Package ‘dplyr’

May 21, 2014

Type Package

Title dplyr: a grammar of data manipulation

Version 0.2

Author
Hadley Wickham <`h.wickham@gmail.com`>,Romain Francois <`romain@r-enthusiasts.com`>

Maintainer Hadley Wickham <`h.wickham@gmail.com`>

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`

Depends R (>= 3.0.2)

Imports assertthat, utils, methods, Rcpp, magrittr, Lahman, hflights

Suggests RSQLite, RSQLite.extfuns, RMySQL, MonetDB.R, RPostgreSQL,data.table, bigr-
query, testthat, knitr, microbenchmark,ggplot2, mgcv

VignetteBuilder knitr

LazyData yes

LinkingTo Rcpp (>= 0.11.1), BH (>= 1.51.0-2)

License MIT + file LICENSE

Collate 'RcppExports.R' 'all-equal.r' 'base.R' 'bench-compare.r'
'cbind.r' 'chain.r' 'cluster.R' 'colwise.R' 'compute-collect.r'
'copy-to.r' 'data-hflights.r' 'data-lahman.r' 'data-nasa.r'
'data-temp.r' 'data.r' 'dbi-s3.r' 'desc.r' 'do.r' 'dplyr.r'
'explain.r' 'failwith.r' 'funs.R' 'glimpse.R' 'group-by.r'
'group-size.r' 'grouped-df.r' 'grouped-dt.r' 'id.r' 'inline.r'
'join-df.r' 'join-dt.r' 'join-sql.r' 'join.r' 'lead-lag.R'
'location.R' 'manip-cube.r' 'manip-df.r' 'manip-dt.r'
'manip-grouped-dt.r' 'manip-sql.r' 'manip.r' 'nth-value.R'
'order-by.R' 'over.R' 'partial-eval.r' 'progress.R' 'query.r'

'query-bq.r' 'rank.R' 'rbind.r' 'rowwise.r' 'sample.R' 'select-vars.R' 'sets.r' 'sql-escape.r' 'sql-star.r' 'src-bigquery.r' 'src-local.r' 'src-monetdb.r' 'src-mysql.r' 'src-postgres.r' 'src-sql.r' 'src-sqlite.r' 'src.r' 'tally.R' 'tbl-cube.r' 'tbl-data-frame.R' 'tbl-df.r' 'tbl-dt.r' 'tbl-sql.r' 'tbl.r' 'top-n.R' 'translate-sql-helpers.r' 'translate-sql-base.r' 'translate-sql-window.r' 'translate-sql.r' 'type-sum.r' 'utils-dt.R' 'utils-format.r' 'utils.r' 'view.r' 'zzz.r'

NeedsCompilation yes

Repository CRAN

Date/Publication 2014-05-21 08:19:55

R topics documented:

`all.equal.tbl_df` *Provide a useful implementation of all.equal for data.frames.*

Description

Provide a useful implementation of all.equal for data.frames.

Usage

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

## S3 method for class 'tbl_dt'
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 the generic.

Value

`TRUE` if equal, otherwise a character vector describing the first reason 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
mtcars_df <- tbl_df(mtcars)
all.equal(mtcars_df, scramble(mtcars_df))

# But those can be overriden if desired
all.equal(mtcars_df, scramble(mtcars_df), ignore_col_order = FALSE)
all.equal(mtcars_df, scramble(mtcars_df), ignore_row_order = FALSE)
```

`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, met_name = deparse(substitute(x)),
  dim_names = names(dimnames(x)), ...)

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

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

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

Arguments

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

`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 `tbl`s.
`op`	A function with a single argument, called often with each element of `tbls`.
`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.
`times`	For benchmarking, the number of times each operation is repeated.
`...`	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

```
if (require("microbenchmark")) {
lahman_local <- lahman_srcs("df", "dt", "cpp")
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(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)

}
```

---

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, ")")
```

`chain` *Chain together multiple operations.*

Description

The downside of the functional nature of dplyr is that when you combine multiple data manipulation operations, you have to read from the inside out and the arguments may be very distant to the function call. These functions providing an alternative way of calling dplyr (and other data manipulation) functions that you read can from left to right.

Usage

```
chain(..., env = parent.frame())

chain_q(calls, env = parent.frame())

lhs %.% rhs

lhs %>% rhs
```

Arguments

`lhs,rhs`	A dataset and function to apply to it
`...,calls`	A sequence of data transformations, starting with a dataset. The first argument of each call should be omitted - the value of the previous step will be substituted in automatically. Use `chain` and `...` when working interactive; use `chain_q` and `calls` when calling from another function.
`env`	Environment in which to evaluation expressions. In ordinary operation you should not need to set this parameter.

Details

The functions work via simple substitution so that `x %.% f(y)` is translated into `f(x, y)`.

Deprecation

`chain` was deprecated in version 0.2, and will be removed in 0.3. It was removed in the interest of making dplyr code more standardised and `%.%` is much more popular.

Examples

```
# If you're performing many operations you can either do step by step
data("hflights", package = "hflights")
a1 <- group_by(hflights, Year, Month, DayofMonth)
a2 <- select(a1, Year:DayofMonth, ArrDelay, DepDelay)
a3 <- summarise(a2,
  arr = mean(ArrDelay, na.rm = TRUE),
  dep = mean(DepDelay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)

# If you don't want to save the intermediate results, you need to
# wrap the functions:
filter(
  summarise(
    select(
      group_by(hflights, Year, Month, DayofMonth),
      Year:DayofMonth, ArrDelay, DepDelay
    ),
    arr = mean(ArrDelay, na.rm = TRUE),
    dep = mean(DepDelay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)

# This is difficult to read because the order of the operations is from
# inside to out, and the arguments are a long way away from the function.
# Alternatively you can use chain or %>% to sequence the operations
# linearly:

hflights %>%
  group_by(Year, Month, DayofMonth) %>%
  select(Year:DayofMonth, ArrDelay, DepDelay) %>%
  summarise(
    arr = mean(ArrDelay, na.rm = TRUE),
    dep = mean(DepDelay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

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,
  ...)
```

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

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 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 fram 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, indexes = NULL, analyze = TRUE, ...)
```

Arguments

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

Value

a sqlite `tbl` object

Examples

```
if (require("RSQLite") && require("RSQLite.extfuns")) {
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)
```

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, ...)

## S3 method for class 'tbl_sql'
do(.data, ..., .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.
`.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.

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 eqvuivalent 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)`)
# 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
data("hflights", package = "hflights")
carriers <- group_by(hflights, UniqueCarrier)
group_size(carriers)

mods <- do(carriers, mod = lm(ArrDelay ~ DepTime, 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 <- hflights %>% group_by(Dest) %>% filter(n() > 100)
library(mgcv)
by_dest %>% do(smooth = gam(ArrDelay ~ s(DepTime) + Month, data = .))

## End(Not run)
```

`dplyr` *The dplyr package.*

Description

The dplyr package.

`explain` *Explain details of an object*

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, ...)
```

Arguments

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

Details

For more details of explaining dplyr sql tbls, see `explain_sql`.

explain_sql *Show sql and query plans.*

Description

Any queries run inside this function will automatically be explained: displaying information about which indexes are used to optimise the query. This requires a little bit of knowledge about how `EXPLAIN` works for your database, but is very useful for diagnosing performance problems.

Usage

```
explain_sql(code)

show_sql(code)

## S3 method for class 'tbl_sql'
explain(x, ...)
```

Arguments

`code`	code to run. All sql queries executed during the running of the code will be shown and explained.
`x`	an sql-based table to explain.
`...`	Ignored. Needed for compatibility with generic.

Examples

```
if (require("RSQLite") && has_lahman("sqlite")) {

batting <- tbl(lahman_sqlite(), "Batting")

# Note that you have to do something that actually triggers a query
# inside the explain function
explain_sql(nrow(batting))
explain_sql(nrow(batting))

# nrow requires two queries the first time because it's the same as dim(x)[1]
# but the results are cached

show_sql(head(batting))
explain_sql(head(batting))

# If you just want to understand the sql for a tbl, use explain
explain(batting)

# The batting database has indices on all ID variables:
# SQLite automatically picks the most restrictive index
explain(filter(batting, lgID == "NL" & yearID == 2000))
# OR's will use multiple indexes
explain(filter(batting, lgID == "NL" | yearID == 2000))
}
```

---

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)
```

funs *Create a list of functions calls.*

Description

`funs` provides a flexible to generate a named list of functions for input to other functions like `colwise`.

Usage

```
funs(..., env = parent.frame())

funs_q(calls, env = parent.frame())
```

Arguments

`calls,...`	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)`
`env`	The environment in which to evaluate calls.

Examples

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

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

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

glimpse *Get a glimpse of your data.*

Description

This is like a transposed version of print: columns run down the page, and data runs across. This makes it possible to see every column in a data frame. It's a little like `str` applied to a data frame but it tries to show you as much data as possible. (And it always shows the underlying data, even when applied to a remote data source.)

Usage

```
glimpse(tbl, width = getOption("width"))
```

Arguments

`tbl`	A data table
`width`	Width of output: defaults to the width of the console.

Examples

```
glimpse(mtcars)

if (require("RSQLite") && has_lahman("sqlite")) {
  batting <- tbl(lahman_sqlite(), "Batting")
  glimpse(batting)
}
```

`grouped_dt` *A grouped data table.*

Description

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

Usage

```
grouped_dt(data, vars)

is.grouped_dt(x)
```

Arguments

`data`	a tbl or data frame.
`vars`	a list of quoted variables.
`x`	an object to check

Examples

```
if (require("data.table")) {
data("hflights", package = "hflights")
hflights_dt <- tbl_dt(hflights)
group_size(group_by(hflights_dt, Year, Month, DayofMonth))
group_size(group_by(hflights_dt, Dest))

monthly <- group_by(hflights_dt, Month)
summarise(monthly, n = n(), delay = mean(ArrDelay))
}
```

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)

regroup(x, value)

ungroup(x)
```

Arguments

`x`	data `tbl`
`value`	a list of symbols

See Also

`group_by` for a version that does non-standard evaluation to save typing

Examples

```
grouped <- group_by(mtcars, cyl)
groups(grouped)
grouped <- regroup(grouped, list(quote(vs)))
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(x, ..., add = FALSE)
```

Arguments

x	a tbl
...	variables to group by. All tbls accept variable names, some will also accept functons 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 = FALSE

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, group 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 followed by a simple group_by
group_by(mtcars, vsam = vs + am)

# 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_size` *Calculate the size of each group*

Description

Calculate the size of each group

Usage

```
group_size(x)
```

Arguments

`x`	a grouped tbl

Examples

```
data("hflights", package = "hflights")
group_size(group_by(hflights, Year, Month, DayofMonth))
group_size(group_by(hflights, Dest))
```

`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, ...)

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

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.
`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 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`. If there are multiple matches between x and y, all combination 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

Grouping

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

`join.tbl_df` *Join data table tbls.*

Description

See `join` for a description of the general purpose of the functions. The data frame implementations are currently not terribly efficient.

Usage

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

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

```
## 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,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, supply a character vector.
`copy`	If `y` is not a data frame or `tbl_df` and `copy` is `TRUE`, `y` will be converted into a data frame
`...`	included for compatibility with the generic; otherwise ignored.

Examples

```
data("Batting", package = "Lahman")
data("Master", package = "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_dt` *Join data table tbls.*

Description

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

Usage

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

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

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

## S3 method for class 'tbl_dt'
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, supply a character vector.
`copy`	If `y` is not a data table or `tbl_dt` and `copy` is `TRUE`, `y` will be converted into a data table.
`...`	Included for compatibility with generic; otherwise ignored.

Examples

```
if (require("RSQLite") && require("RSQLite.extfuns")) {
data("Batting", package = "Lahman")
data("Master", package = "Lahman")

batting_dt <- tbl_dt(Batting)
person_dt <- tbl_dt(Master)

# Inner join: match batting and person data
inner_join(batting_dt, person_dt)

# Left join: keep batting data even if person missing
left_join(batting_dt, person_dt)

# Semi-join: find batting data for top 4 teams, 2010:2012
grid <- expand.grid(
  teamID = c("WAS", "ATL", "PHI", "NYA"),
  yearID = 2010:2012)
top4 <- semi_join(batting_dt, grid, copy = TRUE)

# Anti-join: find batting data with out player data
anti_join(batting_dt, person_dt)
}
```

`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_sql'
inner_join(x, y, by = NULL, copy = FALSE,
  auto_index = FALSE, ...)

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

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

## S3 method for class 'tbl_sql'
anti_join(x, y, by = NULL, copy = FALSE,
  auto_index = 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, supply a character vector.
`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.
`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

```
if (require("RSQLite") && has_lahman("sqlite")) {

# Left joins ----------------------------------------------------------------
batting <- tbl(lahman_sqlite(), "Batting")
team_info <- select(tbl(lahman_sqlite(), "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_sqlite(), "Master")

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

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

# Find all managers
manager <- tbl(lahman_sqlite(), "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)
}
```

lahman — *Cache and retrieve an `src_sqlite` of the Lahman baseball database.*

Description

This creates an interesting database using data from the Lahman baseball data source, provided by Sean Lahman at http://www.seanlahman.com/baseball-archive/statistics/, and made easily available in R through the **Lahman** package by Michael Friendly, Dennis Murphy and Martin Monkman. See the documentation for that package for documentation of the inidividual tables.

Usage

```
lahman_sqlite()

lahman_postgres(...)

lahman_mysql(...)

lahman_monetdb(...)

lahman_df()

lahman_dt()

lahman_bigquery(...)

has_lahman(type, ...)

lahman_srcs(..., quiet = NULL)
```

Arguments

`...`	Arguments passed to `src` on first load. For mysql and postgresql, the defaults assume you have a local server with `lahman` database already created. For bigquery, it assumes you have read/write access to a project called `Sys.getenv("BIGQUERY_PROJECT")` For `lahman_srcs`, character vector of names giving srcs to generate.
`quiet`	if `TRUE`, suppress messages about databases failing to connect.
`type`	src type.

Examples

```
# Connect to a local sqlite database, if already created
if (require("RSQLite") && has_lahman("sqlite")) {
  lahman_sqlite()
  batting <- tbl(lahman_sqlite(), "Batting")
  batting
}
```

```
# Connect to a local postgres database with lahman database, if available
if (require("RPostgreSQL") && has_lahman("postgres")) {
  lahman_postgres()
  batting <- tbl(lahman_postgres(), "Batting")
}
```

`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

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)
```

`manip` *Data manipulation functions.*

Description

These five functions form the backbone of dplyr. They are all S3 generic functions with methods for each individual data type. All functions work exactly the same way: the first argument is the tbl, and the subsequence arguments are interpreted in the context of that tbl.

Usage

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

summarise(.data, ...)

summarize(.data, ...)

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

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

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

Arguments

`.data`	a tbl
`...`	variables interpreted in the context of that data frame.

Manipulation functions

The five key data manipulation functions are:

- filter: return only a subset of the rows. If multiple conditions are supplied they are combined with &.
- select: return only a subset of the columns. If multiple columns are supplied they are all used.
- arrange: reorder the rows. Multiple inputs are ordered from left-to- right.
- mutate: add new columns. Multiple inputs create multiple columns.
- summarise: reduce each group to a single row. Multiple inputs create multiple output summaries.

These are all made significantly more useful when applied by group, as with `group_by`

Tbls

dplyr comes with three built-in tbls. Read the help for the manip methods of that class to get more details:

- data.frame: manip_df
- data.table: manip_dt
- SQLite: `src_sqlite`
- PostgreSQL: `src_postgres`
- MySQL: `src_mysql`

Output

Generally, manipulation functions will return an output object of the same type as their input. The exceptions are:

- `summarise` will return an ungrouped source
- remote sources (like databases) will typically return a local source from at least `summarise` and `mutate`

Row names

dplyr methods do not preserve row names. If have been using row names to store important information, please make them explicit variables.

Arrange

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.

Selection

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

- `starts_with(x, ignore.case = FALSE)`: names starts with `x`
- `ends_with(x, ignore.case = FALSE)`: names ends in `x`
- `contains(x, ignore.case = FALSE)`: selects all variables whose name contains `x`
- `matches(x, ignore.case = FALSE)`: selects all variables whose name matches the regular expression `x`
- `num_range("x", 1:5, width = 2)`: selects all variables (numerically) from x01 to x05.

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

Examples

```
filter(mtcars, cyl == 8)
select(mtcars, mpg, cyl, hp:vs)
arrange(mtcars, cyl, disp)
mutate(mtcars, displ_l = disp / 61.0237)
summarise(mtcars, mean(disp))
summarise(group_by(mtcars, cyl), mean(disp))
# More detailed select 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)

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(iris, petal_length = Petal.Length)
select(iris, petal = starts_with("Petal"))
```

manip_df *Data manipulation for data frames.*

Description

Data manipulation for data frames.

Usage

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

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

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

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

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

Arguments

`.data`	a data frame
`...`	variables interpreted in the context of `.data`

Examples

```
data("hflights", package = "hflights")
filter(hflights, Month == 1, DayofMonth == 1, Dest == "DFW")
head(select(hflights, Year:DayOfWeek))
summarise(hflights, delay = mean(ArrDelay, na.rm = TRUE), n = length(ArrDelay))
head(mutate(hflights, gained = ArrDelay - DepDelay))
head(arrange(hflights, Dest, desc(ArrDelay)))
```

manip_dt *Data manipulation for data tables.*

Description

Data manipulation for data tables.

Usage

```
## S3 method for class 'data.table'
filter(.data, ..., .env = parent.frame())

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

## S3 method for class 'data.table'
mutate(.data, ..., inplace = FALSE)

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

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

Arguments

`.data`	a data table
`...`	variables interpreted in the context of `.data`
`inplace`	if `FALSE` (the default) the data frame will be copied prior to modification to avoid changes propagating via reference.
`.env`	The environment in which to evaluate arguments not included in the data. The default should suffice for ordinary usage.

Examples

```
if (require("data.table")) {
# If you start with a data table, you end up with a data table
data("hflights", package = "hflights")
hflights <- as.data.table(hflights)
filter(hflights, Month == 1, DayofMonth == 1, Dest == "DFW")
head(select(hflights, Year:DayOfWeek))
summarise(hflights, delay = mean(ArrDelay, na.rm = TRUE), n = length(ArrDelay))
head(mutate(hflights, gained = ArrDelay - DepDelay))
head(arrange(hflights, Dest, desc(ArrDelay)))

# If you start with a tbl, you end up with a tbl
hflights2 <- as.tbl(hflights)
filter(hflights2, Month == 1, DayofMonth == 1, Dest == "DFW")
head(select(hflights2, Year:DayOfWeek))
summarise(hflights2, delay = mean(ArrDelay, na.rm = TRUE), n = length(ArrDelay))
head(mutate(hflights2, gained = ArrDelay - DepDelay))
head(arrange(hflights2, Dest, desc(ArrDelay)))
}
```

manip_grouped_dt *Data manipulation for grouped data tables.*

Description

Data manipulation for grouped data tables.

Usage

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

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

## S3 method for class 'grouped_dt'
mutate(.data, ..., inplace = FALSE)

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

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

Arguments

`.data`	a data table
`...`	variables interpreted in the context of `.data`
`inplace`	if `FALSE` (the default) the data frame will be copied prior to modification to avoid changes propagating via reference.

Examples

```
if (require("data.table")) {
data("hflights", package = "hflights")
hflights2 <- tbl_dt(hflights)
by_dest <- group_by(hflights2, Dest)

filter(by_dest, ArrDelay == max(ArrDelay, na.rm = TRUE))
summarise(by_dest, arr = mean(ArrDelay, na.rm = TRUE))

# Normalise arrival and departure delays by airport
scaled <- mutate(by_dest, arr_z = scale(ArrDelay), dep_z = scale(DepDelay))
select(scaled, Year:DayOfWeek, Dest, arr_z:dep_z)

arrange(by_dest, desc(ArrDelay))
select(by_dest, -(DayOfWeek:TailNum))
# All manip functions preserve grouping structure, except for summarise
# which removes a grouping level
by_day <- group_by(hflights, Year, Month, DayofMonth)
by_month <- summarise(by_day, delayed = sum(ArrDelay > 0, na.rm = TRUE))
by_month
summarise(by_month, delayed = sum(delayed))

# You can also manually ungroup:
ungroup(by_day)
}
```

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

```
data("hflights", package = "hflights")
carriers <- group_by(hflights, UniqueCarrier)
summarise(carriers, n())
mutate(carriers, n = n())
filter(carriers, n() == 79)
```

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
```

---

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. If a numeric 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

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

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

Description

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

Usage

```
n_distinct(x)
```

Arguments

`x`	a vector 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
`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)
row_number(x)
min_rank(x)
dense_rank(x)
percent_rank(x)
cume_dist(x)

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

`rbind_all` *Efficiently rbind multiple data frames.*

Description

This is an efficient version of the common pattern of `do.call(rbind, dfs)` for row-binding many data frames together. It works in the same way as `rbind.fill` but is implemented in C++ so avoids many copies and is much much faster.

Usage

```
rbind_all(dots)

rbind_list(...)
```

Arguments

`dots,...`	list of data frames to combine. With `rbind_all`, they should already be in a list, with `rbind_list` you supply them individually.

Examples

```
one <- mtcars[1:10, ]
two <- mtcars[11:32, ]

rbind_list(one, two)
rbind_all(list(one, two))
```

---

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. It's 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)
```

`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, ...)

setdiff(x, y, ...)

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

Arguments

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

`setops-data.frame` *Set operations for data frames.*

Description

These set operations are implemented with an efficeint C++ backend.

Usage

```
## S3 method for class 'data.frame'
intersect(x, y, ...)

## S3 method for class 'data.frame'
union(x, y, ...)

## S3 method for class 'data.frame'
setdiff(x, y, ...)

## S3 method for class 'data.frame'
setequal(x, y, ...)
```

Arguments

`x,y`	Two data frames to compare, igoring order of row and columns
`...`	Needed for compatibility with generic. Otherwise ignored.

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)

setequal(mtcars, mtcars[32:1, ])
```

`src_bigquery` *A bigquery data source.*

Description

Use `src_bigquery` to connect to an existing bigquery dataset, and `tbl` to connect to tables within that database.

Usage

```
src_bigquery(project, dataset, billing = project)

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

Arguments

`project`	project id or name
`dataset`	dataset name
`billing`	billing project, if different to `project`
`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.
`src`	a bigquery src created with `src_bigquery`.

Debugging

To see exactly what SQL is being sent to the database, you can set option `dplyr.show_sql` to true: `options(dplyr.show_sql = TRUE)`. If you're wondering why a particularly query is slow, it can be helpful to see the query plan. You can do this by setting `options(dplyr.explain_sql = TRUE)`.

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_sql to check that mysql 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

```
# Connection basics ---------------------------------------------------------
## Not run:
# To connect to a database first create a src:
my_db <- src_bigquery("myproject", "mydataset")
# 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_bigqueryql() how to
# a database that you can write to

if (has_lahman("bigquery") && interactive()) {
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_bigquery(), "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))
filter(players, AB == max(AB) | G == max(G))
# Not supported yet:
## Not run:
mutate(players, cyear = yearID - min(yearID) + 1,
 cumsum(AB, yearID))

## End(Not run)
mutate(players, rank())

# 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))
# Not supported yet:
## Not run: mutate(stints, cumsum(stints, yearID))
# But other window functions are:
mutate(players, rank = rank(ab))

# Joins ---------------------------------------------------------------------
player_info <- select(tbl(lahman_bigquery(), "Master"), playerID, hofID,
  birthYear)
hof <- select(filter(tbl(lahman_bigquery(), "HallOfFame"), inducted == "Y"),
 hofID, 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_bigqueryql(),
  sql("SELECT * FROM Batting WHERE YearID = 2008"))
batting2008
}
```

---

src_monetdb *Connect to MonetDB (http://www.monetdb.org), an Open Source analytics-focused database*

Description

Use `src_monetdb` to connect to an existing MonetDB database, and `tbl` to connect to tables within that database. Please note that the ORDER BY, LIMIT and OFFSET keywords are not supported in the query when using `tbl` on a connection to a MonetDB database. If you are running a local database, you only need to define the name of the database you want to connect to.

Usage

```
src_monetdb(dbname, host = "localhost", port = 50000L, user = "monetdb",
  password = "monetdb", ...)

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

Arguments

`dbname`	Database name
`host,port`	Host name and port number of database (defaults to localhost:50000)
`user,password`	User name and password (if needed)
`...`	for the src, other arguments passed on to the underlying database connector, `dbConnect`.
`src`	a MonetDB src created with `src_monetdb`.
`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 can set option `dplyr.show_sql` to true: `options(dplyr.show_sql = TRUE)`. If you're wondering why a particularly query is slow, it can be helpful to see the query plan. You can do this by setting `options(dplyr.explain_sql = TRUE)`.

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_sql to check that mysql 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_monetdb(dbname="demo")
# 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" first.

if (has_lahman("monetdb")) {
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_monetdb(), "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))

# 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))

# Joins ---------------------------------------------------------------------
player_info <- select(tbl(lahman_monetdb(), "Master"), playerID, hofID,
  birthYear)
hof <- select(filter(tbl(lahman_monetdb(), "HallOfFame"), inducted == "Y"),
 hofID, 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_monetdb(),
  sql('SELECT * FROM "Batting" WHERE "yearID" = 2008'))
batting2008
}
```

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 (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 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 can set option `dplyr.show_sql` to true: `options(dplyr.show_sql = TRUE)`. If you're wondering why a particularly query is slow, it can be helpful to see the query plan. You can do this by setting `options(dplyr.explain_sql = TRUE)`.

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_sql` to check that mysql 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("mysql")) {
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_mysql(), "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_mysql(), "Master"), playerID, hofID,
  birthYear)
hof <- select(filter(tbl(lahman_mysql(), "HallOfFame"), inducted == "Y"),
 hofID, 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_mysql(),
  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 can set option `dplyr.show_sql` to true: `options(dplyr.show_sql = TRUE)`. If you're wondering why a particularly query is slow, it can be helpful to see the query plan. You can do this by setting `options(dplyr.explain_sql = TRUE)`.

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_sql` to check that mysql 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
# a database that you can write to

if (has_lahman("postgres")) {
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_postgres(), "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))
progress <- mutate(players, cyear = yearID - min(yearID) + 1,
 rank(desc(AB)), cumsum(AB, yearID))

# 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, yearID))
# Joins ---------------------------------------------------------------------
player_info <- select(tbl(lahman_postgres(), "Master"), playerID, hofID,
  birthYear)
hof <- select(filter(tbl(lahman_postgres(), "HallOfFame"), inducted == "Y"),
 hofID, 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_postgres(),
  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.

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`.
`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 can set option `dplyr.show_sql` to true: `options(dplyr.show_sql = TRUE)`. If you're wondering why a particularly query is slow, it can be helpful to see the query plan. You can do this by setting `options(dplyr.explain_sql = TRUE)`.

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_sql` to check that mysql 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()
```

```
if (require("RSQLite") && has_lahman("sqlite")) {
# Methods -------------------------------------------------------------------
batting <- tbl(lahman_sqlite(), "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_sqlite(), "Master"), playerID, hofID,
  birthYear)
hof <- select(filter(tbl(lahman_sqlite(), "HallOfFame"), inducted == "Y"),
 hofID, 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_sqlite(),
  sql("SELECT * FROM Batting WHERE YearID = 2008"))
batting2008
}
```

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_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_q(tbl, funs, vars, env = parent.frame())

mutate_each(tbl, funs, ...)

mutate_each_q(tbl, funs, vars, env = parent.frame())
```

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`. For standard evaluation versions (ending in _q) these can be either a list of expressions or a character vector.
`env`	The environment in which to evaluate the function calls. Should only be modified by advanced users.

Examples

```
# One function
by_species <- iris %>% group_by(Species)
by_species %>% summarise_each(funs(length))
by_species %>% summarise_each(funs(mean))
by_species %>% summarise_each(funs(mean), Petal.Width)
by_species %>% summarise_each(funs(mean), matches("Width"))

by_species %>% mutate_each(funs(half = . / 2))
by_species %>% mutate_each(funs(min_rank))

# Two functions
by_species %>% summarise_each(funs(min, max))
by_species %>% summarise_each(funs(min, max), Petal.Width, Sepal.Width)
by_species %>% summarise_each(funs(min, max), matches("Width"))

# Alternative function specification
iris %>% summarise_each(funs(ul = length(unique(.))))
by_species %>% summarise_each(funs(ul = length(unique(.))))

by_species %>% summarise_each(c("min", "max"))

# Alternative variable specification
summarise_each_q(iris, funs(max), names(iris)[-5])
summarise_each_q(iris, funs(max), list(quote(-Species)))
```

`tally` *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.

Usage

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

Arguments

x	a `tbl` to tally
wt	if not specified, will tally the number of rows. If specified, will perform a "weighted" tally but summing over the specified variable.
sort	if `TRUE` will sort output in descending order of `n`

Examples

```
library(Lahman)
batting_tbl <- tbl_df(Batting)
tally(group_by(batting_tbl, yearID))
tally(group_by(batting_tbl, yearID), sort = TRUE)

# Multiple tallys progressively role 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)
```

---

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

An 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 supported 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 tble.*

Description

A data frame tbl wraps a local data frame. The main advantage to using a `tbl_df` over a regular data frame is the printing: tbl objects only print a few rows and all the columns that fit on one screen, providing describing the rest of it as text.

Usage

```
tbl_df(data)
```

Arguments

`data`	a data frame

Examples

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

data("Batting", package = "Lahman")
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 ---------------------------------------------------------------------
data("Master", "HallOfFame", package = "Lahman")
player_info <- select(tbl_df(Master), playerID, hofID, birthYear)
hof <- select(filter(tbl_df(HallOfFame), inducted == "Y"),
 hofID, 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_dt` *Create a data table tbl.*

Description

A data table tbl wraps a local data table.

Usage

```
tbl_dt(data)
```

Arguments

`data`	a data table

Examples

```
if (require("data.table")) {
ds <- tbl_dt(mtcars)
ds
as.data.table(ds)
as.tbl(mtcars)
}
```

`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 n rows (by value).

Description

This is a convenient wrapper that uses `filter` and `rank` to select the top n entries in each group, ordered by `wt`.

Usage

```
top_n(x, n, wt = NULL)
```

Arguments

`x`	a `tbl` to filter
`n`	number of rows to return. If `x` is grouped, this is the number of rows per group. May include more than `n` if there are ties.
`wt`	the variable to use for ordering. If not specified, defaults to the last variable in the tbl.

Examples

```
data("Batting", package = "Lahman")
players <- group_by(tbl_df(Batting), playerID)
games <- tally(players, G)
top_n(games, 10, n)

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

translate_sql Translate an expression to sql.

Description

Translate an expression to sql.

Usage

```
translate_sql(..., tbl = NULL, env = parent.frame(), variant = NULL,
  window = FALSE)

translate_sql_q(expr, tbl = NULL, env = parent.frame(), variant = NULL,
  window = FALSE)
```

Arguments

`...`	unevaluated expression to translate
`expr`	list of quoted objects to translate
`tbl`	An optional `tbl`. If supplied, will be used to automatically figure out the SQL variant to use.
`env`	environment in which to evaluate expression.
`variant`	used to override default variant provided by source useful for testing/examples
`window`	If `variant` not supplied, used to determine whether the variant is window based or not.

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))

# Logical operators are converted to their sql equivalents
translate_sql(x < 5 & !(y >= 5))

# If is translated into select case
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"))

# Note that variable names will be escaped if needed
translate_sql(like == 7)
```

```
# 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_q:
x <- quote(y + 1 / sin(t))
translate_sql(x)
translate_sql_q(list(x))

# Translation with data source --------------------------------------------

hflights <- tbl(hflights_postgres(), "hflights")
# Note distinction between integers and reals
translate_sql(Month == 1, tbl = hflights)
translate_sql(Month == 1L, tbl = hflights)

# Know how to translate most simple mathematical expressions
translate_sql(Month %in% 1:3, tbl = hflights)
translate_sql(Month >= 1L & Month <= 3L, tbl = hflights)
translate_sql((Month >= 1L & Month <= 3L) | Carrier == "AA", tbl = hflights)

# Some R functions don't have equivalents in SQL: where possible they
# will be translated to the equivalent
translate_sql(xor(Month <= 3L, Carrier == "AA"), tbl = hflights)

# Local variables will be automatically inserted into the SQL
x <- 5L
translate_sql(Month == x, tbl = hflights)

# By default all computation will happen in sql
translate_sql(Month < 1 + 1, source = hflights)
# Use local to force local evaluation
translate_sql(Month < local(1 + 1), source = hflights)

# This is also needed if you call a local function:
inc <- function(x) x + 1
translate_sql(Month == inc(x), source = hflights)
translate_sql(Month == local(inc(x)), source = hflights)

# Windowed translation --------------------------------------------
planes <- arrange(group_by(hflights, TailNum), desc(DepTime))

translate_sql(DepTime > mean(DepTime), tbl = planes, window = TRUE)
translate_sql(DepTime == min(DepTime), tbl = planes, window = TRUE)

translate_sql(rank(), tbl = planes, window = TRUE)
translate_sql(rank(DepTime), tbl = planes, window = TRUE)
translate_sql(ntile(DepTime, 2L), tbl = planes, window = TRUE)
translate_sql(lead(DepTime, 2L), tbl = planes, window = TRUE)
translate_sql(cumsum(DepDelay), tbl = planes, window = TRUE)
translate_sql(order_by(DepDelay, cumsum(DepDelay)), tbl = planes, window = TRUE)
```

Index