Package: tidypolars (via r-universe)

November 19, 2024

Type Package

Title Get the Power of Polars with the Syntax of the Tidyverse

Version 0.12.0

Description Polars is an amazing 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

RoxygenNote 7.3.2

Roxygen list(markdown = TRUE)

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

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

Depends R (>= 4.1.0)

Imports dplyr, glue, lifecycle, polars (>= 0.21.0), rlang, tidyr, tidyselect, utils, vctrs

Suggests arrow, bench, data.table, 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/pak/sysreqs libicu-dev

Repository https://r-multiverse.r-universe.dev

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

RemoteRef v0.12.0

RemoteSha 4508d4d25685eed1ecee8a9450c9b804b34311ef

Contents

`arrange.RPolarsDataFrame` *Order rows using column values*

Description

Order rows using column values

Usage

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

Arguments

`.data`	A Polars Data/LazyFrame
`...`	Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by_group`	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)
```

as_tibble.tidypolars *Convert a Polars DataFrame to an R data.frame or to a tibble*

Description

This makes it easier to convert a polars DataFrame or LazyFrame to a `tibble` in a pipe workflow.

Usage

```
## S3 method for class 'tidypolars'
as_tibble(x, int64_conversion = polars::polars_options()$int64_conversion, ...)
```

Arguments

`x`	A Polars Data/LazyFrame
`int64_conversion`	How should Int64 values be handled when converting a polars object to R? See the documentation in `polars::as.data.frame.RPolarsDataFrame`.
`...`	Options passed to `polars::as.data.frame.RPolarsDataFrame`.

About int64

Int64 is a format accepted in Polars but not natively in R (the package `bit64` helps with that).

Since `tidypolars` is simply a wrapper around `polars`, the behavior of `int64` values will depend on the options set in `polars`. Use `options(polars.int64_conversion =)` to specify how int64 variables should be handled. See the documentation in `polars` for the possible options.

Examples

```
iris |>
  as_polars_df() |>
  filter(Sepal.Length > 6) |>
  as_tibble()
```

bind_cols_polars *Append multiple Data/LazyFrames next to each other*

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` *Stack multiple Data/LazyFrames on top of each other*

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 colum
bind_rows_polars(p1, p2, .id = "id")

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

`complete.RPolarsDataFrame`

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 'RPolarsDataFrame'
complete(data, ..., fill = list())

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

Arguments

`data`	A Polars Data/LazyFrame
`...`	Any expression accepted by `dplyr::select()`: variable names, column numbers, select helpers, etc.
`fill`	A named list that for each variable supplies a single value to use instead of `NA` for missing combinations.

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)

df |>
  complete(
    group, item_id, item_name,
    fill = list(value1 = 0, value2 = 99)
  )

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

---

compute.RPolarsLazyFrame

Collect a LazyFrame

Description

`compute()` checks the query, optimizes it in the background, and runs it. The output is a Polars DataFrame. `collect()` is similar to `compute()` but converts the output to an R data.frame, which consumes more memory.

Until `tidypolars` 0.7.0, there was only `collect()` and it was used to collect a LazyFrame into a Polars DataFrame. This usage is still valid for now but will change in 0.8.0 to automatically convert a DataFrame to a `data.frame`. Use `compute()` to have a Polars DataFrame as output.

Usage

```
## S3 method for class 'RPolarsLazyFrame'
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,
  streaming = FALSE,
  collect_in_background = FALSE
)

## S3 method for class 'RPolarsLazyFrame'
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,
  streaming = FALSE,
  collect_in_background = FALSE
)
```

Arguments

x	A Polars LazyFrame
...	Not used.
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`.

`streaming`
: Run parts of the query in a streaming fashion (this is in an alpha state). Default is `FALSE`.

`collect_in_background`
: Detach this query from the R session. Computation will start in background. Get a handle which later can be converted into the resulting DataFrame. Useful in interactive mode to not lock R session (default is `FALSE`).

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.RPolarsDataFrame

Count the observations in each group

Description

Count the observations in each group

Usage

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

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

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

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

Arguments

x	A Polars Data/LazyFrame
...	Any expression accepted by `dplyr::select()`: variable names, column numbers, select helpers, etc.
sort	If TRUE, will show the largest groups at the top.
name	Name of the new column.

Examples

```
test <- polars::as_polars_df(mtcars)
count(test, cyl)

count(test, cyl, am)

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

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

---

cross_join.RPolarsDataFrame

Cross join

Description

Cross joins match each row in x to every row in y, resulting in a dataset with `nrow(x) * nrow(y)` rows.

Usage

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

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

Arguments

x, y	Two Polars Data/LazyFrames
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.
...	Not used.

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)
```

---

describe *Summary statistics for a Polars DataFrame*

Description

[Deprecated]

This function is deprecated as of tidypolars 0.10.0, it will be removed in a future update. Use `summary()` with the same arguments instead.

Usage

```
describe(.data, percentiles = c(0.25, 0.75))
```

Arguments

.data	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).

describe_plan *Show the optimized and non-optimized query plans*

Description

[Deprecated]

Those functions are deprecated as of tidypolars 0.10.0, they will be removed in a future update. Use `explain()` with `optimized = FALSE` to recover the output of `describe_plan()`, and with `optimized = TRUE` (the default) to get the output of `describe_optimized_plan()`.

Usage

```
describe_plan(.data)

describe_optimized_plan(.data)
```

Arguments

`.data`	A Polars LazyFrame

distinct.RPolarsDataFrame *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 occurence or remove all duplicates.

Usage

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

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

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

Arguments

.data	A Polars Data/LazyFrame
...	Any expression accepted by dplyr::select(): variable names, column numbers, select helpers, etc.
keep	Either "first" (keep the first occurrence of the duplicated row), "last" (last occurrence) or "none" (remove all ofccurences 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.RPolarsDataFrame

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 'RPolarsDataFrame'
drop_na(data, ...)

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

Arguments

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

Examples

```
tmp <- mtcars
tmp[1:3, "mpg"] <- NA
tmp[4, "hp"] <- NA
pl_tmp <- polars::pl$DataFrame(tmp)

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

`explain.RPolarsLazyFrame` *Show the optimized and non-optimized query plans*

Description

This function is available for `LazyFrame`s 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 not the query performed. Note that the plans are read from bottom to top.

Usage

```
## S3 method for class 'RPolarsLazyFrame'
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

Fetch is a way to collect only the first `n` rows of a LazyFrame. It is mainly used to test that a query runs as expected on a subset of the data before using `collect()` on the full query. Note that fetching `n` rows doesn't mean that the output will actually contain `n` rows, see the section 'Details' for more information.

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,
  streaming = FALSE
)
```

Arguments

`.data`	A Polars LazyFrame
`n_rows`	Number of rows to fetch.
`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.

streaming
: Run parts of the query in a streaming fashion (this is in an alpha state). Default is FALSE.

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.

Examples

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

# this will return 30 rows
fetch(dat_lazy, 30)

# this will return less than 30 rows because there are less than 30 matches
# for this filter in the whole dataset
dat_lazy |>
  filter(Sepal.Length > 7.0) |>
  fetch(30)
```

---

fill.RPolarsDataFrame *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 'RPolarsDataFrame'
fill(data, ..., .direction = c("down", "up", "downup", "updown"))
```

Arguments

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

Keep rows that match a condition

Description

This function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of TRUE for all conditions. Note that when a condition evaluates to NA the row will be dropped, unlike base subsetting with [.

Usage

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

## S3 method for class 'RPolarsLazyFrame'
filter(.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.
.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.

Examples

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

filter(pl_iris, Sepal.Length < 5, Species == "setosa")

filter(pl_iris, Sepal.Length < Sepal.Width + Petal.Length)

filter(pl_iris, Species == "setosa" | is.na(Species))

iris2 <- iris
iris2$Species <- as.character(iris2$Species)
iris2 |>
  as_polars_df() |>
  filter(Species %in% c("setosa", "virginica"))

# filter by group
pl_iris |>
  group_by(Species) |>
  filter(Sepal.Length == max(Sepal.Length)) |>
  ungroup()

# an alternative syntax for grouping is to use `.by`
pl_iris |>
  filter(Sepal.Length == max(Sepal.Length), .by = Species)
```

---

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,
  dtypes = 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,
  reuse_downloaded = TRUE
)

scan_csv_polars(
  source,
  ...,
  has_header = TRUE,
  separator = ",",
  comment_prefix = NULL,
  quote_char = "\"",
  skip_rows = 0,
  dtypes = 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,
  reuse_downloaded = TRUE
)
```

Arguments

source	Path to a file or URL. It is possible to provide multiple paths provided that all CSV files have the same schema. It is not possible to provide several URLs.
...	Ignored.
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" 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.
dtypes	Named list of column names - dtypes or dtype - column names. This list is used while reading to overwrite dtypes. Supported types so far are: • "Boolean" or "logical" for DataType::Boolean, • "Categorical" or "factor" for DataType::Categorical, • "Float32" or "double" for DataType::Float32, • "Float64" or "float64" for DataType::Float64, • "Int32" or "integer" for DataType::Int32, • "Int64" or "integer64" for DataType::Int64, • "String" or "character" for DataType::String,
null_values	Values to interpret as NA values. Can be: • a character vector: all values that match one of the values in this vector will be NA; • a named list with column names and null values.
ignore_errors	Keep reading the file even if some lines yield errors. You can also use infer_schema_length = 0 to read all columns as UTF8 to check which values might cause an issue.
cache	Cache the result after reading.

infer_schema_length
: Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If NULL, a full table scan will be done (slow).

n_rows
: Maximum number of rows to read.

encoding
: Either "utf8" or "utf8-lossy". Lossy means that invalid UTF8 values are replaced with "?" characters.

low_memory
: Reduce memory usage (will yield a lower performance).

rechunk
: Reallocate to contiguous memory when all chunks / files are parsed.

skip_rows_after_header
: Parse the first row as headers, and then skip this number of rows.

row_index_name
: If not NULL, this will insert a row index column with the given name into the DataFrame.

row_index_offset
: Offset to start the row index column (only used if the name is set).

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 pl$String.

eol_char
: Single byte end of line character (default: \n). When encountering a file with Windows line endings (\r\n), one can go with the default \n. The extra \r 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.

reuse_downloaded
: If TRUE(default) and a URL was provided, cache the downloaded files in session for an easy reuse.

---

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,
  memory_map = TRUE,
  row_index_name = NULL,
```

```
  row_index_offset = 0L,
  rechunk = FALSE,
  cache = TRUE,
  include_file_paths = NULL
)

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

Arguments

source	Path to a file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
memory_map	A logical. If TRUE, try to memory map the file. This can greatly improve performance on repeated queries as the OS may cache pages. Only uncompressed Arrow IPC files can be memory mapped.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
rechunk	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
cache	Cache the result after reading.
include_file_paths	Character value indicating the column name that will include the path of the source file(s).

Details

Hive-style partitioning is not supported yet.

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,
  reuse_downloaded = TRUE,
  ignore_errors = FALSE
)

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,
  reuse_downloaded = TRUE,
  ignore_errors = FALSE
)
```

Arguments

`source`	Path to a file or URL. It is possible to provide multiple paths provided that all NDJSON files have the same schema. It is not possible to provide several URLs.
`...`	Ignored.
`infer_schema_length`	Maximum number of rows to read to infer the column types. If set to 0, all columns will be read as UTF-8. If `NULL`, a full table scan will be done (slow).

batch_size	Number of rows that will be processed per thread.
n_rows	Maximum number of rows to read.
low_memory	Reduce memory usage (will yield a lower performance).
rechunk	Reallocate to contiguous memory when all chunks / files are parsed.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
reuse_downloaded	If TRUE(default) and a URL was provided, cache the downloaded files in session for an easy reuse.
ignore_errors	Keep reading the file even if some lines yield errors. You can also use infer_schema_length = 0 to read all columns as UTF8 to check which values might cause an issue.

---

from_parquet *Import data from Parquet file(s)*

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 to a file. You can use globbing with * to scan/read multiple files in the same directory (see examples).
...	Ignored.
n_rows	Maximum number of rows to read.
row_index_name	If not NULL, this will insert a row index column with the given name into the DataFrame.
row_index_offset	Offset to start the row index column (only used if the name is set).
parallel	This determines the direction of parallelism. "auto" will try to determine the optimal direction. Can be "auto", "columns", "row_groups", "prefiltered", or "none". See 'Details'.
hive_partitioning	Infer statistics and schema from Hive partitioned URL and use them to prune reads. If NULL (default), it is automatically enabled when a single directory is passed, and otherwise disabled.
hive_schema	A list containing the column names and data types of the columns by which the data is partitioned, e.g. list(a = pl$String, b = pl$Float32). If NULL (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	In case of reading multiple files via a glob pattern, rechunk the final DataFrame into contiguous memory chunks.
low_memory	Reduce memory usage (will yield a lower performance).
storage_options	Experimental. List of options necessary to scan parquet files from different cloud storage providers (GCP, AWS, Azure, HuggingFace). See the 'Details' section.

`use_statistics` Use statistics in the parquet file 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.

Details

On parallel strategies:

The prefiltered 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.

The prefiltered settings falls back to auto if no predicate is given.

Connecting to cloud providers:

Polars supports scanning parquet files from different cloud providers. The cloud providers currently supported are AWS, GCP, and Azure. The supported keys to pass to the `storage_options` argument can be found here:

- aws
- gcp
- azure

Currently it is impossible to scan public parquet files from GCP without a valid service account. Be sure to always include a service account in the `storage_options` argument.

Scanning from HuggingFace:

It is possible to scan data stored on HuggingFace using a path starting with `hf://`. The `hf://` path format is defined as `hf://BUCKET/REPOSITORY@REVISION/PATH`, where:

- BUCKET is one of datasets or spaces
- REPOSITORY is the location of the repository. this is usually in the format of username/repo_name. A branch can also be optionally specified by appending `@branch`.
- REVISION is the name of the branch (or commit) to use. This is optional and defaults to main if not given.
- PATH is a file or directory path, or a glob pattern from the repository root.

A Hugging Face API key can be passed to access private locations using either of the following methods:

- Passing a token in storage_options to the scan function, e.g. `scan_parquet(..., storage_options = list(token =`
- Setting the HF_TOKEN environment variable, e.g. `Sys.setenv(HF_TOKEN = <your HF token>)`.

`group_by.RPolarsDataFrame` *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 'RPolarsDataFrame'
group_by(.data, ..., maintain_order = FALSE, .add = FALSE)

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

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

## S3 method for class 'RPolarsLazyFrame'
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`.
`x`	A Polars Data/LazyFrame

Examples

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

by_cyl

by_cyl |> summarise(
  disp = mean(disp),
  hp = mean(hp)
)
```

```
by_cyl |> filter(disp == max(disp))
```

`group_split.RPolarsDataFrame` *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 'RPolarsDataFrame'
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.RPolarsDataFrame` *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 'RPolarsDataFrame'
group_vars(x)

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

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

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

Arguments

x, .tbl	A Polars Data/LazyFrame
...	Not used.

Examples

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

group_vars(pl_g)

group_keys(pl_g)
```

---

left_join.RPolarsDataFrame

Mutating joins

Description

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

Usage

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

```
  na_matches = "na",
  relationship = NULL
)

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

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

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

## S3 method for class 'RPolarsLazyFrame'
left_join(
  x,
  y,
  by = NULL,
```

```
  copy = NULL,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = "na",
  relationship = NULL
)

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

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

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

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.
copy, keep	Not used.
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.
...	Not used.
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.
relationship	Handling of the expected relationship between the keys of x and y. Must be one of the following: • NULL, the default, is equivalent to "many-to-many". It doesn't expect any relationship between x and y. • "one-to-one" expects each row in x to match at most 1 row in y and each row in y to match at most 1 row in x. • "one-to-many" expects each row in y to match at most 1 row in x. • "many-to-one" expects each row in x matches at most 1 row in y.

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

Create a column with unique id per row values

Usage

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

Arguments

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

Examples

```
mtcars |>
  as_polars_df() |>
  make_unique_id(am, gear)
```

mutate.RPolarsDataFrame *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 'RPolarsDataFrame'
mutate(.data, ..., .by = NULL, .keep = c("all", "used", "unused", "none"))

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

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.
`.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.

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 }})
```

`pivot_longer.RPolarsDataFrame` *Pivot a Data/LazyFrame from wide to long*

Description

Pivot a Data/LazyFrame from wide to long

Usage

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

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

Arguments

data	A Polars Data/LazyFrame
cols	Columns to pivot into longer format. Can be anything accepted by `dplyr::select()`.
...	Not used.
names_to	The (quoted) name of the column that will contain the column names specified by `cols`.
names_prefix	A regular expression used to remove matching text from the start of each variable name.
values_to	A string specifying the name of the column to create from the data stored in cell values.

Examples

```
pl_relig_income <- polars::pl$DataFrame(tidyr::relig_income)
pl_relig_income

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

pl_billboard <- polars::pl$DataFrame(tidyr::billboard)
pl_billboard
```

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

`pivot_wider.RPolarsDataFrame` *Pivot a DataFrame from long to wide*

Description

Pivot a DataFrame from long to wide

Usage

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

Arguments

`data`	A Polars DataFrame (LazyFrames are not supported).
`...`	Not used.
`id_cols`	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 `names_from` and `values_from`. If a tidyselect expression is supplied, it will be evaluated on data after removing the columns specified through `names_from` and `values_from`.
`names_from`	The (quoted or unquoted) column names whose values will be used for the names of the new columns.
`values_from`	The (quoted or unquoted) column names whose values will be used to fill the new columns.

names_prefix	String added to the start of every variable name. This is particularly useful if names_from is a numeric vector and you want to create syntactic variable names.
names_sep	If names_from or values_from contains multiple variables, this will be used to join their values together into a single string to use as a column name.
names_glue	Instead of names_sep and names_prefix, you can supply a glue specification that uses the names_from columns to create custom column names.
values_fill	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.

Examples

```
pl_fish_encounters <- polars::pl$DataFrame(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.RPolarsDataFrame *Extract a variable of a Data/LazyFrame*

Description

This returns an R vector and not a Polars Series.

Usage

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

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

Arguments

.data	A Polars Data/LazyFrame
var	A quoted or unquoted variable name, or a variable index.
...	Not used.

Examples

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

---

relocate.RPolarsDataFrame

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 'RPolarsDataFrame'
relocate(.data, ..., .before = NULL, .after = NULL)

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

Arguments

.data	A Polars Data/LazyFrame
...	Any expression accepted by dplyr::select(): variable names, column numbers, select helpers, etc.
.before, .after	Column name (either quoted or unquoted) that indicates the destination of columns selected by .... 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.RPolarsDataFrame

Rename columns

Description

Rename columns

Usage

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

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

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

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

Arguments

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

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.RPolarsDataFrame

Replace NAs with specified values

Description

Replace NAs with specified values

Usage

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

## S3 method for class 'RPolarsLazyFrame'
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.
...	Not used.

Examples

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

# replace all NA with 0
replace_na(pl_test, 0)

# custom replacement per column
replace_na(pl_test, list(x = 0, y = 999))
```

---

rowwise.RPolarsDataFrame

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 'RPolarsDataFrame'
rowwise(data, ...)

## S3 method for class 'RPolarsLazyFrame'
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.RPolarsDataFrame

Select columns from a Data/LazyFrame

Description

Select columns from a Data/LazyFrame

Usage

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

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

Arguments

.data	A Polars Data/LazyFrame
...	Any expression accepted by `dplyr::select()`: 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.RPolarsDataFrame` *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 'RPolarsDataFrame'
semi_join(x, y, by = NULL, ..., na_matches = "na")

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

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

## S3 method for class 'RPolarsLazyFrame'
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.
`...`	Not used.
`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.

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.RPolarsDataFrame

Separate a character column into multiple columns based on a substring

---

Description

Currently, splitting a column on a regular expression or position is not possible.

Usage

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

## S3 method for class 'RPolarsLazyFrame'
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	String that is used to split the column. Regular expressions are not supported yet.
remove	If TRUE, remove input column from output data frame.
...	Not used.

Examples

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

---

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 = "\"",
  batch_size = 1024,
  datetime_format = NULL,
  date_format = NULL,
  time_format = NULL,
  float_precision = NULL,
  null_values = "",
  quote_style = "necessary",
  maintain_order = TRUE,
```

```
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE
)
```

Arguments

`.data`	A Polars LazyFrame.
`path`	Output file (must be a `.csv` file).
`...`	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`	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 `?strptime()` for accepted values. If no format specified, the default fractional-second precision is inferred from the maximum time unit found in the `Datetime` cols (if any).
`float_precision`	Number of decimal places to write, applied to both `Float32` and `Float64` datatypes.
`null_values`	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 `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.

Value

The input LazyFrame.

Examples

```
## Not run:
# 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)


## End(Not run)
```

---

sink_ipc *Stream output to an IPC file*

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",
  maintain_order = TRUE,
  type_coercion = TRUE,
  predicate_pushdown = TRUE,
  projection_pushdown = TRUE,
  simplify_expression = TRUE,
  slice_pushdown = TRUE,
  no_optimization = FALSE
)
```

Arguments

.data	A Polars LazyFrame.
path	Output file.
...	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.
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.

Value

The input LazyFrame.

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
)
```

Arguments

.data	A Polars LazyFrame.
path	Output file.
...	Ignored.
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.

Value

The input LazyFrame.

`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
)
```

Arguments

`.data`	A Polars LazyFrame.
`path`	Output file (must be a `.parquet` file).
`...`	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`.

Value

The input LazyFrame.

Examples

```
## Not run:
# 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)


## End(Not run)
```

`slice_tail.RPolarsDataFrame` *Subset rows of a Data/LazyFrame*

Description

Subset rows of a Data/LazyFrame

Usage

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

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

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

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

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

Arguments

`.data`	A Polars Data/LazyFrame
`...`	Not used.
`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).

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.RPolarsDataFrame

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 'RPolarsDataFrame'
summarize(.data, ..., .by = NULL, .groups = "drop_last")

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

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

## S3 method for class 'RPolarsLazyFrame'
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.RPolarsDataFrame`

Summary statistics for a Polars DataFrame

Description

Summary statistics for a Polars DataFrame

Usage

```
## S3 method for class 'RPolarsDataFrame'
summary(object, percentiles = c(0.25, 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` *global options*

Description

There is currently one global option:

- `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.

Examples

```
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")
```

---

uncount.RPolarsDataFrame

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 'RPolarsDataFrame'
uncount(data, weights, ..., .remove = TRUE, .id = NULL)

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

Arguments

data	A Polars Data/LazyFrame
weights	A vector of weights. Evaluated in the context of data.
...	Not used.
.remove	If TRUE, and weights is the name of a column in data, then this column is removed.
.id	Supply a string to create a new variable which gives a unique identifier for each created row.

Examples

```
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.RPolarsDataFrame

Unite multiple columns into one by pasting strings together

Description

Unite multiple columns into one by pasting strings together

Usage

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

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

Arguments

data	A Polars Data/LazyFrame
col	The name of the new column, as a string or symbol.

...	Any expression accepted by `dplyr::select()`: variable names, column numbers, select helpers, etc.
sep	Separator to use between values.
remove	If TRUE, remove input columns from the output Data/LazyFrame.
na.rm	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)
```

---

write_csv_polars *Export data to CSV file(s)*

Description

Export data to CSV file(s)

Usage

```
write_csv_polars(
  .data,
  file,
  ...,
  include_bom = FALSE,
  include_header = TRUE,
  separator = ",",
  line_terminator = "\n",
```

```
  quote = "\"",
  batch_size = 1024,
  datetime_format = NULL,
  date_format = NULL,
  time_format = NULL,
  float_precision = NULL,
  null_values = "",
  quote_style = "necessary"
)
```

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`	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_values`	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).

Value

The input DataFrame.

Examples

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

read.csv(dest)
```

---

write_ipc_polars *Export data to IPC file(s)*

Description

Export data to IPC file(s)

Usage

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

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.
future	Setting this to TRUE will write Polars' internal data structures that might not be available by other Arrow implementations.

Value

The input DataFrame.

write_json_polars *Export data to JSON file(s)*

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	Pretty serialize JSON.
row_oriented	Write to row-oriented JSON. This is slower, but more common.

Value

The input DataFrame.

Examples

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

jsonlite::fromJSON(dest)
```

write_ndjson_polars *Export data to NDJSON file(s)*

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.

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 *Export data to Parquet file(s)*

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
)
```

Arguments

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

`partition_by`
: Column(s) to partition by. A partitioned dataset will be written if this is specified.

`partition_chunk_size_bytes`
: 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.

Value

The input DataFrame.

Examples

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

nanoparquet::read_parquet(dest)
```

Index