Package 'dbplyr'

June 27, 2017

Type Package

Version 1.1.0

Title A 'dplyr' Back End for Databases

Description A 'dplyr' back end for databases that allows you to work with remote database tables as if they are in-memory data frames. Basic features works with any database that has a 'DBI' back end; more advanced features require 'SQL' translation to be provided by the package author.

URL https://github.com/tidyverse/dbplyr

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

Depends R (>= 3.1.2)

Imports assertthat, DBI (>= 0.5), dplyr (>= 0.5.0.9004), glue, methods, purrr, rlang (>= 0.1.0), tibble (>= 1.3.0.9007), R6, utils

Suggests covr, knitr, Lahman (>= 3.0-1), nycflights13, rmarkdown, RSQLite (>= 1.0.0), RMySQL, RPostgreSQL, testthat

VignetteBuilder knitr

LazyData yes

License MIT + file LICENSE

Collate 'cache.r' 'compat-purrr.R' 'data-lahman.r' 'data-nycflights13.r' 'db-compute.R' 'db-mysql.r' 'db-odbc-hive.R' 'db-odbc-impala.R' 'db-odbc-mssql.R' 'db-odbc-oracle.R' 'db-odbc-postgres.R' 'db-postgres.r' 'db-sqlite.r' 'dbi-s3.r' 'dbplyr.R' 'do.r' 'explain.r' 'ident.R' 'lazy-ops.R' 'memdb.R' 'partial-eval.r' 'query.r' 'schema.R' 'simulate.r' 'sql-build.R' 'sql-escape.r' 'sql-generic.R' 'sql-optimise.R' 'sql-query.R' 'sql-render.R' 'sql.R' 'src-sql.r' 'src_dbi.R' 'tbl-lazy.R' 'tbl-sql.r' 'test-frame.R' 'testthat.r' 'translate-sql-helpers.r' 'translate-sql-window.r' 'translate-sql-base.r' 'translate-sql-clause.r' 'translate-sql-odbc.R' 'translate-sql.r' 'utils-format.r' 'utils.r' 'window.R' 'zzz.R'

RoxygenNote 6.0.1

NeedsCompilation no

Author Hadley Wickham [aut, cre],
RStudio [cph, fnd]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2017-06-27 06:55:24 UTC

R topics documented:

`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 = sql_current_con())
```

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, ")")
```

`copy_to.src_sql` *Copy a local data frame to a DBI backend.*

Description

This `copy_to()` method works for all DBI sources. It is useful for copying small amounts of data to a database for examples, experiments, and joins. By default, it creates temporary tables which are typically only visible to the current connection to the database.

Usage

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

Arguments

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

Value

A tbl() object (invisibly).

Examples

```
library(dplyr)
set.seed(1014)

mtcars$model <- rownames(mtcars)
mtcars2 <- src_memdb() %>%
  copy_to(mtcars, indexes = list("model"), overwrite = TRUE)
mtcars2 %>% filter(model == "Hornet 4 Drive")

# copy_to is called automatically if you set copy = TRUE
# in the join functions
df <- tibble(cyl = c(6, 8))
mtcars2 %>% semi_join(df, copy = TRUE)
```

---

do.tbl_sql *Perform arbitrary computation on remote backend*

Description

Perform arbitrary computation on remote backend

Usage

```
## 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.

escape *Escape/quote a string.*

Description

Escape/quote a string.

Usage

```
escape(x, parens = NA, collapse = " ", con = NULL)

sql_vector(x, parens = NA, collapse = " ", con = NULL)
```

Arguments

x	An object to escape. Existing sql vectors will be left as is, character vectors are escaped with single quotes, numeric vectors have trailing `.0` added if they're whole numbers, identifiers are escaped with double quotes.
parens, collapse	Controls behaviour when multiple values are supplied. `parens` should be a logical flag, or if `NA`, will wrap in parens if length > 1. Default behaviour: lists are always wrapped in parens and separated by commas, identifiers are separated by commas and never wrapped, atomic vectors are separated by spaces and wrapped in parens if needed.
con	Database connection. If not specified, uses SQL 92 conventions.

Examples

```
# Doubles vs. integers
escape(1:5)
escape(c(1, 5.4))

# String vs known sql vs. sql identifier
escape("X")
escape(sql("X"))
escape(ident("X"))

# Escaping is idempotent
escape("X")
escape(escape("X"))
escape(escape(escape("X")))
```

ident *Flag a character vector as SQL identifiers*

Description

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

Usage

```
ident(...)

ident_q(...)

is.ident(x)
```

Arguments

`...`	A character vector, or name-value pairs
`x`	An object

Details

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

Examples

```
# SQL92 quotes strings with '
escape("x")

# And identifiers with "
ident("x")

# You can supply multiple inputs
ident(a = "x", b = "y")
ident_q(a = "x", b = "y")
```

in_schema *Refer to a table in a schema*

Description

Refer to a table in a schema

Usage

```
in_schema(schema, table)
```

Arguments

schema, table	Names of schema and table.

Examples

```
in_schema("my_schema", "my_table")

# Example using schemas with SQLite
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
src <- src_dbi(con)

# Add auxilary schema
tmp <- tempfile()
DBI::dbExecute(con, paste0("ATTACH '", tmp, "' AS aux"))

library(dplyr, warn.conflicts = FALSE)
copy_to(con, iris, "df", temporary = FALSE)
copy_to(con, mtcars, in_schema("aux", "df"), temporary = FALSE)

con %>% tbl("df")
con %>% tbl(in_schema("aux", "df"))
```

---

join.tbl_sql *Join sql tbls.*

Description

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

Usage

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

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

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

## S3 method for class 'tbl_lazy'
```

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

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

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

Arguments

x	tbls to join
y	tbls to join
by	a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, by = c("a" = "b") will match x.a to y.b.
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into a temporary table in same database as x. *_join() will automatically run ANALYZE on the created table in the hope that this will make you queries as efficient as possible by giving more data to the query planner. This allows you to join tables across srcs, but it's potentially expensive operation so you must opt into it.
suffix	If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to diambiguate them. Should be a character vector of length 2.
auto_index	if copy is TRUE, automatically create indices for the variables in by. This may speed up the join if there are matching indexes in x.
...	other parameters passed onto methods

Implementation notes

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

Examples

```
## Not run:
library(dplyr)
if (has_lahman("sqlite")) {

# Left joins ----------------------------------------------------------------
lahman_s <- lahman_sqlite()
batting <- tbl(lahman_s, "Batting")
```

```
team_info <- select(tbl(lahman_s, "Teams"), yearID, lgID, teamID, G, R:H)

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

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

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

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

people <- tbl(lahman_s, "Master")

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

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

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

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

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

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

## End(Not run)
```

---

memdb_frame *Create a database table in temporary in-memory database.*

Description

memdb_frame() works like tibble::tibble(), but instead of creating a new data frame in R, it creates a table in src_memdb().

Usage

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

src_memdb()
```

Arguments

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

Examples

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

---

sql *SQL escaping.*

Description

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

Usage

```
sql(...)

is.sql(x)

as.sql(x)
```

Arguments

...	Character vectors that will be combined into a single SQL expression.
x	Object to coerce

src_dbi *dplyr backend for any DBI-compatible database*

Description

`src_dbi()` is a general dplyr backend that connects to any DBI driver. `src_memdb()` connects to a temporary in-memory SQLite database, that's useful for testing and experimenting.

You can generate a `tbl()` directly from the DBI connection, or go via `src_dbi()`.

Usage

```
src_dbi(con, auto_disconnect = FALSE)

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

Arguments

`con`	An object that inherits from DBI::DBIConnection, typically generated by DBI::dbConnect
`auto_disconnect`	Should the connection be automatically closed when the src is deleted. This is useful for older DBI backends that don't clean up themselves.
`src`	Either a `src_dbi` or `DBIConnection`
`from`	Either a string (giving a table name) or literal `sql()`.
`...`	Needed for compatibility with generic; currently ignored.

Details

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

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

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

Value

An S3 object with class `src_dbi`, `src_sql`, `src`.

Examples

```
# Basic connection using DBI ----------------------------------------
library(dplyr)

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
src <- src_dbi(con)

# Add some data
copy_to(src, mtcars)
src
DBI::dbListTables(con)

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

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

# Alternatively, you can use the `src_sqlite()` helper
src2 <- src_sqlite(":memory:", create = TRUE)

# If you just want a temporary in-memory database, use src_memdb()
src3 <- src_memdb()

# To show off the full features of dplyr's database integration,
# we'll use the Lahman database. lahman_sqlite() takes care of
# creating the database.

if (has_lahman("sqlite")) {
lahman_p <- lahman_sqlite()
batting <- lahman_p %>% tbl("Batting")
batting

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

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

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

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

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

`translate_sql` *Translate an expression to sql.*

Description

Translate an expression to sql.

Usage

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

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

Arguments

`..., dots`	Expressions to translate. `translate_sql()` automatically quotes them for you. `translate_sql_()` expects a list of already quoted objects.
`con`	An optional database connection to control the details of the translation. The default, `NULL`, generates ANSI SQL.
`vars`	Deprecated. Now call `partial_eval()` directly.
`vars_group, vars_order, vars_frame`	Parameters used in the `OVER` expression of windowed functions.
`window`	Use `FALSE` to suppress generation of the `OVER` statement used for window functions. This is necessary when generating SQL for a grouped summary.

Base translation

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

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

All other functions will be preserved as is. R's infix functions (e.g. `%like%`) will be converted to their SQL equivalents (e.g. `LIKE`). You can use this to access SQL string concatenation: `||` is mapped to `OR`, but `%||%` is mapped to `||`. To suppress this behaviour, and force errors immediately

when dplyr doesn't know how to translate a function it encounters, using set the `dplyr.strict_sql` option to `TRUE`.

You can also use `sql()` to insert a raw sql string.

SQLite translation

The SQLite variant currently only adds one additional function: a mapping from `sd()` to the SQL aggregation function `STDEV`.

Examples

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

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

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

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

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

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

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

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

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

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

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

```
translate_sql(cumsum(mpg), vars_order = "mpg")
```

`window_order` *Override window order and frame*

Description

Override window order and frame

Usage

```
window_order(.data, ...)

window_frame(.data, from = -Inf, to = Inf)
```

Arguments

`.data`	A remote tibble
`...`	Name-value pairs of expressions.
`from, to`	Bounds of the frame.

Examples

```
library(dplyr)
df <- lazy_frame(g = rep(1:2, each = 5), y = runif(10), z = 1:10)

df %>%
  window_order(y) %>%
  mutate(z = cumsum(y)) %>%
  sql_build()

df %>%
  group_by(g) %>%
  window_frame(-3, 0) %>%
  window_order(z) %>%
  mutate(z = sum(x)) %>%
  sql_build()
```

Index