or within a quosure, a string naming a function, or a function.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with explicit splicing.
`.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()`, or a character vector of column names, or a numeric vector of column positions.

Examples

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

```
# Selection drops unselected variables:
is_whole <- function(x) all(floor(x) == x)
select_if(mtcars, is_whole, toupper)

# But renaming retains them:
rename_if(mtcars, is_whole, toupper)

# The renaming function is optional for selection:
select_if(mtcars, is_whole)
```

`select_helpers` *Select helpers*

Description

These functions allow you to select variables based on their names.

- `starts_with()`: starts with a prefix
- `ends_with()`: ends with a prefix
- `contains()`: contains a literal string
- `matches()`: matches a regular expression
- `num_range()`: a numerical range like x01, x02, x03.
- `one_of()`: variables in character vector.
- `everything()`: all variables.

Usage

```
current_vars()

starts_with(match, ignore.case = TRUE, vars = current_vars())

ends_with(match, ignore.case = TRUE, vars = current_vars())

contains(match, ignore.case = TRUE, vars = current_vars())

matches(match, ignore.case = TRUE, vars = current_vars())

num_range(prefix, range, width = NULL, vars = current_vars())

one_of(..., vars = current_vars())

everything(vars = current_vars())
```

Arguments

match	A string.
ignore.case	If TRUE, the default, ignores case when matching names.
vars	A character vector of variable names. When called from inside select() these are automatically set to the names of the table.
prefix	A prefix that starts the numeric range.
range	A sequence of integers, like 1:5
width	Optionally, the "width" of the numeric range. For example, a range of 2 gives "01", a range of three "001", etc.
...	One or more character vectors.

Value

An integer vector giving the position of the matched variables.

Examples

```
iris <- tbl_df(iris) # so it prints a little nicer
select(iris, starts_with("Petal"))
select(iris, ends_with("Width"))
select(iris, contains("etal"))
select(iris, matches(".t."))
select(iris, Petal.Length, Petal.Width)
select(iris, everything())
vars <- c("Petal.Length", "Petal.Width")
select(iris, one_of(vars))
```

---

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.

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, ])
```

---

slice *Select rows by position*

Description

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()`.

Usage

```
slice(.data, ...)
```

Arguments

.data	A tbl.
...	Integer row values. These arguments 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.

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)
slice(mtcars, n())
slice(mtcars, 5:n())

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 excpetion: 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)

Examples

```
storms
```

---

summarise *Reduces multiple values down to a single value*

Description

`summarise()` is typically used on grouped data created by `group_by()`. The output will have one row for each group.

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))`. These arguments 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`. 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

Data frames are the only backend that supports creating a variable and using it in the same summary. See examples for more details.

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()

# Note that with data frames, newly created summaries immediately
# overwrite existing variables
mtcars %>%
  group_by(cyl) %>%
  summarise(disp = mean(disp), sd = sd(disp))
```

```
# summarise() supports quasiquotation. You can unquote raw
# expressions or quosures:
var <- quo(mean(cyl))
summarise(mtcars, !! var)
```

---

summarise_all *Summarise and mutate multiple columns.*

Description

These verbs are scoped variants of `summarise()`, `mutate()` and `transmute()`. They apply operations on a selection of variables.

- `summarise_all()`, `mutate_all()` and `transmute_all()` apply the functions to all (non-grouping) columns.
- `summarise_at()`, `mutate_at()` and `transmute_at()` allow you to select columns using the same name-based select_helpers just like with `select()`.
- `summarise_if()`, `mutate_if()` and `transmute_if()` operate on columns for which a predicate returns `TRUE`.

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)

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	List of function calls generated by funs(), or a character vector of function names, or simply a function. Bare formulas are passed to rlang::as_function() to create purrr-style lambda functions. Note that these lambda prevent hybrid evaluation from happening and it is thus more efficient to supply functions like mean() directly rather than in a lambda-formula.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with explicit splicing.
.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(), or a character vector of column names, or a numeric vector of column positions.
.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).

See Also

vars(), funs()

Examples

```
# The scoped variants of summarise() and mutate() make it easy to
# apply the same transformation to multiple variables:

iris %>%
  group_by(Species) %>%
  summarise_all(mean)

# 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:

starwars %>% summarise_at(vars(height:mass), mean, na.rm = TRUE)
starwars %>% summarise_at(c("height", "mass"), mean, na.rm = TRUE)
starwars %>% summarise_if(is.numeric, mean, na.rm = TRUE)

# mutate_if is particularly useful for transforming variables from
```

```
# one type to another
iris %>% as_tibble() %>% mutate_if(is.factor, as.character)
iris %>% as_tibble() %>% mutate_if(is.double, as.integer)

# ------------------------------------------------------------------------
# If you want apply multiple transformations, use funs()
by_species <- iris %>% group_by(Species)

by_species %>% summarise_all(funs(min, max))
# Note that output variable name now includes the function name, in order to
# keep things distinct.

# You can express more complex inline transformations using .
by_species %>% mutate_all(funs(. / 2.54))

# Function names will be included if .funs has names or multiple inputs
by_species %>% mutate_all(funs(cm = . / 2.54))
by_species %>% summarise_all(funs(med = median))
by_species %>% summarise_all(funs(Q3 = quantile), probs = 0.75)
by_species %>% summarise_all(c("min", "max"))
```

`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.

`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, sort = FALSE)

count(x, ..., wt = NULL, sort = FALSE)

add_tally(x, wt, sort = FALSE)

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

Arguments

`x`	a `tbl()` to tally/count.

wt	(Optional) If omitted, will count the number of rows. If specified, will perform a "weighted" tally by summing the (non-missing) values of variable wt. 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
...	Variables to group by.

Value

A tbl, grouped the same way as x.

Note

The column name in the returned data is usually n, even if you have supplied a weight.

If the data already already has a column named n, the output column will be called nn. If the table already has columns called n and nn then the column returned will be nnn, and so on.

There is currently no way to control the output variable name - if you need to change the default, you'll have to write the summarise() yourself.

Examples

```
# tally() is short-hand for mutate()
mtcars %>% tally()
# count() is a short-hand for group_by() + tally()
mtcars %>% count(cyl)

# 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)

# add_count() is useful for groupwise filtering
# e.g.: show only 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 experiement 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)
```

Arguments

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

Examples

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

# 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
# A little nicer with %>%
tbl_df(Batting) %>%
  group_by(playerID) %>%
  tally(G) %>%
  top_n(10)

# Find year with most games for each player
tbl_df(Batting) %>% group_by(playerID) %>% top_n(1, G)
}
```

`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 an integerish vector of positions or a character vector of column names.

See Also

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

Index