Package ‘dbplyr’

November 2, 2020

Type Package

Title A 'dplyr' Back End for Databases

Version 2.0.0

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.

License MIT + file LICENSE

URL `https://dbplyr.tidyverse.org/`, `https://github.com/tidyverse/dbplyr`

BugReports `https://github.com/tidyverse/dbplyr/issues`

Depends R (>= 3.1)

Imports assertthat (>= 0.2.0),
DBI (>= 1.0.0),
dplyr (>= 0.8.0),
glue (>= 1.2.0),
lifecycle,
magrittr,
methods,
purrr (>= 0.2.5),
R6 (>= 2.2.2),
rlang (>= 0.2.0),
tibble (>= 1.4.2),
tidyselect (>= 0.2.4),
blob (>= 1.2.0),
utils,
withr

Suggests bit64,
covr,
knitr,
Lahman,
nycflights13,
odbc,
RMariaDB (>= 1.0.2),
rmarkdown,
RPostgres (>= 1.1.3),
RPostgreSQL,

RSQLite (>= 2.1.0),
testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Encoding UTF-8

Language en-gb

LazyData yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

Collate 'utils.R'
'sql.R'
'escape.R'
'translate-sql-quantile.R'
'translate-sql-string.R'
'translate-sql-paste.R'
'translate-sql-helpers.R'
'translate-sql-window.R'
'translate-sql-conditional.R'
'backend-.R'
'backend-access.R'
'backend-hana.R'
'backend-hive.R'
'backend-impala.R'
'backend-mssql.R'
'backend-mysql.R'
'backend-odbc.R'
'backend-oracle.R'
'backend-postgres.R'
'backend-postgres-old.R'
'backend-redshift.R'
'backend-sqlite.R'
'backend-teradata.R'
'build-sql.R'
'data-cache.R'
'data-lahman.R'
'data-nycflights13.R'
'db-escape.R'
'db-io.R'
'db-sql.R'
'db.R'
'dbplyr.R'
'explain.R'
'ident.R'
'lazy-ops.R'
'memdb.R'
'partial-eval.R'
'progress.R'
'query-join.R'
'query-select.R'
'query-semi-join.R'

'query-set-op.R'
'query.R'
'reexport.R'
'remote.R'
'schema.R'
'simulate.R'
'sql-build.R'
'sql-clause.R'
'sql-expr.R'
'src-sql.R'
'src_dbi.R'
'tbl-lazy.R'
'tbl-sql.R'
'test-frame.R'
'testthat.R'
'translate-sql.R'
'utils-format.R'
'verb-arrange.R'
'verb-compute.R'
'verb-copy-to.R'
'verb-distinct.R'
'verb-do-query.R'
'verb-do.R'
'verb-filter.R'
'verb-group_by.R'
'verb-head.R'
'verb-joins.R'
'verb-mutate.R'
'verb-pull.R'
'verb-select.R'
'verb-set-ops.R'
'verb-slice.R'
'verb-summarise.R'
'verb-window.R'
'zzz.R'

RdMacros lifecycle

R topics documented:

arrange.tbl_lazy 4
backend-access 5
backend-hana 6
backend-hive 6
backend-impala 7
backend-mssql 7
backend-mysql 8
backend-odbc 9
backend-oracle 9
backend-postgres 10
backend-redshift 10
backend-sqlite 11

backend-teradata 11
collapse.tbl_sql 12
copy_to.src_sql 13
dbplyr-slice 14
distinct.tbl_lazy 15
do.tbl_sql 16
escape 16
filter.tbl_lazy 17
group_by.tbl_lazy 18
head.tbl_lazy 19
ident 19
intersect.tbl_lazy 20
in_schema 21
join.tbl_sql 22
memdb_frame 25
mutate.tbl_lazy 25
pull.tbl_sql 26
remote_name 27
select.tbl_lazy 28
sql 29
summarise.tbl_lazy 29
tbl.src_dbi 30
translate_sql 31
window_order 33

Index **35**

`arrange.tbl_lazy` *Arrange rows by column values*

Description

This is an method for the dplyr `arrange()` generic. It generates the ORDER BY clause of the SQL query. It also affects the `window_order()` of windowed expressions in `mutate.tbl_lazy()`.

Note that ORDER BY clauses can not generally appear in subqueries, which means that you should `arrange()` as late as possible in your pipelines.

Usage

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

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Missing values

Unlike R, most databases sorts `NA` (`NULL`s) at the front. You can can override this behaviour by explicitly sorting on `is.na(x)`.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db %>% arrange(a) %>% show_query()

# Note that NAs are sorted first
db %>% arrange(b)
# override by sorting on is.na() first
db %>% arrange(is.na(b), b)
```

`backend-access` *Backend: MS Access*

Description

See `vignette("translate-function")` and `vignette("translate-verb")` for details of overall translation technology. Key differences for this backend are:

- `SELECT` uses `TOP`, not `LIMIT`
- Non-standard types and mathematical functions
- String concatenation uses `&`
- No `ANALYZE` equivalent
- `TRUE` and `FALSE` converted to 1 and 0

Use `simulate_access()` with `lazy_frame()` to see simulated SQL without converting to live access database.

Usage

```
simulate_access()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)
lf <- lazy_frame(x = 1, y = 2, z = "a", con = simulate_access())

lf %>% head()
lf %>% mutate(y = as.numeric(y), z = sqrt(x^2 + 10))
lf %>% mutate(a = paste0(z, " times"))
```

backend-hana *Backend: SAP HANA*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- Temporary tables get # prefix and use LOCAL TEMPORARY COLUMN.
- No table analysis performed in copy_to().
- paste() uses ||
- Note that you can't create new boolean columns from logical expressions; you need to wrap with explicit ifelse: ifelse(x > y,TRUE,FALSE).

Use simulate_hana() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_hana()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_hana())
lf %>% transmute(x = paste0(z, " times"))
```

backend-hive *Backend: Hive*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are a scattering of custom translations provided by users.

Use simulate_hive() with lazy_frame() to see simulated SQL without converting to live access database.

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = simulate_hive())
lf %>% transmute(x = cot(b))
lf %>% transmute(x = bitwShiftL(c, 1L))
lf %>% transmute(x = str_replace_all(z, "a", "b"))

lf %>% summarise(x = median(d, na.rm = TRUE))
lf %>% summarise(x = var(c, na.rm = TRUE))
```

backend-impala *Backend: Impala*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are a scattering of custom translations provided by users, mostly focussed on bitwise operations.

Use simulate_impala() with lazy_frame() to see simulated SQL without converting to live access database.

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_impala())
lf %>% transmute(X = bitwNot(bitwOr(b, c)))
```

backend-mssql *Backend: SQL server*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- SELECT uses TOP not LIMIT
- Automatically prefixes # to create temporary tables. Add the prefix yourself to avoid the message.
- String basics: paste(), substr(), nchar()
- Custom types for as.* functions
- Lubridate extraction functions, year(), month(), day() etc
- Semi-automated bit <-> boolean translation (see below)

Use simulate_mssql() with lazy_frame() to see simulated SQL without converting to live access database.

Arguments

version	Version of MS SQL to simulate. Currently only, difference is that 15.0 and above will use TRY_CAST() instead of CAST().

Bit vs boolean

SQL server uses two incompatible types to represent TRUE and FALSE values:

- The BOOLEAN type is the result of logical comparisons (e.g. x > y) and can be used WHERE but not to create new columns in SELECT. https://docs.microsoft.com/en-us/sql/t-sql/language-elements/comparison-operators-transact-sql
- The BIT type is a special type of numeric column used to store TRUE and FALSE values, but can't be used in WHERE clauses. https://docs.microsoft.com/en-us/sql/t-sql/data-types/bit-transact-sql?view=sql-server-ver15

dbplyr does its best to automatically create the correct type when needed, but can't do it 100% correctly because it does not have a full type inference system. This means that you many need to manually do conversions from time to time.

- To convert from bit to boolean use x == 1
- To convert from boolean to bit use as.logical(if(x, 0, 1))

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_mssql())
lf %>% head()
lf %>% transmute(x = paste(b, c, d))

# Can use boolean as is:
lf %>% filter(c > d)
# Need to convert from boolean to bit:
lf %>% transmute(x = c > d)
# Can use boolean as is:
lf %>% transmute(x = ifelse(c > d, "c", "d"))
```

backend-mysql *Backend: MySQL/MariaDB*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- paste() uses CONCAT_WS()
- String translations for str_detect(), str_locate(), and str_replace_all()
- Clear error message for unsupported full joins

Use simulate_mysql() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_mysql()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_mysql())
lf %>% transmute(x = paste0(z, " times"))
```

---

backend-odbc *Backend: ODBC*

Description

See `vignette("translate-function")` and `vignette("translate-verb")` for details of overall translation technology. Key differences for this backend are minor translations for common data types.

Use `simulate_odbc()` with `lazy_frame()` to see simulated SQL without converting to live access database.

Usage

```
simulate_odbc()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = simulate_odbc())
lf %>% transmute(x = as.numeric(b))
lf %>% transmute(x = as.integer(b))
lf %>% transmute(x = as.character(b))
```

---

backend-oracle *Backend: Oracle*

Description

See `vignette("translate-function")` and `vignette("translate-verb")` for details of overall translation technology. Key differences for this backend are:

- Use FETCH FIRST instead of `LIMIT`
- Custom types
- `paste()` uses `||`
- Custom subquery generation (no `AS`)
- `setdiff()` uses `MINUS` instead of `EXCEPT`

Use `simulate_oracle()` with `lazy_frame()` to see simulated SQL without converting to live access database.

Usage

```
simulate_oracle()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_oracle())
lf %>% transmute(x = paste0(c, " times"))
lf %>% setdiff(lf)
```

---

backend-postgres *Backend: PostgreSQL*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- Many stringr functions
- lubridate date-time extraction functions
- More standard statistical summaries

Use simulate_postgres() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_postgres()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_postgres())
lf %>% summarise(x = sd(b, na.rm = TRUE))
lf %>% summarise(y = cor(b, c), y = cov(b, c))
```

---

backend-redshift *Backend: Redshift*

Description

Base translations come from PostgreSQL backend. There are generally few differences, apart from string manipulation.

Use simulate_redshift() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_redshift()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_redshift())
lf %>% transmute(x = paste(c, " times"))
lf %>% transmute(x = substr(c, 2, 3))
lf %>% transmute(x = str_replace_all(c, "a", "z"))
```

backend-sqlite *Backend: SQLite*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- Uses non-standard LOG() function
- Date-time extraction functions from lubridate
- Custom median translation

Use simulate_sqlite() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_sqlite()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_sqlite())
lf %>% transmute(x = paste(c, " times"))
lf %>% transmute(x = log(b), y = log(b, base = 2))
```

backend-teradata *Backend: Teradata*

Description

See vignette("translate-function") and vignette("translate-verb") for details of overall translation technology. Key differences for this backend are:

- Uses TOP instead of LIMIT
- Selection of user supplied translations

Use simulate_teradata() with lazy_frame() to see simulated SQL without converting to live access database.

Usage

```
simulate_teradata()
```

Examples

```
library(dplyr, warn.conflicts = FALSE)

lf <- lazy_frame(a = TRUE, b = 1, c = 2, d = "z", con = simulate_teradata())
lf %>% head()
```

collapse.tbl_sql *Compute results of a query*

Description

These are methods for the dplyr generics collapse(), compute(), and collect(). collapse() creates a subquery, compute() stores the results in a remote table, and collect() executes the query and downloads the data into R.

Usage

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

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

## S3 method for class 'tbl_sql'
collect(x, ..., n = Inf, warn_incomplete = TRUE)
```

Arguments

x	A lazy data frame backed by a database query.
...	other parameters passed to methods.
name	Table name in remote database.
temporary	Should the table be temporary (TRUE, the default) or persistent (FALSE')?
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.

n	Number of rows to fetch. Defaults to Inf, meaning all rows.
warn_incomplete	Warn if n is less than the number of result rows?

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(a = c(3, 4, 1, 2), b = c(5, 1, 2, NA))
db %>% filter(a <= 2) %>% collect()
```

copy_to.src_sql *Copy a local data frame to a remote database*

Description

This is an implementation of the dplyr copy_to() generic and it mostly a wrapper around DBI::dbWriteTable().

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 only visible within 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,
  ...,
  in_transaction = TRUE
)
```

Arguments

dest	remote data source
df	A local data frame, a tbl_sql from same source, or a tbl_sql from another source. If from another source, all data must transition through R in one pass, so it is only suitable for transferring small amounts of data.
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 https://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.
`in_transaction`	Should the table creation be wrapped in a transaction? This typically makes things faster, but you may want to suppress if the database doesn't support transactions, or you're wrapping in a transaction higher up (and your database doesn't support nested transactions.)

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

df <- data.frame(x = 1:5, y = letters[5:1])
db <- copy_to(src_memdb(), df)
db

df2 <- data.frame(y = c("a", "d"), fruit = c("apple", "date"))
# copy_to() is called automatically if you set copy = TRUE
# in the join functions
db %>% left_join(df2, copy = TRUE)
```

---

dbplyr-slice *Subset rows using their positions*

Description

These are methods for the dplyr generics `slice_min()`, `slice_max()`, and `slice_sample()`. They are translated to SQL using `filter()` and window functions (`ROWNUMBER`, `MIN_RANK`, or `CUME_DIST` depending on arguments). `slice()`, `slice_head()`, and `slice_tail()` are not supported since database tables have no intrinsic order.

If data is grouped, the operation will be performed on each group so that (e.g.) `slice_min(db,x,n = 3)` will select the three rows with the smallest value of `x` in each group.

Usage

```
## S3 method for class 'tbl_lazy'
slice_min(.data, order_by, ..., n, prop, with_ties = TRUE)

## S3 method for class 'tbl_lazy'
slice_max(.data, order_by, ..., n, prop, with_ties = TRUE)

## S3 method for class 'tbl_lazy'
slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE)
```

Arguments

`.data` A lazy data frame backed by a database query.

`order_by` Variable or function of variables to order by.

`...` Not used.

`n, prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used.

If `n` is greater than the number of rows in the group (or `prop` > 1), the result will be silently truncated to the group size. If the proportion of a group size is not an integer, it is rounded down.

`with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use FALSE to ignore ties, and return the first n rows.

`weight_by, replace`
Not supported for database backends.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:3, y = c(1, 1, 2))
db %>% slice_min(x) %>% show_query()
db %>% slice_max(x) %>% show_query()
db %>% slice_sample() %>% show_query()

db %>% group_by(y) %>% slice_min(x) %>% show_query()

# By default, ties are includes so you may get more rows
# than you expect
db %>% slice_min(y, n = 1)
db %>% slice_min(y, n = 1, with_ties = FALSE)

# Non-integer group sizes are rounded down
db %>% slice_min(x, prop = 0.5)
```

`distinct.tbl_lazy` *Subset distinct/unique rows*

Description

This is a method for the dplyr `distinct()` generic. It adds the `DISTINCT` clause to the SQL query.

Usage

```
## S3 method for class 'tbl_lazy'
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

`.data` A lazy data frame backed by a database query.

`...` <`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.

`.keep_all` If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

Value

Another tbl_lazy. Use show_query() to see the generated query, and use collect() to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = c(1, 1, 2, 2), y = c(1, 2, 1, 1))
db %>% distinct() %>% show_query()
db %>% distinct(x) %>% show_query()
```

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() requires you to provide a database connection to control the details of escaping. escape_ansi() uses the SQL 92 ANSI standard.

Usage

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

escape_ansi(x, parens = NA, collapse = "")

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.

Examples

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

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

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

`filter.tbl_lazy` *Subset rows using column values*

Description

This is a method for the dplyr `filter()` generic. It generates the `WHERE` clause of the SQL query.

Usage

```
## S3 method for class 'tbl_lazy'
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data` A lazy data frame backed by a database query.

`...` <`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.

`.preserve` Not supported by this method.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = c(2, NA, 5, NA, 10), y = 1:5)
db %>% filter(x < 5) %>% show_query()
db %>% filter(is.na(x)) %>% show_query()
```

group_by.tbl_lazy — *Group by one or more variables*

Description

This is a method for the dplyr `group_by()` generic. It is translated to the GROUP BY clause of the SQL query when used with `summarise()` and to the PARTITION BY clause of window functions when used with `mutate()`.

Usage

```
## S3 method for class 'tbl_lazy'
group_by(.data, ..., .add = FALSE, add = NULL, .drop = TRUE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called add, but that prevented creating a new grouping variable called add, and conflicts with our naming conventions.
`add`	Deprecated. Please use `.add` instead.
`.drop`	Not supported by this method.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db %>%
  group_by(g) %>%
  summarise(n()) %>%
  show_query()

db %>%
  group_by(g) %>%
  mutate(x2 = x / sum(x, na.rm = TRUE)) %>%
  show_query()
```

head.tbl_lazy *Subset the first rows*

Description

This is a method for the head() generic. It is usually translated to the LIMIT clause of the SQL query. Because LIMIT is not an official part of the SQL specification, some database use other clauses like TOP or FETCH ROWS.

Note that databases don't really have a sense of row order, so what "first" means is subject to interpretation. Most databases will respect ordering performed with arrange(), but it's not guaranteed. tail() is not supported at all because the situation is even murkier for the "last" rows.

Usage

```
## S3 method for class 'tbl_lazy'
head(x, n = 6L, ...)
```

Arguments

x	A lazy data frame backed by a database query.
n	Number of rows to return
...	Not used.

Value

Another tbl_lazy. Use show_query() to see the generated query, and use collect() to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:100)
db %>% head() %>% show_query()

# Pretend we have data in a SQL server database
db2 <- lazy_frame(x = 1:100, con = simulate_mssql())
db2 %>% head() %>% show_query()
```

ident *Flag a character vector as SQL identifiers*

Description

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

Usage

```
ident(...)

is.ident(x)
```

Arguments

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

Examples

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

# And identifiers with "
ident("x")
escape_ansi(ident("x"))

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

`intersect.tbl_lazy` *SQL set operations*

Description

These are methods for the dplyr generics `dplyr::intersect()`, `dplyr::union()`, and `dplyr::setdiff()`. They are translated to `INTERSECT`, `UNION`, and `EXCEPT` respectively.

Usage

```
## S3 method for class 'tbl_lazy'
intersect(x, y, copy = FALSE, ..., all = FALSE)

## S3 method for class 'tbl_lazy'
union(x, y, copy = FALSE, ..., all = FALSE)

## S3 method for class 'tbl_lazy'
union_all(x, y, copy = FALSE, ...)

## S3 method for class 'tbl_lazy'
setdiff(x, y, copy = FALSE, ..., all = FALSE)
```

Arguments

`x`	A pair of lazy data frames backed by database queries.
`y`	A pair of lazy data frames backed by database queries.

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.
...	Not currently used; provided for future extensions.
all	If TRUE, includes all matches in output, not just unique rows.

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. These will be automatically quoted; use sql() to pass a raw name that won't get quoted.

Examples

```
in_schema("my_schema", "my_table")
# eliminate quotes
in_schema(sql("my_schema"), sql("my_table"))

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

# 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 tables*

Description

These are methods for the dplyr join generics. They are translated to the following SQL queries:

- `inner_join(x,y)`: SELECT * FROM x JOIN y ON x.a = y.a
- `left_join(x,y)`: SELECT * FROM x LEFT JOIN y ON x.a = y.a
- `right_join(x,y)`: SELECT * FROM x RIGHT JOIN y ON x.a = y.a
- `full_join(x,y)`: SELECT * FROM x FULL JOIN y ON x.a = y.a
- `semi_join(x,y)`: SELECT * FROM x WHERE EXISTS (SELECT 1 FROM y WHERE x.a = y.a)
- `anti_join(x,y)`: SELECT * FROM x WHERE NOT EXISTS (SELECT 1 FROM y WHERE x.a = y.a)

Usage

```
## S3 method for class 'tbl_lazy'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)

## S3 method for class 'tbl_lazy'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)

## S3 method for class 'tbl_lazy'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)

## S3 method for class 'tbl_lazy'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)

## S3 method for class 'tbl_lazy'
semi_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)

## S3 method for class 'tbl_lazy'
anti_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  auto_index = FALSE,
  ...,
  sql_on = NULL,
  na_matches = c("never", "na")
)
```

Arguments

`x, y` A pair of lazy data frames backed by database queries.

`by` A character vector of variables to join by.

If `NULL`, the default, *_join() will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

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`.

To join by multiple variables, use a vector with length > 1. For example, `by =`

c("a","b") will match x$a to y$a and x$b to y$b. Use a named vector to match different variables in x and y. For example, by = c("a" = "b","c" = "d") will match x$a to y$b and x$c to y$d.

To perform a cross-join, generating all combinations of x and y, use by = character().

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 disambiguate 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.

sql_on A custom join predicate as an SQL expression. Usually joins use column equality, but you can perform more complex queries by supply sql_on which should be a SQL expression that uses LHS and RHS aliases to refer to the left-hand side or right-hand side of the join respectively.

na_matches Should NA (NULL) values match one another? The default, "never", is how databases usually work. "na" makes the joins behave like the dplyr join functions, merge(), match(), and %in%.

Value

Another tbl_lazy. Use show_query() to see the generated query, and use collect() to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

band_db <- tbl_memdb(dplyr::band_members)
instrument_db <- tbl_memdb(dplyr::band_instruments)
band_db %>% left_join(instrument_db) %>% show_query()

# Can join with local data frames by setting copy = TRUE
band_db %>%
  left_join(dplyr::band_instruments, copy = TRUE)

# Unlike R, joins in SQL don't usually match NAs (NULLs)
db <- memdb_frame(x = c(1, 2, NA))
label <- memdb_frame(x = c(1, NA), label = c("one", "missing"))
db %>% left_join(label, by = "x")
# But you can activate R's usual behaviour with the na_matches argument
db %>% left_join(label, by = "x", na_matches = "na")

# By default, joins are equijoins, but you can use `sql_on` to
# express richer relationships
db1 <- memdb_frame(x = 1:5)
db2 <- memdb_frame(x = 1:3, y = letters[1:3])
db1 %>% left_join(db2) %>% show_query()
db1 %>% left_join(db2, sql_on = "LHS.x < RHS.x") %>% show_query()
```

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 = unique_table_name())

tbl_memdb(df, name = deparse(substitute(df)))

src_memdb()
```

Arguments

...	<dynamic-dots> A set of name-value pairs. These arguments are processed with rlang::quos() and support unquote via !! and unquote-splice via !!!. Use := to create columns that start with a dot. Arguments are evaluated sequentially. You can refer to previously created elements directly or using the .data pronoun. An existing .data pronoun, provided e.g. inside dplyr::mutate(), is not available.
df	Data frame to copy
name, .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()

mtcars_db <- tbl_memdb(mtcars)
mtcars_db %>% group_by(cyl) %>% summarise(n = n()) %>% show_query()
```

mutate.tbl_lazy *Create, modify, and delete columns*

Description

These are methods for the dplyr mutate() and transmute() generics. They are translated to computed expressions in the SELECT clause of the SQL query.

Usage

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

Arguments

`.data` A lazy data frame backed by a database query.

`...` <`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:5, y = 5:1)
db %>%
  mutate(a = (x + y) / 2, b = sqrt(x^2L + y^2L)) %>%
  show_query()

# dbplyr automatically creates subqueries as needed
db %>%
  mutate(x1 = x + 1, x2 = x1 * 2) %>%
  show_query()
```

`pull.tbl_sql` *Extract a single column*

Description

This is a method for the dplyr `pull()` generic. It evaluates the query retrieving just the specified column.

Usage

```
## S3 method for class 'tbl_sql'
pull(.data, var = -1)
```

Arguments

`.data` A lazy data frame backed by a database query.

`var` A variable specified as:

- a literal variable name
- a positive integer, giving the position counting from the left
- a negative integer, giving the position counting from the right.

The default returns the last column (on the assumption that's the column you've created most recently).

This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).

Value

A vector of data.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1:5, y = 5:1)
db %>%
  mutate(z = x + y * 2) %>%
  pull()
```

---

remote_name *Metadata about a remote table*

Description

`remote_name()` gives the name remote table, or `NULL` if it's a query. `remote_query()` gives the text of the query, and `remote_query_plan()` the query plan (as computed by the remote database). `remote_src()` and `remote_con()` give the dplyr source and DBI connection respectively.

Usage

```
remote_name(x)

remote_src(x)

remote_con(x)

remote_query(x)

remote_query_plan(x)
```

Arguments

x	Remote table, currently must be a tbl_sql.

Value

The value, or `NULL` if not remote table, or not applicable. For example, computed queries do not have a "name"

Examples

```
mf <- memdb_frame(x = 1:5, y = 5:1, .name = "blorp")
remote_name(mf)
remote_src(mf)
remote_con(mf)
remote_query(mf)

mf2 <- dplyr::filter(mf, x > 3)
remote_name(mf2)
remote_src(mf2)
```

```
remote_con(mf2)
remote_query(mf2)
```

select.tbl_lazy *Subset, rename, and reorder columns using their names*

Description

These are methods for the dplyr `select()`, `rename()`, and `relocate()` generics. They generate the `SELECT` clause of the SQL query.

These functions do not support predicate functions, i.e. you can not use `where(is.numeric)` to select all numeric variables.

Usage

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

## S3 method for class 'tbl_lazy'
rename(.data, ...)

## S3 method for class 'tbl_lazy'
rename_with(.data, .fn, .cols = everything(), ...)

## S3 method for class 'tbl_lazy'
relocate(.data, ..., .before = NULL, .after = NULL)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.
`.fn`	A function used to transform the selected `.cols`. Should return a character vector the same length as the input.
`.cols`	<`tidy-select`> Columns to rename; defaults to all columns.
`.before`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.after`	<`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(x = 1, y = 2, z = 3)
db %>% select(-y) %>% show_query()
db %>% relocate(z) %>% show_query()
db %>% rename(first = x, last = z) %>% 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, con)
```

Arguments

`...`	Character vectors that will be combined into a single SQL expression.
`x`	Object to coerce
`con`	Needed when `x` is directly supplied from the user so that schema specifications can be quoted using the correct identifiers.

summarise.tbl_lazy *Summarise each group to one row*

Description

This is a method for the dplyr `summarise()` generic. It generates the `SELECT` clause of the SQL query, and generally needs to be combined with `group_by()`.

Usage

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

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions or variables. Use `desc()` to sort a variable in descending order.

Value

Another `tbl_lazy`. Use `show_query()` to see the generated query, and use `collect()` to execute the query and return data to R.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db %>%
  summarise(n()) %>%
  show_query()

db %>%
  group_by(g) %>%
  summarise(n()) %>%
  show_query()
```

---

tbl.src_dbi *Use dplyr verbs with a remote database table*

Description

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()`.

Usage

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

Arguments

`src` A `DBIConnection` object produced by `DBI::dbConnect()`.

`from` Either a string (giving a table name), a fully qualified table name created by `in_schema()` or a literal `sql()` string.

`...` Needed for compatibility with generic; currently ignored.

Details

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 verb that is not lazy: `do()` is eager because it must pull the data into R.

Examples

```
library(dplyr)

# Connect to a temporary in-memory SQLite database
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

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

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

# Use `in_schema()` for fully qualified table names
con %>% tbl(in_schema("temp", "mtcars")) %>% head(1)

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

# If you just want a temporary in-memory database, use src_memdb()
src2 <- 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 (requireNamespace("Lahman", quietly = TRUE)) {
batting <- copy_to(con, Lahman::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,
  context = list()
)
```

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.

`context` Use to carry information for special translation cases. For example, MS SQL needs a different conversion for is.na() in WHERE vs. SELECT clauses. Expects a list.

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% NA)
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), con = simulate_dbi())

# 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

These allow you to override the PARTITION BY and ORDER BY clauses of window functions generated by grouped mutates.

Usage

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

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

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	Variables to order by
`from, to`	Bounds of the frame.

Examples

```
library(dplyr, warn.conflicts = FALSE)

db <- memdb_frame(g = rep(1:2, each = 5), y = runif(10), z = 1:10)
db %>%
  window_order(y) %>%
  mutate(z = cumsum(y)) %>%
  show_query()

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

Index

.data, *25*

anti_join.tbl_lazy *(*join.tbl_sql*)*, 22
arrange(), *4*
arrange.tbl_lazy, 4
as.sql *(*sql*)*, 29

backend-access, 5
backend-hana, 6
backend-hive, 6
backend-impala, 7
backend-mssql, 7
backend-mysql, 8
backend-odbc, 9
backend-oracle, 9
backend-postgres, 10
backend-redshift, 10
backend-sqlite, 11
backend-teradata, 11

collapse(), *12*
collapse.tbl_sql, 12
collect(), *5*, *12*, *14*, *16*, *17*, *19*, *24*, *26*, *29*, *30*
collect.tbl_sql *(*collapse.tbl_sql*)*, 12
compute(), *12*, *30*
compute.tbl_sql *(*collapse.tbl_sql*)*, 12
copy_to(), *6*, *13*
copy_to.src_sql, 13

DBI::dbWriteTable(), *13*
dbplyr-slice, 14
desc(), *4*, *15*, *17*, *18*, *26*, *28*, *29*
distinct(), *15*
distinct.tbl_lazy, 15
do(), *30*
do.tbl_sql, 16
dplyr::mutate(), *25*

escape, 16
escape_ansi *(*escape*)*, 16
explain(), *30*

filter(), *14*, *17*
filter.tbl_lazy, 17
full_join.tbl_lazy *(*join.tbl_sql*)*, 22

group_by(), *18*
group_by.tbl_lazy, 18

head(), *19*
head.tbl_lazy, 19

ident, 19
in_schema, 21
in_schema(), *30*
inner_join.tbl_lazy *(*join.tbl_sql*)*, 22
intersect.tbl_lazy, 20
is.ident *(*ident*)*, 19
is.sql *(*sql*)*, 29

join, *22*
join.tbl_sql, 22

left_join.tbl_lazy *(*join.tbl_sql*)*, 22

match(), *24*
memdb_frame, 25
merge(), *24*
mutate(), *18*, *25*
mutate.tbl_lazy, 25
mutate.tbl_lazy(), *4*

partial_eval(), *32*
PostgreSQL backend, *10*
pull(), *26*
pull.tbl_sql, 26

quasiquotation, *26*

relocate(), *28*
relocate.tbl_lazy *(*select.tbl_lazy*)*, 28
remote_con *(*remote_name*)*, 27
remote_name, 27
remote_query *(*remote_name*)*, 27
remote_query_plan *(*remote_name*)*, 27
remote_src *(*remote_name*)*, 27
rename(), *28*
rename.tbl_lazy *(*select.tbl_lazy*)*, 28
rename_with.tbl_lazy *(*select.tbl_lazy*)*, 28
right_join.tbl_lazy *(*join.tbl_sql*)*, 22

rlang::quos(), *25*

select(), *28*
select.tbl_lazy, 28
semi_join.tbl_lazy *(join.tbl_sql)*, 22
setdiff.tbl_lazy *(intersect.tbl_lazy)*, 20
show_query(), *5*, *14*, *16*, *17*, *19*, *24*, *26*, *29*, *30*
simulate_access *(backend-access)*, 5
simulate_hana *(backend-hana)*, 6
simulate_mysql *(backend-mysql)*, 8
simulate_odbc *(backend-odbc)*, 9
simulate_oracle *(backend-oracle)*, 9
simulate_postgres *(backend-postgres)*, 10
simulate_redshift *(backend-redshift)*, 10
simulate_sqlite *(backend-sqlite)*, 11
simulate_teradata *(backend-teradata)*, 11
slice_max(), *14*
slice_max.tbl_lazy *(dbplyr-slice)*, 14
slice_min(), *14*
slice_min.tbl_lazy *(dbplyr-slice)*, 14
slice_sample(), *14*
slice_sample.tbl_lazy *(dbplyr-slice)*, 14
sql, 29
sql(), *30*, *32*
sql_vector *(escape)*, 16
src_memdb *(memdb_frame)*, 25
src_memdb(), *25*
summarise(), *18*, *29*
summarise.tbl_lazy, 29

tbl.src_dbi, 30
tbl_dbi *(tbl.src_dbi)*, 30
tbl_memdb *(memdb_frame)*, 25
tbl_sql, *27*
tibble::tibble(), *25*
translate_sql, 31
translate_sql_ *(translate_sql)*, 31
transmute(), *25*

union.tbl_lazy *(intersect.tbl_lazy)*, 20
union_all.tbl_lazy *(intersect.tbl_lazy)*, 20

window_frame *(window_order)*, 33
window_order, 33
window_order(), *4*