Package 'dbplyr'

September 10, 2025

Type Package

Title A 'dplyr' Back End for Databases

Version 2.5.1

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 (>= 3.6)

Imports blob (>= 1.2.0), cli (>= 3.6.1), DBI (>= 1.1.3), dplyr (>= 1.1.2), glue (>= 1.6.2), lifecycle (>= 1.0.3), magrittr, methods, pillar (>= 1.9.0), purrr (>= 1.0.1), R6 (>= 2.2.2), rlang (>= 1.1.1), tibble (>= 3.2.1), tidyr (>= 1.3.0), tidyselect (>= 1.2.1), utils, vctrs (>= 0.6.3), withr (>= 2.5.0)

Suggests bit64, covr, knitr, Lahman, nycflights13, odbc (>= 1.4.2), RMariaDB (>= 1.2.2), rmarkdown, RPostgres (>= 1.4.5), RPostgreSQL, RSQLite (>= 2.3.8), testthat (>= 3.1.10)

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel TRUE

Encoding UTF-8

Language en-gb

RoxygenNote 7.3.3

Collate 'db-sql.R' 'utils-check.R' 'import-standalone-types-check.R'
'import-standalone-obj-type.R' 'utils.R' 'sql.R' 'escape.R'
'translate-sql-cut.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'
'verb-copy-to.R' 'backend-mssql.R' 'backend-mysql.R'
'backend-odbc.R' 'backend-oracle.R' 'backend-postgres.R'
'backend-postgres-old.R' 'backend-redshift.R'
'backend-snowflake.R' 'backend-spark-sql.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.R' 'dbplyr.R' 'explain.R' 'ident.R'
'import-standalone-s3-register.R' 'join-by-compat.R'
'join-cols-compat.R' 'lazy-join-query.R' 'lazy-ops.R'
'lazy-query.R' 'lazy-select-query.R' 'lazy-set-op-query.R'
'memdb.R' 'optimise-utils.R' 'pillar.R' 'progress.R'
'sql-build.R' 'query-join.R' 'query-select.R'
'query-semi-join.R' 'query-set-op.R' 'query.R' 'reexport.R'
'remote.R' 'rows.R' 'schema.R' 'simulate.R' 'sql-clause.R'
'sql-expr.R' 'src-sql.R' 'src_dbi.R' 'table-name.R'
'tbl-lazy.R' 'tbl-sql.R' 'test-frame.R' 'testthat.R'
'tidyeval-across.R' 'tidyeval.R' 'translate-sql.R'
'utils-format.R' 'verb-arrange.R' 'verb-compute.R'
'verb-count.R' 'verb-distinct.R' 'verb-do-query.R' 'verb-do.R'
'verb-expand.R' 'verb-fill.R' 'verb-filter.R' 'verb-group_by.R'
'verb-head.R' 'verb-joins.R' 'verb-mutate.R'
'verb-pivot-longer.R' 'verb-pivot-wider.R' 'verb-pull.R'
'verb-select.R' 'verb-set-ops.R' 'verb-slice.R'
'verb-summarise.R' 'verb-uncount.R' 'verb-window.R' 'zzz.R'

NeedsCompilation no

Author Hadley Wickham [aut, cre],
Maximilian Girlich [aut],
Edgar Ruiz [aut],
Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2025-09-10 07:51:05 UTC

Contents

arrange.tbl_lazy 4
backend-access 5
backend-hana 5
backend-hive 6

backend-impala . . . 7
backend-mssql . . . 7
backend-mysql . . . 8
backend-odbc . . . 9
backend-oracle . . . 10
backend-postgres . . . 10
backend-redshift . . . 11
backend-snowflake . . . 11
backend-spark-sql . . . 12
backend-sqlite . . . 13
backend-teradata . . . 13
collapse.tbl_sql . . . 14
complete.tbl_lazy . . . 15
copy_inline . . . 16
copy_to.src_sql . . . 17
count.tbl_lazy . . . 18
dbplyr-slice . . . 19
dbplyr_uncount . . . 21
distinct.tbl_lazy . . . 22
do.tbl_sql . . . 22
escape . . . 23
expand.tbl_lazy . . . 24
fill.tbl_lazy . . . 25
filter.tbl_lazy . . . 26
get_returned_rows . . . 27
group_by.tbl_lazy . . . 28
head.tbl_lazy . . . 29
intersect.tbl_lazy . . . 30
join.tbl_sql . . . 30
memdb_frame . . . 35
mutate.tbl_lazy . . . 36
pivot_longer.tbl_lazy . . . 37
pivot_wider.tbl_lazy . . . 39
pull.tbl_sql . . . 42
remote_name . . . 43
replace_na.tbl_lazy . . . 44
rows_insert.tbl_lazy . . . 45
select.tbl_lazy . . . 48
sql . . . 49
sql_options . . . 50
sql_query_insert . . . 51
summarise.tbl_lazy . . . 54
tbl.src_dbi . . . 55
translate_sql . . . 56
window_order . . . 58

Index **60**

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

Description

This is an method for the dplyr `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 of 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 `dplyr::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("translation-function")` and `vignette("translation-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("translation-function")` and `vignette("translation-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 `dplyr::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(d, " times"))
```

---

backend-hive *Backend: Hive*

Description

See `vignette("translation-function")` and `vignette("translation-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.

Usage

```
simulate_hive()
```

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(c, "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("translation-function")` and `vignette("translation-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.

Usage

```
simulate_impala()
```

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("translation-function")` and `vignette("translation-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.

Usage

```
simulate_mssql(version = "15.0")
```

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://learn.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://learn.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("translation-function") and vignette("translation-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()

simulate_mariadb()
```

Examples

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

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

---

backend-odbc *Backend: ODBC*

Description

See `vignette("translation-function")` and `vignette("translation-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("translation-function")` and `vignette("translation-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`

Note that versions of Oracle prior to 23c have limited supported for `TRUE` and `FALSE` and you may need to use `1` and `0` instead. See https://oracle-base.com/articles/23/boolean-data-type-23 for more details.

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("translation-function")` and `vignette("translation-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), z = 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-snowflake *Backend: Snowflake*

Description

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology.

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

Usage

```
simulate_snowflake()
```

Examples

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

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

---

backend-spark-sql *Backend: Databricks Spark SQL*

Description

See `vignette("translation-function")` and `vignette("translation-verb")` for details of overall translation technology. Key differences for this backend are better translation of statistical aggregate functions (e.g. `var()`, `median()`) and use of temporary views instead of temporary tables when copying data.

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

Usage

```
simulate_spark_sql()
```

Examples

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

lf <- lazy_frame(a = TRUE, b = 1, d = 2, c = "z", con = simulate_spark_sql())

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

lf %>% mutate(x = first(c))
lf %>% mutate(x = first(c), .by = d)
```

backend-sqlite *Backend: SQLite*

Description

See `vignette("translation-function")` and `vignette("translation-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
- Right and full joins are simulated using left joins

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("translation-function")` and `vignette("translation-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 `dplyr::collapse()`, `dplyr::compute()`, and `dplyr::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 = NULL,
  temporary = TRUE,
  unique_indexes = list(),
  indexes = list(),
  analyze = TRUE,
  ...,
  cte = FALSE
)

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

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.
cte	**[Experimental]** Use common table expressions in the generated SQL?
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()
```

---

complete.tbl_lazy *Complete a SQL table with missing combinations of data*

Description

Turns implicit missing values into explicit missing values. This is a method for the tidyr::complete() generic.

Usage

```
## S3 method for class 'tbl_lazy'
complete(data, ..., fill = list())
```

Arguments

data	A lazy data frame backed by a database query.
...	Specification of columns to expand. See tidyr::expand for more details.
fill	A named list that for each variable supplies a single value to use instead of NA for missing combinations.

Value

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

Examples

```
df <- memdb_frame(
  group = c(1:2, 1),
  item_id = c(1:2, 2),
  item_name = c("a", "b", "b"),
  value1 = 1:3,
  value2 = 4:6
```

```
)

df %>% tidyr::complete(group, nesting(item_id, item_name))

# You can also choose to fill in missing values
df %>% tidyr::complete(group, nesting(item_id, item_name), fill = list(value1 = 0))
```

`copy_inline` *Use a local data frame in a dbplyr query*

Description

This is an alternative to `dplyr::copy_to()` that does not need write access and is faster for small data.

Usage

```
copy_inline(con, df, types = NULL)
```

Arguments

`con`	A database connection.
`df`	A local data frame. The data is written directly in the SQL query so it should be small.
`types`	A named character vector of SQL data types to use for the columns. The data types are backend specific. For example for Postgres this could be `c(id = "bigint", created_at = "timestamp", values = "integer[]")`. If `NULL`, the default, the types are determined from `df`.

Details

It writes the data directly in the SQL query via the `VALUES` clause.

Value

A `tbl_lazy`.

See Also

`dplyr::copy_to()` to copy the data into a new database table.

Examples

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

copy_inline(con, df)

copy_inline(con, df) %>% dplyr::show_query()
```

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

Description

This is an implementation of the dplyr `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 of new remote table. Use a string to create the table in the current catalog/schema. Use `I()` if you want to create it in a specific catalog/schema, e.g. `I("schema.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 dplyr::show_query() to see the generated query, and use collect() to execute the query and return data to R.

See Also

copy_inline() to use small data in an SQL query without actually writing to a table.

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)
```

---

count.tbl_lazy *Count observations by group*

Description

These are methods for the dplyr dplyr::count() and dplyr::tally() generics. They wrap up group_by.tbl_lazy(), summarise.tbl_lazy() and, optionally, arrange.tbl_lazy().

Usage

```
## S3 method for class 'tbl_lazy'
count(x, ..., wt = NULL, sort = FALSE, name = NULL)

## S3 method for class 'tbl_lazy'
add_count(x, ..., wt = NULL, sort = FALSE, name = NULL, .drop = NULL)

## S3 method for class 'tbl_lazy'
tally(x, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

x	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
...	<data-masking> Variables, or functions of variables. Use desc() to sort a variable in descending order.
wt	<data-masking> Frequency weights. Can be NULL or a variable: • If NULL (the default), counts the number of rows in each group. • If a variable, computes sum(wt) for each group.
sort	If TRUE, will show the largest groups at the top.
name	The name of the new column in the output. If omitted, it will default to n. If there's already a column called n, it will use nn. If there's a column called n and nn, it'll use nnn, and so on, adding ns until it gets a new name.
.drop	Not supported for lazy tables.

Examples

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

db <- memdb_frame(g = c(1, 1, 1, 2, 2), x = c(4, 3, 6, 9, 2))
db %>% count(g) %>% show_query()
db %>% count(g, wt = x) %>% show_query()
db %>% count(g, wt = x, sort = TRUE) %>% show_query()
```

---

dbplyr-slice *Subset rows using their positions*

Description

These are methods for the dplyr generics dplyr::slice_min(), dplyr::slice_max(), and dplyr::slice_sample(). They are translated to SQL using dplyr::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,
```

```
  by = NULL,
  with_ties = TRUE,
  na_rm = TRUE
)

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

## S3 method for class 'tbl_lazy'
slice_sample(.data, ..., n, prop, by = NULL, 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.
`by`	**[Experimental]** <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`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.
`na_rm`	Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.
`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)
```

---

dbplyr_uncount *"Uncount" a database table*

Description

This is a method for the tidyr `uncount()` generic. It uses a temporary table, so your database user needs permissions to create one.

Usage

```
dbplyr_uncount(data, weights, .remove = TRUE, .id = NULL)
```

Arguments

`data`	A lazy data frame backed by a database query.
`weights`	A vector of weights. Evaluated in the context of `data`; supports quasiquotation.
`.remove`	If `TRUE`, and `weights` is the name of a column in `data`, then this column is removed.
`.id`	Supply a string to create a new variable which gives a unique identifier for each created row.

Examples

```
df <- memdb_frame(x = c("a", "b"), n = c(1, 2))
dbplyr_uncount(df, n)
dbplyr_uncount(df, n, .id = "id")

# You can also use constants
dbplyr_uncount(df, 2)

# Or expressions
dbplyr_uncount(df, 2 / n)
```

distinct.tbl_lazy *Subset distinct/unique rows*

Description

This is a method for the dplyr `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 of 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 `dplyr::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, must 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")))
```

`expand.tbl_lazy` *Expand SQL tables to include all possible combinations of values*

Description

This is a method for the tidyr::expand generics. It doesn't sort the result explicitly, so the order might be different to what `expand()` returns for data frames.

Usage

```
## S3 method for class 'tbl_lazy'
expand(data, ..., .name_repair = "check_unique")
```

Arguments

`data`	A lazy data frame backed by a database query.
`...`	Specification of columns to expand. See tidyr::expand for more details.
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic • `"unique_quiet"`: Same as `"unique"`, but "quiet" • `"universal_quiet"`: Same as `"universal"`, but "quiet" • a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R). • A purrr-style anonymous function, see `rlang::as_function()` This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

Value

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

Examples

```
fruits <- memdb_frame(
  type   = c("apple", "orange", "apple", "orange", "orange", "orange"),
  year   = c(2010, 2010, 2012, 2010, 2010, 2012),
  size = c("XS", "S",  "M", "S", "S", "M"),
  weights = rnorm(6)
)

# All possible combinations ---------------------------------------
fruits %>% tidyr::expand(type)
fruits %>% tidyr::expand(type, size)

# Only combinations that already appear in the data ---------------
fruits %>% tidyr::expand(nesting(type, size))
```

`fill.tbl_lazy` *Fill in missing values with previous or next value*

Description

Fill in missing values with previous or next value

Usage

```
## S3 method for class 'tbl_lazy'
fill(.data, ..., .direction = c("down", "up", "updown", "downup"))
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	Columns to fill.
`.direction`	Direction in which to fill missing values. Currently either "down" (the default) or "up". Note that "up" does not work when `.data` is sorted by non-numeric columns. As a workaround revert the order yourself beforehand; for example replace `arrange(x, desc(y))` by `arrange(desc(x), y)`.

Examples

```
squirrels <- tibble::tribble(
  ~group,    ~name,     ~role,     ~n_squirrels, ~ n_squirrels2,
  1,      "Sam",    "Observer",   NA,                 1,
  1,     "Mara", "Scorekeeper",    8,                NA,
  1,    "Jesse",    "Observer",   NA,                NA,
  1,      "Tom",    "Observer",   NA,                 4,
  2,     "Mike",    "Observer",   NA,                NA,
  2,  "Rachael",    "Observer",   NA,                 6,
  2,  "Sydekea", "Scorekeeper",   14,                NA,
```

```
  2, "Gabriela",    "Observer",   NA,               NA,
  3,  "Derrick",    "Observer",   NA,               NA,
  3,     "Kara", "Scorekeeper",    9,                10,
  3,    "Emily",    "Observer",   NA,               NA,
  3, "Danielle",    "Observer",   NA,               NA
)
squirrels$id <- 1:12

tbl_memdb(squirrels) %>%
  window_order(id) %>%
  tidyr::fill(
    n_squirrels,
    n_squirrels2,
  )
```

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

Description

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

Usage

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

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	**[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.preserve`	Not supported by this method.

Value

Another `tbl_lazy`. Use `dplyr::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()
```

---

get_returned_rows *Extract and check the* RETURNING *rows*

Description

[Experimental]

`get_returned_rows()` extracts the RETURNING rows produced by `dplyr::rows_insert()`, `dplyr::rows_append()`, `dplyr::rows_update()`, `dplyr::rows_upsert()`, or `dplyr::rows_delete()` if these are called with the `returning` argument. An error is raised if this information is not available.

`has_returned_rows()` checks if `x` has stored RETURNING rows produced by `dplyr::rows_insert()`, `dplyr::rows_append()`, `dplyr::rows_update()`, `dplyr::rows_upsert()`, or `dplyr::rows_delete()`.

Usage

```
get_returned_rows(x)

has_returned_rows(x)
```

Arguments

`x`	A lazy tbl.

Value

For `get_returned_rows()`, a tibble.

For `has_returned_rows()`, a scalar logical.

Examples

```
library(dplyr)

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbExecute(con, "CREATE TABLE Info (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   number INTEGER
)")
info <- tbl(con, "Info")

rows1 <- copy_inline(con, data.frame(number = c(1, 5)))
rows_insert(info, rows1, conflict = "ignore", in_place = TRUE)
```

```
info

# If the table has an auto incrementing primary key, you can use
# the returning argument + `get_returned_rows()` its value
rows2 <- copy_inline(con, data.frame(number = c(13, 27)))
info <- rows_insert(
  info,
  rows2,
  conflict = "ignore",
  in_place = TRUE,
  returning = id
)
info
get_returned_rows(info)
```

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

Description

This is a method for the dplyr `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 = deprecated(), .drop = TRUE)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<`data-masking`> Variables, or functions of 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 `dplyr::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()
```

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, 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.

join.tbl_sql *Join SQL tables*

Description

These are methods for the dplyr 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,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
right_join(
  x,
```

```
  y,
  by = NULL,
  copy = FALSE,
  suffix = NULL,
  ...,
  keep = NULL,
  na_matches = c("never", "na"),
  multiple = NULL,
  unmatched = "drop",
  relationship = NULL,
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

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

## S3 method for class 'tbl_lazy'
cross_join(
  x,
  y,
  ...,
  copy = FALSE,
  suffix = c(".x", ".y"),
  x_as = NULL,
  y_as = NULL
)

## S3 method for class 'tbl_lazy'
semi_join(
  x,
```

```
  y,
  by = NULL,
  copy = FALSE,
  ...,
  na_matches = c("never", "na"),
  sql_on = NULL,
  auto_index = FALSE,
  x_as = NULL,
  y_as = NULL
)

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

Arguments

x, y	A pair of lazy data frames backed by database queries.
by	A join specification created with join_by(), or 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 on different variables between x and y, use a join_by() specification. For example, join_by(a == b) will match x$a to y$b. To join by multiple variables, use a join_by() specification with multiple expressions. For example, join_by(a == b, c == d) will match x$a to y$b and x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c). join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b"). To perform a cross-join, generating all combinations of x and y, see cross_join().
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 auto-

matically 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.

`...` Other parameters passed onto methods.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`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()`, `bit64::match()`, and `%in%`.

`multiple`, `unmatched`

Unsupported in database backends. As a workaround for multiple use a unique key and for unmatched a foreign key constraint.

`relationship` Unsupported in database backends.

`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.

`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`.

`x_as`, `y_as` Alias to use for `x` resp. `y`. Defaults to `"LHS"` resp. `"RHS"`

Value

Another `tbl_lazy`. Use `dplyr::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 `rlang::!!` and unquote-splice via `rlang::!!!`. Use `:=` to create columns that start with a dot. Arguments are evaluated sequentially. You can refer to previously created elements directly or using the rlang::.data pronoun. To refer explicitly to objects in the calling environment, use `rlang::!!` or rlang::.env, e.g. `!!.data` or `.env$.data` for the special case of an object named `.data`.
`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 `dplyr::mutate()` and `dplyr::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,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<data-masking> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	**[Experimental]** <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.keep`	Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept. • `"all"` retains all columns from `.data`. This is the default. • `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side.

- `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them.
- `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept.

`.before, .after` <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

Value

Another `tbl_lazy`. Use `dplyr::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()
```

---

pivot_longer.tbl_lazy *Pivot data from wide to long*

Description

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `tidyr::pivot_wider()`.

Learn more in `vignette("pivot", "tidyr")`.

While most functionality is identical there are some differences to `pivot_longer()` on local data frames:

- the output is sorted differently/not explicitly,
- the coercion of mixed column types is left to the database,
- `values_ptypes` NOT supported.

Note that `build_longer_spec()` and `pivot_longer_spec()` do not work with remote tables.

Usage

```
## S3 method for class 'tbl_lazy'
pivot_longer(
  data,
  cols,
  ...,
  cols_vary,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = NULL,
  names_transform = NULL,
  names_repair = "check_unique",
  values_to = "value",
  values_drop_na = FALSE,
  values_ptypes,
  values_transform = NULL
)
```

Arguments

data	A data frame to pivot.
cols	Columns to pivot into longer format.
...	Additional arguments passed on to methods.
cols_vary	Unsupported; included for compatibility with the generic.
names_to	A string specifying the name of the column to create from the data stored in the column names of data.
names_prefix	A regular expression used to remove matching text from the start of each variable name.
names_sep, names_pattern	If names_to contains multiple values, these arguments control how the column name is broken up.
names_ptypes	A list of column name-prototype pairs.
names_transform, values_transform	A list of column name-function pairs.
names_repair	What happens if the output has invalid column names?
values_to	A string specifying the name of the column to create from the data stored in cell values. If names_to is a character containing the special .value sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.
values_drop_na	If TRUE, will drop rows that contain only NAs in the value_to column.
values_ptypes	Not supported.

Details

The SQL translation basically works as follows:

1. split the specification by its key columns i.e. by variables crammed into the column names.
2. for each part in the split specification `transmute()` `data` into the following columns

- id columns i.e. columns that are not pivotted
- key columns
- value columns i.e. columns that are pivotted

1. combine all the parts with `union_all()`

Examples

```
# See vignette("pivot") for examples and explanation

# Simplest case where column names are character data
memdb_frame(
  id = c("a", "b"),
  x = 1:2,
  y = 3:4
) %>%
  tidyr::pivot_longer(-id)
```

---

pivot_wider.tbl_lazy *Pivot data from long to wide*

Description

`pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`. Learn more in `vignette("pivot", "tidyr")`.

Note that `pivot_wider()` is not and cannot be lazy because we need to look at the data to figure out what the new column names will be. If you have a long running query you have two options:

- (temporarily) store the result of the query via `compute()`.
- Create a spec before and use `dbplyr_pivot_wider_spec()` - dbplyr's version of `tidyr::pivot_wider_spec()`. Note that this function is only a temporary solution until `pivot_wider_spec()` becomes a generic. It will then be removed soon afterwards.

Usage

```
## S3 method for class 'tbl_lazy'
pivot_wider(
  data,
  ...,
  id_cols = NULL,
  id_expand = FALSE,
  names_from = name,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_vary = "fastest",
  names_expand = FALSE,
  names_repair = "check_unique",
  values_from = value,
  values_fill = NULL,
  values_fn = ~max(.x, na.rm = TRUE),
  unused_fn = NULL
)

dbplyr_pivot_wider_spec(
  data,
  spec,
  ...,
  names_repair = "check_unique",
  id_cols = NULL,
  id_expand = FALSE,
  values_fill = NULL,
  values_fn = ~max(.x, na.rm = TRUE),
  unused_fn = NULL,
  error_call = current_env()
)
```

Arguments

data	A lazy data frame backed by a database query.
...	Unused; included for compatibility with generic.
id_cols	A set of columns that uniquely identifies each observation.
id_expand	Unused; included for compatibility with the generic.
names_from, values_from	A pair of arguments describing which column (or columns) to get the name of the output column (names_from), and which column (or columns) to get the cell values from (values_from). If values_from contains multiple values, the value will be added to the front of the output column.
names_prefix	String added to the start of every variable name.

names_sep
: If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.

names_glue
: Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns (and special `.value`) to create custom column names.

names_sort
: Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.

names_vary
: When `names_from` identifies a column (or columns) with multiple unique values, and multiple `values_from` columns are provided, in what order should the resulting column names be combined?

  - `"fastest"` varies `names_from` values fastest, resulting in a column naming scheme of the form: `value1_name1, value1_name2, value2_name1, value2_name2`. This is the default.
  - `"slowest"` varies `names_from` values slowest, resulting in a column naming scheme of the form: `value1_name1, value2_name1, value1_name2, value2_name2`.

names_expand
: Should the values in the `names_from` columns be expanded by `tidyr::expand()` before pivoting? This results in more columns, the output will contain column names corresponding to a complete expansion of all possible values in `names_from`. Additionally, the column names will be sorted, identical to what `names_sort` would produce.

names_repair
: What happens if the output has invalid column names?

values_fill
: Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing.

values_fn
: A function, the default is `max()`, applied to the `value` in each cell in the output. In contrast to local data frames it must not be `NULL`.

unused_fn
: Optionally, a function applied to summarize the values from the unused columns (i.e. columns not identified by `id_cols`, `names_from`, or `values_from`).

  The default drops all unused columns from the result.

  This can be a named list if you want to apply different aggregations to different unused columns.

  `id_cols` must be supplied for `unused_fn` to be useful, since otherwise all unspecified columns will be considered `id_cols`.

  This is similar to grouping by the `id_cols` then summarizing the unused columns using `unused_fn`.

spec
: A specification data frame. This is useful for more complex pivots because it gives you greater control on how metadata stored in the columns become column names in the result.

  Must be a data frame containing character `.name` and `.value` columns. Additional columns in `spec` should be named to match columns in the long format of the dataset and contain values corresponding to columns pivoted from the wide format. The special `.seq` variable is used to disambiguate rows internally; it is automatically removed after pivoting.

error_call
: The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Details

The big difference to `pivot_wider()` for local data frames is that `values_fn` must not be `NULL`. By default it is `max()` which yields the same results as for local data frames if the combination of `id_cols` and `value` column uniquely identify an observation. Mind that you also do not get a warning if an observation is not uniquely identified.

The translation to SQL code basically works as follows:

1. Get unique keys in `names_from` column.
2. For each key value generate an expression of the form:

```
value_fn(
  CASE WHEN (`names from column` == `key value`)
  THEN (`value column`)
  END
) AS `output column`
```

3. Group data by id columns.
4. Summarise the grouped data with the expressions from step 2.

Examples

```
memdb_frame(
  id = 1,
  key = c("x", "y"),
  value = 1:2
) %>%
  tidyr::pivot_wider(
    id_cols = id,
    names_from = key,
    values_from = value
  )
```

---

pull.tbl_sql *Extract a single column*

Description

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

Usage

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

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).
name	An optional parameter that specifies the column to be used as names for a named vector. Specified in a similar manner as var.
...	<data-masking> Variables, or functions of variables. Use desc() to sort a variable in descending order.

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 unescaped name of the remote table, or NULL if it is a query (created by sql()) or already escape (created by ident_q()). remote_table() gives the remote table or the 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, null_if_local = TRUE)

remote_table(x, null_if_local = TRUE)
```

```
remote_src(x)

remote_con(x)

remote_query(x, cte = FALSE, sql_options = NULL)

remote_query_plan(x, ...)
```

Arguments

x	Remote table, currently must be a tbl_sql.
null_if_local	Return NULL if the remote table is created via tbl_lazy() or lazy_frame()?
cte	**[Deprecated]** Use the sql_options argument instead.
sql_options	**[Experimental]** SQL rendering options generated by sql_options().
...	Additional arguments passed on to methods.

Value

A string, or NULL if not a 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)
```

---

replace_na.tbl_lazy *Replace NAs with specified values*

Description

This is a method for the tidyr::replace_na() generic.

Usage

```
## S3 method for class 'tbl_lazy'
replace_na(data, replace = list(), ...)
```

Arguments

data	A pair of lazy data frame backed by database queries.
replace	A named list of values, with one value for each column that has NA values to be replaced.
...	Unused; included for compatibility with generic.

Value

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

Examples

```
df <- memdb_frame(x = c(1, 2, NA), y = c("a", NA, "b"))
df %>% tidyr::replace_na(list(x = 0, y = "unknown"))
```

---

rows_insert.tbl_lazy *Edit individual rows in the underlying database table*

Description

These are methods for the dplyr dplyr::rows_insert(), dplyr::rows_append(), dplyr::rows_update(), dplyr::rows_patch(), dplyr::rows_upsert(), and dplyr::rows_delete() generics.

When in_place = TRUE these verbs do not generate SELECT queries, but instead directly modify the underlying data using INSERT, UPDATE, or DELETE operators. This will require that you have write access to the database: the connection needs permission to insert, modify or delete rows, but not to alter the structure of the table.

The default, in_place = FALSE, generates equivalent lazy tables (using SELECT queries) that allow previewing the result without actually modifying the underlying table on the database.

Usage

```
## S3 method for class 'tbl_lazy'
rows_insert(
  x,
  y,
  by = NULL,
  ...,
  conflict = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE,
  returning = NULL,
  method = NULL
)
```

```
## S3 method for class 'tbl_lazy'
rows_append(x, y, ..., copy = FALSE, in_place = FALSE, returning = NULL)

## S3 method for class 'tbl_lazy'
rows_update(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE,
  returning = NULL
)

## S3 method for class 'tbl_lazy'
rows_patch(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE,
  returning = NULL
)

## S3 method for class 'tbl_lazy'
rows_upsert(
  x,
  y,
  by = NULL,
  ...,
  copy = FALSE,
  in_place = FALSE,
  returning = NULL,
  method = NULL
)

## S3 method for class 'tbl_lazy'
rows_delete(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE,
```

```
  returning = NULL
)
```

Arguments

x	A lazy table. For `in_place = TRUE`, this must be a table instantiated with `dplyr::tbl()` or `dplyr::compute()`, not to a lazy query. The `remote_name()` function is used to determine the name of the table to be updated.
y	A lazy table, data frame, or data frame extensions (e.g. a tibble).
by	An unnamed character vector giving the key columns. The key columns must exist in both `x` and `y`. Keys typically uniquely identify each row, but this is only enforced for the key values of `y` when `rows_update()`, `rows_patch()`, or `rows_upsert()` are used. By default, we use the first column in `y`, since the first column is a reasonable place to put an identifier variable.
...	Other parameters passed onto methods.
conflict	For `rows_insert()`, how should keys in `y` that conflict with keys in `x` be handled? A conflict arises if there is a key in `y` that already exists in `x`. One of: • `"error"`, the default, is not supported for database tables. To get the same behaviour add a unique index on the `by` columns and use `rows_append()`. • `"ignore"` will ignore rows in `y` with keys that conflict with keys in `x`.
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.
in_place	Should `x` be modified in place? If `FALSE` will generate a `SELECT` query that returns the modified table; if `TRUE` will modify the underlying table using a DML operation (`INSERT`, `UPDATE`, `DELETE` or similar).
returning	Columns to return. See `get_returned_rows()` for details.
method	A string specifying the method to use. This is only relevant for `in_place = TRUE`.
unmatched	For `rows_update()`, `rows_patch()`, and `rows_delete()`, how should keys in `y` that are unmatched by the keys in `x` be handled? One of: • `"error"`, the default, is not supported for database tables. Add a foreign key constraint on the `by` columns of `y` to let the database check this behaviour for you. • `"ignore"` will ignore rows in `y` with keys that are unmatched by the keys in `x`.

Value

A new `tbl_lazy` of the modified data. With `in_place = FALSE`, the result is a lazy query that prints visibly, because the purpose of this operation is to preview the results. With `in_place = TRUE`, `x` is returned invisibly, because the purpose of this operation is the side effect of modifying rows in the table behind `x`.

Examples

```
library(dplyr)

con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbExecute(con, "CREATE TABLE Ponies (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   name TEXT,
   cutie_mark TEXT
)")

ponies <- tbl(con, "Ponies")

applejack <- copy_inline(con, data.frame(
  name = "Apple Jack",
  cutie_mark = "three apples"
))

# The default behavior is to generate a SELECT query
rows_insert(ponies, applejack, conflict = "ignore")
# And the original table is left unchanged:
ponies

# You can also choose to modify the table with in_place = TRUE:
rows_insert(ponies, applejack, conflict = "ignore", in_place = TRUE)
# In this case `rows_insert()` returns nothing and the underlying
# data is modified
ponies
```

---

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

Description

These are methods for the dplyr `dplyr::select()`, `dplyr::rename()`, and `dplyr::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 of 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, .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.

sql_options *Options for generating SQL*

Description

Options for generating SQL

Usage

```
sql_options(cte = FALSE, use_star = TRUE, qualify_all_columns = FALSE)
```

Arguments

cte	If FALSE, the default, subqueries are used. If TRUE common table expressions are used.
use_star	If TRUE, the default, * is used to select all columns of a table. If FALSE all columns are explicitly selected.
qualify_all_columns	If FALSE, the default, columns are only qualified with the table they come from if the same column name appears in multiple tables.

Value

A <dbplyr_sql_options> object.

Examples

```
library(dplyr, warn.conflicts = FALSE)
lf1 <- lazy_frame(key = 1, a = 1, b = 2)
lf2 <- lazy_frame(key = 1, a = 1, c = 3)

result <- left_join(lf1, lf2, by = "key") %>%
  filter(c >= 3)

show_query(result)
sql_options <- sql_options(cte = TRUE, qualify_all_columns = TRUE)
show_query(result, sql_options = sql_options)
```

sql_query_insert *Generate SQL for Insert, Update, Upsert, and Delete*

Description

These functions generate the SQL used in `rows_*(in_place = TRUE)`.

Usage

```
sql_query_insert(
  con,
  table,
  from,
  insert_cols,
  by,
  ...,
  conflict = c("error", "ignore"),
  returning_cols = NULL,
  method = NULL
)

sql_query_append(con, table, from, insert_cols, ..., returning_cols = NULL)

sql_query_update_from(
  con,
  table,
  from,
  by,
  update_values,
  ...,
  returning_cols = NULL
)

sql_query_upsert(
  con,
  table,
  from,
  by,
  update_cols,
  ...,
  returning_cols = NULL,
  method = NULL
)

sql_query_delete(con, table, from, by, ..., returning_cols = NULL)
```

Arguments

con	Database connection.
table	Table to update. Must be a table identifier. Use a string to refer to tables in the current schema/catalog or `I()` to refer to tables in other schemas/catalogs.
from	Table or query that contains the new data. Either a table identifier or SQL.
insert_cols	Names of columns to insert.
by	An unnamed character vector giving the key columns. The key columns must exist in both `x` and `y`. Keys typically uniquely identify each row, but this is only enforced for the key values of `y` when `rows_update()`, `rows_patch()`, or `rows_upsert()` are used. By default, we use the first column in `y`, since the first column is a reasonable place to put an identifier variable.
...	Other parameters passed onto methods.
conflict	For `rows_insert()`, how should keys in `y` that conflict with keys in `x` be handled? A conflict arises if there is a key in `y` that already exists in `x`. One of: • `"error"`, the default, will error if there are any keys in `y` that conflict with keys in `x`. • `"ignore"` will ignore rows in `y` with keys that conflict with keys in `x`.
returning_cols	Optional. Names of columns to return.
method	Optional. The method to use.
update_values	A named SQL vector that specify how to update the columns.
update_cols	Names of columns to update.

Details

Insert Methods

`"where_not_exists"`:

The default for most databases.

```
INSERT INTO x_name
SELECT *
FROM y
WHERE NOT EXISTS <match on by columns>
```

`"on_conflict"`:

Supported by:

- Postgres
- SQLite

This method uses the `ON CONFLICT` clause and therefore requires a unique index on the columns specified in `by`.

Upsert Methods

`"merge"`:

The upsert method according to the SQL standard. It uses the `MERGE` statement

```
MERGE INTO x_name
USING y
  ON <match on by columns>
WHEN MATCHED THEN
  UPDATE SET ...
WHEN NOT MATCHED THEN
  INSERT ...
```

`"on_conflict"`:

Supported by:

- Postgres
- SQLite

This method uses the `ON CONFLICT` clause and therefore requires a unique index on the columns specified in by.

`"cte_update"`:

Supported by:

- Postgres
- SQLite
- Oracle

The classical way to upsert in Postgres and SQLite before support for `ON CONFLICT` was added. The update is done in a CTE clause and the unmatched values are then inserted outside of the CTE.

Value

A SQL query.

Examples

```
sql_query_upsert(
  con = simulate_postgres(),
  table = ident("airlines"),
  from = ident("df"),
  by = "carrier",
  update_cols = "name"
)
```

summarise.tbl_lazy *Summarise each group to one row*

Description

This is a method for the dplyr `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, ..., .by = NULL, .groups = NULL)
```

Arguments

`.data`	A lazy data frame backed by a database query.
`...`	<data-masking> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by`	**[Experimental]** <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.groups`	**[Experimental]** Grouping structure of the result. • "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0. • "drop": All levels of grouping are dropped. • "keep": Same grouping structure as `.data`. When `.groups` is not specified, it defaults to "drop_last". In addition, a message informs you of that choice, unless the result is ungrouped, the option "dplyr.summarise.inform" is set to `FALSE`, or when `summarise()` is called from a function in a package.

Value

Another `tbl_lazy`. Use `dplyr::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 `dplyr::compute()` to run the query and save the results in a temporary table in the database, or use `dplyr::collect()` to retrieve the results to R. You can see the query with `dplyr::show_query()`.

Usage

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

Arguments

`src`	A `DBIConnection` object produced by `DBI::dbConnect()`.
`from`	Either a table identifier or a literal `sql()` string. Use a string to identify a table in the current schema/catalog. We recommend using `I()` to identify a table outside the default catalog or schema, e.g. `I("schema.table")` or `I("catalog.schema.table")`. You can also use `in_schema()`/`in_catalog()` or `DBI::Id()`.
`...`	Passed on to `tbl_sql()`

Details

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

There is one verb that is not lazy: `dplyr::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 `I()` for qualified table names
con %>% tbl(I("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

dbplyr translates commonly used base functions including logical (`!`, `&`, `|`), arithmetic (`^`), and comparison (`!=`) operators, as well as common summary (`mean()`, `var()`), and transformation (`log()`) functions. 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`).

Learn more in `vignette("translation-function")`.

Usage

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

translate_sql_(
  dots,
  con,
  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_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.

Examples

```
con <- simulate_dbi()

# Regular maths is translated in a very straightforward way
```

```
translate_sql(x + 1, con = con)
translate_sql(sin(x) + tan(y), con = con)

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

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

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

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

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

# 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), con = con)

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

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

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

`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(y)) %>%
  show_query()
```

Index

?dplyr_by, *20*, *26*, *36*, *54*
?join_by, *33*

abort(), *41*
add_count.tbl_lazy *(*count.tbl_lazy*)*, 18
anti_join.tbl_lazy *(*join.tbl_sql*)*, 30
arrange(), *20*
arrange.tbl_lazy, 4
arrange.tbl_lazy(), *18*
as.sql *(*sql*)*, 49

backend-access, 5
backend-hana, 5
backend-hive, 6
backend-impala, 7
backend-mssql, 7
backend-mysql, 8
backend-odbc, 9
backend-oracle, 10
backend-postgres, 10
backend-redshift, 11
backend-snowflake, 11
backend-spark-sql, 12
backend-sqlite, 13
backend-teradata, 13
bit64::match(), *34*

collapse.tbl_sql, 14
collect(), *4*, *15*, *18*, *22*, *24*, *26*, *29*, *34*, *37*, *45*, *54*
collect.tbl_sql *(*collapse.tbl_sql*)*, 14
complete.tbl_lazy, 15
compute.tbl_sql *(*collapse.tbl_sql*)*, 14
copy_inline, 16
copy_inline(), *18*
copy_to.src_sql, 17
count.tbl_lazy, 18
cross_join(), *33*
cross_join.tbl_lazy *(*join.tbl_sql*)*, 30

DBI::dbWriteTable(), *17*
DBI::Id(), *55*
dbplyr-slice, 19
dbplyr_pivot_wider_spec *(*pivot_wider.tbl_lazy*)*, 39
dbplyr_uncount, 21
desc(), *4*, *19*, *22*, *26*, *28*, *36*, *43*, *49*, *54*
distinct.tbl_lazy, 22
do.tbl_sql, 22
dplyr::arrange(), *4*
dplyr::collapse(), *14*
dplyr::collect(), *14*, *55*
dplyr::compute(), *14*, *47*, *55*
dplyr::copy_to(), *5*, *16, 17*
dplyr::count(), *18*
dplyr::distinct(), *22*
dplyr::do(), *55*
dplyr::explain(), *55*
dplyr::filter(), *19*, *26*
dplyr::group_by(), *28*
dplyr::join, *30*
dplyr::mutate(), *36*
dplyr::pull(), *42*
dplyr::relocate(), *48*
dplyr::rename(), *48*
dplyr::rows_append(), *27*, *45*
dplyr::rows_delete(), *27*, *45*
dplyr::rows_insert(), *27*, *45*
dplyr::rows_patch(), *45*
dplyr::rows_update(), *27*, *45*
dplyr::rows_upsert(), *27*, *45*
dplyr::select(), *48*
dplyr::show_query(), *4*, *15*, *18*, *22*, *24*, *26*, *29*, *34*, *37*, *45*, *54, 55*
dplyr::slice_max(), *19*
dplyr::slice_min(), *19*
dplyr::slice_sample(), *19*
dplyr::summarise(), *54*
dplyr::tally(), *18*
dplyr::tbl(), *47*

dplyr::transmute(), *36*

escape, 23
escape_ansi *(*escape*)*, 23
expand.tbl_lazy, 24

fill.tbl_lazy, 25
filter.tbl_lazy, 26
full_join.tbl_lazy *(*join.tbl_sql*)*, 30

get_returned_rows, 27
get_returned_rows(), *47*
group_by(), *20*, *26*, *36*, *54*
group_by.tbl_lazy, 28
group_by.tbl_lazy(), *18*

has_returned_rows *(*get_returned_rows*)*, 27
head(), *29*
head.tbl_lazy, 29

in_catalog(), *55*
in_schema(), *55*
inner_join.tbl_lazy *(*join.tbl_sql*)*, 30
intersect.tbl_lazy, 30
is.sql *(*sql*)*, 49

join.tbl_sql, 30
join_by(), *33*

lazy_frame(), *44*
left_join.tbl_lazy *(*join.tbl_sql*)*, 30

memdb_frame, 35
merge(), *34*
mutate(), *28*
mutate.tbl_lazy, 36
mutate.tbl_lazy(), *4*

pivot_longer.tbl_lazy, 37
pivot_wider.tbl_lazy, 39
PostgreSQL backend, *11*
pull.tbl_sql, 42

quasiquotation, *43*

relocate(), *37*
relocate.tbl_lazy *(*select.tbl_lazy*)*, 48
remote_con *(*remote_name*)*, 43
remote_name, 43
remote_name(), *47*
remote_query *(*remote_name*)*, 43
remote_query_plan *(*remote_name*)*, 43
remote_src *(*remote_name*)*, 43
remote_table *(*remote_name*)*, 43
rename.tbl_lazy *(*select.tbl_lazy*)*, 48
rename_with.tbl_lazy *(*select.tbl_lazy*)*, 48
replace_na.tbl_lazy, 44
right_join.tbl_lazy *(*join.tbl_sql*)*, 30
rlang::.data, *35*
rlang::.env, *35*
rlang::as_function(), *24*
rlang::quos(), *35*
rows_append.tbl_lazy *(*rows_insert.tbl_lazy*)*, 45
rows_delete.tbl_lazy *(*rows_insert.tbl_lazy*)*, 45
rows_insert.tbl_lazy, 45
rows_patch.tbl_lazy *(*rows_insert.tbl_lazy*)*, 45
rows_update.tbl_lazy *(*rows_insert.tbl_lazy*)*, 45
rows_upsert.tbl_lazy *(*rows_insert.tbl_lazy*)*, 45

select.tbl_lazy, 48
semi_join.tbl_lazy *(*join.tbl_sql*)*, 30
setdiff.tbl_lazy *(*intersect.tbl_lazy*)*, 30
simulate_access *(*backend-access*)*, 5
simulate_hana *(*backend-hana*)*, 5
simulate_hive *(*backend-hive*)*, 6
simulate_impala *(*backend-impala*)*, 7
simulate_mariadb *(*backend-mysql*)*, 8
simulate_mssql *(*backend-mssql*)*, 7
simulate_mysql *(*backend-mysql*)*, 8
simulate_odbc *(*backend-odbc*)*, 9
simulate_oracle *(*backend-oracle*)*, 10
simulate_postgres *(*backend-postgres*)*, 10
simulate_redshift *(*backend-redshift*)*, 11
simulate_snowflake *(*backend-snowflake*)*, 11
simulate_spark_sql *(*backend-spark-sql*)*, 12
simulate_sqlite *(*backend-sqlite*)*, 13
simulate_teradata *(*backend-teradata*)*, 13
slice_max.tbl_lazy *(*dbplyr-slice*)*, 19
slice_min.tbl_lazy *(*dbplyr-slice*)*, 19
slice_sample.tbl_lazy *(*dbplyr-slice*)*, 19

sql, 49
sql(), *55*
sql_options, 50
sql_options(), *44*
sql_query_append *(*sql_query_insert*)*, 51
sql_query_delete *(*sql_query_insert*)*, 51
sql_query_insert, 51
sql_query_update_from
        *(*sql_query_insert*)*, 51
sql_query_upsert *(*sql_query_insert*)*, 51
sql_vector *(*escape*)*, 23
src_memdb *(*memdb_frame*)*, 35
src_memdb(), *35*
summarise(), *28*
summarise.tbl_lazy, 54
summarise.tbl_lazy(), *18*

tally.tbl_lazy *(*count.tbl_lazy*)*, 18
tbl.src_dbi, 55
tbl_dbi *(*tbl.src_dbi*)*, 55
tbl_lazy(), *44*
tbl_memdb *(*memdb_frame*)*, 35
tbl_sql, *44*
tbl_sql(), *55*
tibble::tibble(), *35*
tidyr::complete(), *15*
tidyr::expand, *15*, *24*
tidyr::expand(), *41*
tidyr::pivot_wider(), *37*
tidyr::replace_na(), *44*
translate_sql, 56
translate_sql_ *(*translate_sql*)*, 56

union.tbl_lazy *(*intersect.tbl_lazy*)*, 30
union_all.tbl_lazy
        *(*intersect.tbl_lazy*)*, 30

vctrs::vec_as_names(), *24*

window_frame *(*window_order*)*, 58
window_order, 58
window_order(), *4*