Package ‘dtplyr’

December 3, 2021

Title Data Table Back-End for 'dplyr'

Version 1.2.0

Description Provides a data.table backend for 'dplyr'. The goal
of 'dtplyr' is to allow you to write 'dplyr' code that is
automatically translated to the equivalent, but usually much faster,
data.table code.

License MIT + file LICENSE

URL https://dtplyr.tidyverse.org, https://github.com/tidyverse/dtplyr

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

Depends R (>= 3.3)

Imports crayon,
data.table (>= 1.13.0),
dplyr (>= 1.0.3),
ellipsis,
glue,
lifecycle,
rlang,
tibble,
tidyselect,
vctrs

Suggests bench,
covr,
knitr,
rmarkdown,
testthat (>= 3.0.0),
tidyr (>= 1.1.0)

VignetteBuilder knitr

Encoding UTF-8

Roxygen {library(tidyr); list(markdown = TRUE)}

RoxygenNote 7.1.2

Config/testthat/edition 3

R topics documented:

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

Description

This is a method for dplyr generic `arrange()`. It is translated to an `order()` call in the `i` argument of [.data.table.

Usage

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

Arguments

`.data`	A `lazy_dt()`.
`...`	<`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.

Examples

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

dt <- lazy_dt(mtcars)
dt %>% arrange(vs, cyl)
dt %>% arrange(desc(vs), cyl)
dt %>% arrange(across(mpg:disp))
```

`collect.dtplyr_step` *Force computation of a lazy data.table*

Description

- `collect()` returns a tibble, grouped if needed.
- `compute()` generates an intermediate assignment in the translation.
- `as.data.table()` returns a data.table.
- `as.data.frame()` returns a data frame.
- `as_tibble()` returns a tibble.

Usage

```
## S3 method for class 'dtplyr_step'
collect(x, ...)

## S3 method for class 'dtplyr_step'
compute(x, name = unique_name(), ...)

## S3 method for class 'dtplyr_step'
as.data.table(x, keep.rownames = FALSE, ...)

## S3 method for class 'dtplyr_step'
as.data.frame(x, ...)

## S3 method for class 'dtplyr_step'
as_tibble(x, ..., .name_repair = "check_unique")
```

Arguments

`x`	A lazy_dt
`...`	Arguments used by other methods.
`name`	Name of intermediate data.table.
`keep.rownames`	Ignored as dplyr never preserves rownames.
`.name_repair`	Treatment of problematic column names

Examples

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

dt <- lazy_dt(mtcars)

# Generate translation
avg_mpg <- dt %>%
  filter(am == 1) %>%
  group_by(cyl) %>%
  summarise(mpg = mean(mpg))

# Show translation and temporarily compute result
avg_mpg

# compute and return tibble
avg_mpg_tb <- as_tibble(avg_mpg)
avg_mpg_tb

# compute and return data.table
avg_mpg_dt <- data.table::as.data.table(avg_mpg)
avg_mpg_dt

# modify translation to use intermediate assignment
compute(avg_mpg)
```

`complete.dtplyr_step` *Complete a data frame with missing combinations of data*

Description

This is a method for the tidyr `complete()` generic. This is a wrapper around `dtplyr` translations for `expand()`, `full_join()`, and `replace_na()` that's useful for completing missing combinations of data.

Usage

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

Arguments

`data`	A `lazy_dt()`.
`...`	Specification of columns to expand. Columns can be atomic vectors or lists. • To find all unique combinations of `x`, `y` and `z`, including those not present in the data, supply each variable as a separate argument: `expand(df,x,y,z)`. • To find only the combinations that occur in the data, use `nesting`: `expand(df,nesting(x,y,z))` • You can combine the two forms. For example, `expand(df,nesting(school_id,student_id),` would produce a row for each present school-student combination for all possible dates.

	When used with factors, `expand()` uses the full set of levels, not just those that appear in the data. If you want to use only the values seen in the data, use `forcats::fct_drop()`. When used with continuous variables, you may need to fill in values that do not appear in the data: to do so use expressions like `year = 2010:2020` or `year = full_seq(year,1)`.
`fill`	A named list that for each variable supplies a single value to use instead of `NA` for missing combinations.

Examples

```
library(tidyr)
tbl <- tibble(x = 1:2, y = 1:2, z = 3:4)
dt <- lazy_dt(tbl)

dt %>%
  complete(x, y)

dt %>%
  complete(x, y, fill = list(z = 10L))
```

`count.dtplyr_step` *Count observations by group*

Description

This is a method for the dplyr `count()` generic. It is translated using `.N` in the `j` argument, and supplying groups to `keyby` as appropriate.

Usage

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

Arguments

`.data`	A `lazy_dt()`
`...`	<`data-masking`> Variables to group by.
`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 error, and require you to specify the name.

Examples

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

dt <- lazy_dt(dplyr::starwars)
dt %>% count(species)
dt %>% count(species, sort = TRUE)
dt %>% count(species, wt = mass, sort = TRUE)
```

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

Description

This is a method for the dplyr `distinct()` generic. It is translated to `data.table::unique.data.table()`.

Usage

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

Arguments

`.data`	A `lazy_dt()`
`...`	<`data-masking`> Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables.
`.keep_all`	If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

Examples

```
library(dplyr, warn.conflicts = FALSE)
df <- lazy_dt(data.frame(
  x = sample(10, 100, replace = TRUE),
  y = sample(10, 100, replace = TRUE)
))

df %>% distinct(x)
df %>% distinct(x, y)
df %>% distinct(x, .keep_all = TRUE)
```

`drop_na.dtplyr_step` *Drop rows containing missing values*

Description

This is a method for the tidyr `drop_na()` generic. It is translated to `data.table::na.omit()`

Usage

```
## S3 method for class 'dtplyr_step'
drop_na(data, ...)
```

Arguments

`data`	A `lazy_dt()`.
`...`	<`tidy-select`> Columns to inspect for missing values.

Examples

```
library(dplyr)
library(tidyr)

dt <- lazy_dt(tibble(x = c(1, 2, NA), y = c("a", NA, "b")))
dt %>% drop_na()
dt %>% drop_na(x)

vars <- "y"
dt %>% drop_na(x, any_of(vars))
```

`expand.dtplyr_step` *Expand data frame to include all possible combinations of values.*

Description

This is a method for the tidyr `expand()` generic. It is translated to `data.table::CJ()`.

Usage

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

Arguments

`data`	A `lazy_dt()`.
`...`	Specification of columns to expand. Columns can be atomic vectors or lists. • To find all unique combinations of `x`, `y` and `z`, including those not present in the data, supply each variable as a separate argument: `expand(df,x,y,z)`. • To find only the combinations that occur in the data, use `nesting`: `expand(df,nesting(x,y,z)`

- You can combine the two forms. For example, `expand(df,nesting(school_id,student_id),` would produce a row for each present school-student combination for all possible dates.

Unlike the data.frame method, this method does not use the full set of levels, just those that appear in the data.

When used with continuous variables, you may need to fill in values that do not appear in the data: to do so use expressions like `year = 2010:2020` or `year = full_seq(year,1)`.

`.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
- 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.

Examples

```
library(tidyr)

fruits <- lazy_dt(tibble(
  type   = c("apple", "orange", "apple", "orange", "orange", "orange"),
  year   = c(2010, 2010, 2012, 2010, 2010, 2012),
  size  =  factor(
    c("XS", "S",  "M", "S", "S", "M"),
    levels = c("XS", "S", "M", "L")
  ),
  weights = rnorm(6, as.numeric(size) + 2)
))

# All possible combinations ---------------------------------------
# Note that only present levels of the factor variable `size` are retained.
fruits %>% expand(type)
fruits %>% expand(type, size)

# This is different from the data frame behaviour:
fruits %>% dplyr::collect() %>% expand(type, size)

# Other uses -------------------------------------------------------
fruits %>% expand(type, size, 2010:2012)

# Use `anti_join()` to determine which observations are missing
all <- fruits %>% expand(type, size, year)
all
all %>% dplyr::anti_join(fruits)

# Use with `right_join()` to fill in missing rows
fruits %>% dplyr::right_join(all)
```

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

Description

This is a method for the tidyr `fill()` generic. It is translated to `data.table::nafill()`. Note that `data.table::nafill()` currently only works for integer and double columns.

Usage

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

Arguments

`data`	A data frame.
`...`	<`tidy-select`> Columns to fill.
`.direction`	Direction in which to fill missing values. Currently either "down" (the default), "up", "downup" (i.e. first down and then up) or "updown" (first up and then down).

Examples

```
library(tidyr)

# Value (year) is recorded only when it changes
sales <- lazy_dt(tibble::tribble(
  ~quarter, ~year, ~sales,
  "Q1",    2000,    66013,
  "Q2",      NA,    69182,
  "Q3",      NA,    53175,
  "Q4",      NA,    21001,
  "Q1",    2001,    46036,
  "Q2",      NA,    58842,
  "Q3",      NA,    44568,
  "Q4",      NA,    50197,
  "Q1",    2002,    39113,
  "Q2",      NA,    41668,
  "Q3",      NA,    30144,
  "Q4",      NA,    52897,
  "Q1",    2004,    32129,
  "Q2",      NA,    67686,
  "Q3",      NA,    31768,
  "Q4",      NA,    49094
))

# `fill()` defaults to replacing missing data from top to bottom
sales %>% fill(year)

# Value (n_squirrels) is missing above and below within a group
squirrels <- lazy_dt(tibble::tribble(
  ~group,    ~name,     ~role,     ~n_squirrels,
  1,      "Sam",    "Observer",   NA,
  1,     "Mara", "Scorekeeper",    8,
  1,    "Jesse",    "Observer",   NA,
  1,      "Tom",    "Observer",   NA,
  2,     "Mike",    "Observer",   NA,
  2,  "Rachael",    "Observer",   NA,
  2,  "Sydekea", "Scorekeeper",   14,
  2, "Gabriela",    "Observer",   NA,
  3,  "Derrick",    "Observer",   NA,
  3,     "Kara", "Scorekeeper",    9,
  3,    "Emily",    "Observer",   NA,
  3, "Danielle",    "Observer",   NA
))

# The values are inconsistently missing by position within the group
# Use .direction = "downup" to fill missing values in both directions
squirrels %>%
  dplyr::group_by(group) %>%
  fill(n_squirrels, .direction = "downup") %>%
  dplyr::ungroup()

# Using `.direction = "updown"` accomplishes the same goal in this example
```

filter.dtplyr_step *Subset rows using column values*

Description

This is a method for the dplyr `arrange()` generic. It is translated to the `i` argument of [.data.table

Usage

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

Arguments

`.data`	A `lazy_dt()`.
`...`	<`data-masking`> Expressions that return a logical value, and are defined in terms of the variables in `.data`. If multiple expressions are included, they are combined with the `&` operator. Only rows for which all conditions evaluate to `TRUE` are kept.
`.preserve`	Ignored

Examples

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

dt <- lazy_dt(mtcars)
dt %>% filter(cyl == 4)
dt %>% filter(vs, am)

dt %>%
  group_by(cyl) %>%
  filter(mpg > mean(mpg))
```

group_by.dtplyr_step *Group and ungroup*

Description

These are methods for dplyr's group_by() and ungroup() generics. Grouping is translated to the either keyby and by argument of [.data.table depending on the value of the arrange argument.

Usage

```
## S3 method for class 'dtplyr_step'
group_by(.data, ..., .add = FALSE, add = deprecated(), arrange = TRUE)

## S3 method for class 'dtplyr_step'
ungroup(.data, ...)
```

Arguments

.data	A lazy_dt()
...	In group_by(), variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate mutate() step before the group_by(). Computations are not allowed in nest_by(). In ungroup(), variables to remove from the grouping.
.add, 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.
arrange	If TRUE, will automatically arrange the output of subsequent grouped operations by group. If FALSE, output order will be left unchanged. In the generated data.table code this switches between using the keyby (TRUE) and by (FALSE) arguments.

Examples

```
library(dplyr, warn.conflicts = FALSE)
dt <- lazy_dt(mtcars)

# group_by() is usually translated to `keyby` so that the groups
# are ordered in the output
dt %>%
 group_by(cyl) %>%
 summarise(mpg = mean(mpg))

# use `arrange = FALSE` to instead use `by` so the original order
# or groups is preserved
dt %>%
 group_by(cyl, arrange = FALSE) %>%
 summarise(mpg = mean(mpg))
```

group_modify.dtplyr_step

Apply a function to each group

Description

These are methods for the dplyr `group_map()` and `group_modify()` generics. They are both translated to [.data.table.

Usage

```
## S3 method for class 'dtplyr_step'
group_modify(.tbl, .f, ..., keep = FALSE)

## S3 method for class 'dtplyr_step'
group_map(.tbl, .f, ..., keep = FALSE)
```

Arguments

`.tbl`	A `lazy_dt()`
`.f`	The name of a two argument function. The first argument is passed `.SD`,the data.table representing the current group; the second argument is passed `.BY`, a list giving the current values of the grouping variables. The function should return a list or data.table.
`...`	Additional arguments passed to `.f`
`keep`	Not supported for lazy_dt.

Value

`group_map()` applies `.f` to each group, returning a list. `group_modify()` replaces each group with the results of `.f`, returning a modified `lazy_dt()`.

Examples

```
library(dplyr)

dt <- lazy_dt(mtcars)

dt %>%
  group_by(cyl) %>%
  group_modify(head, n = 2L)

dt %>%
  group_by(cyl) %>%
  group_map(head, n = 2L)
```

head.dtplyr_step *Subset first or last rows*

Description

These are methods for the base generics head() and tail(). They are not translated.

Usage

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

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

Arguments

x	A lazy_dt()
n	Number of rows to select. Can use a negative number to instead drop rows from the other end.
...	Passed on to head()/tail().

Examples

```
library(dplyr, warn.conflicts = FALSE)
dt <- lazy_dt(data.frame(x = 1:10))

# first three rows
head(dt, 3)
# last three rows
tail(dt, 3)

# drop first three rows
tail(dt, -3)
```

intersect.dtplyr_step *Set operations*

Description

These are methods for the dplyr generics intersect(), union(), union_all(), and setdiff().
They are translated to data.table::fintersect(), data.table::funion(), and data.table::fsetdiff().

Usage

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

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

## S3 method for class 'dtplyr_step'
union_all(x, y, ...)

## S3 method for class 'dtplyr_step'
setdiff(x, y, ...)
```

Arguments

`x, y`	A pair of `lazy_dt()`s.
`...`	Ignored

Examples

```
dt1 <- lazy_dt(data.frame(x = 1:4))
dt2 <- lazy_dt(data.frame(x = c(2, 4, 6)))

intersect(dt1, dt2)
union(dt1, dt2)
setdiff(dt1, dt2)
```

`lazy_dt` *Create a "lazy" data.table for use with dplyr verbs*

Description

A lazy data.table lazy captures the intent of dplyr verbs, only actually performing computation when requested (with `collect()`, `pull()`, `as.data.frame()`, `data.table::as.data.table()`, or `tibble::as_tibble()`). This allows dtplyr to convert dplyr verbs into as few data.table expressions as possible, which leads to a high performance translation.

See `vignette("translation")` for the details of the translation.

Usage

```
lazy_dt(x, name = NULL, immutable = TRUE, key_by = NULL)
```

Arguments

`x`	A data table (or something can can be coerced to a data table).
`name`	Optionally, supply a name to be used in generated expressions. For expert use only.
`immutable`	If `TRUE`, `x` is treated as immutable and will never be modified by any code generated by dtplyr. Alternatively, you can set `immutable = FALSE` to allow dtplyr to modify the input object.

key_by	Set keys for data frame, using select() semantics (e.g. key_by = c(key1,key2). This uses data.table::setkey() to sort the table and build an index. This will considerably improve performance for subsets, summaries, and joins that use the keys. See vignette("datatable-keys-fast-subset") for more details.

Examples

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

# If you have a data.table, using it with any dplyr generic will
# automatically convert it to a lazy_dt object
dt <- data.table::data.table(x = 1:10, y = 10:1)
dt %>% filter(x == y)
dt %>% mutate(z = x + y)

# Note that dtplyr will avoid mutating the input data.table, so the
# previous translation includes an automatic copy(). You can avoid this
# with a manual call to lazy_dt()
dt %>%
  lazy_dt(immutable = FALSE) %>%
  mutate(z = x + y)

# If you have a data frame, you can use lazy_dt() to convert it to
# a data.table:
mtcars2 <- lazy_dt(mtcars)
mtcars2
mtcars2 %>% select(mpg:cyl)
mtcars2 %>% select(x = mpg, y = cyl)
mtcars2 %>% filter(cyl == 4) %>% select(mpg)
mtcars2 %>% select(mpg, cyl) %>% filter(cyl == 4)
mtcars2 %>% mutate(cyl2 = cyl * 2, cyl4 = cyl2 * 2)
mtcars2 %>% transmute(cyl2 = cyl * 2, vs2 = vs * 2)
mtcars2 %>% filter(cyl == 8) %>% mutate(cyl2 = cyl * 2)

# Learn more about translation in vignette("translation")
by_cyl <- mtcars2 %>% group_by(cyl)
by_cyl %>% summarise(mpg = mean(mpg))
by_cyl %>% mutate(mpg = mean(mpg))
by_cyl %>%
  filter(mpg < mean(mpg)) %>%
  summarise(hp = mean(hp))
```

---

left_join.dtplyr_step *Join data tables*

Description

These are methods for the dplyr generics left_join(), right_join(), inner_join(), full_join(), anti_join(), and semi_join(). Left, right, inner, and anti join are translated to the [.data.table equivalent, full joins to data.table::merge.data.table(). Left, right, and full joins are in some cases followed by calls to data.table::setcolorder() and data.table::setnames() to ensure that column order and names match dplyr conventions. Semi-joins don't have a direct data.table equivalent.

Usage

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

Arguments

`x, y`	A pair of `lazy_dt()`s.
`...`	Other parameters passed onto methods.
`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 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.
`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.

Examples

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

band_dt <- lazy_dt(dplyr::band_members)
instrument_dt <- lazy_dt(dplyr::band_instruments)

band_dt %>% left_join(instrument_dt)
band_dt %>% right_join(instrument_dt)
band_dt %>% inner_join(instrument_dt)
band_dt %>% full_join(instrument_dt)

band_dt %>% semi_join(instrument_dt)
band_dt %>% anti_join(instrument_dt)
```

`mutate.dtplyr_step` *Create and modify columns*

Description

This is a method for the dplyr `mutate()` generic. It is translated to the `j` argument of [.data.table, using := to modify "in place". If `.before` or `.after` is provided, the new columns are relocated with a call to `data.table::setcolorder()`.

Usage

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

Arguments

`.data`	A `lazy_dt()`.
`...`	<data-masking> Name-value pairs. The name gives the name of the column in the output, and the value should evaluate to a vector.
`.before, .after`	**[Experimental]** <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

Examples

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

dt <- lazy_dt(data.frame(x = 1:5, y = 5:1))
dt %>%
  mutate(a = (x + y) / 2, b = sqrt(x^2 + y^2))

# It uses a more sophisticated translation when newly created variables
# are used in the same expression
dt %>%
  mutate(x1 = x + 1, x2 = x1 + 1)
```

`nest.dtplyr_step` *Nest*

Description

This is a method for the tidyr `tidyr::nest()` generic. It is translated using the non-nested variables in the by argument and `.SD` in the `j` argument.

Usage

```
## S3 method for class 'dtplyr_step'
nest(.data, ..., .names_sep = NULL, .key = deprecated())
```

Arguments

`.data`	A data frame.
`...`	<`tidy-select`> Columns to nest, specified using name-variable pairs of the form `new_col = c(col1,col2,col3)`. The right hand side can be any valid tidy select expression.
`.names_sep`	If `NULL`, the default, the names will be left as is. In `nest()`, inner names will come from the former outer names; in `unnest()`, the new outer names will come from the inner names. If a string, the inner and outer names will be used together. In `nest()`, the names of the new outer columns will be formed by pasting together the outer

	and the inner column names, separated by names_sep. In unnest(), the new inner names will have the outer names (+ names_sep) automatically stripped. This makes names_sep roughly symmetric between nesting and unnesting.
.key	Not supported.
data	A lazy_dt().

Examples

```
if (require("tidyr", quietly = TRUE)) {
  dt <- lazy_dt(tibble(x = c(1, 2, 1), y = c("a", "a", "b")))
  dt %>% nest(data = y)

  dt %>% dplyr::group_by(x) %>% nest()
}
```

---

pivot_longer.dtplyr_step *Pivot data from wide to long*

Description

This is a method for the tidyr pivot_longer() generic. It is translated to data.table::melt()

Usage

```
## S3 method for class 'dtplyr_step'
pivot_longer(
  data,
  cols,
  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 = NULL,
  values_transform = NULL,
  ...
)
```

Arguments

data	A lazy_dt().
cols	<tidy-select> Columns to pivot into longer format.
names_to	A string specifying the name of the column to create from the data stored in the column names of data. Can be a character vector, creating multiple columns, if names_sep or names_pattern is provided. In this case, there are two special values you can take advantage of:

	• `NA` will discard that component of the name. • `.value` indicates that component of the name defines the name of the column containing the cell values, overriding `values_to`.
`names_prefix`	A regular expression used to remove matching text from the start of each variable name.
`names_sep`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (()). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_pattern`	If `names_to` contains multiple values, these arguments control how the column name is broken up. `names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on). `names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (()). If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.
`names_ptypes, names_transform, values_ptypes, values_transform`	Not currently supported by dtplyr.
`names_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.
`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 `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in `data` were created by its structure.
`...`	Additional arguments passed on to methods.

Examples

```
library(tidyr)

# Simplest case where column names are character data
relig_income_dt <- lazy_dt(relig_income)
relig_income_dt %>%
  pivot_longer(!religion, names_to = "income", values_to = "count")

# Slightly more complex case where columns have common prefix,
# and missing missings are structural so should be dropped.
```

```
billboard_dt <- lazy_dt(billboard)
billboard %>%
 pivot_longer(
   cols = starts_with("wk"),
   names_to = "week",
   names_prefix = "wk",
   values_to = "rank",
   values_drop_na = TRUE
 )

# Multiple variables stored in column names
lazy_dt(who) %>%
  pivot_longer(
    cols = new_sp_m014:newrel_f65,
    names_to = c("diagnosis", "gender", "age"),
    names_pattern = "new_?(.*)_(.)(.*)",
    values_to = "count"
  )

# Multiple observations per row
anscombe_dt <- lazy_dt(anscombe)
anscombe_dt %>%
 pivot_longer(
   everything(),
   names_to = c(".value", "set"),
   names_pattern = "(.)(.)"
 )
```

---

pivot_wider.dtplyr_step

Pivot data from long to wide

Description

This is a method for the tidyr `pivot_wider()` generic. It is translated to `data.table::dcast()`

Usage

```
## S3 method for class 'dtplyr_step'
pivot_wider(
  data,
  id_cols = NULL,
  names_from = name,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  names_sort = FALSE,
  names_repair = "check_unique",
  values_from = value,
  values_fill = NULL,
  values_fn = NULL,
  ...
)
```

Arguments

`data`	A `lazy_dt()`.
`id_cols`	<`tidy-select`> A set of columns that uniquely identifies each observation. Defaults to all columns in `data` except for the columns specified in `names_from` and `values_from`. Typically used when you have redundant variables, i.e. variables whose values are perfectly correlated with existing variables.
`names_from`	<`tidy-select`> 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. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.
`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_repair`	What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.
`values_from`	<`tidy-select`> 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.
`values_fill`	Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing. This can be a named list if you want to apply different aggregations to different value columns.
`values_fn`	A function, the default is `length()`. Note this is different behavior than `tidyr::pivot_wider()`, which returns a list column by default.
`...`	Additional arguments passed on to methods.

Examples

```
library(tidyr)

fish_encounters_dt <- lazy_dt(fish_encounters)
fish_encounters_dt
fish_encounters_dt %>%
  pivot_wider(names_from = station, values_from = seen)
# Fill in missing values
fish_encounters_dt %>%
  pivot_wider(names_from = station, values_from = seen, values_fill = 0)
```

```
# Generate column names from multiple variables
us_rent_income_dt <- lazy_dt(us_rent_income)
us_rent_income_dt
us_rent_income_dt %>%
  pivot_wider(names_from = variable, values_from = c(estimate, moe))

# When there are multiple `names_from` or `values_from`, you can use
# use `names_sep` or `names_glue` to control the output variable names
us_rent_income_dt %>%
  pivot_wider(
    names_from = variable,
    names_sep = ".",
    values_from = c(estimate, moe)
  )

# Can perform aggregation with values_fn
warpbreaks_dt <- lazy_dt(as_tibble(warpbreaks[c("wool", "tension", "breaks")]))
warpbreaks_dt
warpbreaks_dt %>%
  pivot_wider(
    names_from = wool,
    values_from = breaks,
    values_fn = mean
  )
```

---

relocate.dtplyr_step *Relocate variables using their names*

Description

This is a method for the dplyr `relocate()` generic. It is translated to the `j` argument of [.data.table.

Usage

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

Arguments

`.data`	A `lazy_dt()`.
`...`	<`tidy-select`> Columns to move.
`.before`	<`tidy-select`> Destination of columns selected by .... Supplying neither will move columns to the left-hand side; specifying both is an error.
`.after`	<`tidy-select`> Destination of columns selected by .... Supplying neither will move columns to the left-hand side; specifying both is an error.

Examples

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

dt <- lazy_dt(data.frame(x = 1, y = 2, z = 3))
dt %>% relocate(z)
dt %>% relocate(y, .before = x)
dt %>% relocate(y, .after = y)
```

`rename.dtplyr_step` *Rename columns using their names*

Description

These are methods for the dplyr generics `rename()` and `rename_with()`. They are both translated to `data.table::setnames()`.

Usage

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

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

Arguments

`.data`	A `lazy_dt()`
`...`	For `rename()`: <`tidy-select`> Use `new_name = old_name` to rename selected variables. For `rename_with()`: additional arguments passed onto `.fn`.
`.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.

Examples

```
library(dplyr, warn.conflicts = FALSE)
dt <- lazy_dt(data.frame(x = 1, y = 2, z = 3))
dt %>% rename(new_x = x, new_y = y)
dt %>% rename_with(toupper)
```

`replace_na.dtplyr_step` *Replace NAs with specified values*

Description

This is a method for the tidyr `replace_na()` generic. It is translated to `data.table::fcoalesce()`.

Note that unlike `tidyr::replace_na()`, `data.table::fcoalesce()` cannot replace `NULL` values in lists.

Usage

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

Arguments

`data`	A `lazy_dt()`.
`replace`	If `data` is a data frame, `replace` takes a list of values, with one value for each column that has `NA` values to be replaced. If `data` is a vector, `replace` takes a single value. This single value replaces all of the `NA` values in the vector.

Examples

```
library(tidyr)

# Replace NAs in a data frame
dt <- lazy_dt(tibble(x = c(1, 2, NA), y = c("a", NA, "b")))
dt %>% replace_na(list(x = 0, y = "unknown"))

# Replace NAs using `dplyr::mutate()`
dt %>% dplyr::mutate(x = replace_na(x, 0))
```

`select.dtplyr_step` *Subset columns using their names*

Description

This is a method for the dplyr `select()` generic. It is translated to the `j` argument of [.data.table.

Usage

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

Arguments

`.data`	A `lazy_dt()`.
`...`	<`tidy-select`> One or more unquoted expressions separated by commas. Variable names can be used as if they were positions in the data frame, so expressions like `x:y` can be used to select a range of variables.

Examples

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

dt <- lazy_dt(data.frame(x1 = 1, x2 = 2, y1 = 3, y2 = 4))

dt %>% select(starts_with("x"))
dt %>% select(ends_with("2"))
dt %>% select(z1 = x1, z2 = x2)
```

separate.dtplyr_step *Separate a character column into multiple columns with a regular expression or numeric locations*

Description

This is a method for the `tidyr::separate()` generic. It is translated to `data.table::tstrsplit()` in the `j` argument of [.data.table.

Usage

```
## S3 method for class 'dtplyr_step'
separate(
  data,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  ...
)
```

Arguments

`data`	A `lazy_dt()`.
`col`	Column name or position. This argument is passed by expression and supports quasiquotation (you can unquote column names or column positions).
`into`	Names of new variables to create as character vector. Use `NA` to omit the variable in the output.
`sep`	Separator between columns. The default value is a regular expression that matches any sequence of non-alphanumeric values.
`remove`	If TRUE, remove the input column from the output data frame.
`convert`	If TRUE, will run type.convert() with as.is = TRUE on new columns. This is useful if the component columns are integer, numeric or logical. NB: this will cause string "NA"s to be converted to NAs.
`...`	Arguments passed on to methods

Examples

```
library(tidyr)
# If you want to split by any non-alphanumeric value (the default):
df <- lazy_dt(data.frame(x = c(NA, "x.y", "x.z", "y.z")), "DT")
df %>% separate(x, c("A", "B"))

# If you just want the second variable:
df %>% separate(x, c(NA, "B"))

# Use regular expressions to separate on multiple characters:
df <- lazy_dt(data.frame(x = c(NA, "x?y", "x.z", "y:z")), "DT")
df %>% separate(x, c("A","B"), sep = "([.?:])")

# convert = TRUE detects column classes:
df <- lazy_dt(data.frame(x = c("x:1", "x:2", "y:4", "z", NA)), "DT")
df %>% separate(x, c("key","value"), ":") %>% str
df %>% separate(x, c("key","value"), ":", convert = TRUE) %>% str
```

---

slice.dtplyr_step *Subset rows using their positions*

Description

These are methods for the dplyr `slice()`, `slice_head()`, `slice_tail()`, `slice_min()`, `slice_max()` and `slice_sample()` generics. They are translated to the `i` argument of [.data.table.

Unlike dplyr, `slice()` (and `slice()` alone) returns the same number of rows per group, regardless of whether or not the indices appear in each group.

Usage

```
## S3 method for class 'dtplyr_step'
slice(.data, ...)

## S3 method for class 'dtplyr_step'
slice_head(.data, ..., n, prop)

## S3 method for class 'dtplyr_step'
slice_tail(.data, ..., n, prop)

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

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

Arguments

`.data`	A `lazy_dt()`.
`...`	Positive integers giving rows to select, or negative integers giving rows to drop.
`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 a negative value of `n` or `prop` is provided, the specified number or proportion of rows will be removed. 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 does not yield an integer number of rows, the absolute value of `prop*n()` is rounded down.
`order_by`	Variable or function of variables to order 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.

Examples

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

dt <- lazy_dt(mtcars)
dt %>% slice(1, 5, 10)
dt %>% slice(-(1:4))

# First and last rows based on existing order
dt %>% slice_head(n = 5)
dt %>% slice_tail(n = 5)

# Rows with minimum and maximum values of a variable
dt %>% slice_min(mpg, n = 5)
dt %>% slice_max(mpg, n = 5)

# slice_min() and slice_max() may return more rows than requested
# in the presence of ties. Use with_ties = FALSE to suppress
dt %>% slice_min(cyl, n = 1)
dt %>% slice_min(cyl, n = 1, with_ties = FALSE)

# slice_sample() allows you to random select with or without replacement
dt %>% slice_sample(n = 5)
dt %>% slice_sample(n = 5, replace = TRUE)

# you can optionally weight by a variable - this code weights by the
# physical weight of the cars, so heavy cars are more likely to get
# selected
dt %>% slice_sample(weight_by = wt, n = 5)
```

---

summarise.dtplyr_step *Summarise each group to one row*

Description

This is a method for the dplyr summarise() generic. It is translated to the j argument of [.data.table.

Usage

```
## S3 method for class 'dtplyr_step'
summarise(.data, ..., .groups = NULL)
```

Arguments

.data	A lazy_dt().
...	<data-masking> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. min(x), n(), or sum(is.na(y)). • A vector of length n, e.g. quantile(). • A data frame, to add multiple columns from a single expression.
.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.

Examples

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

dt <- lazy_dt(mtcars)

dt %>%
  group_by(cyl) %>%
  summarise(vs = mean(vs))

dt %>%
  group_by(cyl) %>%
  summarise(across(disp:wt, mean))
```

`transmute.dtplyr_step` *Create new columns, dropping old*

Description

This is a method for the dplyr `transmute()` generic. It is translated to the `j` argument of [.data.table.

Usage

```
## S3 method for class 'dtplyr_step'
transmute(.data, ...)
```

Arguments

`.data`	A `lazy_dt()`.
`...`	<data-masking> Name-value pairs. The name gives the name of the column in the output, and the value should evaluate to a vector.

Examples

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

dt <- lazy_dt(dplyr::starwars)
dt %>% transmute(name, sh = paste0(species, "/", homeworld))
```