Package 'dbplyr'

April 18, 2020

Type Package

Title A 'dplyr' Back End for Databases

Version 1.4.3

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,
methods,
purrr (>= 0.2.5),
R6 (>= 2.2.2),
rlang (>= 0.2.0),
tibble (>= 1.4.2),
tidyselect (>= 0.2.4),
utils

Suggests bit64,
covr,
knitr,
Lahman,
nycflights13,
RMariaDB (>= 1.0.2),
rmarkdown,
RPostgres (>= 1.1.3),
RSQLite (>= 2.1.0),
testthat (>= 2.0.0)

VignetteBuilder knitr

Encoding UTF-8

Language en-gb

LazyData yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.0

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-athena.R'
'backend-hive.R'
'backend-impala.R'
'backend-mssql.R'
'backend-mysql.R'
'backend-odbc.R'
'backend-oracle.R'
'backend-postgres.R'
'backend-presto.R'
'backend-sqlite.R'
'backend-teradata.R'
'build-sql.R'
'data-cache.R'
'data-lahman.R'
'data-nycflights13.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'
'remote.R'
'schema.R'
'simulate.R'
'sql-build.R'
'sql-expr.R'
'src-sql.R'
'src_dbi.R'
'tbl-lazy.R'
'tbl-sql.R'

'test-frame.R'
'testthat.R'
'translate-sql-clause.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-summarise.R'
'verb-window.R'
'zzz.R'

RdMacros lifecycle

R topics documented:

`arrange.tbl_lazy` *Arrange rows by variables in a remote database table*

Description

Order rows of database tables by an expression involving its variables.

Usage

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

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	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

An object of the same class as .data.

Missing values

Compared to its sorting behaviour on local data, the arrange() method for most database tables sorts NA at the beginning unless wrapped with desc(). Users can override this behaviour by explicitly sorting on is.na(x).

Examples

```
library(dplyr)

dbplyr::memdb_frame(a = c(3, 4, 1, 2)) %>%
  arrange(a)

# NA sorted first
dbplyr::memdb_frame(a = c(3, 4, NA, 2)) %>%
  arrange(a)

# override by sorting on is.na() first
dbplyr::memdb_frame(a = c(3, 4, NA, 2)) %>%
  arrange(is.na(a), a)
```

---

collapse.tbl_sql *Force computation of query*

Description

collapse() creates a subquery; compute() stores the results in a remote table; collect() downloads the results into the current R session.

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 tbl_sql
...	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?

---

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

Description

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

Usage

```
## S3 method for class 'src_sql'
copy_to(
  dest,
  df,
```

```
  name = deparse(substitute(df)),
  overwrite = FALSE,
  types = NULL,
  temporary = TRUE,
  unique_indexes = NULL,
  indexes = NULL,
  analyze = TRUE,
  ...
)
```

Arguments

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

Value

A tbl() object (invisibly).

Examples

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

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

cyl8 <- mtcars2 %>% filter(cyl == 8)
cyl8_cached <- copy_to(src_memdb(), cyl8)

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

do.tbl_sql *Perform arbitrary computation on remote backend*

Description

Perform arbitrary computation on remote backend

Usage

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

Arguments

.data	a tbl
...	Expressions to apply to each group. If named, results will be stored in a new column. If unnamed, should return a data frame. You can use . to refer to the current group. You can not mix named and unnamed arguments.
.chunk_size	The size of each chunk to pull into R. If this number is too big, the process will be slow because R has to allocate and free a lot of memory. If it's too small, it will be slow, because of the overhead of talking to the database.

escape *Escape/quote a string.*

Description

escape() 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")))
```

`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(...)

ident_q(...)

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")
```

in_schema *Refer to a table in a schema*

Description

Refer to a table in a schema

Usage

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

Arguments

`schema, table`	Names of schema and table.

Examples

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

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

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

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

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

join.tbl_sql *Join sql tbls.*

Description

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

Usage

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

```
  sql_on = NULL
)

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

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

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

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

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

```
  x,
  y,
  by = NULL,
  copy = FALSE,
  auto_index = FALSE,
  ...,
  sql_on = NULL
)
```

Arguments

x	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
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. The SQL can refer to the LHS and RHS aliases to disambiguate column names.

Implementation notes

Semi-joins are implemented using WHERE EXISTS, and anti-joins with WHERE NOT EXISTS.

All joins use column equality by default. An arbitrary join predicate can be specified by passing an SQL expression to the sql_on argument. Use LHS and RHS to refer to the left-hand side or right-hand side table, respectively.

Examples

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

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

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

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

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

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

people <- tbl(lahman_s, "Master")

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

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

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

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

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

# batters without person covariates
anti_join(batting, people)

# Arbitrary predicates --------------------------------------------------------

# Find all pairs of awards given to the same player
```

```
# with at least 18 years between the awards:
awards_players <- tbl(lahman_s, "AwardsPlayers")
inner_join(
  awards_players, awards_players,
  sql_on = paste0(
    "(LHS.playerID = RHS.playerID) AND ",
    "(LHS.yearID < RHS.yearID - 18)"
  )
)
}

## End(Not run)
```

---

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

Description

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

Usage

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

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)
```

sql *SQL escaping.*

Description

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

Usage

```
sql(...)

is.sql(x)

as.sql(x)
```

Arguments

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

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) or 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")

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

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

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

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

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

translate_sql *Translate an expression to sql.*

Description

Translate an expression to sql.

Usage

```
translate_sql(
  ...,
  con = simulate_dbi(),
  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

Override window order and frame

Usage

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

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

Arguments

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

Examples

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

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

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

Index