

Package: tidypolars (via r-universe)

August 3, 2026

Type Package

Title More Efficient Tidyverse Code, Using Polars in the Background

Version 0.19.0

Description Polars is a cross-language tool for manipulating very large data. However, one drawback is that the R implementation has a syntax that will look odd to many R users who are not used to Python syntax. The objective of tidypolars is to improve the ease-of-use of Polars in R by providing tidyverse syntax to polars.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

URL <https://tidypolars.etiennebacher.com>,
<https://etiennebacher.r-universe.dev/tidypolars>

BugReports <https://github.com/etiennebacher/tidypolars/issues>

Depends R (>= 4.3.0)

Imports cli, dplyr (> 1.1.4), glue, lifecycle, polars (>= 1.13.0),
rlang (>= 1.1.0), tidyr, tidyselect, vctrs

Suggests arrow, bench, data.table, fs, knitr, jsonlite, lubridate,
nanoparquet, patrick, quickcheck, rmarkdown, rstudioapi,
stringr, testthat (>= 3.0.0), tibble, withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel true

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libicu-dev

Repository <https://test.r-universe.dev>

Date/Publication 2026-07-04 13:44:40 UTC

RemoteUrl <https://github.com/etiennebacher/tidypolars>

RemoteRef v0.19.0

RemoteSha e79dda50194dfd00a43d66afedd57072cab2ef54

Contents

.tp	3
arrange.polars_data_frame	4
bind_cols.polars	5
bind_rows.polars	5
complete.polars_data_frame	6
compute.polars_lazy_frame	8
count.polars_data_frame	10
cross_join.polars_data_frame	11
distinct.polars_data_frame	13
drop_na.polars_data_frame	14
explain.polars_lazy_frame	14
fetch	15
fill.polars_data_frame	17
filter.polars_data_frame	18
from_csv	20
from_ipc	24
from_ndjson	26
from_parquet	29
group_by.polars_data_frame	32
group_split.polars_data_frame	33
group_vars.polars_data_frame	34
left_join.polars_data_frame	35
make_unique_id	39
mutate.polars_data_frame	40
partitioned_output	42
pivot_longer.polars_data_frame	44
pivot_wider.polars_data_frame	45
pull.polars_data_frame	47
relocate.polars_data_frame	48
rename.polars_data_frame	49
replace_na.polars_data_frame	50
rowwise.polars_data_frame	51
select.polars_data_frame	52
semi_join.polars_data_frame	52
separate.polars_data_frame	54
separate_longer	55
sink_csv	57
sink_ipc	60
sink_ndjson	62
sink_parquet	64
slice_tail.polars_data_frame	67

summarize.polars_data_frame	68
summary.polars_data_frame	69
tidypolars_options	70
uncount.polars_data_frame	71
unite.polars_data_frame	72
unnest_longer_polars	73
write_csv_polars	75
write_ipc_polars	77
write_json_polars	78
write_ndjson_polars	79
write_parquet_polars	79

Index	82
--------------	-----------

.tp	<i>Get tidypolars function translation without loading their original package</i>
-----	---

Description

Use `.tp$function_name()` to get access to the functions that are translated by `tidypolars` without loading the package these functions originally come from.

This may be useful in cases where you want to benefit from the interface of these functions but don't want to add some tidyverse dependencies to your project (e.g. `stringr` because it might be slow to build the package in some cases).

Note that the name of the package that originally provided the function must be appended to the function name. For instance, if you want to use `stringr::str_extract()` without loading `stringr`, you can do so with `.tp$str_extract_stringr()`. This is because multiple packages may have a function named `str_extract()`, so we need to inform `tidypolars` of which translation we want exactly.

Note: using `.tp` will make it harder to convert `tidypolars` code to run with other tidyverse-based backends because `.tp` will be unknown to those backends. If you expect to switch between `tidypolars`, the original tidyverse, and tidyverse-based backends, you should avoid using `.tp` and load the original packages in the session instead.

This is similar to the `dd` object in `duckplyr` and to the `.sql` object in `dbplyr`.

Usage

```
.tp
```

Examples

```
# List of all functions stored in this object
sort(names(.tp))

dat <- polars::pl$DataFrame(x = c("abc12", "def3"))
dat |>
```

```
mutate(y = .tp$str_extract_stringr(x, "\\d+"))
```

```
arrange.polars_data_frame
```

Order rows using column values

Description

Order rows using column values

Usage

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

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	Variables, or functions of variables. Use <code>desc()</code> to sort a variable in descending order.
<code>.by_group</code>	If TRUE, will sort data within groups.

Examples

```
pl_test <- polars::pl$DataFrame(
  x1 = c("a", "a", "b", "a", "c"),
  x2 = c(2, 1, 5, 3, 1),
  value = sample(1:5)
)

arrange(pl_test, x1)
arrange(pl_test, x1, -x2)

# if the data is grouped, you need to specify `by_group = TRUE` to sort by
# the groups first
pl_test |>
  group_by(x1) |>
  arrange(-x2, .by_group = TRUE)
```

bind_cols_polars	<i>Append multiple Data/LazyFrames next to each other</i>
------------------	---

Description

Append multiple Data/LazyFrames next to each other

Usage

```
bind_cols_polars(..., .name_repair = "unique")
```

Arguments

...	Polars DataFrames or LazyFrames to combine. Each argument can either be a Data/LazyFrame, or a list of Data/LazyFrames. Columns are matched by name. All Data/LazyFrames must have the same number of rows and there mustn't be duplicated column names.
.name_repair	Can be "unique", "universal", "check_unique", "minimal". See vctrs::vec_as_names() for the explanations for each value.

Examples

```
p1 <- polars::pl$DataFrame(
  x = sample(letters, 20),
  y = sample(1:100, 20)
)
p2 <- polars::pl$DataFrame(
  z = sample(letters, 20),
  w = sample(1:100, 20)
)

bind_cols_polars(p1, p2)
bind_cols_polars(list(p1, p2))
```

bind_rows_polars	<i>Stack multiple Data/LazyFrames on top of each other</i>
------------------	--

Description

Stack multiple Data/LazyFrames on top of each other

Usage

```
bind_rows_polars(..., .id = NULL)
```

Arguments

`...` Polars DataFrames or LazyFrames to combine. Each argument can either be a Data/LazyFrame, or a list of Data/LazyFrames. Columns are matched by name, and any missing columns will be filled with NA.

`.id` The name of an optional identifier column. Provide a string to create an output column that identifies each input. If all elements in `...` are named, the identifier will use their names. Otherwise, it will be a simple count.

Examples

```
library(polars)
p1 <- pl$DataFrame(
  x = c("a", "b"),
  y = 1:2
)
p2 <- pl$DataFrame(
  y = 3:4,
  z = c("c", "d")
)$with_columns(pl$col("y")$cast(pl$Int16))

bind_rows_polars(p1, p2)

# this is equivalent
bind_rows_polars(list(p1, p2))

# create an id column
bind_rows_polars(p1, p2, .id = "id")

# create an id column with named elements
bind_rows_polars(p1 = p1, p2 = p2, .id = "id")
```

complete.polars_data_frame

Complete a data frame with missing combinations of data

Description

Turns implicit missing values into explicit missing values. This is useful for completing missing combinations of data.

Usage

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

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

Arguments

data	A Polars Data/LazyFrame
...	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc. 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 <code>year = 2010:2020</code> or <code>year = full_seq(year, 1)</code> .
fill	A named list that for each variable supplies a single value to use instead of NA for missing combinations.
explicit	Should both implicit (newly created) and explicit (pre-existing) missing values be filled by <code>fill</code> ? By default, this is TRUE, but if set to FALSE this will limit the fill to only implicit missing values.

Examples

```
df <- polars::pl$DataFrame(
  group = c(1:2, 1, 2),
  item_id = c(1:2, 2, 3),
  item_name = c("a", "a", "b", "b"),
  value1 = c(1, NA, 3, 4),
  value2 = 4:7
)
df

df |> complete(group, item_id, item_name)

# Use `fill` to replace NAs with some value. By default, affects both new
# (implicit) and pre-existing (explicit) missing values.
df |>
  complete(
    group, item_id, item_name,
    fill = list(value1 = 0, value2 = 99)
  )

# Limit the fill to only the newly created (i.e. previously implicit)
# missing values with `explicit = FALSE`
df |>
  complete(
    group, item_id, item_name,
    fill = list(value1 = 0, value2 = 99),
    explicit = FALSE
  )

df |>
  group_by(group, maintain_order = TRUE) |>
  complete(item_id, item_name)
```

`compute.polars_lazy_frame`*Run computations on a LazyFrame*

Description

`collect()` and `compute()` can be applied on a `LazyFrame` only. They both check the validity of the query (for instance raising an error if a string operation would be applied on a numeric column), optimize it in the background, and perform computations.

These two functions differ in their output type:

- `compute()` returns a [Polars DataFrame](#);
- `collect()` returns a [tibble](#). This operation consumes more memory and takes longer than `compute()` because it also needs to convert the data from Polars to R.

Usage

```
## S3 method for class 'polars_lazy_frame'
compute(
  x,
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
  no_optimization = FALSE,
  engine = c("auto", "in-memory", "streaming"),
  streaming = FALSE
)
```

```
## S3 method for class 'polars_lazy_frame'
collect(
  x,
  ...,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  comm_subplan_elim = TRUE,
  comm_subexpr_elim = TRUE,
  cluster_with_columns = TRUE,
```

```

no_optimization = FALSE,
engine = c("auto", "in-memory", "streaming"),
streaming = FALSE,
.name_repair = "check_unique",
uint8 = "integer",
int64 = "double",
date = "Date",
time = "hms",
decimal = "double",
as_clock_class = FALSE,
ambiguous = "raise",
non_existent = "raise"
)

```

Arguments

x	A Polars LazyFrame
...	Dots which should be empty.
type_coercion	Coerce types such that operations succeed and run on minimal required memory (default is TRUE).
predicate_pushdown	Applies filters as early as possible at scan level (default is TRUE).
projection_pushdown	Select only the columns that are needed at the scan level (default is TRUE).
simplify_expression	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is TRUE).
slice_pushdown	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is TRUE).
comm_subplan_elim	Cache branching subplans that occur on self-joins or unions (default is TRUE).
comm_subexpr_elim	Cache common subexpressions (default is TRUE).
cluster_with_columns	Combine sequential independent calls to \$with_columns().
no_optimization	Sets the following optimizations to FALSE: predicate_pushdown, projection_pushdown, slice_pushdown, simplify_expression. Default is FALSE.
engine	The engine name to use for processing the query. One of the followings: • "auto" (default): Select the engine automatically. The "in-memory" engine will be selected for most cases. • "in-memory": Use the in-memory engine. • "streaming": Use the streaming engine, usually faster and can handle larger-than-memory data.
streaming	[Deprecated] Deprecated, use engine instead.

.name_repair, uint8, int64, date, time, decimal, as_clock_class,
ambiguous, non_existent

Parameters to control the conversion from polars types to R. See `?polars:::as.data.frame.polars_lazy` for explanations and accepted values.

Value

`compute()` returns a Polars DataFrame, `collect()` returns a tibble.

See Also

[fetch\(\)](#) for applying a lazy query on a subset of the data.

Examples

```
dat_lazy <- polars::as_polars_df(iris)$lazy()

compute(dat_lazy)

# you can build a query and add compute() as the last piece
dat_lazy |>
  select(starts_with("Sepal")) |>
  filter(between(Sepal.Length, 5, 6)) |>
  compute()

# call collect() instead to return a data.frame (note that this is more
# expensive than compute())
dat_lazy |>
  select(starts_with("Sepal")) |>
  filter(between(Sepal.Length, 5, 6)) |>
  collect()
```

count.polars_data_frame

Count the observations in each group

Description

Count the observations in each group

Usage

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

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

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

## S3 method for class 'polars_lazy_frame'
tally(x, wt = NULL, sort = FALSE, name = "n")

## S3 method for class 'polars_data_frame'
add_count(x, ..., wt = NULL, sort = FALSE, name = "n")

## S3 method for class 'polars_lazy_frame'
add_count(x, ..., wt = NULL, sort = FALSE, name = "n")
```

Arguments

x	A Polars Data/LazyFrame
...	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
wt	Not supported by tidypolars.
sort	If TRUE, will show the largest groups at the top.
name	Name of the new column.

Examples

```
test <- polars::as_polars_df(mtcars)

# grouping variables must be specified in count() and add_count()
count(test, cyl)
count(test, cyl, am)
count(test, cyl, am, sort = TRUE, name = "count")

add_count(test, cyl, am, sort = TRUE, name = "count")

# tally() directly uses grouping variables of the input
test |>
  group_by(cyl) |>
  tally()

test |>
  group_by(cyl, am) |>
  tally(sort = TRUE, name = "count")
```

Description

Cross joins match each row in x to every row in y, resulting in a dataset with $\text{nrow}(x) * \text{nrow}(y)$ rows.

Usage

```
## S3 method for class 'polars_data_frame'
cross_join(x, y, ..., suffix = c(".x", ".y"))

## S3 method for class 'polars_lazy_frame'
cross_join(x, y, ..., suffix = c(".x", ".y"))
```

Arguments

x, y	Two Polars Data/LazyFrames
...	Dots which should be empty.
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.

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use `options(tidypolars_unknown_args = "error")`.

Examples

```
test <- polars::pl$DataFrame(
  origin = c("ALG", "FRA", "GER"),
  year = c(2020, 2020, 2021)
)

test2 <- polars::pl$DataFrame(
  destination = c("USA", "JPN", "BRA"),
  language = c("english", "japanese", "portuguese")
)

test

test2

cross_join(test, test2)
```

distinct.polars_data_frame

Remove or keep only duplicated rows in a Data/LazyFrame

Description

By default, duplicates are looked for in all variables. It is possible to specify a subset of variables where duplicates should be looked for. It is also possible to keep either the first occurrence, the last occurrence or remove all duplicates.

Usage

```
## S3 method for class 'polars_data_frame'
distinct(.data, ..., .keep_all = FALSE, keep = "first", maintain_order = TRUE)

## S3 method for class 'polars_lazy_frame'
distinct(.data, ..., .keep_all = FALSE, keep = "first", maintain_order = TRUE)

duplicated_rows(.data, ...)
```

Arguments

.data	A Polars Data/LazyFrame
...	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
.keep_all	If TRUE, keep all variables in .data after duplicated rows are removed.
keep	Either "first" (keep the first occurrence of the duplicated row), "last" (last occurrence) or "none" (remove all occurrences of duplicated rows).
maintain_order	Maintain row order. This is the default but it can slow down the process with large datasets and it prevents the use of streaming.

Examples

```
pl_test <- polars::pl$DataFrame(
  iso_o = c(rep(c("AA", "AB"), each = 2), "AC", "DC"),
  iso_d = rep(c("BA", "BB", "BC"), each = 2),
  value = c(2, 2, 3, 4, 5, 6)
)

distinct(pl_test)
distinct(pl_test, iso_o)

duplicated_rows(pl_test)
duplicated_rows(pl_test, iso_o, iso_d)
```

```
drop_na.polars_data_frame
```

Drop missing values

Description

By default, this will drop rows that contain any missing values. It is possible to specify a subset of variables so that only missing values in these variables will be considered.

Usage

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

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

Arguments

<code>data</code>	A Polars Data/LazyFrame
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.

Examples

```
tmp <- mtcars
tmp[1:3, "mpg"] <- NA
tmp[4, "hp"] <- NA
pl_tmp <- as_polars_df(tmp)

drop_na(pl_tmp)
drop_na(pl_tmp, hp, mpg)
```

```
explain.polars_lazy_frame
```

Show the optimized and non-optimized query plans

Description

This function is available for LazyFrames only.

By default, `explain()` shows the query plan that is optimized and then run by Polars. Setting `optimized = FALSE` shows the query plan as-is, without any optimization done, but this is not the query performed. Note that the plans are read from bottom to top.

Usage

```
## S3 method for class 'polars_lazy_frame'
explain(x, optimized = TRUE, ...)
```

Arguments

x	A Polars LazyFrame.
optimized	Logical. If TRUE (default), show the query optimized by Polars. Otherwise, show the initial query.
...	Ignored.

Examples

```
query <- mtcars |>
  as_polars_lf() |>
  arrange(drat) |>
  filter(cyl == 3) |>
  select(mpg)

# unoptimized query plan:
no_opt <- explain(query, optimized = FALSE)
no_opt

# better printing with cat():
cat(no_opt)

# optimized query run by polars
cat(explain(query))
```

 fetch

Fetch n rows of a LazyFrame

Description**[Deprecated]**

Use `head()` before `collect()` to only get a subset of the data.

Usage

```
fetch(
  .data,
  n_rows = 500,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
```

```

    slice_pushdown = TRUE,
    comm_subplan_elim = TRUE,
    comm_subexpr_elim = TRUE,
    cluster_with_columns = TRUE,
    no_optimization = FALSE,
    engine = c("auto", "in-memory", "streaming"),
    streaming = FALSE
)

```

Arguments

<code>.data</code>	A Polars LazyFrame
<code>n_rows</code>	Number of rows to fetch.
<code>type_coercion</code>	Coerce types such that operations succeed and run on minimal required memory (default is TRUE).
<code>predicate_pushdown</code>	Applies filters as early as possible at scan level (default is TRUE).
<code>projection_pushdown</code>	Select only the columns that are needed at the scan level (default is TRUE).
<code>simplify_expression</code>	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is TRUE).
<code>slice_pushdown</code>	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is TRUE).
<code>comm_subplan_elim</code>	Cache branching subplans that occur on self-joins or unions (default is TRUE).
<code>comm_subexpr_elim</code>	Cache common subexpressions (default is TRUE).
<code>cluster_with_columns</code>	Combine sequential independent calls to <code>\$with_columns()</code> .
<code>no_optimization</code>	Sets the following optimizations to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>simplify_expression</code> . Default is FALSE.
<code>engine</code>	The engine name to use for processing the query. One of the followings: • "auto" (default): Select the engine automatically. The "in-memory" engine will be selected for most cases. • "in-memory": Use the in-memory engine. • "streaming": Use the streaming engine, usually faster and can handle larger-than-memory data.
<code>streaming</code>	[Deprecated] Deprecated, use <code>engine</code> instead.

Details

The parameter `n_rows` indicates how many rows from the LazyFrame should be used at the beginning of the query, but it doesn't guarantee that `n_rows` will be returned. For example, if the query

contains a filter or join operations with other datasets, then the final number of rows can be lower than `n_rows`. On the other hand, appending some rows during the query can lead to an output that has more rows than `n_rows`.

See Also

`collect()` for applying a lazy query on the full data.

```
fill.polars_data_frame
```

Fill in missing values with previous or next value

Description

Fills missing values in selected columns using the next or previous entry. This is useful in the common output format where values are not repeated, and are only recorded when they change.

Usage

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

Arguments

<code>data</code>	A Polars Data/LazyFrame
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
<code>.by</code>	Optionally, a selection of columns to group by for just this operation, functioning as an alternative to <code>group_by()</code> . The group order is not maintained, use <code>group_by()</code> if you want more control over it.
<code>.direction</code>	Direction in which to fill missing values. Either "down" (the default), "up", "downup" (i.e. first down and then up) or "updown" (first up and then down).

Details

With grouped Data/LazyFrames, `fill()` will be applied within each group, meaning that it won't fill across group boundaries.

Examples

```
pl_test <- polars::pl$DataFrame(x = c(NA, 1), y = c(2, NA))

fill(pl_test, everything(), .direction = "down")
fill(pl_test, everything(), .direction = "up")

# with grouped data, it doesn't use values from other groups
pl_grouped <- polars::pl$DataFrame(
```

```

grp = rep(c("A", "B"), each = 3),
x = c(1, NA, NA, NA, 2, NA),
y = c(3, NA, 4, NA, 3, 1)
) |>
  group_by(grp)

fill(pl_grouped, x, y, .direction = "down")

```

filter.polars_data_frame

Keep or drop rows that match a condition

Description

These functions are used to subset a data frame, applying the expressions in ... to determine which rows should be kept (for filter()) or dropped (for filter_out()).

Multiple conditions can be supplied separated by a comma. These will be combined with the & operator.

Both filter() and filter_out() treat NA like FALSE. This subtle behavior can impact how you write your conditions when missing values are involved. See the section on Missing values for important details and examples.

filter_out() is available for dplyr >= 1.2.0.

Usage

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

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

```
## S3 method for class 'polars_data_frame'
filter_out(.data, ..., .by = NULL)
```

```
## S3 method for class 'polars_lazy_frame'
filter_out(.data, ..., .by = NULL)
```

Arguments

.data	A Polars Data/LazyFrame
...	Expressions that return a logical value, and are defined in terms of the variables in the data. If multiple expressions are included, they will be combined with the & operator. Only rows for which all conditions evaluate to TRUE are kept (for filter()) or dropped (for filter_out()).
.by	Optionally, a selection of columns to group by for just this operation, functioning as an alternative to group_by(). The group order is not maintained, use group_by() if you want more control over it.

Missing values

Read this section in the [dplyr documentation](#).

Examples

```
starwars <- as_polars_df(dplyr::starwars)

# Filtering for one criterion
filter(starwars, species == "Human")

# Filtering for multiple criteria within a single logical expression
filter(starwars, hair_color == "none" & eye_color == "black")
filter(starwars, hair_color == "none" | eye_color == "black")

# When multiple expressions are used, they are combined using &
filter(starwars, hair_color == "none", eye_color == "black")

# Filtering out to drop rows
filter_out(starwars, hair_color == "none")

# When filtering out, it can be useful to first interactively filter for the
# rows you want to drop, just to double check that you've written the
# conditions correctly. Then, just change `filter()` to `filter_out()`.
filter(starwars, mass > 1000, eye_color == "orange")
filter_out(starwars, mass > 1000, eye_color == "orange")

# The filtering operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
#
# The following keeps rows where `mass` is greater than the
# global average:
starwars |> filter(mass > mean(mass, na.rm = TRUE))

# Whereas this keeps rows with `mass` greater than the per `gender`
# average:
starwars |> filter(mass > mean(mass, na.rm = TRUE), .by = gender)

# If you find yourself trying to use a `filter()` to drop rows, then
# you should consider if switching to `filter_out()` can simplify your
# conditions. For example, to drop blond individuals, you might try:
starwars |> filter(hair_color != "blond")

# But this also drops rows with an `NA` hair color! To retain those:
starwars |> filter(hair_color != "blond" | is.na(hair_color))

# But explicit `NA` handling like this can quickly get unwieldy, especially
# with multiple conditions. Since your intent was to specify rows to drop
# rather than rows to keep, use `filter_out()`. This also removes the need
# for any explicit `NA` handling.
starwars |> filter_out(hair_color == "blond")

# To refer to column names that are stored as strings, use the `.data`
```

```
# pronoun:
vars <- c("mass", "height")
cond <- c(80, 150)
starwars |>
  filter(
    .data[[vars[[1]]]] > cond[[1]],
    .data[[vars[[2]]]] > cond[[2]]
  )
```

from_csv

Import data from CSV file(s)

Description

`read_csv_polars()` imports the data as a Polars DataFrame.

`scan_csv_polars()` imports the data as a Polars LazyFrame.

Usage

```
read_csv_polars(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = "\"",
  skip_rows = 0,
  schema = NULL,
  schema_overrides = NULL,
  null_values = NULL,
  ignore_errors = FALSE,
  cache = FALSE,
  infer_schema_length = 100,
  n_rows = NULL,
  encoding = "utf8",
  low_memory = FALSE,
  rechunk = TRUE,
  skip_rows_after_header = 0,
  row_index_name = NULL,
  row_index_offset = 0,
  try_parse_dates = FALSE,
  eol_char = "\n",
  raise_if_empty = TRUE,
  truncate_ragged_lines = FALSE,
  include_file_paths = NULL,
  dtypes,
```

```

        reuse_downloaded
    )

scan_csv_polars(
    source,
    ...,
    has_header = TRUE,
    separator = ",",
    comment_prefix = NULL,
    quote_char = "\"",
    skip_rows = 0,
    schema = NULL,
    schema_overrides = NULL,
    null_values = NULL,
    ignore_errors = FALSE,
    cache = FALSE,
    infer_schema_length = 100,
    n_rows = NULL,
    encoding = "utf8",
    low_memory = FALSE,
    rechunk = TRUE,
    skip_rows_after_header = 0,
    row_index_name = NULL,
    row_index_offset = 0,
    try_parse_dates = FALSE,
    eol_char = "\n",
    raise_if_empty = TRUE,
    truncate_ragged_lines = FALSE,
    include_file_paths = NULL,
    dtypes,
    reuse_downloaded
)

```

Arguments

source	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.
...	These dots are for future extensions and must be empty.
has_header	Indicate if the first row of dataset is a header or not. If FALSE, column names will be autogenerated in the following format: "column_x" with x being an enumeration over every column in the dataset starting at 1.
separator	Single byte character to use as separator in the file.
comment_prefix	A string, which can be up to 5 symbols in length, used to indicate the start of a comment line. For instance, it can be set to # or //.
quote_char	Single byte character used for quoting. Set to NULL to turn off special handling and escaping of quotes.
skip_rows	Start reading after a particular number of rows. The header will be parsed at this offset.

schema	Provide the schema. This means that polars doesn't do schema inference. This argument expects the complete schema, whereas <code>schema_overrides</code> can be used to partially overwrite a schema. This must be a list. Names of list elements are used to match to inferred columns.
schema_overrides	Overwrite dtypes during inference. This must be a list. Names of list elements are used to match to inferred columns.
null_values	Character vector specifying the values to interpret as NA values. It can be named, in which case names specify the columns in which this replacement must be made (e.g. <code>c(col1 = "a")</code>).
ignore_errors	Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</code> to read all columns as UTF8 to check which values might cause an issue.
cache	Cache the result after reading.
infer_schema_length	The maximum number of rows to scan for schema inference. If NULL, the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$String</code> .
n_rows	Stop reading from the source after reading <code>n_rows</code> .
encoding	Either <code>"utf8"</code> or <code>"utf8-lossy"</code> . Lossy means that invalid UTF8 values are replaced with <code>"?"</code> characters.
low_memory	Reduce memory pressure at the expense of performance.
rechunk	Reallocate to contiguous memory when all chunks/files are parsed.
skip_rows_after_header	Skip this number of rows when the header is parsed.
row_index_name	If not NULL, this will insert a row index column with the given name.
row_index_offset	Offset to start the row index column (only used if the name is set by <code>row_index_name</code>).
try_parse_dates	Try to automatically parse dates. Most ISO8601-like formats can be inferred, as well as a handful of others. If this does not succeed, the column remains of data type <code>pl\$String</code> .
eol_char	Single byte end of line character (default: <code>"\n"</code>). When encountering a file with Windows line endings (<code>"\r\n"</code>), one can go with the default <code>"\n"</code> . The extra <code>"\r"</code> will be removed when processed.
raise_if_empty	If FALSE, parsing an empty file returns an empty DataFrame or LazyFrame.
truncate_ragged_lines	Truncate lines that are longer than the schema.
include_file_paths	Include the path of the source file(s) as a column with this name.
dtypes	[Deprecated] Deprecated, use <code>schema_overrides</code> instead.
reuse_downloaded	[Deprecated] Deprecated with no replacement.

Value

The scan function returns a LazyFrame, the read function returns a DataFrame.

Examples

```
### Read or scan a single CSV file -----

# Setup: create a CSV file
dest <- withr::local_tempfile(fileext = ".csv")
write.csv(mtcars, dest, row.names = FALSE)

# Import this file as a DataFrame for eager evaluation
read_csv_polars(dest) |>
  arrange(mpg)

# Import this file as a LazyFrame for lazy evaluation
scan_csv_polars(dest) |>
  arrange(mpg) |>
  compute()

### Change the datatype of some columns when reading the file -----

scan_csv_polars(
  dest,
  schema_overrides = list(gear = polars::pl$String, carb = polars::pl$Float32)
) |>
  arrange(mpg) |>
  compute()

### Read or scan several all CSV files in a folder -----

# Setup: create a folder "output" that contains two CSV files
dest_folder <- withr::local_tempdir(tmpdir = "output")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.csv")
dest2 <- file.path(dest_folder, "output_2.csv")

write.csv(mtcars[1:16, ], dest1, row.names = FALSE)
write.csv(mtcars[17:32, ], dest2, row.names = FALSE)
list.files(dest_folder)

# Import all files as a LazyFrame
scan_csv_polars(dest_folder) |>
  arrange(mpg) |>
  compute()

# Include the file path to know where each row comes from
scan_csv_polars(dest_folder, include_file_paths = "file_path") |>
  arrange(mpg) |>
  compute()
```

```

### Read or scan all CSV files that match a glob pattern -----

# Setup: create a folder "output_glob" that contains three CSV files,
# two of which follow the pattern "output_XXX.csv"
dest_folder <- withr::local_tempdir(tmpdir = "output_glob")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.csv")
dest2 <- file.path(dest_folder, "output_2.csv")
dest3 <- file.path(dest_folder, "other_output.csv")

write.csv(mtcars[1:16, ], dest1, row.names = FALSE)
write.csv(mtcars[17:32, ], dest2, row.names = FALSE)
write.csv(iris, dest3, row.names = FALSE)
list.files(dest_folder)

# Import only the files whose name match "output_XXX.csv" as a LazyFrame
scan_csv_polars(paste0(dest_folder, "/output_*.csv")) |>
  arrange(mpg) |>
  compute()

```

from_ipc

Import data from IPC file(s)

Description

`read_ipc_polars()` imports the data as a Polars DataFrame.

`scan_ipc_polars()` imports the data as a Polars LazyFrame.

Usage

```

read_ipc_polars(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  rechunk = FALSE,
  cache = TRUE,
  include_file_paths = NULL,
  memory_map
)

scan_ipc_polars(
  source,
  ...,

```

```

    n_rows = NULL,
    row_index_name = NULL,
    row_index_offset = 0L,
    rechunk = FALSE,
    cache = TRUE,
    include_file_paths = NULL,
    memory_map
  )

```

Arguments

source	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.
...	These dots are for future extensions and must be empty.
n_rows	Stop reading from the source after reading <code>n_rows</code> .
row_index_name	If not <code>NULL</code> , this will insert a row index column with the given name.
row_index_offset	Offset to start the row index column (only used if the name is set by <code>row_index_name</code>).
rechunk	Reallocate to contiguous memory when all chunks/files are parsed.
cache	Cache the result after reading.
include_file_paths	Include the path of the source file(s) as a column with this name.
memory_map	[Deprecated] Deprecated with no replacement.

Value

The scan function returns a `LazyFrame`, the read function returns a `DataFrame`.

Examples

```

### Read or scan a single IPC file -----

# Setup: create an IPC file
dest <- withr::local_tempfile(fileext = ".ipc")
arrow::write_ipc_file(mtcars, file(dest))

# Import this file as a DataFrame for eager evaluation
read_ipc_polars(dest) |>
  arrange(mpg)

# Import this file as a LazyFrame for lazy evaluation
scan_ipc_polars(dest) |>
  arrange(mpg) |>
  compute()

### Read or scan several all IPC files in a folder -----

```

```

# Setup: create a folder "output" that contains two IPC files
dest_folder <- withr::local_tempdir(tmpdir = "output")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.ipc")
dest2 <- file.path(dest_folder, "output_2.ipc")

arrow::write_ipc_file(mtcars[1:16, ], dest1)
arrow::write_ipc_file(mtcars[17:32, ], dest2)
list.files(dest_folder)

# Import all files as a LazyFrame
scan_ipc_polars(dest_folder) |>
  arrange(mpg) |>
  compute()

### Read or scan all IPC files that match a glob pattern -----

# Setup: create a folder "output_glob" that contains three IPC files,
# two of which follow the pattern "output_XXX.ipc"
dest_folder <- withr::local_tempdir(tmpdir = "output_glob")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.ipc")
dest2 <- file.path(dest_folder, "output_2.ipc")
dest3 <- file.path(dest_folder, "other_output.ipc")

arrow::write_ipc_file(mtcars[1:16, ], dest1)
arrow::write_ipc_file(mtcars[17:32, ], dest2)
arrow::write_ipc_file(iris, dest3)
list.files(dest_folder)

# Import only the files whose name match "output_XXX.ipc" as a LazyFrame
scan_ipc_polars(paste0(dest_folder, "/output_*.ipc")) |>
  arrange(mpg) |>
  compute()

```

from_ndjson

Import data from NDJSON file(s)

Description

`read_ndjson_polars()` imports the data as a Polars DataFrame.

`scan_ndjson_polars()` imports the data as a Polars LazyFrame.

Usage

```

read_ndjson_polars(
  source,
  ...,

```

```

    infer_schema_length = 100,
    batch_size = NULL,
    n_rows = NULL,
    low_memory = FALSE,
    rechunk = FALSE,
    row_index_name = NULL,
    row_index_offset = 0,
    ignore_errors = FALSE,
    reuse_downloaded
)

scan_ndjson_polars(
  source,
  ...,
  infer_schema_length = 100,
  batch_size = NULL,
  n_rows = NULL,
  low_memory = FALSE,
  rechunk = FALSE,
  row_index_name = NULL,
  row_index_offset = 0,
  ignore_errors = FALSE,
  reuse_downloaded
)

```

Arguments

source	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.
...	These dots are for future extensions and must be empty.
infer_schema_length	The maximum number of rows to scan for schema inference. If <code>NULL</code> , the full data may be scanned (this is slow). Set <code>infer_schema = FALSE</code> to read all columns as <code>pl\$String</code> .
batch_size	Number of rows to read in each batch.
n_rows	Stop reading from the source after reading <code>n_rows</code> .
low_memory	Reduce memory pressure at the expense of performance.
rechunk	Reallocate to contiguous memory when all chunks/files are parsed.
row_index_name	If not <code>NULL</code> , this will insert a row index column with the given name.
row_index_offset	Offset to start the row index column (only used if the name is set by <code>row_index_name</code>).
ignore_errors	Keep reading the file even if some lines yield errors. You can also use <code>infer_schema = FALSE</code> to read all columns as <code>UTF8</code> to check which values might cause an issue.
reuse_downloaded	[Deprecated] Deprecated with no replacement.

Value

The scan function returns a LazyFrame, the read function returns a DataFrame.

Examples

```
### Read or scan a single NDJSON file -----

# Setup: create a NDJSON file
dest <- withr::local_tempfile(fileext = ".json")
jsonlite::stream_out(mtcars, file(dest), verbose = FALSE)

# Import this file as a DataFrame for eager evaluation
read_ndjson_polars(dest) |>
  arrange(mpg)

# Import this file as a LazyFrame for lazy evaluation
scan_ndjson_polars(dest) |>
  arrange(mpg) |>
  compute()

### Read or scan several all NDJSON files in a folder -----

# Setup: create a folder "output" that contains two NDJSON files
dest_folder <- withr::local_tempdir(tmpdir = "output")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.json")
dest2 <- file.path(dest_folder, "output_2.json")

jsonlite::stream_out(mtcars[1:16, ], file(dest1), verbose = FALSE)
jsonlite::stream_out(mtcars[17:32, ], file(dest2), verbose = FALSE)
list.files(dest_folder)

# Import all files as a LazyFrame
scan_ndjson_polars(dest_folder) |>
  arrange(mpg) |>
  compute()

### Read or scan all NDJSON files that match a glob pattern -----

# Setup: create a folder "output_glob" that contains three NDJSON files,
# two of which follow the pattern "output_XXX.json"
dest_folder <- withr::local_tempdir(tmpdir = "output_glob")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.json")
dest2 <- file.path(dest_folder, "output_2.json")
dest3 <- file.path(dest_folder, "other_output.json")

jsonlite::stream_out(mtcars[1:16, ], file(dest1), verbose = FALSE)
jsonlite::stream_out(mtcars[17:32, ], file(dest2), verbose = FALSE)
jsonlite::stream_out(iris, file(dest3), verbose = FALSE)
```

```
list.files(dest_folder)

# Import only the files whose name match "output_XXX.json" as a LazyFrame
scan_ndjson_polars(paste0(dest_folder, "/output_*.json")) |>
  arrange(mpg) |>
  compute()
```

from_parquet	<i>Import data from Parquet file(s)</i>
--------------	---

Description

`read_parquet_polars()` imports the data as a Polars DataFrame.

`scan_parquet_polars()` imports the data as a Polars LazyFrame.

Usage

```
read_parquet_polars(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = "auto",
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
  glob = TRUE,
  rechunk = TRUE,
  low_memory = FALSE,
  storage_options = NULL,
  use_statistics = TRUE,
  cache = TRUE,
  include_file_paths = NULL
)
```

```
scan_parquet_polars(
  source,
  ...,
  n_rows = NULL,
  row_index_name = NULL,
  row_index_offset = 0L,
  parallel = "auto",
  hive_partitioning = NULL,
  hive_schema = NULL,
  try_parse_hive_dates = TRUE,
```

```

    glob = TRUE,
    rechunk = FALSE,
    low_memory = FALSE,
    storage_options = NULL,
    use_statistics = TRUE,
    cache = TRUE,
    include_file_paths = NULL
  )

```

Arguments

source	Path(s) to a file or directory. When needing to authenticate for scanning cloud locations, see the <code>storage_options</code> parameter.
...	These dots are for future extensions and must be empty.
n_rows	Stop reading from the source after reading <code>n_rows</code> .
row_index_name	If not <code>NULL</code> , this will insert a row index column with the given name.
row_index_offset	Offset to start the row index column (only used if the name is set by <code>row_index_name</code>).
parallel	This determines the direction and strategy of parallelism. • "auto" (default): Will try to determine the optimal direction. • "prefiltered": [Experimental] Strategy first evaluates the pushed-down predicates in parallel and determines a mask of which rows to read. Then, it parallelizes over both the columns and the row groups while filtering out rows that do not need to be read. This can provide significant speedups for large files (i.e. many row-groups) with a predicate that filters clustered rows or filters heavily. In other cases, prefiltered may slow down the scan compared other strategies. Falls back to "auto" if no predicate is given. • "columns", "row_groups": Use the specified direction. • "none": No parallelism.
hive_partitioning	Infer statistics and schema from Hive partitioned sources and use them to prune reads. If <code>NULL</code> (default), it is automatically enabled when a single directory is passed, and otherwise disabled.
hive_schema	[Experimental] A list containing the column names and data types of the columns by which the data is partitioned, e.g. <code>list(a = pl\$String, b = pl\$Float32)</code> . If <code>NULL</code> (default), the schema of the Hive partitions is inferred.
try_parse_hive_dates	Whether to try parsing hive values as date / datetime types.
glob	Expand path given via globbing rules.
rechunk	Reallocate to contiguous memory when all chunks/files are parsed.
low_memory	Reduce memory pressure at the expense of performance
storage_options	Named vector containing options that indicate how to connect to a cloud provider. The cloud providers currently supported are AWS, GCP, and Azure. See supported keys here:

- **aws**
- **gcp**
- **azure**
- Hugging Face (hf://): Accepts an API key under the token parameter `c(token = YOUR_TOKEN)` or by setting the `HF_TOKEN` environment variable.

If `storage_options` is not provided, Polars will try to infer the information from environment variables.

`use_statistics` Use statistics in the parquet to determine if pages can be skipped from reading.

`cache` Cache the result after reading.

`include_file_paths` Include the path of the source file(s) as a column with this name.

Value

The scan function returns a LazyFrame, the read function returns a DataFrame.

Examples

```
### Read or scan a single Parquet file -----

# Setup: create a Parquet file
dest <- withr::local_tempfile(fileext = ".parquet")
dat <- as_polars_df(mtcars)
write_parquet_polars(dat, dest)

# Import this file as a DataFrame for eager evaluation
read_parquet_polars(dest) |>
  arrange(mpg)

# Import this file as a LazyFrame for lazy evaluation
scan_parquet_polars(dest) |>
  arrange(mpg) |>
  compute()

### Read or scan several all Parquet files in a folder -----

# Setup: create a folder "output" that contains two Parquet files
dest_folder <- withr::local_tempdir(tmpdir = "output")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.parquet")
dest2 <- file.path(dest_folder, "output_2.parquet")

write_parquet_polars(as_polars_df(mtcars[1:16, ]), dest1)
write_parquet_polars(as_polars_df(mtcars[17:32, ]), dest2)
list.files(dest_folder)

# Import all files as a LazyFrame
scan_parquet_polars(dest_folder) |>
  arrange(mpg) |>
```

```

compute()

# Include the file path to know where each row comes from
scan_parquet_polars(dest_folder, include_file_paths = "file_path") |>
  arrange(mpg) |>
  compute()

### Read or scan all Parquet files that match a glob pattern -----

# Setup: create a folder "output_glob" that contains three Parquet files,
# two of which follow the pattern "output_XXX.parquet"
dest_folder <- withr::local_tempdir(tmpdir = "output_glob")
dir.create(dest_folder, showWarnings = FALSE)
dest1 <- file.path(dest_folder, "output_1.parquet")
dest2 <- file.path(dest_folder, "output_2.parquet")
dest3 <- file.path(dest_folder, "other_output.parquet")

write_parquet_polars(as_polars_df(mtcars[1:16, ]), dest1)
write_parquet_polars(as_polars_df(mtcars[17:32, ]), dest2)
write_parquet_polars(as_polars_df(iris), dest3)
list.files(dest_folder)

# Import only the files whose name match "output_XXX.parquet" as a LazyFrame
scan_parquet_polars(paste0(dest_folder, "/output_*.parquet")) |>
  arrange(mpg) |>
  compute()

```

group_by.polars_data_frame

Group by one or more variables

Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing Polars Data/LazyFrame and converts it into a grouped one where operations are performed "by group". `ungroup()` removes grouping.

Usage

```

## S3 method for class 'polars_data_frame'
group_by(.data, ..., maintain_order = FALSE, .add = FALSE, .drop = TRUE)

## S3 method for class 'polars_data_frame'
ungroup(x, ...)

## S3 method for class 'polars_lazy_frame'
group_by(.data, ..., maintain_order = FALSE, .add = FALSE, .drop = TRUE)

```

```
## S3 method for class 'polars_lazy_frame'
ungroup(x, ...)
```

Arguments

.data	A Polars Data/LazyFrame
...	Variables to group by (used in group_by() only). Not used in ungroup().
maintain_order	Maintain row order. For performance reasons, this is FALSE by default). Setting it to TRUE can slow down the process with large datasets and prevents the use of streaming.
.add	When FALSE (default), group_by() will override existing groups. To add to the existing groups, use .add = TRUE.
.drop	Unsupported. It is only present to provide a good error message if specified by the user.
x	A Polars Data/LazyFrame

Examples

```
by_cyl <- mtcars |>
  as_polars_df() |>
  group_by(cyl)

by_cyl

by_cyl |> summarise(
  disp = mean(displacement),
  hp = mean(horsepower)
)
by_cyl |> filter(displacement == max(displacement))
```

```
group_split.polars_data_frame
      Grouping metadata
```

Description

group_vars() returns a character vector with the names of the grouping variables. group_keys() returns a data frame with one row per group.

Usage

```
## S3 method for class 'polars_data_frame'
group_split(.tbl, ..., .keep = TRUE)
```

Arguments

.tbl	A Polars Data/LazyFrame
...	If .tbl is not grouped, variables to group by. If .tbl is already grouped, this is ignored.
.keep	Should the grouping columns be kept?

Examples

```
pl_g <- polars::as_polars_df(iris) |>
  group_by(Species)

group_split(pl_g)
```

```
group_vars.polars_data_frame
      Grouping metadata
```

Description

group_vars() returns a character vector with the names of the grouping variables. group_keys() returns a data frame with one row per group.

Usage

```
## S3 method for class 'polars_data_frame'
group_vars(x)

## S3 method for class 'polars_lazy_frame'
group_vars(x)

## S3 method for class 'polars_data_frame'
group_keys(.tbl, ...)

## S3 method for class 'polars_lazy_frame'
group_keys(.tbl, ...)
```

Arguments

x, .tbl	A Polars Data/LazyFrame
...	These dots are for future extensions and must be empty.

Examples

```
pl_g <- polars::as_polars_df(mtcars) |>
  group_by(cyl, am)

group_vars(pl_g)

group_keys(pl_g)
```

```
left_join.polars_data_frame
  Mutating joins
```

Description

Mutating joins add columns from y to x, matching observations based on the keys.

Usage

```
## S3 method for class 'polars_data_frame'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = "na",
  relationship = NULL
)

## S3 method for class 'polars_data_frame'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = "na",
  relationship = NULL
)

## S3 method for class 'polars_data_frame'
full_join(
```

```
x,  
y,  
by = NULL,  
copy = FALSE,  
suffix = c(".x", ".y"),  
...,  
keep = NULL,  
na_matches = "na",  
relationship = NULL  
)  
  
## S3 method for class 'polars_data_frame'  
inner_join(  
  x,  
  y,  
  by = NULL,  
  copy = FALSE,  
  suffix = c(".x", ".y"),  
  ...,  
  keep = NULL,  
  na_matches = "na",  
  relationship = NULL  
)  
  
## S3 method for class 'polars_lazy_frame'  
left_join(  
  x,  
  y,  
  by = NULL,  
  copy = FALSE,  
  suffix = c(".x", ".y"),  
  ...,  
  keep = NULL,  
  na_matches = "na",  
  relationship = NULL  
)  
  
## S3 method for class 'polars_lazy_frame'  
right_join(  
  x,  
  y,  
  by = NULL,  
  copy = FALSE,  
  suffix = c(".x", ".y"),  
  ...,  
  keep = NULL,  
  na_matches = "na",  
  relationship = NULL
```

```

)

## S3 method for class 'polars_lazy_frame'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = "na",
  relationship = NULL
)

## S3 method for class 'polars_lazy_frame'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = "na",
  relationship = NULL
)

```

Arguments

<code>x, y</code>	Two Polars Data/LazyFrames
<code>by</code>	Variables to join by. If <code>NULL</code> (default), <code>*_join()</code> will perform a natural join, using all variables in common across <code>x</code> and <code>y</code>. A message lists the variables so that you can check they're correct; suppress the message by supplying <code>by</code> explicitly. <code>by</code> can take a character vector, like <code>c("x", "y")</code> if <code>x</code> and <code>y</code> are in both datasets. To join on variables that don't have the same name, use equalities in the character vector, like <code>c("x1" = "x2", "y")</code>. If you use a character vector, the join can only be done using strict equality. <code>by</code> can also be a specification created by <code>dplyr::join_by()</code>. Contrary to the input as character vector shown above, <code>join_by()</code> uses unquoted column names, e.g. <code>join_by(x1 == x2, y)</code>. Finally, <code>inner_join()</code> also supports inequality joins, e.g. <code>join_by(x1 >= x2)</code>, and the helpers <code>between()</code>, <code>overlaps()</code>, and <code>within()</code>. See the documentation of dplyr::join_by() for more information. Other join types will likely support inequality joins in the future.
<code>copy, keep</code>	Not supported.

<code>suffix</code>	If there are non-joined duplicate variables in <code>x</code> and <code>y</code> , these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
<code>...</code>	Dots which should be empty.
<code>na_matches</code>	Should two NA values match? • "na", the default, treats two NA values as equal. • "never" treats two NA values as different and will never match them together or to any other values. Note that when joining Polars Data/LazyFrames, NaN are always considered equal, no matter the value of <code>na_matches</code>. This differs from the original dplyr implementation.
<code>relationship</code>	Handling of the expected relationship between the keys of <code>x</code> and <code>y</code> . Must be one of the following: • NULL, the default, is equivalent to "many-to-many". It doesn't expect any relationship between <code>x</code> and <code>y</code>. • "one-to-one" expects each row in <code>x</code> to match at most 1 row in <code>y</code> and each row in <code>y</code> to match at most 1 row in <code>x</code>. • "one-to-many" expects each row in <code>y</code> to match at most 1 row in <code>x</code>. • "many-to-one" expects each row in <code>x</code> matches at most 1 row in <code>y</code>.

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use `options(tidypolars_unknown_args = "error")`.

Examples

```
test <- polars::pl$DataFrame(
  x = c(1, 2, 3),
  y1 = c(1, 2, 3),
  z = c(1, 2, 3)
)

test2 <- polars::pl$DataFrame(
  x = c(1, 2, 4),
  y2 = c(1, 2, 4),
  z2 = c(4, 5, 7)
)

test

test2

# default is to use common columns, here "x" only
left_join(test, test2)

# we can specify the columns on which to join with join_by()...
left_join(test, test2, by = join_by(x, y1 == y2))
```

```

# ... or with a character vector
left_join(test, test2, by = c("x", "y1" = "y2"))

# we can customize the suffix of common column names not used to join
test2 <- polars::pl$DataFrame(
  x = c(1, 2, 4),
  y1 = c(1, 2, 4),
  z = c(4, 5, 7)
)

left_join(test, test2, by = "x", suffix = c("_left", "_right"))

# the argument "relationship" ensures the join matches the expectation
country <- polars::pl$DataFrame(
  iso = c("FRA", "DEU"),
  value = 1:2
)
country

country_year <- polars::pl$DataFrame(
  iso = rep(c("FRA", "DEU"), each = 2),
  year = rep(2019:2020, 2),
  value2 = 3:6
)
country_year

# We expect that each row in "x" matches only one row in "y" but, it's not
# true as each row of "x" matches two rows of "y"
tryCatch(
  left_join(country, country_year, join_by(iso), relationship = "one-to-one"),
  error = function(e) e
)

# A correct expectation would be "one-to-many":
left_join(country, country_year, join_by(iso), relationship = "one-to-many")

```

make_unique_id

Create a column with unique id per row values

Description

[Deprecated]

The underlying Polars function isn't guaranteed to give the same results across different versions. Therefore, this function will be removed and has no replacement in tidypolars.

Usage

```
make_unique_id(.data, ..., new_col = "hash")
```

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
<code>new_col</code>	Name of the new column

```
mutate.polars_data_frame
```

Create, modify, and delete columns

Description

This creates new columns that are functions of existing variables. It can also modify (if the name is the same as an existing column) and delete columns (by setting their value to NULL).

Usage

```
## S3 method for class 'polars_data_frame'
mutate(
  .data,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)

## S3 method for class 'polars_lazy_frame'
mutate(
  .data,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector the same length as the current group (or the whole data frame if ungrouped). • NULL, to remove the column.

- `across()` is mostly supported, except in a few cases. In particular, if the `.cols` argument is `where(...)`, it will *not* select variables that were created before `across()`. Other select helpers are supported. See the examples.
- `.by` Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. The group order is not maintained, use `group_by()` if you want more control over it.
- `.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.

Details

A lot of functions available in base R (`cos`, `mean`, `multiplying`, etc.) or in other packages (`dplyr::lag()`, etc.) are implemented in an efficient way in Polars. These functions are automatically translated to Polars syntax under the hood so that you can continue using the classic R syntax and functions.

If a Polars built-in replacement doesn't exist (for example for custom functions), then `tidypolars` will throw an error. See the vignette on Polars expressions to know how to write custom functions that are accepted by `tidypolars`.

Examples

```
pl_iris <- polars::as_polars_df(iris)

# classic operation
mutate(pl_iris, x = Sepal.Width + Sepal.Length)

# logical operation
mutate(pl_iris, x = Sepal.Width > Sepal.Length & Petal.Width > Petal.Length)

# overwrite existing variable
mutate(pl_iris, Sepal.Width = Sepal.Width * 2)

# grouped computation
pl_iris |>
  group_by(Species) |>
  mutate(foo = mean(Sepal.Length))

# an alternative syntax for grouping is to use `.by`
```

```

pl_iris |>
  mutate(foo = mean(Sepal.Length), .by = Species)

# across() is available
pl_iris |>
  mutate(
    across(.cols = contains("Sepal"), .fns = mean, .names = "{.fn}_of_{.col}")
  )
#
# It can receive several types of functions:
pl_iris |>
  mutate(
    across(
      .cols = contains("Sepal"),
      .fns = list(mean = mean, sd = ~ sd(.x)),
      .names = "{.fn}_of_{.col}"
    )
  )

# Be careful when using across(.cols = where(...), ...) as it will not include
# variables created in the same `...` (this is only the case for `where()`):
## Not run:
pl_iris |>
  mutate(
    foo = 1,
    across(
      .cols = where(is.numeric),
      \(x) x - 1000 # <<<<<<<< this will not be applied on variable "foo"
    )
  )

## End(Not run)
# Warning message:
# In `across()`, the argument `.cols = where(is.numeric)` will not take into account
# variables created in the same `mutate()`/`summarize` call.

# Embracing an external variable works
some_value <- 1
mutate(pl_iris, x = {{ some_value }})

```

partitioned_output	<i>Helper functions to export data as a partitioned output</i>
--------------------	--

Description

[Experimental]

Partitioning schemes are used to write multiple files with `sink_*`() and `write*_polars()` functions.

- `partition_by()`: Configuration for writing to multiple output files. Supports partitioning by key, file size limits, or both.

The following functions are deprecated and will be removed in a future release:

- **[Deprecated]** `partition_by_key()`: use `partition_by(key = ...)` instead.
- **[Deprecated]** `partition_by_max_size()`: use `partition_by(max_rows_per_file = ...)` instead.

Usage

```
partition_by(
  base_path,
  ...,
  key = NULL,
  include_key = NULL,
  max_rows_per_file = NULL,
  approximate_bytes_per_file = NULL
)

partition_by_key(base_path, ..., by, include_key = TRUE)

partition_by_max_size(base_path, ..., max_size)
```

Arguments

<code>base_path</code>	The base path for the output files. Use the <code>mkdir</code> option of the <code>sink_*</code> or <code>write_*_polars()</code> functions to ensure directories in the path are created.
<code>...</code>	These dots are for future extensions and must be empty.
<code>key</code>	Something can be coerced to a list of Polars expressions. Used to partition by.
<code>include_key</code>	A bool indicating whether to include the key columns in the output files. Can only be used if <code>key</code> is specified, otherwise should be <code>NULL</code> .
<code>max_rows_per_file</code>	An integer-ish value indicating the maximum size in rows of each of the generated files.
<code>approximate_bytes_per_file</code>	An integer-ish value indicating approximate number of bytes to write to each file, or <code>NULL</code> . This is measured as the estimated size of the DataFrame in memory. Defaults to approximately 4GB when <code>key</code> is specified without <code>max_rows_per_file</code> , otherwise unlimited.
<code>by</code>	[Deprecated] Something can be coerced to a list of Polars expressions. Used to partition by. Use the <code>key</code> property of <code>partition_by()</code> instead.
<code>max_size</code>	[Deprecated] An integer-ish value indicating the maximum size in rows of each of the generated files. Use the <code>max_rows_per_file</code> property of <code>partition_by()</code> instead.

`pivot_longer.polars_data_frame`

Pivot a Data/LazyFrame from wide to long

Description

Pivot a Data/LazyFrame from wide to long

Usage

```
## S3 method for class 'polars_data_frame'
pivot_longer(
  data,
  cols,
  ...,
  names_to = "name",
  names_prefix = NULL,
  values_to = "value"
)

## S3 method for class 'polars_lazy_frame'
pivot_longer(
  data,
  cols,
  ...,
  names_to = "name",
  names_prefix = NULL,
  values_to = "value"
)
```

Arguments

<code>data</code>	A Polars Data/LazyFrame
<code>cols</code>	Columns to pivot into longer format. Can be anything accepted by <code>dplyr::select()</code> .
<code>...</code>	Dots which should be empty.
<code>names_to</code>	The (quoted) name of the column that will contain the column names specified by <code>cols</code> .
<code>names_prefix</code>	A regular expression used to remove matching text from the start of each variable name.
<code>values_to</code>	A string specifying the name of the column to create from the data stored in cell values.

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use `options(tidypolars_unknown_args = "error")`.

Examples

```
pl_relig_income <- as_polars_df(tidyr::relig_income)
pl_relig_income

pl_relig_income |>
  pivot_longer(!religion, names_to = "income", values_to = "count")

pl_billboard <- as_polars_df(tidyr::billboard)
pl_billboard

pl_billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    names_prefix = "wk",
    values_to = "rank",
  )
```

```
pivot_wider.polars_data_frame
```

Pivot a Data/LazyFrame from long to wide

Description

Pivot a Data/LazyFrame from long to wide

Usage

```
## S3 method for class 'polars_data_frame'
pivot_wider(
  data,
  ...,
  id_cols = NULL,
  names_from = name,
  values_from = value,
  names_prefix = "",
  names_sep = "_",
  names_glue = NULL,
  values_fill = NULL
)

## S3 method for class 'polars_lazy_frame'
pivot_wider(
  data,
  ...,
  id_cols = NULL,
  names_from = name,
```

```

    values_from = value,
    names_prefix = "",
    names_sep = "_",
    names_glue = NULL,
    values_fill = NULL
  )

```

Arguments

<code>data</code>	A Polars Data/LazyFrame.
<code>...</code>	Dots which should be empty.
<code>id_cols</code>	A set of columns that uniquely identify each observation. Typically used when you have redundant variables, i.e. variables whose values are perfectly correlated with existing variables. Defaults to all columns in data except for the columns specified through <code>names_from</code> and <code>values_from</code> . If a tidyselect expression is supplied, it will be evaluated on data after removing the columns specified through <code>names_from</code> and <code>values_from</code> .
<code>names_from</code>	The (quoted or unquoted) column names whose values will be used for the names of the new columns.
<code>values_from</code>	The (quoted or unquoted) column names whose values will be used to fill the new columns.
<code>names_prefix</code>	String added to the start of every variable name. This is particularly useful if <code>names_from</code> is a numeric vector and you want to create syntactic variable names.
<code>names_sep</code>	If <code>names_from</code> or <code>values_from</code> contains multiple variables, this will be used to join their values together into a single string to use as a column name.
<code>names_glue</code>	Instead of <code>names_sep</code> and <code>names_prefix</code> , you can supply a glue specification that uses the <code>names_from</code> columns to create custom column names.
<code>values_fill</code>	A scalar that will be used to replace missing values in the new columns. Note that the type of this value will be applied to new columns. For example, if you provide a character value to fill numeric columns, then all these columns will be converted to character.

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use `options(tidypolars_unknown_args = "error")`.

Examples

```

pl_fish_encounters <- as_polars_df(tidyr::fish_encounters)

pl_fish_encounters |>
  pivot_wider(names_from = station, values_from = seen)

pl_fish_encounters |>

```

```

    pivot_wider(names_from = station, values_from = seen, values_fill = 0)

# be careful about the type of the replacement value!
pl_fish_encounters |>
  pivot_wider(names_from = station, values_from = seen, values_fill = "a")

# using "names_glue" to specify the names of new columns
production <- expand_grid(
  product = c("A", "B"),
  country = c("AI", "EI"),
  year = 2000:2014
) |>
  filter((product == "A" & country == "AI") | product == "B") |>
  mutate(production = 1:45) |>
  as_polars_df()

production

production |>
  pivot_wider(
    names_from = c(product, country),
    values_from = production,
    names_glue = "prod_{product}_{country}"
  )

```

pull.polars_data_frame

Extract a variable of a Data/LazyFrame

Description

This returns an R vector and not a Polars Series.

Usage

```
## S3 method for class 'polars_data_frame'
pull(.data, var, ...)
```

```
## S3 method for class 'polars_lazy_frame'
pull(.data, var, ...)
```

Arguments

.data	A Polars Data/LazyFrame
var	A quoted or unquoted variable name, or a variable index.
...	Dots which should be empty.

Examples

```
pl_test <- as_polars_df(iris)
pull(pl_test, Sepal.Length)
pull(pl_test, "Sepal.Length")
```

```
relocate.polars_data_frame
      Change column order
```

Description

Use `relocate()` to change column positions, using the same syntax as `select()` to make it easy to move blocks of columns at once.

Usage

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

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

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
<code>.before</code> , <code>.after</code>	Column name (either quoted or unquoted) that indicates the destination of columns selected by <code>...</code> . Supplying neither will move columns to the left-hand side; specifying both is an error.

Examples

```
dat <- as_polars_df(mtcars)

dat |>
  relocate(hp, vs, .before = cyl)

# if .before and .after are not specified, selected columns are moved to the
# first positions
dat |>
  relocate(hp, vs)

# .before and .after can be quoted or unquoted
dat |>
  relocate(hp, vs, .after = "gear")
```

```
# select helpers are also available
dat |>
  relocate(contains("[aeiou]"))

dat |>
  relocate(hp, vs, .after = last_col())
```

 rename.polars_data_frame

Rename columns

Description

Rename columns

Usage

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

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

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

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

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	For <code>rename()</code> , use <code>new_name = old_name</code> to rename selected variables. It is also possible to use quotation marks, e.g. <code>"new_name" = "old_name"</code> . For <code>rename_with</code> , additional arguments passed to <code>fn</code> .
<code>.fn</code>	Function to apply on column names
<code>.cols</code>	Columns on which to apply <code>fn</code> . Can be anything accepted by <code>dplyr::select()</code> .

Examples

```
pl_test <- polars::as_polars_df(mtcars)

rename(pl_test, miles_per_gallon = mpg, horsepower = "hp")

rename(pl_test, `Miles per gallon` = "mpg", `Horse power` = "hp")
```

```

rename_with(pl_test, toupper, .cols = contains("p"))

pl_test_2 <- polars::as_polars_df(iris)

rename_with(pl_test_2, function(x) tolower(gsub(".", "_", x, fixed = TRUE)))

rename_with(pl_test_2, \(x) tolower(gsub(".", "_", x, fixed = TRUE)))

```

```
replace_na.polars_data_frame
```

Replace NAs with specified values

Description

Replace NAs with specified values

Usage

```

## S3 method for class 'polars_data_frame'
replace_na(data, replace, ...)

## S3 method for class 'polars_lazy_frame'
replace_na(data, replace, ...)

```

Arguments

data	A Polars Data/LazyFrame
replace	Either a scalar that will be used to replace NA in all columns, or a named list with the column name and the value that will be used to replace NA in it.
...	Dots which should be empty.

Examples

```

pl_test <- polars::pl$DataFrame(x = c(NA, 1), y = c(2, NA))

replace_na(pl_test, list(x = 0, y = 999))

```

rowwise.polars_data_frame

Group input by rows

Description

[EXPERIMENTAL]

rowwise() allows you to compute on a Data/LazyFrame a row-at-a-time. This is most useful when a vectorised function doesn't exist. rowwise() produces another type of grouped data, and therefore can be removed with ungroup().

Usage

```
## S3 method for class 'polars_data_frame'
rowwise(data, ...)
```

```
## S3 method for class 'polars_lazy_frame'
rowwise(data, ...)
```

Arguments

data	A Polars Data/LazyFrame
...	Any expression accepted by dplyr::select(): variable names, column numbers, select helpers, etc.

Value

A Polars Data/LazyFrame.

Examples

```
df <- polars::pl$DataFrame(x = c(1, 3, 4), y = c(2, 1, 5), z = c(2, 3, 1))

# Compute the mean of x, y, z in each row
df |>
  rowwise() |>
  mutate(m = mean(c(x, y, z)))

# Compute the min and max of x and y in each row
df |>
  rowwise() |>
  mutate(min = min(c(x, y)), max = max(c(x, y)))
```

```
select.polars_data_frame
```

Select columns from a Data/LazyFrame

Description

Select columns from a Data/LazyFrame

Usage

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

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

Arguments

<code>.data</code>	A Polars Data/LazyFrame
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc. Renaming is also possible.

Examples

```
pl_iris <- polars::as_polars_df(iris)

select(pl_iris, c("Sepal.Length", "Sepal.Width"))
select(pl_iris, Sepal.Length, Sepal.Width)
select(pl_iris, 1:3)
select(pl_iris, starts_with("Sepal"))
select(pl_iris, -ends_with("Length"))

# Renaming while selecting is also possible
select(pl_iris, foo1 = Sepal.Length, Sepal.Width)
```

```
semi_join.polars_data_frame
```

Filtering joins

Description

Filtering joins filter rows from `x` based on the presence or absence of matches in `y`:

- `semi_join()` return all rows from `x` with a match in `y`.
- `anti_join()` return all rows from `x` without a match in `y`.

Usage

```
## S3 method for class 'polars_data_frame'
semi_join(x, y, by = NULL, ..., na_matches = "na")

## S3 method for class 'polars_data_frame'
anti_join(x, y, by = NULL, ..., na_matches = "na")

## S3 method for class 'polars_lazy_frame'
semi_join(x, y, by = NULL, ..., na_matches = "na")

## S3 method for class 'polars_lazy_frame'
anti_join(x, y, by = NULL, ..., na_matches = "na")
```

Arguments

x, y	Two Polars Data/LazyFrames
by	Variables to join by. If NULL (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. by can take a character vector, like c("x", "y") if x and y are in both datasets. To join on variables that don't have the same name, use equalities in the character vector, like c("x1" = "x2", "y"). If you use a character vector, the join can only be done using strict equality. by can also be a specification created by dplyr::join_by(). Contrary to the input as character vector shown above, join_by() uses unquoted column names, e.g join_by(x1 == x2, y). Finally, inner_join() also supports inequality joins, e.g. join_by(x1 >= x2), and the helpers between(), overlaps(), and within(). See the documentation of dplyr::join_by() for more information. Other join types will likely support inequality joins in the future.
...	Dots which should be empty.
na_matches	Should two NA values match? • "na", the default, treats two NA values as equal. • "never" treats two NA values as different and will never match them together or to any other values. Note that when joining Polars Data/LazyFrames, NaN are always considered equal, no matter the value of na_matches. This differs from the original dplyr implementation.

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use options(tidypolars_unknown_args = "error").

Examples

```
test <- polars::pl$DataFrame(
  x = c(1, 2, 3),
  y = c(1, 2, 3),
  z = c(1, 2, 3)
)

test2 <- polars::pl$DataFrame(
  x = c(1, 2, 4),
  y = c(1, 2, 4),
  z2 = c(1, 2, 4)
)

test

test2

# only keep the rows of `test` that have matching keys in `test2`
semi_join(test, test2, by = c("x", "y"))

# only keep the rows of `test` that don't have matching keys in `test2`
anti_join(test, test2, by = c("x", "y"))
```

```
separate.polars_data_frame
```

Separate a character column into multiple columns based on a substring

Description

Currently, splitting a column on a position is not possible.

Usage

```
## S3 method for class 'polars_data_frame'
separate(data, col, into, sep = " ", remove = TRUE, ...)

## S3 method for class 'polars_lazy_frame'
separate(data, col, into, sep = " ", remove = TRUE, ...)
```

Arguments

data	A Polars Data/LazyFrame
col	Column to split
into	Character vector containing the names of new variables to create. Use NA to omit the variable in the output.
sep	A regular expression that is used to split the column.

remove If TRUE, remove input column from output data frame.
... Dots which should be empty.

Examples

```
# Split at each dot
test <- polars::pl$DataFrame(x = c(NA, "x.y", "x.z", "y.z"))
separate(test, x, into = c("foo", "foo2"), sep = "\\.")

# Split on any number of whitespace
test <- polars::pl$DataFrame(x = c(NA, "x y", "x y", "x y z"))
separate(test, x, into = c("foo", "foo2"), sep = "\\s+")
```

separate_longer	<i>Split a string column into rows</i>
-----------------	--

Description

Each of these functions takes a string and splits it into multiple rows:

- `separate_longer_delim_polars()` splits by delimiter. It is the polars equivalent of `tidyr::separate_longer_delim`.
- `separate_longer_position_polars()` splits by fixed width. It is the polars equivalent of `tidyr::separate_longer_position()`.

Usage

```
separate_longer_delim_polars(data, cols, delim, ...)

separate_longer_position_polars(data, cols, width, ..., keep_empty = FALSE)
```

Arguments

data A Polars DataFrame or LazyFrame.
cols `<tidy-select>` Column(s) to separate.
delim The delimiter string to split on. For `separate_longer_delim_polars()` only.
... These dots are for future extensions and must be empty.
width The width of each piece. For `separate_longer_position_polars()` only.
keep_empty If TRUE, empty strings are kept in the output. If FALSE (the default), empty strings are dropped. NA values are always kept.

Value

A Polars DataFrame or LazyFrame with the specified column(s) split into rows.

See Also

`tidyr::separate_longer_delim()`, `tidyr::separate_longer_position()`

Examples

```
library(polars)
library(tidypolars)

# separate_longer_delim_polars: split by delimiter
df <- pl$DataFrame(
  id = 1:3,
  x = c("a,b,c", "d,e", "f")
)
separate_longer_delim_polars(df, x, delim = ",")

# Multiple columns with broadcasting, the same as `tidyr` behavior
df2 <- pl$DataFrame(
  id = 1:2,
  x = c("a,b", "c,d"),
  y = c("1,2", "3,4")
)
separate_longer_delim_polars(df2, c(x, y), delim = ",")

# Multiple columns with broadcasting
df3 <- pl$DataFrame(
  id = 1:5,
  x = c("a,b", NA, "", "c", ""),
  y = c("1", "2,3", "4,5", NA, "")
)
separate_longer_delim_polars(df3, c(x, y), delim = ",")

# separate_longer_position_polars: split by fixed width
df4 <- pl$DataFrame(
  id = 1:3,
  x = c("abcd", "efg", "hi")
)
separate_longer_position_polars(df4, x, width = 2)

# keep_empty example: control whether empty strings are preserved
df5 <- pl$DataFrame(
  id = 1:4,
  x = c("ab", "", "ef", NA)
)
separate_longer_position_polars(df5, x, width = 2)
separate_longer_position_polars(df5, x, width = 2, keep_empty = TRUE)

# Multiple columns with broadcasting
df6 <- pl$DataFrame(
  id = 1:3,
  x = c("a", "bc", "def"),
  y = c("12", "345", "67")
)
```

```
# Shorter strings are recycled to match the longest in each row
separate_longer_position_polars(df6, c(x, y), width = 2)
```

sink_csv

Stream output to a CSV file

Description

This function allows to stream a LazyFrame that is larger than RAM directly to a .csv file without collecting it in the R session, thus preventing crashes because of too small memory.

Usage

```
sink_csv(
  .data,
  path,
  ...,
  include_bom = FALSE,
  include_header = TRUE,
  separator = ",",
  line_terminator = "\n",
  quote_char = "\"",
  batch_size = 1024,
  datetime_format = NULL,
  date_format = NULL,
  time_format = NULL,
  float_precision = NULL,
  null_value = "",
  quote_style = "necessary",
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE,
  mkdir = FALSE,
  quote,
  null_values
)
```

Arguments

.data	A Polars LazyFrame.
path	Output file. Can also be a partition_*() function to export the output to multiple files (see Details section below).

...	Ignored.
include_bom	Whether to include UTF-8 BOM (byte order mark) in the CSV output.
include_header	Whether to include header in the CSV output.
separator	Separate CSV fields with this symbol.
line_terminator	String used to end each row.
quote_char	Byte to use as quoting character.
batch_size	Number of rows that will be processed per thread.
datetime_format, date_format, time_format	A format string used to format date and time values. See <code>?strftime()</code> for accepted values. If no format specified, the default fractional-second precision is inferred from the maximum time unit found in the <code>Datetime</code> cols (if any).
float_precision	Number of decimal places to write, applied to both <code>Float32</code> and <code>Float64</code> datatypes.
null_value	A string representing null values (defaulting to the empty string).
quote_style	Determines the quoting strategy used: • "necessary" (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). • "always": This puts quotes around every field. • "non_numeric": This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary.
maintain_order	Whether maintain the order the data was processed (default is <code>TRUE</code>). Setting this to <code>FALSE</code> will be slightly faster.
type_coercion	Coerce types such that operations succeed and run on minimal required memory (default is <code>TRUE</code>).
predicate_pushdown	Applies filters as early as possible at scan level (default is <code>TRUE</code>).
projection_pushdown	Select only the columns that are needed at the scan level (default is <code>TRUE</code>).
simplify_expression	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is <code>TRUE</code>).
slice_pushdown	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is <code>TRUE</code>).
no_optimization	Sets the following optimizations to <code>FALSE</code> : <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>simplify_expression</code> . Default is <code>FALSE</code> .
mkdir	Recursively create all the directories in the path.
quote	[Deprecated] Deprecated, use <code>quote_char</code> instead.
null_values	[Deprecated] Deprecated, use <code>null_value</code> instead.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input LazyFrame.

Examples

```
# This is an example workflow where sink_csv() is not very useful because
# the data would fit in memory. It simply is an example of using it at the
# end of a piped workflow.

# Create files for the CSV input and output:
file_csv <- tempfile(fileext = ".csv")
file_csv2 <- tempfile(fileext = ".csv")

# Write some data in a CSV file
fake_data <- do.call("rbind", rep(list(mtcars), 1000))
write.csv(fake_data, file = file_csv, row.names = FALSE)

# In a new R session, we could read this file as a LazyFrame, do some operations,
# and write it to another CSV file without ever collecting it in the R session:
scan_csv_polars(file_csv) |>
  filter(cyl %in% c(4, 6), mpg > 22) |>
  mutate(
    hp_gear_ratio = hp / gear
  ) |>
  sink_csv(path = file_csv2)

#-----
# Write a LazyFrame to multiple files depending on various strategies.
my_lf <- as_polars_lf(mtcars)

# Split the LazyFrame by key(s) and write each split to a different file:
out_path <- withr::local_tempdir()
sink_csv(my_lf, partition_by_key(out_path, by = c("am", "cyl")), mkdir = TRUE)
fs::dir_tree(out_path)

# Split the LazyFrame by max number of rows per file:
out_path <- withr::local_tempdir()
sink_csv(my_lf, partition_by_max_size(out_path, max_size = 5), mkdir = TRUE)
fs::dir_tree(out_path) # mtcars has 32 rows so we have 7 output files
```

sink_ipc	<i>Stream output to an IPC file</i>
----------	-------------------------------------

Description

This function allows to stream a LazyFrame that is larger than RAM directly to an IPC file without collecting it in the R session, thus preventing crashes because of too small memory.

Usage

```
sink_ipc(
  .data,
  path,
  ...,
  compression = "zstd",
  compat_level = "newest",
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE,
  mkdir = FALSE
)
```

Arguments

.data	A Polars LazyFrame.
path	Output file. Can also be a partition_*() function to export the output to multiple files (see Details section below).
...	Ignored.
compression	NULL or a character of the compression method, "uncompressed" or "lz4" or "zstd". NULL is equivalent to "uncompressed". Choose "zstd" for good compression performance. Choose "lz4" for fast compression/decompression.
compat_level	Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., "newest") or as a scalar integer (currently, 0 or 1 is supported). "newest" (default): Use the highest level, currently same as 1 (Low compatibility). "oldest": Same as 0 (High compatibility).
maintain_order	Whether maintain the order the data was processed (default is TRUE). Setting this to FALSE will be slightly faster.

type_coercion	Coerce types such that operations succeed and run on minimal required memory (default is TRUE).
predicate_pushdown	Applies filters as early as possible at scan level (default is TRUE).
projection_pushdown	Select only the columns that are needed at the scan level (default is TRUE).
simplify_expression	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is TRUE).
slice_pushdown	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is TRUE).
no_optimization	Sets the following optimizations to FALSE: predicate_pushdown, projection_pushdown, slice_pushdown, simplify_expression. Default is FALSE.
mkdir	Recursively create all the directories in the path.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input LazyFrame.

Examples

```
# This is an example workflow where sink_ipc() is not very useful because
# the data would fit in memory. It simply is an example of using it at the
# end of a piped workflow.

# Create files for the IPC input and output:
file_ipc <- tempfile(fileext = ".ipc")
file_ipc2 <- tempfile(fileext = ".ipc")

# Write some data in an IPC file
fake_data <- do.call("rbind", rep(list(mtcars), 1000))
arrow::write_ipc_file(fake_data, file_ipc)

# In a new R session, we could read this file as a LazyFrame, do some operations,
# and write it to another IPC file without ever collecting it in the R session:
scan_ipc_polars(file_ipc) |>
  filter(cyl %in% c(4, 6), mpg > 22) |>
  mutate(
    hp_gear_ratio = hp / gear
  ) |>
  sink_ipc(path = file_ipc2)
```

```
#-----
# Write a LazyFrame to multiple files depending on various strategies.
my_lf <- as_polars_lf(mtcars)

# Split the LazyFrame by key(s) and write each split to a different file:
out_path <- withr::local_tempdir()
sink_ipc(my_lf, partition_by_key(out_path, by = c("am", "cyl")), mkdir = TRUE)
fs::dir_tree(out_path)

# Split the LazyFrame by max number of rows per file:
out_path <- withr::local_tempdir()
sink_ipc(my_lf, partition_by_max_size(out_path, max_size = 5), mkdir = TRUE)
fs::dir_tree(out_path) # mtcars has 32 rows so we have 7 output files
```

sink_ndjson

Stream output to a NDJSON file

Description

This writes the output of a query directly to a NDJSON file without collecting it in the R session first. This is useful if the output of the query is still larger than RAM as it would crash the R session if it was collected into R.

Usage

```
sink_ndjson(
  .data,
  path,
  ...,
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE,
  mkdir = FALSE
)
```

Arguments

.data	A Polars LazyFrame.
path	Output file. Can also be a partition_*() function to export the output to multiple files (see Details section below).
...	Ignored.

<code>maintain_order</code>	Whether maintain the order the data was processed (default is TRUE). Setting this to FALSE will be slightly faster.
<code>type_coercion</code>	Coerce types such that operations succeed and run on minimal required memory (default is TRUE).
<code>predicate_pushdown</code>	Applies filters as early as possible at scan level (default is TRUE).
<code>projection_pushdown</code>	Select only the columns that are needed at the scan level (default is TRUE).
<code>simplify_expression</code>	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is TRUE).
<code>slice_pushdown</code>	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is TRUE).
<code>no_optimization</code>	Sets the following optimizations to FALSE: <code>predicate_pushdown</code> , <code>projection_pushdown</code> , <code>slice_pushdown</code> , <code>simplify_expression</code> . Default is FALSE.
<code>mkdir</code>	Recursively create all the directories in the path.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input LazyFrame.

Examples

```
# This is an example workflow where sink_ndjson() is not very useful because
# the data would fit in memory. It simply is an example of using it at the
# end of a piped workflow.

# Create files for the NDJSON input and output:
file_ndjson <- tempfile(fileext = ".ndjson")
file_ndjson2 <- tempfile(fileext = ".ndjson")

# Write some data in a CSV file
fake_data <- do.call("rbind", rep(list(mtcars), 1000))
jsonlite::stream_out(fake_data, file(file_ndjson), verbose = FALSE)

# In a new R session, we could read this file as a LazyFrame, do some operations,
# and write it to another NDJSON file without ever collecting it in the R session:
scan_ndjson_polars(file_ndjson) |>
  filter(cyl %in% c(4, 6), mpg > 22) |>
  mutate(
    hp_gear_ratio = hp / gear
```

```

) |>
sink_ndjson(path = file_ndjson2)

#-----
# Write a LazyFrame to multiple files depending on various strategies.
my_lf <- as_polars_lf(mtcars)

# Split the LazyFrame by key(s) and write each split to a different file:
out_path <- withr::local_tempdir()
sink_ndjson(my_lf, partition_by_key(out_path, by = c("am", "cyl")), mkdir = TRUE)
fs::dir_tree(out_path)

# Split the LazyFrame by max number of rows per file:
out_path <- withr::local_tempdir()
sink_ndjson(my_lf, partition_by_max_size(out_path, max_size = 5), mkdir = TRUE)
fs::dir_tree(out_path) # mtcars has 32 rows so we have 7 output files

```

sink_parquet

Stream output to a parquet file

Description

This function allows to stream a LazyFrame that is larger than RAM directly to a .parquet file without collecting it in the R session, thus preventing crashes because of too small memory.

Usage

```

sink_parquet(
  .data,
  path,
  ...,
  compression = "zstd",
  compression_level = 3,
  statistics = FALSE,
  row_group_size = NULL,
  data_page_size = NULL,
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE,
  mkdir = FALSE
)

```

Arguments

.data	A Polars LazyFrame.
path	Output file. Can also be a partition_*() function to export the output to multiple files (see Details section below).
...	Ignored.
compression	The compression method. One of : • "uncompressed" • "zstd" (default): good compression performance • "lz4": fast compression / decompression • "snappy": more backwards compatibility guarantees when you deal with older parquet readers. • "gzip", "lzo", "brotli"
compression_level	The level of compression to use (default is 3). Only used if compression is one of "gzip", "brotli", or "zstd". Higher compression means smaller files on disk. • "gzip" : min-level = 0, max-level = 10 • "brotli" : min-level = 0, max-level = 11 • "zstd" : min-level = 1, max-level = 22.
statistics	Whether to compute and write column statistics (default is FALSE). This requires more computations.
row_group_size	Size of the row groups in number of rows. If NULL (default), the chunks of the DataFrame are used. Writing in smaller chunks may reduce memory pressure and improve writing speeds.
data_page_size	If NULL (default), the limit will be around 1MB.
maintain_order	Whether maintain the order the data was processed (default is TRUE). Setting this to FALSE will be slightly faster.
type_coercion	Coerce types such that operations succeed and run on minimal required memory (default is TRUE).
predicate_pushdown	Applies filters as early as possible at scan level (default is TRUE).
projection_pushdown	Select only the columns that are needed at the scan level (default is TRUE).
simplify_expression	Various optimizations, such as constant folding and replacing expensive operations with faster alternatives (default is TRUE).
slice_pushdown	Only load the required slice from the scan. Don't materialize sliced outputs level. Don't materialize sliced outputs (default is TRUE).
no_optimization	Sets the following optimizations to FALSE: predicate_pushdown, projection_pushdown, slice_pushdown, simplify_expression. Default is FALSE.
mkdir	Recursively create all the directories in the path.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input LazyFrame.

Examples

```
# This is an example workflow where sink_parquet() is not very useful because
# the data would fit in memory. It simply is an example of using it at the
# end of a piped workflow.

# Create files for the CSV input and the Parquet output:
file_csv <- tempfile(fileext = ".csv")
file_parquet <- tempfile(fileext = ".parquet")

# Write some data in a CSV file
fake_data <- do.call("rbind", rep(list(mtcars), 1000))
write.csv(fake_data, file = file_csv, row.names = FALSE)

# In a new R session, we could read this file as a LazyFrame, do some operations,
# and write it to a parquet file without ever collecting it in the R session:
scan_csv_polars(file_csv) |>
  filter(cyl %in% c(4, 6), mpg > 22) |>
  mutate(
    hp_gear_ratio = hp / gear
  ) |>
  sink_parquet(path = file_parquet)

#-----
# Write a LazyFrame to multiple files depending on various strategies.
my_lf <- as_polars_lf(mtcars)

# Split the LazyFrame by key(s) and write each split to a different file:
out_path <- withr::local_tempdir()
sink_parquet(my_lf, partition_by_key(out_path, by = c("am", "cyl")), mkdir = TRUE)
fs::dir_tree(out_path)

# Split the LazyFrame by max number of rows per file:
out_path <- withr::local_tempdir()
sink_parquet(my_lf, partition_by_max_size(out_path, max_size = 5), mkdir = TRUE)
fs::dir_tree(out_path) # mtcars has 32 rows so we have 7 output files
```

 slice_tail.polars_data_frame

Subset rows of a Data/LazyFrame

Description

Subset rows of a Data/LazyFrame

Usage

```
## S3 method for class 'polars_data_frame'
slice_tail(.data, ..., n, by = NULL)

## S3 method for class 'polars_lazy_frame'
slice_tail(.data, ..., n, by = NULL)

## S3 method for class 'polars_data_frame'
slice_head(.data, ..., n, by = NULL)

## S3 method for class 'polars_lazy_frame'
slice_head(.data, ..., n, by = NULL)

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

Arguments

.data	A Polars Data/LazyFrame
...	Dots which should be empty.
n	The number of rows to select from the start or the end of the data. Cannot be used with prop.
by	Optionally, a selection of columns to group by for just this operation, functioning as an alternative to group_by(). The group order is not maintained, use group_by() if you want more control over it.
prop	Proportion of rows to select. Cannot be used with n.
replace	Perform the sampling with replacement (TRUE) or without (FALSE).

Unknown arguments

Arguments that are supported by the original implementation in the tidyverse but are not listed above will throw a warning by default if they are specified. To change this behavior to error instead, use `options(tidypolars_unknown_args = "error")`.

Examples

```
pl_test <- polars::as_polars_df(iris)
slice_head(pl_test, n = 3)
slice_tail(pl_test, n = 3)
slice_sample(pl_test, n = 5)
slice_sample(pl_test, prop = 0.1)
```

```
summarize.polars_data_frame
```

Summarize each group down to one row

Description

summarize() returns one row for each combination of grouping variables (one difference with dplyr::summarize() is that summarize() only accepts grouped data). It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

Usage

```
## S3 method for class 'polars_data_frame'
summarize(.data, ..., .by = NULL, .groups = "drop_last")

## S3 method for class 'polars_data_frame'
summarise(.data, ..., .by = NULL, .groups = "drop_last")

## S3 method for class 'polars_lazy_frame'
summarize(.data, ..., .by = NULL, .groups = "drop_last")

## S3 method for class 'polars_lazy_frame'
summarise(.data, ..., .by = NULL, .groups = "drop_last")
```

Arguments

.data	A Polars Data/LazyFrame
...	Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector the same length as the current group (or the whole data frame if ungrouped). • NULL, to remove the column.

across() is mostly supported, except in a few cases. In particular, if the .cols argument is where(...), it will *not* select variables that were created before across(). Other select helpers are supported. See the examples.

- `.by` Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. The group order is not maintained, use `group_by()` if you want more control over it.
- `.groups` Grouping structure of the result. Must be one of:
- "drop_last" (default): drop the last level of grouping;
 - "drop": all levels of grouping are dropped;
 - "keep": keep the same grouping structure as `.data`.
- For now, "rowwise" is not supported. Note that `dplyr` uses `.groups = NULL` by default, whose behavior depends on the number of rows by group in the output. However, returning several rows by group in `summarize()` is deprecated (one should use `reframe()` instead), which is why `.groups = NULL` is not supported by `tidypolars`.

Examples

```
mtcars |>
  as_polars_df() |>
  group_by(cyl) |>
  summarize(m_gear = mean(gear), sd_gear = sd(gear))

# an alternative syntax is to use `.by`
mtcars |>
  as_polars_df() |>
  summarize(m_gear = mean(gear), sd_gear = sd(gear), .by = cyl)
```

summary.polars_data_frame

Summary statistics for a Polars DataFrame

Description

Summary statistics for a Polars DataFrame

Usage

```
## S3 method for class 'polars_data_frame'
summary(object, percentiles = c(0.25, 0.5, 0.75), ...)
```

Arguments

- `object` A Polars DataFrame.
- `percentiles` One or more percentiles to include in the summary statistics. All values must be between 0 and 1 (NULL are ignored).
- `...` Ignored.

Examples

```
mtcars |>
  as_polars_df() |>
  summary(percentiles = c(0.2, 0.4, 0.6, 0.8))
```

tidypolars_options	tidypolars <i>global options</i>
--------------------	----------------------------------

Description

tidypolars has the following global options:

- `tidypolars_unknown_args` controls what happens when some arguments passed in an expression are unknown, e.g the argument `prob` in `sample()`. The default ("warn") only warns the user that some arguments are ignored by tidypolars. The only other accepted value is "error" to throw an error when this happens.
- `tidypolars_fallback_to_r` controls what happens when an unknown function (that isn't translated to use polars syntax) is passed in an expression. The default is FALSE, meaning that unknown functions will trigger an error. Setting this option to TRUE will convert the data to R, apply the unknown function, and convert the output back to polars. **Using the fallback to R has several drawbacks:**
 - it loses some of polars built-in parallelism and other optimizations;
 - the session may crash or experience a severe slowdown when the data is converted to R (especially if the input is a LazyFrame).

The package polars also contains several global options that may be useful, such as changing the default behavior when converting Int64 values to R: https://pola-rs.github.io/r-polars/man/polars_options.html.

Examples

```
##### Unknown arguments

options(tidypolars_unknown_args = "warn")
test <- polars::pl$DataFrame(x = c(2, 1, 5, 3, 1))

# The default is to warn the user
mutate(test, x2 = sample(x, prob = 0.5))

# But one can make this stricter and throw an error when this happens
options(tidypolars_unknown_args = "error")
try(mutate(test, x2 = sample(x, prob = 0.5)))

options(tidypolars_unknown_args = "warn")

##### Fallback to R

test <- polars::pl$DataFrame(x = c(2, 1, 5, 3, 1))
```

```
# The default is to error because mad() isn't translated internally
try(mutate(test, x2 = mad(x)))

# But one can allow fallback to R to apply this function and then convert
# the output back to polars (see drawbacks in the "description" section
# above)
options(tidypolars_fallback_to_r = TRUE)
mutate(test, x2 = mad(x))

options(tidypolars_fallback_to_r = FALSE)
```

```
uncount.polars_data_frame
```

Uncount a Data/LazyFrame

Description

This duplicates rows according to a weighting variable (or expression). This is the opposite of `count()`.

Usage

```
## S3 method for class 'polars_data_frame'
uncount(data, weights, ..., .remove = TRUE, .id = NULL)
```

```
## S3 method for class 'polars_lazy_frame'
uncount(data, weights, ..., .remove = TRUE, .id = NULL)
```

Arguments

<code>data</code>	A Polars Data/LazyFrame
<code>weights</code>	A vector of weights. Evaluated in the context of data.
<code>...</code>	Dots which should be empty.
<code>.remove</code>	If TRUE, and weights is the name of a column in data, then this column is removed.
<code>.id</code>	Supply a string to create a new variable which gives a unique identifier for each created row.

Examples

```
test <- polars::pl$DataFrame(x = c("a", "b"), y = 100:101, n = c(1, 2))
test

uncount(test, n)
```

```

uncount(test, n, .id = "id")

# using constants
uncount(test, 2)

# using expressions
uncount(test, 2 / n)

```

```
unite.polars_data_frame
```

Unite multiple columns into one by pasting strings together

Description

Unite multiple columns into one by pasting strings together

Usage

```

## S3 method for class 'polars_data_frame'
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)

## S3 method for class 'polars_lazy_frame'
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)

```

Arguments

<code>data</code>	A Polars Data/LazyFrame
<code>col</code>	The name of the new column, as a string or symbol.
<code>...</code>	Any expression accepted by <code>dplyr::select()</code> : variable names, column numbers, select helpers, etc.
<code>sep</code>	Separator to use between values.
<code>remove</code>	If TRUE, remove input columns from the output Data/LazyFrame.
<code>na.rm</code>	If TRUE, missing values will be replaced with an empty string prior to uniting each value.

Examples

```

test <- polars::pl$DataFrame(
  year = 2009:2011,
  month = 10:12,
  day = c(11L, 22L, 28L),
  name_day = c("Monday", "Thursday", "Wednesday")
)

# By default, united columns are dropped
unite(test, col = "full_date", year, month, day, sep = "-")

```

```

unite(test, col = "full_date", year, month, day, sep = "-", remove = FALSE)

test2 <- polars::pl$DataFrame(
  name = c("John", "Jack", "Thomas"),
  middlename = c("T.", NA, "F."),
  surname = c("Smith", "Thompson", "Jones")
)

# By default, NA values are kept in the character output
unite(test2, col = "full_name", everything(), sep = " ")
unite(test2, col = "full_name", everything(), sep = " ", na.rm = TRUE)

```

unnest_longer_polars *Unnest a list-column into rows*

Description

`unnest_longer_polars()` turns each element of a list-column into a row. This is the equivalent of `tidyr::unnest_longer()`.

Usage

```

unnest_longer_polars(
  data,
  col,
  ...,
  values_to = NULL,
  indices_to = NULL,
  keep_empty = FALSE
)

```

Arguments

<code>data</code>	A Polars DataFrame or LazyFrame.
<code>col</code>	<code><tidy-select></code> Column(s) to unnest. Can be bare column names, character strings, or tidyselect expressions. When selecting multiple columns, the list elements in each row must have the same length across all selected columns.
<code>...</code>	These dots are for future extensions and must be empty.
<code>values_to</code>	A string giving the column name to store the unnested values in. If NULL (the default), the original column name is used. When multiple columns are selected, this can be a glue string containing "{col}" to provide a template for the column names (e.g., <code>values_to = "{col}_val"</code>).
<code>indices_to</code>	A string giving the column name to store the index of the values. If NULL (the default), no index column is created. When multiple columns are selected, this can be a glue string containing "{col}" to create separate index columns for each unnested column (e.g., <code>indices_to = "{col}_idx"</code>).
<code>keep_empty</code>	If TRUE, empty values (NULL or empty lists) are kept as NA in the output. If FALSE (the default), empty values are dropped.

Details

When multiple columns are selected, the corresponding list elements from each row are expanded together. This requires that all selected columns have lists of the same length in each row.

The `indices_to` parameter creates an integer column with the position (1-indexed) of each element within the original list. Named elements in the list are not currently supported for index names (they will use integer positions).

When using "{col}" templates with multiple columns, the template is applied to each column name to generate the output column names.

Value

A Polars DataFrame or LazyFrame with the list-column(s) unnested into rows.

See Also

[tidyr::unnest_longer\(\)](#) for the tidyr equivalent.

Examples

```
library(polars)

# Basic example with a list column
df <- pl$DataFrame(
  id = 1:3,
  values = list(c(1, 2), c(3, 4, 5), 6)
)
df

unnest_longer_polars(df, values)

# With indices
unnest_longer_polars(df, values, indices_to = "idx")

# Rename the output column
unnest_longer_polars(df, values, values_to = "val")

# Multiple columns - list elements must have same length per row
df2 <- pl$DataFrame(
  id = 1:2,
  a = list(c(1, 2), c(3, 4)),
  b = list(c("x", "y"), c("z", "w"))
)
unnest_longer_polars(df2, c(a, b))

# Multiple columns with values_to template
unnest_longer_polars(df2, c(a, b), values_to = "{col}_val")

# Multiple columns with indices_to template
unnest_longer_polars(df2, c(a, b), indices_to = "{col}_idx")

# keep_empty example
```

```
df4 <- pl$DataFrame(
  id = 1:3,
  values = list(c(1, 2), NULL, 3)
)

# By default, NULL/empty values are dropped
unnest_longer_polars(df4, values)

# Use keep_empty = TRUE to keep them as NA
unnest_longer_polars(df4, values, keep_empty = TRUE)
```

write_csv_polars	<i>Export data to CSV file(s)</i>
------------------	-----------------------------------

Description

Export data to CSV file(s)

Usage

```
write_csv_polars(
  .data,
  file,
  ...,
  include_bom = FALSE,
  include_header = TRUE,
  separator = ",",
  line_terminator = "\n",
  quote_char = "\"",
  batch_size = 1024,
  datetime_format = NULL,
  date_format = NULL,
  time_format = NULL,
  float_precision = NULL,
  null_value = "",
  quote_style = "necessary",
  quote,
  null_values
)
```

Arguments

.data	A Polars DataFrame.
file	File path to which the result should be written.
...	Ignored.
include_bom	Whether to include UTF-8 BOM (byte order mark) in the CSV output.

include_header	Whether to include header in the CSV output.
separator	Separate CSV fields with this symbol.
line_terminator	String used to end each row.
quote_char	Byte to use as quoting character.
batch_size	Number of rows that will be processed per thread.
datetime_format	A format string, with the specifiers defined by the chrono Rust crate. If no format specified, the default fractional-second precision is inferred from the maximum timeunit found in the frame's Datetime cols (if any).
date_format	A format string, with the specifiers defined by the chrono Rust crate.
time_format	A format string, with the specifiers defined by the chrono Rust crate.
float_precision	Number of decimal places to write, applied to both Float32 and Float64 datatypes.
null_value	A string representing null values (defaulting to the empty string).
quote_style	Determines the quoting strategy used. • "necessary" (default): This puts quotes around fields only when necessary. They are necessary when fields contain a quote, delimiter or record terminator. Quotes are also necessary when writing an empty record (which is indistinguishable from a record with one empty field). This is the default. • "always": This puts quotes around every field. • "non_numeric": This puts quotes around all fields that are non-numeric. Namely, when writing a field that does not parse as a valid float or integer, then quotes will be used even if they aren't strictly necessary. • "never": This never puts quotes around fields, even if that results in invalid CSV data (e.g. by not quoting strings containing the separator).
quote	[Deprecated] Deprecated, use quote_char instead.
null_values	[Deprecated] Deprecated, use null_value instead.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input DataFrame.

Examples

```
dest <- tempfile(fileext = ".csv")
mtcars |>
  as_polars_df() |>
```

```
write_csv_polars(dest)

read_csv(dest)
```

write_ipc_polars	<i>Export data to IPC file(s)</i>
------------------	-----------------------------------

Description

Export data to IPC file(s)

Usage

```
write_ipc_polars(
    .data,
    file,
    compression = "uncompressed",
    ...,
    compat_level = "newest",
    future
)
```

Arguments

.data	A Polars DataFrame.
file	File path to which the result should be written.
compression	NULL or a character of the compression method, "uncompressed" or "lz4" or "zstd". NULL is equivalent to "uncompressed". Choose "zstd" for good compression performance. Choose "lz4" for fast compression/decompression.
...	Ignored.
compat_level	Determines the compatibility level when exporting Polars' internal data structures. When specifying a new compatibility level, Polars exports its internal data structures that might not be interpretable by other Arrow implementations. The level can be specified as the name (e.g., "newest") or as a scalar integer (currently, 0 or 1 is supported). "newest" (default): Use the highest level, currently same as 1 (Low compatibility). "oldest": Same as 0 (High compatibility).
future	[Deprecated] Deprecated, use compat_level instead.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input DataFrame.

write_json_polars	<i>Export data to JSON file(s)</i>
-------------------	------------------------------------

Description

Export data to JSON file(s)

Usage

```
write_json_polars(.data, file, ..., pretty = FALSE, row_oriented = FALSE)
```

Arguments

.data	A Polars DataFrame.
file	File path to which the result should be written.
...	Ignored.
pretty	[Deprecated] Deprecated with no replacement.
row_oriented	[Deprecated] Deprecated with no replacement.

Value

The input DataFrame.

Examples

```
dest <- tempfile(fileext = ".json")
mtcars |>
  as_polars_df() |>
  write_json_polars(dest)

jsonlite::fromJSON(dest)
```

write_ndjson_polars	<i>Export data to NDJSON file(s)</i>
---------------------	--------------------------------------

Description

Export data to NDJSON file(s)

Usage

```
write_ndjson_polars(.data, file)
```

Arguments

.data	A Polars DataFrame.
file	File path to which the result should be written.

Details

Partitioned output:

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input DataFrame.

Examples

```
dest <- tempfile(fileext = ".ndjson")
mtcars |>
  as_polars_df() |>
  write_ndjson_polars(dest)

jsonlite::stream_in(file(dest), verbose = FALSE)
```

write_parquet_polars	<i>Export data to Parquet file(s)</i>
----------------------	---------------------------------------

Description

Export data to Parquet file(s)

Usage

```

write_parquet_polars(
    .data,
    file,
    ...,
    compression = "zstd",
    compression_level = 3,
    statistics = TRUE,
    row_group_size = NULL,
    data_page_size = NULL,
    partition_by = NULL,
    partition_chunk_size_bytes = 4294967296,
    mkdir = FALSE
)

```

Arguments

<code>.data</code>	A Polars DataFrame.
<code>file</code>	File path to which the result should be written.
<code>...</code>	Ignored.
<code>compression</code>	The compression method. One of : • "uncompressed" • "zstd" (default): good compression performance • "lz4": fast compression / decompression • "snappy": more backwards compatibility guarantees when you deal with older parquet readers. • "gzip", "lzo", "brotli"
<code>compression_level</code>	The level of compression to use (default is 3). Only used if <code>compression</code> is one of "gzip", "brotli", or "zstd". Higher compression means smaller files on disk. • "gzip" : min-level = 0, max-level = 10 • "brotli" : min-level = 0, max-level = 11 • "zstd" : min-level = 1, max-level = 22.
<code>statistics</code>	Whether to compute and write column statistics (default is FALSE). This requires more computations.
<code>row_group_size</code>	Size of the row groups in number of rows. If NULL (default), the chunks of the DataFrame are used. Writing in smaller chunks may reduce memory pressure and improve writing speeds.
<code>data_page_size</code>	If NULL (default), the limit will be around 1MB.
<code>partition_by</code>	Column(s) to partition by. A partitioned dataset will be written if this is specified.
<code>partition_chunk_size_bytes</code>	Approximate size to split DataFrames within a single partition when writing. Note this is calculated using the size of the DataFrame in memory - the size of the output file may differ depending on the file format / compression.
<code>mkdir</code>	Recursively create all the directories in the path.

Details**Partitioned output:**

It is possible to export data to multiple files based on various parameters, such as the values of some variables, or such that each file has a maximum number of rows. See [partition_by\(\)](#) for more details.

Value

The input DataFrame.

Examples

```
dest <- tempfile(fileext = ".parquet")
mtcars |>
  as_polars_df() |>
  write_parquet_polars(dest)

nanoparquet::read_parquet(dest)
```

Index

.tp, [3](#)

add_count.polars_data_frame
 (count.polars_data_frame), [10](#)

add_count.polars_lazy_frame
 (count.polars_data_frame), [10](#)

anti_join.polars_data_frame
 (semi_join.polars_data_frame),
 [52](#)

anti_join.polars_lazy_frame
 (semi_join.polars_data_frame),
 [52](#)

arrange.polars_data_frame, [4](#)

bind_cols_polars, [5](#)

bind_rows_polars, [5](#)

collect(), [17](#)

collect.polars_lazy_frame
 (compute.polars_lazy_frame), [8](#)

complete.polars_data_frame, [6](#)

complete.polars_lazy_frame
 (complete.polars_data_frame), [6](#)

compute.polars_lazy_frame, [8](#)

count.polars_data_frame, [10](#)

count.polars_lazy_frame
 (count.polars_data_frame), [10](#)

cross_join.polars_data_frame, [11](#)

cross_join.polars_lazy_frame
 (cross_join.polars_data_frame),
 [11](#)

distinct.polars_data_frame, [13](#)

distinct.polars_lazy_frame
 (distinct.polars_data_frame),
 [13](#)

dplyr::join_by(), [37](#), [53](#)

drop_na.polars_data_frame, [14](#)

drop_na.polars_lazy_frame
 (drop_na.polars_data_frame), [14](#)

duplicated_rows
 (distinct.polars_data_frame),
 [13](#)

explain.polars_lazy_frame, [14](#)

fetch, [15](#)

fetch(), [10](#)

fill.polars_data_frame, [17](#)

filter.polars_data_frame, [18](#)

filter.polars_lazy_frame
 (filter.polars_data_frame), [18](#)

filter_out.polars_data_frame
 (filter.polars_data_frame), [18](#)

filter_out.polars_lazy_frame
 (filter.polars_data_frame), [18](#)

from_csv, [20](#)

from_ipc, [24](#)

from_ndjson, [26](#)

from_parquet, [29](#)

full_join.polars_data_frame
 (left_join.polars_data_frame),
 [35](#)

full_join.polars_lazy_frame
 (left_join.polars_data_frame),
 [35](#)

group_by.polars_data_frame, [32](#)

group_by.polars_lazy_frame
 (group_by.polars_data_frame),
 [32](#)

group_keys.polars_data_frame
 (group_vars.polars_data_frame),
 [34](#)

group_keys.polars_lazy_frame
 (group_vars.polars_data_frame),
 [34](#)

group_split.polars_data_frame, [33](#)

group_vars.polars_data_frame, [34](#)

group_vars.polars_lazy_frame
 (group_vars.polars_data_frame),
 34

inner_join.polars_data_frame
 (left_join.polars_data_frame),
 35

inner_join.polars_lazy_frame
 (left_join.polars_data_frame),
 35

left_join.polars_data_frame, 35
 left_join.polars_lazy_frame
 (left_join.polars_data_frame),
 35

make_unique_id, 39
 mutate.polars_data_frame, 40
 mutate.polars_lazy_frame
 (mutate.polars_data_frame), 40

partition_by(partitioned_output), 42
 partition_by(), 59, 61, 63, 66, 76, 77, 79, 81
 partition_by_key(partitioned_output),
 42

partition_by_max_size
 (partitioned_output), 42

partitioned_output, 42

pivot_longer.polars_data_frame, 44
 pivot_longer.polars_lazy_frame
 (pivot_longer.polars_data_frame),
 44

pivot_wider.polars_data_frame, 45
 pivot_wider.polars_lazy_frame
 (pivot_wider.polars_data_frame),
 45

Polars DataFrame, 8
 pull.polars_data_frame, 47
 pull.polars_lazy_frame
 (pull.polars_data_frame), 47

read_csv_polars(from_csv), 20
 read_ipc_polars(from_ipc), 24
 read_ndjson_polars(from_ndjson), 26
 read_parquet_polars(from_parquet), 29
 relocate(), 41
 relocate.polars_data_frame, 48
 relocate.polars_lazy_frame
 (relocate.polars_data_frame),
 48

rename.polars_data_frame, 49
 rename.polars_lazy_frame
 (rename.polars_data_frame), 49

rename_with.polars_data_frame
 (rename.polars_data_frame), 49
 rename_with.polars_lazy_frame
 (rename.polars_data_frame), 49

replace_na.polars_data_frame, 50
 replace_na.polars_lazy_frame
 (replace_na.polars_data_frame),
 50

right_join.polars_data_frame
 (left_join.polars_data_frame),
 35

right_join.polars_lazy_frame
 (left_join.polars_data_frame),
 35

rowwise.polars_data_frame, 51
 rowwise.polars_lazy_frame
 (rowwise.polars_data_frame), 51

scan_csv_polars(from_csv), 20
 scan_ipc_polars(from_ipc), 24
 scan_ndjson_polars(from_ndjson), 26
 scan_parquet_polars(from_parquet), 29

select.polars_data_frame, 52
 select.polars_lazy_frame
 (select.polars_data_frame), 52

semi_join.polars_data_frame, 52
 semi_join.polars_lazy_frame
 (semi_join.polars_data_frame),
 52

separate.polars_data_frame, 54
 separate.polars_lazy_frame
 (separate.polars_data_frame),
 54

separate_longer, 55
 separate_longer_delim_polars
 (separate_longer), 55

separate_longer_position_polars
 (separate_longer), 55

sink_csv, 57
 sink_ipc, 60
 sink_ndjson, 62
 sink_parquet, 64

slice_head.polars_data_frame
 (slice_tail.polars_data_frame),
 67

[slice_head.polars_lazy_frame](#)
 ([slice_tail.polars_data_frame](#)),
 [67](#)
[slice_sample.polars_data_frame](#)
 ([slice_tail.polars_data_frame](#)),
 [67](#)
[slice_tail.polars_data_frame](#), [67](#)
[slice_tail.polars_lazy_frame](#)
 ([slice_tail.polars_data_frame](#)),
 [67](#)
[summarise.polars_data_frame](#)
 ([summarize.polars_data_frame](#)),
 [68](#)
[summarise.polars_lazy_frame](#)
 ([summarize.polars_data_frame](#)),
 [68](#)
[summarize.polars_data_frame](#), [68](#)
[summarize.polars_lazy_frame](#)
 ([summarize.polars_data_frame](#)),
 [68](#)
[summary.polars_data_frame](#), [69](#)

[tally.polars_data_frame](#)
 ([count.polars_data_frame](#)), [10](#)
[tally.polars_lazy_frame](#)
 ([count.polars_data_frame](#)), [10](#)
[tibble](#), [8](#)
[tidypolars_options](#), [70](#)
[tidyr::separate_longer_delim\(\)](#), [56](#)
[tidyr::separate_longer_position\(\)](#), [56](#)
[tidyr::unnest_longer\(\)](#), [74](#)

[uncount.polars_data_frame](#), [71](#)
[uncount.polars_lazy_frame](#)
 ([uncount.polars_data_frame](#)), [71](#)
[ungroup.polars_data_frame](#)
 ([group_by.polars_data_frame](#)),
 [32](#)
[ungroup.polars_lazy_frame](#)
 ([group_by.polars_data_frame](#)),
 [32](#)
[unite.polars_data_frame](#), [72](#)
[unite.polars_lazy_frame](#)
 ([unite.polars_data_frame](#)), [72](#)
[unnest_longer_polars](#), [73](#)

[vctrs::vec_as_names\(\)](#), [5](#)

[write_csv_polars](#), [75](#)

[write_ipc_polars](#), [77](#)
[write_json_polars](#), [78](#)
[write_ndjson_polars](#), [79](#)
[write_parquet_polars](#), [79](#)