Package 'dplyr'

June 24, 2016

Type Package

Version 0.5.0

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 https://github.com/hadley/dplyr

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

Depends R (>= 3.1.2)

Imports assertthat, utils, R6, Rcpp (>= 0.12.3), tibble, magrittr, lazyeval (>= 0.1.10), DBI (>= 0.4.1)

Suggests RSQLite (>= 1.0.0), RMySQL, RPostgreSQL, testthat, knitr, microbenchmark, ggplot2, mgcv, Lahman (>= 3.0-1), nycflights13, methods, rmarkdown, covr, dtplyr

VignetteBuilder knitr

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

LazyData yes

License MIT + file LICENSE

Collate 'RcppExports.R' 'all-equal.r' 'bench-compare.r' 'bind.r' 'case_when.R' 'coalesce.R' 'colwise.R' 'compute-collect.r' 'copy-to.r' 'data-lahman.r' 'data-nasa.r' 'data-nycflights13.r' 'data-temp.r' 'data.r' 'dataframe.R' 'dbi-s3.r' 'desc.r' 'distinct.R' 'do.r' 'dplyr.r' 'explain.r' 'failwith.r' 'funs.R' 'group-by.r' 'group-indices.R' 'group-size.r' 'grouped-df.r' 'id.r' 'if_else.R' 'inline.r' 'join.r' 'lazy-ops.R' 'lead-lag.R' 'location.R' 'manip.r' 'na_if.R' 'near.R' 'nth-value.R' 'order-by.R' 'over.R' 'partial-eval.r' 'progress.R' 'query.r' 'rank.R' 'recode.R' 'rowwise.r' 'sample.R' 'select-utils.R' 'select-vars.R' 'sets.r' 'sql-build.R' 'sql-escape.r' 'sql-generic.R' 'sql-query.R' 'sql-render.R' 'sql-star.r' 'src-local.r' 'src-mysql.r' 'src-postgres.r' 'src-sql.r' 'src-sqlite.r' 'src-test.r'

'src.r' 'tally.R' 'tbl-cube.r' 'tbl-df.r' 'tbl-lazy.R'
'tbl-sql.r' 'tbl.r' 'tibble-reexport.r' 'top-n.R'
'translate-sql-helpers.r' 'translate-sql-base.r'
'translate-sql-window.r' 'translate-sql.r' 'utils-format.r'
'utils-replace-with.R' 'utils.r' 'view.r' 'zzz.r'

RoxygenNote 5.0.1

NeedsCompilation yes

Author Hadley Wickham [aut, cre],
Romain Francois [aut],
RStudio [cph]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2016-06-24 15:37:11

R topics documented:

add_rownames *Convert row names to an explicit variable.*

Description

Deprecated, use rownames_to_column instead.

Usage

```
add_rownames(df, var = "rowname")
```

Arguments

df	Input data frame with rownames.
var	Name of variable to use

Examples

```
mtcars %>% tbl_df()

mtcars %>% add_rownames()
```

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)
```

---

arrange *Arrange rows by variables.*

Description

Use desc to sort a variable in descending order. Generally, this will not also automatically order by grouping variables.

Usage

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

arrange_(.data, ..., .dots)
```

Arguments

.data	A tbl. All main verbs are S3 generics and provide methods for tbl_df, tbl_dt and tbl_sql.
...	Comma separated list of unquoted variable names. Use desc to sort a variable in descending order.
.dots	Used to work around non-standard evaluation. See vignette("nse") for details.

Value

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

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

Locales

Note that for local data frames, the ordering is done in C++ code which does not have access to the local specific ordering usually done in R. This means that strings are ordered as if in the C locale.

See Also

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

Examples

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

---

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 `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.

---

bench_compare *Evaluate, compare, benchmark operations of a set of srcs.*

Description

These functions support the comparison of results and timings across multiple sources.

Usage

```
bench_tbls(tbls, op, ..., times = 10)

compare_tbls(tbls, op, ref = NULL, compare = equal_data_frame, ...)

eval_tbls(tbls, op)
```

Arguments

tbls	A list of tbls.
op	A function with a single argument, called often with each element of tbls.
times	For benchmarking, the number of times each operation is repeated.
ref	For checking, an data frame to test results against. If not supplied, defaults to the results from the first src.
compare	A function used to compare the results. Defaults to equal_data_frame which ignores the order of rows and columns.
...	For compare_tbls: additional parameters passed on the compare function For bench_tbls: additional benchmarks to run.

Value

eval_tbls: a list of data frames.

compare_tbls: an invisible TRUE on success, otherwise an error is thrown.

bench_tbls: an object of class microbenchmark

See Also

src_local for working with local data

Examples

```
## Not run:
if (require("microbenchmark") && has_lahman()) {
lahman_local <- lahman_srcs("df", "sqlite")
teams <- lapply(lahman_local, function(x) x %>% tbl("Teams"))

compare_tbls(teams, function(x) x %>% filter(yearID == 2010))
bench_tbls(teams, function(x) x %>% filter(yearID == 2010))

# You can also supply arbitrary additional arguments to bench_tbls
# if there are other operations you'd like to compare.
bench_tbls(teams, function(x) x %>% filter(yearID == 2010),
   base = subset(Lahman::Teams, yearID == 2010))

# A more complicated example using multiple tables
setup <- function(src) {
  list(
    src %>% tbl("Batting") %>% filter(stint == 1) %>% select(playerID:H),
    src %>% tbl("Master") %>% select(playerID, birthYear)
  )
}
two_tables <- lapply(lahman_local, setup)

op <- function(tbls) {
  semi_join(tbls[[1]], tbls[[2]], by = "playerID")
}
# compare_tbls(two_tables, op)
bench_tbls(two_tables, op, times = 2)

}

## End(Not run)
```

`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 column-binding, rows are matched by position, not value so all data frames must have the same number of rows. To match by value, not position, see `left_join` etc. When row-binding, columns are matched by name, and any values that don't match will be filled with NA.
`.id`	Data frames 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.

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 either supply data frames as arguments
bind_rows(one, two)
# Or a single argument containing a list of data frames
bind_rows(list(one, two))
bind_rows(split(mtcars, mtcars$cyl))

# 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))
```

---

build_sql *Build a SQL string.*

Description

This is a convenience function that should prevent sql injection attacks (which in the context of dplyr are most likely to be accidental not deliberate) by automatically escaping all expressions in the input, while treating bare strings as sql. This is unlikely to prevent any serious attack, but should make it unlikely that you produce invalid sql.

Usage

```
build_sql(..., .env = parent.frame(), con = NULL)
```

Arguments

...	input to convert to SQL. Use sql to preserve user input as is (dangerous), and ident to label user input as sql identifiers (safe)
.env	the environment in which to evalute the arguments. Should not be needed in typical use.
con	database connection; used to select correct quoting characters.

Examples

```
build_sql("SELECT * FROM TABLE")
x <- "TABLE"
build_sql("SELECT * FROM ", x)
build_sql("SELECT * FROM ", ident(x))
build_sql("SELECT * FROM ", sql(x))

# http://xkcd.com/327/
name <- "Robert'); DROP TABLE Students;--"
build_sql("INSERT INTO Students (Name) VALUES (", name, ")")
```

---

case_when *A general vectorised if.*

Description

This function allows you to vectorise mutiple `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.

Value

A vector as long as the longest LHS, with the type (and attributes) of the first RHS. Inconsistent lengths of 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"
)
```

---

coalesce *Find first non-missing element*

Description

Given a set of vectors, `coelesce` 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(x, ...)
```

Arguments

`x, ...`	Vectors. All inputs should either be length 1, or the same length as `x`

Value

A vector the same length as `x` 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)
```

`compute` *Compute a lazy tbl.*

Description

`compute` forces computation of lazy tbls, leaving data in the remote source. `collect` also forces computation, but will bring data back into an R data.frame (stored in a `tbl_df`). `collapse` doesn't force computation, but collapses a complex tbl into a form that additional restrictions can be placed on.

Usage

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

collect(x, ...)

collapse(x, ...)

## S3 method for class 'tbl_sql'
compute(x, name = random_table_name(), temporary = TRUE,
  unique_indexes = list(), indexes = list(), ...)
```

Arguments

`x`	a data tbl
`name`	name of temporary table on database.
`...`	other arguments passed on to methods
`temporary`	if `TRUE`, will create a temporary table that is local to this connection and will be automatically deleted when the connection expires
`unique_indexes`	a list of character vectors. Each element of the list will create a new unique index over the specified column(s). Duplicate rows will result in failure.
`indexes`	a list of character vectors. Each element of the list will create a new index.

Grouping

`compute` and `collect` preserve grouping, `collapse` drops it.

See Also

`copy_to` which is the conceptual opposite: it takes a local data frame and makes it available to the remote source.

Examples

```
if (require("RSQLite") && has_lahman("sqlite")) {
  batting <- tbl(lahman_sqlite(), "Batting")
  remote <- select(filter(batting, yearID > 2010 && stint == 1), playerID:H)
  remote2 <- collapse(remote)
  cached <- compute(remote)
  local  <- collect(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)), ...)
```

Arguments

`dest`	remote data source
`df`	local data frame
`name`	name for new remote table.
`...`	other parameters passed to methods.

Value

a `tbl` object in the remote source

`copy_to.src_sql` *Copy a local data frame to a sqlite src.*

Description

This standard method works for all sql sources.

Usage

```
## S3 method for class 'src_sql'
copy_to(dest, df, name = deparse(substitute(df)),
  types = NULL, temporary = TRUE, unique_indexes = NULL, indexes = NULL,
  analyze = TRUE, ...)
```

Arguments

`dest`	remote data source
`df`	local data frame
`name`	name for new remote table.
`types`	a character vector giving variable types to use for the columns. See http://www.sqlite.org/datatype3.html for available types.
`temporary`	if `TRUE`, will create a temporary table that is local to this connection and will be automatically deleted when the connection expires
`unique_indexes`	a list of character vectors. Each element of the list will create a new unique index over the specified column(s). Duplicate rows will result in failure.
`indexes`	a list of character vectors. Each element of the list will create a new index.
`analyze`	if `TRUE` (the default), will automatically ANALYZE the new table so that the query optimiser has useful information.
`...`	other parameters passed to methods.

Value

a sqlite `tbl` object

Examples

```
if (requireNamespace("RSQLite")) {
db <- src_sqlite(tempfile(), create = TRUE)

iris2 <- copy_to(db, iris)
mtcars$model <- rownames(mtcars)
mtcars2 <- copy_to(db, mtcars, indexes = list("model"))

explain(filter(mtcars2, model == "Hornet 4 Drive"))

# Note that tables are temporary by default, so they're not
```

```
# visible from other connections to the same database.
src_tbls(db)
db2 <- src_sqlite(db$path)
src_tbls(db2)
}
```

`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` & `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.

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)
```

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)

distinct_(.data, ..., .dots, .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.
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.

Examples

```
df <- data.frame(
  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))
```

do *Do arbitrary operations on a tbl.*

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, ...)

do_(.data, ..., .dots)

## S3 method for class 'tbl_sql'
do_(.data, ..., .dots, .chunk_size = 10000L)
```

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.
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.
`.chunk_size`	The size of each chunk to pull into R. If this number is too big, the process will be slow because R has to allocate and free a lot of memory. If it's too small, it will be slow, because of the overhead of talking to the database.

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 dlply, and do with a single unnamed argument is basically equivalent to 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)
}
```

dplyr *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")`

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

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("RSQLite") && has_lahman("sqlite")) {

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()
}
```

---

failwith *Fail with specified value.*

Description

Modify a function so that it returns a default value when there is an error.

Usage

```
failwith(default = NULL, f, quiet = FALSE)
```

Arguments

default	default value
f	function
quiet	all error messages be suppressed?

Value

a function

See Also

try_default

Examples

```
f <- function(x) if (x == 1) stop("Error!") else 1
## Not run:
f(1)
f(2)

## End(Not run)

safef <- failwith(NULL, f)
safef(1)
safef(2)
```

`filter` *Return rows with matching conditions.*

Description

Return rows with matching conditions.

Usage

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

filter_(.data, ..., .dots)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df`, `tbl_dt` and `tbl_sql`.
`...`	Logical predicates. Multiple conditions are combined with `&`.
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.

Value

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

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

See Also

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

Examples

```
filter(mtcars, cyl == 8)
filter(mtcars, cyl < 6)

# Multiple criteria
filter(mtcars, cyl < 6 & vs == 1)
filter(mtcars, cyl < 6 | vs == 1)

# Multiple arguments are equivalent to and
filter(mtcars, cyl < 6, vs == 1)
```

---

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_each.

Usage

```
funs(...)

funs_(dots, args = list(), env = baseenv())
```

Arguments

dots, ...	A list of functions specified by: • Their name, "mean" • The function itself, mean • A call to the function with . as a dummy parameter, mean(., na.rm = TRUE)
args	A named list of additional arguments to be added to all function calls.
env	The environment in which functions should be evaluated.

Examples

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

# Overide 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 *Get/set the grouping variables for tbl.*

Description

These functions do not perform non-standard evaluation, and so are useful when programming against `tbl` objects. `ungroup` is a convenient inline way of removing existing grouping.

Usage

```
groups(x)

ungroup(x, ...)
```

Arguments

`x`	data `tbl`
`...`	Additional arguments that maybe used by methods.

Examples

```
grouped <- group_by(mtcars, cyl)
groups(grouped)
groups(ungroup(grouped))
```

group_by *Group a tbl by one or more variables.*

Description

Most data operations are useful done on groups defined by variables in the the dataset. The `group_by` function takes an existing tbl and converts it into a grouped tbl where operations are performed "by group".

Usage

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

group_by_(.data, ..., .dots, add = FALSE)
```

Arguments

.data	a tbl
...	variables to group by. All tbls accept variable names, some will also accept functions of variables. Duplicated groups will be silently dropped.
add	By default, when add = FALSE, group_by will override existing groups. To instead add to the existing groups, use add = TRUE
.dots	Used to work around non-standard evaluation. See vignette("nse") for details.

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: grouped_dt
- SQLite: src_sqlite
- PostgreSQL: src_postgres
- MySQL: src_mysql

See Also

ungroup for the inverse operation, groups for accessors that don't do special evaluation.

Examples

```
by_cyl <- group_by(mtcars, cyl)
summarise(by_cyl, mean(disp), mean(hp))
filter(by_cyl, disp == max(disp))

# summarise peels off a single layer of grouping
by_vs_am <- group_by(mtcars, vs, am)
by_vs <- summarise(by_vs_am, n = n())
by_vs
summarise(by_vs, n = sum(n))
# use ungroup() to remove if not wanted
summarise(ungroup(by_vs), n = sum(n))

# You can group by expressions: this is just short-hand for
# a mutate/rename followed by a simple group_by
group_by(mtcars, vsam = vs + am)
group_by(mtcars, vs2 = vs)

# You can also group by a constant, but it's not very useful
group_by(mtcars, "vs")

# By default, group_by sets groups. Use add = TRUE to add groups
groups(group_by(by_cyl, vs, am))
```

```
groups(group_by(by_cyl, vs, am, add = TRUE))

# Duplicate groups are silently dropped
groups(group_by(by_cyl, cyl, cyl))
```

`group_indices` *Group id.*

Description

Generate a unique id for each group

Usage

```
group_indices(.data, ...)

group_indices_(.data, ..., .dots)
```

Arguments

`.data`	a tbl
`...`	variables to group by. All tbls accept variable names, some will also accept functions of variables. Duplicated groups will be silently dropped.
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.

See Also

`group_by`

Examples

```
group_indices(mtcars, cyl)
```

`group_size` *Calculate group sizes.*

Description

Calculate group sizes.

Usage

```
group_size(x)

n_groups(x)
```

Arguments

x	a grouped tbl

Examples

```
if (require("nycflights13")) {

by_day <- flights %>% group_by(year, month, day)
n_groups(by_day)
group_size(by_day)

by_dest <- flights %>% group_by(dest)
n_groups(by_dest)
group_size(by_dest)
}
```

---

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` - this may be an expensive operation.

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.
...	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.

---

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"), ...)

## S3 method for class 'tbl_df'
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x",
  ".y"), ...)
```

```
## S3 method for class 'tbl_df'
right_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...)

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

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

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

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.
...	included for compatibility with the generic; otherwise ignored.

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)
}
```

`join.tbl_sql` *Join sql tbls.*

Description

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

Usage

```
## S3 method for class 'tbl_lazy'
inner_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), auto_index = FALSE, ...)

## S3 method for class 'tbl_lazy'
left_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), auto_index = FALSE, ...)

## S3 method for class 'tbl_lazy'
right_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), auto_index = FALSE, ...)

## S3 method for class 'tbl_lazy'
full_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), auto_index = FALSE, ...)

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

## S3 method for class 'tbl_lazy'
anti_join(x, y, by = NULL, copy = FALSE,
  auto_index = FALSE, ...)
```

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 a temporary table in same database as x. join will automatically run ANALYZE on the created table in the hope that this will make you queries as efficient as possible by giving more data to the query planner. This allows you to join tables across srcs, but it's 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.
auto_index	if copy is TRUE, automatically create indices for the variables in by. This may speed up the join if there are matching indexes in x.
...	other parameters passed onto methods

Implementation notes

Semi-joins are implemented using WHERE EXISTS, and anti-joins with WHERE NOT EXISTS. Support for semi-joins is somewhat partial: you can only create semi joins where the x and y columns are compared with = not with more general operators.

Examples

```
## Not run:
if (require("RSQLite") && has_lahman("sqlite")) {

# Left joins ------------------------------------------------------------------
lahman_s <- lahman_sqlite()
batting <- tbl(lahman_s, "Batting")
team_info <- select(tbl(lahman_s, "Teams"), yearID, lgID, teamID, G, R:H)

# Combine player and whole team statistics
first_stint <- select(filter(batting, stint == 1), playerID:H)
both <- left_join(first_stint, team_info, type = "inner", by = c("yearID", "teamID", "lgID"))
head(both)
explain(both)

# Join with a local data frame
grid <- expand.grid(
  teamID = c("WAS", "ATL", "PHI", "NYA"),
  yearID = 2010:2012)
top4a <- left_join(batting, grid, copy = TRUE)
explain(top4a)

# Indices don't really help here because there's no matching index on
# batting
top4b <- left_join(batting, grid, copy = TRUE, auto_index = TRUE)
explain(top4b)

# Semi-joins ------------------------------------------------------------------

people <- tbl(lahman_s, "Master")
```

```
# All people in half of fame
hof <- tbl(lahman_s, "HallOfFame")
semi_join(people, hof)

# All people not in the hall of fame
anti_join(people, hof)

# Find all managers
manager <- tbl(lahman_s, "Managers")
semi_join(people, manager)

# Find all managers in hall of fame
famous_manager <- semi_join(semi_join(people, manager), hof)
famous_manager
explain(famous_manager)

# Anti-joins -----------------------------------------------------------------

# batters without person covariates
anti_join(batting, people)
}

## End(Not run)
```

`lead-lag` *Lead and lag.*

Description

Lead and lag are useful for comparing values offset by a constant (e.g. the previous or next value)

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)
```

---

location *Print the location in memory of a data frame*

Description

This is useful for understand how and when dplyr makes copies of data frames

Usage

```
location(df)

changes(x, y)
```

Arguments

df	a data frame
x, y	two data frames to compare

Examples

```
location(mtcars)

mtcars2 <- mutate(mtcars, cyl2 = cyl * 2)
location(mtcars2)

changes(mtcars, mtcars)
changes(mtcars, mtcars2)
```

`mutate` *Add new variables.*

Description

Mutate adds new variables and preserves existing; transmute drops existing variables.

Usage

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

mutate_(.data, ..., .dots)

transmute(.data, ...)

transmute_(.data, ..., .dots)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df`, `tbl_dt` and `tbl_sql`.
`...`	Name-value pairs of expressions. Use `NULL` to drop a variable.
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.

Value

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

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

See Also

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

Examples

```
mutate(mtcars, displ_l = disp / 61.0237)
transmute(mtcars, displ_l = disp / 61.0237)

mutate(mtcars, cyl = NULL)
```

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

Description

This function is implemented special 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	If th

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 atomic vectors, where a missing value of the appropriate type is return, and for lists, where a NULL is return. For more complicated objects, you'll need to supply this value.

Value

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

Examples

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

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

last(x)
last(x, y)
```

---

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

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

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 is a useful function to control the order 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)
```

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)
```

---

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`

Usage

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

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

Arguments

`.x`	A vector to modify
`...`	Replacments. These should be named for character and factor `.x`, and can be named for numeric `.x`. All replacements must be the same type, and must have either length one or the same length as x.
`.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.

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's 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 = parent.frame())

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

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 expression is evaluated in the context of the data frame. It must return a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.
.env	Environment in which to look for non-data names used in weight. Non-default settings for experts only.

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)
```

---

select *Select/rename variables by name.*

Description

`select()` keeps only the variables you mention; `rename()` keeps all variables.

Usage

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

select_(.data, ..., .dots)

rename(.data, ...)

rename_(.data, ..., .dots)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df`, `tbl_dt` and `tbl_sql`.
`...`	Comma separated list of unquoted expressions. You can treat variable names like they are positions. Use positive values to select variables; use negative values to drop variables.
`.dots`	Use `select_()` to do standard evaluation. See `vignette("nse")` for details

Value

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

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

Special functions

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

To drop variables, use `-`. You can rename variables with named arguments.

See Also

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

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)
vars <- c("Petal.Length", "Petal.Width")
select(iris, one_of(vars))

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
select(iris, -starts_with("Petal"))
select(iris, -ends_with("Width"))
select(iris, -contains("etal"))
select(iris, -matches(".t."))
select(iris, -Petal.Length, -Petal.Width)

# Rename variables:
# * select() keeps only the variables you specify
select(iris, petal_length = Petal.Length)
# Renaming multiple variables uses a prefix:
select(iris, petal = starts_with("Petal"))

# Reorder variables: keep the variable "Species" in the front
select(iris, Species, everything())

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

# Programming with select ----------------------------------------------------
select_(iris, ~Petal.Length)
```

```
select_(iris, "Petal.Length")
select_(iris, lazyeval::interp(~matches(x), x = ".t."))
select_(iris, quote(-Petal.Length), quote(-Petal.Width))
select_(iris, .dots = list(quote(-Petal.Length), quote(-Petal.Width)))
```

`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 given 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))
```

---

select_if *Select columns using a predicate*

Description

This verb is analogous to summarise_if() and mutate_if() in that it lets you use a predicate on the columns of a data frame. Only those columns for which the predicate returns TRUE will be selected.

Usage

```
select_if(.data, .predicate, ...)
```

Arguments

.data	A local tbl source.
.predicate	A predicate function to be applied to the columns or a logical vector. The columns for which .predicate is or returns TRUE will be summarised or mutated.
...	Additional arguments passed to .predicate.

Details

Predicates can only be used with local sources like a data frame.

Examples

```
iris %>% select_if(is.factor)
iris %>% select_if(is.numeric)
iris %>% select_if(function(col) is.numeric(col) && mean(col) > 3.5)
```

---

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, ...)

slice_(.data, ..., .dots)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df`, `tbl_dt` and `tbl_sql`.
`...`	Integer row values
`.dots`	Used to work around non-standard evaluation. See `vignette("nse")` for details.

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()))
```

src-test *A set of DBI methods to ease unit testing dplyr with DBI*

Description

A set of DBI methods to ease unit testing dplyr with DBI

Usage

```
## S3 method for class 'DBITestConnection'
db_query_fields(con, sql, ...)

## S3 method for class 'DBITestConnection'
sql_escape_ident(con, x)

## S3 method for class 'DBITestConnection'
sql_translate_env(con)
```

Arguments

con	A database connection.
sql	A string containing an sql query.
...	Other arguments passed on to the individual methods
x	Object to transform

src_memdb *Per-session in-memory SQLite databases.*

Description

src_memdb lets you easily access a sessio-temporary in-memory SQLite database. memdb_frame() works like data_frame, but instead of creating a new data frame in R, it creates a table in src_memdb

Usage

```
src_memdb()

memdb_frame(..., .name = random_table_name())
```

Arguments

...	A set of name-value pairs. Arguments are evaluated sequentially, so you can refer to previously created variables.
.name	Name of table in database: defaults to a random name that's unlikely to conflict with exist

Examples

```
if (require("RSQLite")) {
src_memdb()

df <- memdb_frame(x = runif(100), y = runif(100))
df %>% arrange(x)
df %>% arrange(x) %>% show_query()
}
```

`src_mysql` *Connect to mysql/mariadb.*

Description

Use `src_mysql` to connect to an existing mysql or mariadb database, and `tbl` to connect to tables within that database. If you are running a local mysqlql database, leave all parameters set as their defaults to connect. If you're connecting to a remote database, ask your database administrator for the values of these variables.

Usage

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

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

Arguments

`dbname`	Database name
`host, port`	Host name and port number of database
`user, password`	User name and password. Rather than supplying a username and password here, it's better to save them in `my.cnf`, as described in `MySQL`. In that case, supply `NULL` to both `user` and `password`.
`...`	for the src, other arguments passed on to the underlying database connector, `dbConnect`. For the tbl, included for compatibility with the generic, but otherwise ignored.
`src`	a mysql src created with `src_mysql`.
`from`	Either a string giving the name of table in database, or `sql` described a derived table or compound join.

Debugging

To see exactly what SQL is being sent to the database, you see `show_query` and `explain`.

Grouping

Typically you will create a grouped data table is to call the `group_by` method on a mysql tbl: this will take care of capturing the unevalated expressions for you.

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.

Output

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_sql` 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.

Note that do is not lazy since it must pull the data into R. It returns a `tbl_df` or `grouped_df`, with one column for each grouping variable, and one list column that contains the results of the operation. do never simplifies its output.

Query principles

This section attempts to lay out the principles governing the generation of SQL queries from the manipulation verbs. The basic principle is that a sequence of operations should return the same value (modulo class) regardless of where the data is stored.

- `arrange(arrange(df, x), y)` should be equivalent to `arrange(df, y, x)`
- `select(select(df, a:x), n:o)` should be equivalent to `select(df, n:o)`
- `mutate(mutate(df, x2 = x * 2), y2 = y * 2)` should be equivalent to `mutate(df, x2 = x * 2, y2 = y * 2)`
- `filter(filter(df, x == 1), y == 2)` should be equivalent to `filter(df, x == 1, y == 2)`
- `summarise` should return the summarised output with one level of grouping peeled off.

Examples

```
## Not run:
# Connection basics ---------------------------------------------------------
# To connect to a database first create a src:
my_db <- src_mysql(host = "blah.com", user = "hadley",
  password = "pass")
# Then reference a tbl within that src
my_tbl <- tbl(my_db, "my_table")

## End(Not run)

# Here we'll use the Lahman database: to create your own local copy,
# create a local database called "lahman", or tell lahman_mysql() how to
# a database that you can write to

if (!has_lahman("postgres") && has_lahman("mysql")) {
lahman_m <- lahman_mysql()
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_m, "Batting")
dim(batting)
colnames(batting)
```

```
head(batting)

# Data manipulation verbs ------------------------------------------------
filter(batting, yearID > 2005, G > 130)
select(batting, playerID:lgID)
arrange(batting, playerID, desc(yearID))
summarise(batting, G = mean(G), n = n())
mutate(batting, rbi2 = 1.0 * R / AB)

# note that 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), by looking at the `head()`, or `collect()` the results locally.

system.time(recent <- filter(batting, yearID > 2010))
system.time(collect(recent))

# Group by operations ----------------------------------------------------
# To perform operations by group, create a grouped object with group_by
players <- group_by(batting, playerID)
group_size(players)

# MySQL doesn't support windowed functions, which means that only
# grouped summaries are really useful:
summarise(players, mean_g = mean(G), best_ab = max(AB))

# When you group by multiple level, each summarise peels off one level
per_year <- group_by(batting, playerID, yearID)
stints <- summarise(per_year, stints = max(stint))
filter(ungroup(stints), stints > 3)
summarise(stints, max(stints))

# Joins ------------------------------------------------------------------
player_info <- select(tbl(lahman_m, "Master"), playerID,
  birthYear)
hof <- select(filter(tbl(lahman_m, "HallOfFame"), inducted == "Y"),
 playerID, votedBy, category)

# Match players and their hall of fame data
inner_join(player_info, hof)
# Keep all players, match hof data where available
left_join(player_info, hof)
# Find only players in hof
semi_join(player_info, hof)
# Find players not in hof
anti_join(player_info, hof)

# Arbitrary SQL ----------------------------------------------------------
# You can also provide sql as is, using the sql function:
batting2008 <- tbl(lahman_m,
  sql("SELECT * FROM Batting WHERE YearID = 2008"))
batting2008
}
```

`src_postgres` *Connect to postgresql.*

Description

Use `src_postgres` to connect to an existing postgresql database, and `tbl` to connect to tables within that database. If you are running a local postgresql database, leave all parameters set as their defaults to connect. If you're connecting to a remote database, ask your database administrator for the values of these variables.

Usage

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

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

Arguments

`dbname`	Database name
`host, port`	Host name and port number of database
`user, password`	User name and password (if needed)
`...`	for the src, other arguments passed on to the underlying database connector, `dbConnect`. For the tbl, included for compatibility with the generic, but otherwise ignored.
`src`	a postgres src created with `src_postgres`.
`from`	Either a string giving the name of table in database, or `sql` described a derived table or compound join.

Debugging

To see exactly what SQL is being sent to the database, you see `show_query` and `explain`.

Grouping

Typically you will create a grouped data table is to call the `group_by` method on a mysql tbl: this will take care of capturing the unevalated expressions for you.

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.

Output

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_sql` 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.

Note that do is not lazy since it must pull the data into R. It returns a `tbl_df` or `grouped_df`, with one column for each grouping variable, and one list column that contains the results of the operation. do never simplifies its output.

Query principles

This section attempts to lay out the principles governing the generation of SQL queries from the manipulation verbs. The basic principle is that a sequence of operations should return the same value (modulo class) regardless of where the data is stored.

- `arrange(arrange(df, x), y)` should be equivalent to `arrange(df, y, x)`
- `select(select(df, a:x), n:o)` should be equivalent to `select(df, n:o)`
- `mutate(mutate(df, x2 = x * 2), y2 = y * 2)` should be equivalent to `mutate(df, x2 = x * 2, y2 = y * 2)`
- `filter(filter(df, x == 1), y == 2)` should be equivalent to `filter(df, x == 1, y == 2)`
- `summarise` should return the summarised output with one level of grouping peeled off.

Examples

```
## Not run:
# Connection basics ---------------------------------------------------------
# To connect to a database first create a src:
my_db <- src_postgres(host = "blah.com", user = "hadley",
  password = "pass")
# Then reference a tbl within that src
my_tbl <- tbl(my_db, "my_table")

## End(Not run)

# Here we'll use the Lahman database: to create your own local copy,
# create a local database called "lahman", or tell lahman_postgres() how to
# access a database that you can write to

if (has_lahman("postgres")) {
lahman_p <- lahman_postgres()
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_p, "Batting")
dim(batting)
colnames(batting)
head(batting)

# Data manipulation verbs ---------------------------------------------------
filter(batting, yearID > 2005, G > 130)
select(batting, playerID:lgID)
arrange(batting, playerID, desc(yearID))
summarise(batting, G = mean(G), n = n())
```

```
mutate(batting, rbi2 = if(is.null(AB)) 1.0 * R / AB else 0)

# note that 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), by looking at the `head()`, or `collect()` the results locally.

system.time(recent <- filter(batting, yearID > 2010))
system.time(collect(recent))

# Group by operations ----------------------------------------------------
# To perform operations by group, create a grouped object with group_by
players <- group_by(batting, playerID)
group_size(players)

summarise(players, mean_g = mean(G), best_ab = max(AB))
best_year <- filter(players, AB == max(AB) | G == max(G))
best_year

progress <- mutate(players,
  cyear = yearID - min(yearID) + 1,
  ab_rank = rank(desc(AB)),
  cumulative_ab = order_by(yearID, cumsum(AB)))

# When you group by multiple level, each summarise peels off one level
per_year <- group_by(batting, playerID, yearID)
stints <- summarise(per_year, stints = max(stint))
filter(stints, stints > 3)
summarise(stints, max(stints))
mutate(stints, order_by(yearID, cumsum(stints)))

# Joins ------------------------------------------------------------------
player_info <- select(tbl(lahman_p, "Master"), playerID, birthYear)
hof <- select(filter(tbl(lahman_p, "HallOfFame"), inducted == "Y"),
 playerID, votedBy, category)

# Match players and their hall of fame data
inner_join(player_info, hof)
# Keep all players, match hof data where available
left_join(player_info, hof)
# Find only players in hof
semi_join(player_info, hof)
# Find players not in hof
anti_join(player_info, hof)

# Arbitrary SQL ----------------------------------------------------------
# You can also provide sql as is, using the sql function:
batting2008 <- tbl(lahman_p,
  sql('SELECT * FROM "Batting" WHERE "yearID" = 2008'))
batting2008
}
```

src_sqlite *Connect to a sqlite database.*

Description

Use src_sqlite to connect to an existing sqlite database, and tbl to connect to tables within that database. If you are running a local sqliteql database, leave all parameters set as their defaults to connect. If you're connecting to a remote database, ask your database administrator for the values of these variables. src_memdb is an easy way to use an in-memory SQLite database that is scoped to the current session.

Usage

```
src_sqlite(path, create = FALSE)

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

Arguments

path	Path to SQLite 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.
src	a sqlite src created with src_sqlite.
from	Either a string giving the name of table in database, or sql described a derived table or compound join.
...	Included for compatibility with the generic, but otherwise ignored.

Debugging

To see exactly what SQL is being sent to the database, you see show_query and explain.

Grouping

Typically you will create a grouped data table is to call the group_by method on a mysql tbl: this will take care of capturing the unevalated expressions for you.

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.

Output

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_sql 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.

Note that do is not lazy since it must pull the data into R. It returns a tbl_df or grouped_df, with one column for each grouping variable, and one list column that contains the results of the operation. do never simplifies its output.

Query principles

This section attempts to lay out the principles governing the generation of SQL queries from the manipulation verbs. The basic principle is that a sequence of operations should return the same value (modulo class) regardless of where the data is stored.

- `arrange(arrange(df, x), y)` should be equivalent to `arrange(df, y, x)`
- `select(select(df, a:x), n:o)` should be equivalent to `select(df, n:o)`
- `mutate(mutate(df, x2 = x * 2), y2 = y * 2)` should be equivalent to `mutate(df, x2 = x * 2, y2 = y * 2)`
- `filter(filter(df, x == 1), y == 2)` should be equivalent to `filter(df, x == 1, y == 2)`
- `summarise` should return the summarised output with one level of grouping peeled off.

Examples

```
## Not run:
# Connection basics ---------------------------------------------------------
# To connect to a database first create a src:
my_db <- src_sqlite(path = tempfile(), create = TRUE)
# Then reference a tbl within that src
my_tbl <- tbl(my_db, "my_table")

## End(Not run)

# Here we'll use the Lahman database: to create your own local copy,
# run lahman_sqlite()

## Not run:
if (requireNamespace("RSQLite") && has_lahman("sqlite")) {
lahman_s <- lahman_sqlite()
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_s, "Batting")
dim(batting)
colnames(batting)
head(batting)

# Data manipulation verbs ---------------------------------------------------
filter(batting, yearID > 2005, G > 130)
select(batting, playerID:lgID)
arrange(batting, playerID, desc(yearID))
summarise(batting, G = mean(G), n = n())
mutate(batting, rbi2 = 1.0 * R / AB)

# note that 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), by looking at the `head()`, or `collect()` the results locally.

system.time(recent <- filter(batting, yearID > 2010))
```

```
system.time(collect(recent))

# Group by operations ----------------------------------------------------------
# To perform operations by group, create a grouped object with group_by
players <- group_by(batting, playerID)
group_size(players)

# sqlite doesn't support windowed functions, which means that only
# grouped summaries are really useful:
summarise(players, mean_g = mean(G), best_ab = max(AB))

# When you group by multiple level, each summarise peels off one level
per_year <- group_by(batting, playerID, yearID)
stints <- summarise(per_year, stints = max(stint))
filter(ungroup(stints), stints > 3)
summarise(stints, max(stints))

# Joins ------------------------------------------------------------------------
player_info <- select(tbl(lahman_s, "Master"), playerID, birthYear)
hof <- select(filter(tbl(lahman_s, "HallOfFame"), inducted == "Y"),
 playerID, votedBy, category)

# Match players and their hall of fame data
inner_join(player_info, hof)
# Keep all players, match hof data where available
left_join(player_info, hof)
# Find only players in hof
semi_join(player_info, hof)
# Find players not in hof
anti_join(player_info, hof)

# Arbitrary SQL ----------------------------------------------------------------
# You can also provide sql as is, using the sql function:
batting2008 <- tbl(lahman_s,
  sql("SELECT * FROM Batting WHERE YearID = 2008"))
batting2008
}

## End(Not run)
```

`src_tbls` *List all tbls provided by a source.*

Description

This is a generic method which individual src's will provide methods for. Most methods will not be documented because it's usually pretty obvious what possible results will be.

Usage

```
src_tbls(x)
```

Arguments

x	a data src.

`summarise` *Summarise multiple values to a single value.*

Description

Summarise multiple values to a single value.

Usage

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

summarise_(.data, ..., .dots)

summarize(.data, ...)

summarize_(.data, ..., .dots)
```

Arguments

.data	A tbl. All main verbs are S3 generics and provide methods for `tbl_df`, `tbl_dt` and `tbl_sql`.
...	Name-value pairs of summary functions like `min()`, `mean()`, `max()` etc.
.dots	Used to work around non-standard evaluation. See `vignette("nse")` for details.

Value

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

Data frame row names are silently dropped. To preserve, convert to an explicit variable.

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.

See Also

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

Examples

```
summarise(mtcars, mean(disp))
summarise(group_by(mtcars, cyl), mean(disp))
summarise(group_by(mtcars, cyl), m = mean(disp), sd = sd(disp))

# With data frames, you can create and immediately use summaries
by_cyl <- mtcars %>% group_by(cyl)
by_cyl %>% summarise(a = n(), b = a + 1)

## Not run:
# You can't with data tables or databases
by_cyl_dt <- mtcars %>% dtplyr::tbl_dt() %>% group_by(cyl)
by_cyl_dt %>% summarise(a = n(), b = a + 1)

by_cyl_db <- src_sqlite(":memory:", create = TRUE) %>%
  copy_to(mtcars) %>% group_by(cyl)
by_cyl_db %>% summarise(a = n(), b = a + 1)

## End(Not run)
```

---

summarise_all *Summarise and mutate multiple columns.*

Description

`summarise_all()` and `mutate_all()` apply the functions to all (non-grouping) columns. `summarise_at()` and `mutate_at()` allow you to select columns using the same name-based `select_helpers` as with `select()`. `summarise_if()` and `mutate_if()` operate on columns for which a predicate returns `TRUE`. Finally, `summarise_each()` and `mutate_each()` are older variants that will be deprecated in the future.

Usage

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

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

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

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

summarise_at(.tbl, .cols, .funs, ...)

mutate_at(.tbl, .cols, .funs, ...)

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

summarize_at(.tbl, .cols, .funs, ...)
```

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

Arguments

.tbl	a tbl
.funs	List of function calls generated by funs(), or a character vector of function names, or simply a function (only for local sources).
...	Additional arguments for the function calls. These are evaluated only once.
.predicate	A predicate function to be applied to the columns or a logical vector. The columns for which .predicate is or returns TRUE will be summarised or mutated.
.cols	A list of columns generated by vars(), or a character vector of column names, or a numeric vector of column positions.

Value

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

See Also

vars(), funs()

Examples

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

# One function
by_species %>% summarise_all(n_distinct)
by_species %>% summarise_all(mean)

# Use the _at and _if variants for conditional mapping.
by_species %>% summarise_if(is.numeric, mean)

# summarise_at() can use select() helpers with the vars() function:
by_species %>% summarise_at(vars(Petal.Width), mean)
by_species %>% summarise_at(vars(matches("Width")), mean)

# You can also specify columns with column names or column positions:
by_species %>% summarise_at(c("Sepal.Width", "Petal.Width"), mean)
by_species %>% summarise_at(c(1, 3), mean)

# You can provide additional arguments. Those are evaluated only once:
by_species %>% summarise_all(mean, trim = 1)
by_species %>% summarise_at(vars(Petal.Width), mean, trim = 1)

# You can provide an expression or multiple functions with the funs() helper.
by_species %>% mutate_all(funs(. * 0.4))
```

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

# Function names will be included if .funs has names or whenever multiple
# functions are used.
by_species %>% mutate_all(funs("in" = . / 2.54))
by_species %>% mutate_all(funs(rg = diff(range(.))))
by_species %>% summarise_all(funs(med = median))
by_species %>% summarise_all(funs(Q3 = quantile), probs = 0.75)
by_species %>% summarise_all(c("min", "max"))

# Two functions, continued
by_species %>% summarise_at(vars(Petal.Width, Sepal.Width), funs(min, max))
by_species %>% summarise_at(vars(matches("Width")), funs(min, max))
```

`summarise_each` *Summarise and mutate multiple columns.*

Description

Apply one or more functions to one or more columns. Grouping variables are always excluded from modification.

Usage

```
summarise_each(tbl, funs, ...)

summarise_each_(tbl, funs, vars)

summarize_each(tbl, funs, ...)

summarize_each_(tbl, funs, vars)

mutate_each(tbl, funs, ...)

mutate_each_(tbl, funs, vars)
```

Arguments

`tbl`	a tbl
`funs`	List of function calls, generated by `funs`, or a character vector of function names.
`vars, ...`	Variables to include/exclude in mutate/summarise. You can use same specifications as in `select`. If missing, defaults to all non-grouping variables. For standard evaluation versions (ending in _) these can be either a list of expressions or a character vector.

Details

In the future mutate_each() and summarise_each() will be deprecated in favour of a more featureful family of functions: mutate_all(), mutate_at(), mutate_if(), summarise_all(), summarise_at() and summarise_if().

---

tally *Counts/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 also does the group_by for you.

Usage

```
tally(x, wt, sort = FALSE)

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

count_(x, vars, 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.
sort	if TRUE will sort output in descending order of n
..., vars	Variables to group by.

Examples

```
if (require("Lahman")) {
batting_tbl <- tbl_df(Batting)
tally(group_by(batting_tbl, yearID))
tally(group_by(batting_tbl, yearID), sort = TRUE)

# Multiple tallys progressively roll up the groups
plays_by_year <- tally(group_by(batting_tbl, playerID, stint), sort = TRUE)
tally(plays_by_year, sort = TRUE)
tally(tally(plays_by_year))

# This looks a little nicer if you use the infix %>% operator
batting_tbl %>% group_by(playerID) %>% tally(sort = TRUE)

# count is even more succinct - it also does the grouping for you
batting_tbl %>% count(playerID)
```

```
batting_tbl %>% count(playerID, wt = G)
batting_tbl %>% count(playerID, wt = G, sort = TRUE)
}
```

`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))
```

`tbl_df` *Create a data frame tbl.*

Description

Forwards the argument to `as_data_frame`, see tibble-package for more details.

Usage

```
tbl_df(data)
```

Arguments

`data`	a data frame

Examples

```
ds <- tbl_df(mtcars)
ds
as.data.frame(ds)

if (require("Lahman") && packageVersion("Lahman") >= "3.0.1") {
batting <- tbl_df(Batting)
dim(batting)
colnames(batting)
head(batting)
```

```
# Data manipulation verbs ---------------------------------------------------
filter(batting, yearID > 2005, G > 130)
select(batting, playerID:lgID)
arrange(batting, playerID, desc(yearID))
summarise(batting, G = mean(G), n = n())
mutate(batting, rbi2 = if(is.null(AB)) 1.0 * R / AB else 0)

# Group by operations -------------------------------------------------------
# To perform operations by group, create a grouped object with group_by
players <- group_by(batting, playerID)
head(group_size(players), 100)

summarise(players, mean_g = mean(G), best_ab = max(AB))
best_year <- filter(players, AB == max(AB) | G == max(G))
progress <- mutate(players, cyear = yearID - min(yearID) + 1,
 rank(desc(AB)), cumsum(AB))

# When you group by multiple level, each summarise peels off one level

per_year <- group_by(batting, playerID, yearID)
stints <- summarise(per_year, stints = max(stint))
filter(stints, stints > 3)
summarise(stints, max(stints))
mutate(stints, cumsum(stints))


# Joins ---------------------------------------------------------------------
player_info <- select(tbl_df(Master), playerID, birthYear)
hof <- select(filter(tbl_df(HallOfFame), inducted == "Y"),
 playerID, votedBy, category)

# Match players and their hall of fame data
inner_join(player_info, hof)
# Keep all players, match hof data where available
left_join(player_info, hof)
# Find only players in hof
semi_join(player_info, hof)
# Find players not in hof
anti_join(player_info, hof)
}
```

`tbl_vars` *List variables provided by a tbl.*

Description

List variables provided by a tbl.

Usage

```
tbl_vars(x)
```

Arguments

x	A tbl object

---

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.

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)
}
```

translate_sql *Translate an expression to sql.*

Description

Translate an expression to sql.

Usage

```
translate_sql(..., con = NULL, vars = character(), vars_group = NULL,
  vars_order = NULL, window = TRUE)

translate_sql_(dots, con = NULL, vars = character(), vars_group = NULL,
  vars_order = NULL, window = TRUE)
```

Arguments

..., dots	Expressions to translate. sql_translate automatically quotes them for you. sql_translate_ expects a list of already quoted objects.
con	An optional database connection to control the details of the translation. The default, NULL, generates ANSI SQL.
vars	A character vector giving variable names in the remote data source. If this is supplied, translate_sql will call partial_eval to interpolate in the values from local variables.
vars_group, vars_order	Grouping and ordering variables used for windowed functions.
window	Use FALSE to suppress generation of the OVER statement used for window functions. This is necessary when generating SQL for a grouped summary.

Base translation

The base translator, base_sql, provides custom mappings for ! (to NOT), && and & to AND, || and | to OR, ^ to POWER, %>% to %, ceiling to CEIL, mean to AVG, var to VARIANCE, tolower to LOWER, toupper to UPPER and nchar to length.

c and : keep their usual R behaviour so you can easily create vectors that are passed to sql.

All other functions will be preserved as is. R's infix functions (e.g. %like%) will be converted to their sql equivalents (e.g. LIKE). You can use this to access SQL string concatenation: || is mapped to OR, but %||% is mapped to ||. To suppress this behaviour, and force errors immediately when dplyr doesn't know how to translate a function it encounters, using set the dplyr.strict_sql option to TRUE.

You can also use sql to insert a raw sql string.

SQLite translation

The SQLite variant currently only adds one additional function: a mapping from sd to the SQL aggregation function stdev.

Examples

```
# Regular maths is translated in a very straightforward way
translate_sql(x + 1)
translate_sql(sin(x) + tan(y))

# Note that all variable names are escaped
translate_sql(like == "x")
# In ANSI SQL: "" quotes variable _names_, '' quotes strings

# Logical operators are converted to their sql equivalents
translate_sql(x < 5 & !(y >= 5))
# xor() doesn't have a direct SQL equivalent
translate_sql(xor(x, y))

# If is translated into case when
translate_sql(if (x > 5) "big" else "small")

# Infix functions are passed onto SQL with % removed
translate_sql(first %like% "Had*")
translate_sql(first %is% NULL)
translate_sql(first %in% c("John", "Roger", "Robert"))


# And be careful if you really want integers
translate_sql(x == 1)
translate_sql(x == 1L)

# If you have an already quoted object, use translate_sql_:
x <- quote(y + 1 / sin(t))
translate_sql_(list(x))

# Translation with known variables ----------------------------------------

# If the variables in the dataset are known, translate_sql will interpolate
# in literal values from the current environment
x <- 10
translate_sql(mpg > x)
translate_sql(mpg > x, vars = names(mtcars))

# By default all computations happens in sql
translate_sql(cyl == 2 + 2, vars = names(mtcars))
# Use local to force local evaluation
translate_sql(cyl == local(2 + 2), vars = names(mtcars))

# This is also needed if you call a local function:
inc <- function(x) x + 1
translate_sql(mpg > inc(x), vars = names(mtcars))
translate_sql(mpg > local(inc(x)), vars = names(mtcars))

# Windowed translation ----------------------------------------
# Known window functions automatically get OVER()
translate_sql(mpg > mean(mpg))
```

```
# Suppress this with window = FALSE
translate_sql(mpg > mean(mpg), window = FALSE)

# vars_group controls partition:
translate_sql(mpg > mean(mpg), vars_group = "cyl")

# and vars_order controls ordering for those functions that need it
translate_sql(cumsum(mpg))
translate_sql(cumsum(mpg), vars_order = "mpg")
```

`vars` *Select columns*

Description

This helper has equivalent semantics to `select()`. Its purpose is to provide `select()` semantics to the colwise summarising and mutating verbs.

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.

See Also

`summarise_all()`

Index