Package 'fastplyr'

November 21, 2025

Title Fast Alternatives to 'tidyverse' Functions

Version 0.9.91

Description A full set of fast data manipulation tools with a tidy front-end and a fast back-end using 'collapse' and 'cheapr'.

License MIT + file LICENSE

BugReports https://github.com/NicChr/fastplyr/issues

Depends R (>= 4.1.0)

Imports cheapr (>= 1.3.2), cli, collapse (>= 2.0.0), dplyr (>= 1.1.0), lifecycle, purrr, rlang, stringr, tidyselect, vctrs (>= 0.6.0)

Suggests data.table, nycflights13, testthat (>= 3.0.0), tidyr

LinkingTo cheapr, cpp11

SystemRequirements C++17

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation yes

Author Nick Christofides [aut, cre] (ORCID: <https://orcid.org/0000-0002-9743-7342>)

Maintainer Nick Christofides <nick.christofides.r@gmail.com>

Repository CRAN

Date/Publication 2025-11-21 10:50:02 UTC

Contents

`add_group_id` *Add a column of useful IDs (group IDs, row IDs & consecutive IDs)*

Description

Add a column of useful IDs (group IDs, row IDs & consecutive IDs)

Usage

```
add_group_id(.data, ...)

## S3 method for class 'data.frame'
add_group_id(
  .data,
  ...,
  .order = group_by_order_default(.data),
  .ascending = TRUE,
  .by = NULL,
  .cols = NULL,
  .name = NULL,
  as_qg = FALSE
```

```
)

add_row_id(.data, ...)

## S3 method for class 'data.frame'
add_row_id(
  .data,
  ...,
  .ascending = TRUE,
  .by = NULL,
  .cols = NULL,
  .name = NULL
)

add_consecutive_id(.data, ...)

## S3 method for class 'data.frame'
add_consecutive_id(
  .data,
  ...,
  .order = group_by_order_default(.data),
  .by = NULL,
  .cols = NULL,
  .name = NULL
)
```

Arguments

.data	A data frame.
...	Additional groups using tidy data-masking rules. To specify groups using tidyselect, simply use the .by argument.
.order	Should the groups be ordered? When .order is TRUE (the default) the group IDs will be ordered but not sorted. If FALSE the order of the group IDs will be based on first appearance.
.ascending	Should the order be ascending or descending? The default is TRUE. For add_row_id() this determines if the row IDs are in increasing or decreasing order. **NOTE** - When order = FALSE, the ascending argument is ignored. This is something that will be fixed in a later version.
.by	Alternative way of supplying groups using tidyselect notation.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.name	Name of the added ID column which should be a character vector of length 1. If .name = NULL (the default), add_group_id() will add a column named "group_id", and if one already exists, a unique name will be used.
as_qg	Should the group IDs be returned as a collapse "qG" class? The default (FALSE) always returns an integer vector.

Value

A data frame with the requested ID column.

See Also

group_id row_id f_consecutive_id

`desc` *Helpers to sort variables in ascending or descending order*

Description

An alternative to `dplyr::desc()` which is much faster for character vectors and factors.

Usage

```
desc(x)
```

Arguments

`x`	Vector.

Value

A numeric vector that can be ordered in ascending or descending order.
Useful in `dplyr::arrange()` or `f_arrange()`.

`fastplyr_options` *Setting global fastplyr options*

Description

Helper functions to allow users to:

- Enable or disable optimisations for common functions package-wide
- Enable or disable informative messages

Usage

```
fastplyr_enable_optimisations()

fastplyr_disable_optimisations()

fastplyr_enable_informative_msgs()

fastplyr_disable_informative_msgs()
```

Value

Enables or disables fastplyr global options invisibly.

See Also

get_group_unaware_fns

`f_arrange` *A* `collapse` *version of* `dplyr::arrange()`

Description

This is a fast and near-identical alternative to `dplyr::arrange()` using the `collapse` package.

`desc()` is like `dplyr::desc()` but works faster when called directly on vectors.

Usage

```
f_arrange(
  .data,
  ...,
  .by = NULL,
  .by_group = FALSE,
  .cols = NULL,
  .descending = FALSE,
  .in_place = FALSE
)
```

Arguments

`.data`	A data frame.
`...`	Variables to arrange by.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using `tidyselect`.
`.by_group`	If `TRUE` the sorting will be first done by the group variables.
`.cols`	(Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
`.descending`	`[logical(1)]` data frame be arranged in descending order? Default is `FALSE`. In simple cases this can be easily achieved through `desc()` but for a mixture of ascending and descending variables, it's easier to use the `.descending` arg to reverse the order.
`.in_place`	Should data be sorted in-place? This can be very efficient for large data frames and can be safely used when overwriting a freshly allocated data frame. If you're unsure whether the data frame is a freshly allocated object, use `cheapr::semi_copy()` before sorting. Please note that no new vectors and no copies are created, data is directly sorted in-memory. This only works on data frames consisting of atomic vectors.

Value

A sorted `data.frame`.

`f_bind` *Bind data frame rows and columns*

Description

Faster bind rows and columns.

Usage

```
f_bind_rows(...)

f_bind_cols(..., .repair_names = TRUE, .recycle = TRUE)
```

Arguments

`...`	Data frames to bind.
`.repair_names`	Should duplicate column names be made unique? Default is `TRUE`.
`.recycle`	Should inputs be recycled to a common row size? Default is `TRUE`.

Value

`f_bind_rows()` performs a union of the data frames specified via `...` and joins the rows of all the data frames, without removing duplicates.

`f_bind_cols()` joins the columns, creating unique column names if there are any duplicates by default.

`f_count` *A fast replacement to dplyr::count()*

Description

Near-identical alternative to `dplyr::count()`.

Usage

```
f_count(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  .order = group_by_order_default(.data),
  name = NULL,
  .by = NULL,
  .cols = NULL
)

f_add_count(
  .data,
  ...,
  wt = NULL,
  sort = FALSE,
  .order = group_by_order_default(.data),
  name = NULL,
  .by = NULL,
  .cols = NULL
)
```

Arguments

.data	A data frame.
...	Variables to group by.
wt	Frequency weights. Can be NULL or a variable: • If NULL (the default), counts the number of rows in each group. • If a variable, computes sum(wt) for each group.
sort	If TRUE, will show the largest groups at the top.
.order	Should the groups be calculated as ordered groups? If FALSE, this will return the groups in order of first appearance, and in many cases is faster. If TRUE (the default), the groups are returned in sorted order, exactly the same way as dplyr::count.
name	The name of the new column in the output. If there's already a column called n, it will use nn. If there's a column called n and nn, it'll use nnn, and so on, adding ns until it gets a new name.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.

Details

This is a fast and near-identical alternative to dplyr::count() using the collapse package. Unlike collapse::fcount(), this works very similarly to dplyr::count(). The only main difference is

that anything supplied to `wt` is recycled and added as a data variable. Other than that everything works exactly as the dplyr equivalent.

`f_count()` and `f_add_count()` can be up to >100x faster than the dplyr equivalents.

Value

A `data.frame` of frequency counts by group.

`f_distinct` *Find distinct rows*

Description

Like `dplyr::distinct()` but faster when lots of groups are involved.

Usage

```
f_distinct(
  .data,
  ...,
  .keep_all = FALSE,
  .order = FALSE,
  .sort = deprecated(),
  .by = NULL,
  .cols = NULL
)
```

Arguments

`.data`	A data frame.
`...`	Variables used to find distinct rows.
`.keep_all`	If `TRUE` then all columns of data frame are kept, default is `FALSE`.
`.order`	Should the groups be calculated as ordered groups? Setting to `TRUE` here implies that the groups are returned sorted.
`.sort`	**[Deprecated]** Use `.order` instead.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.cols`	(Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.

Value

A `data.frame` of distinct groups.

`f_duplicates` *Find duplicate rows*

Description

Find duplicate rows

Usage

```
f_duplicates(
  .data,
  ...,
  .keep_all = FALSE,
  .both_ways = FALSE,
  .add_count = FALSE,
  .drop_empty = FALSE,
  .order = FALSE,
  .sort = deprecated(),
  .by = NULL,
  .cols = NULL
)
```

Arguments

`.data`	A data frame.
`...`	Variables used to find duplicate rows.
`.keep_all`	If `TRUE` then all columns of data frame are kept, default is `FALSE`.
`.both_ways`	If `TRUE` then duplicates and non-duplicate first instances are retained. The default is `FALSE` which returns only duplicate rows. Setting this to `TRUE` can be particularly useful when examining the differences between duplicate rows.
`.add_count`	If `TRUE` then a count column is added to denote the number of duplicates (including first non-duplicate instance). The naming convention of this column follows `dplyr::add_count()`.
`.drop_empty`	If `TRUE` then empty rows with all `NA` values are removed. The default is `FALSE`.
`.order`	Should the groups be calculated as ordered groups? Setting to `TRUE` here implies that the groups are returned sorted.
`.sort`	**[Deprecated]** Use `.order` instead.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.cols`	(Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.

Details

This function works like `dplyr::distinct()` in its handling of arguments and data-masking but returns duplicate rows. In certain situations in can be much faster than `data |> group_by() |> filter(n() > 1)` when there are many groups.

Value

A `data.frame` of duplicate rows.

See Also

f_count f_distinct

`f_expand` *Fast versions of* `tidyr::expand()` *and* `tidyr::complete()`.

Description

Fast versions of `tidyr::expand()` and `tidyr::complete()`.

Usage

```
f_expand(.data, ..., .sort = FALSE, .by = NULL, .cols = NULL)

f_complete(.data, ..., .sort = FALSE, .by = NULL, .cols = NULL, fill = NA)

crossing(..., .sort = FALSE)

nesting(..., .sort = FALSE)
```

Arguments

`.data`	A data frame
`...`	Variables to expand.
`.sort`	Logical. If `TRUE` expanded/completed variables are sorted. The default is `FALSE`.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.cols`	(Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
`fill`	A named list containing value-name pairs to fill the named implicit missing values.

Details

`crossing` and `nesting` are helpers that are basically identical to tidyr's `crossing` and `nesting`.

Value

A data.frame of expanded groups.

---

f_fill *Fill* NA *values forwards and backwards*

Description

Fill NA values forwards and backwards

Usage

```
f_fill(
  .data,
  ...,
  .by = NULL,
  .cols = NULL,
  .direction = c("forwards", "backwards"),
  .fill_limit = Inf,
  .new_names = "{.col}"
)
```

Arguments

.data	A data frame.
...	Cols to fill NA values specified through tidyselect notation. If left empty all cols are used by default.
.by	Cols to group by for this operation. Specified through tidyselect.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.direction	Which direction should NA values be filled? By default, "forwards" (Last-Observation-Carried-Forward) is used. "backwards" is (Next-Observation-Carried-Backward).
.fill_limit	The maximum number of consecutive NA values to fill. Default is Inf.
.new_names	A name specification for the names of filled variables. The default "{.col}" replaces the given variables with the imputed ones. New variables can be created alongside the originals if we give a different specification, e.g. .new_names = "{.col}_imputed". This follows the specification of dplyr::across if .fns were an empty string "".

Value

A data frame with NA values filled forward or backward.

f_filter *Alternative to* dplyr::filter()

Description

Alternative to dplyr::filter()

Usage

```
f_filter(.data, ..., .by = NULL)
```

Arguments

.data	A data frame.
...	Expressions used to filter the data frame with.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.

Value

A filtered data frame.

f_group_by *'collapse' version of* dplyr::group_by()

Description

This works the exact same as dplyr::group_by() and typically performs around the same speed but uses slightly less memory.

Usage

```
f_group_by(
  .data,
  ...,
  .add = FALSE,
  .order = group_by_order_default(.data),
  .by = NULL,
  .cols = NULL,
  .drop = df_group_by_drop_default(.data)
)
```

Arguments

.data	data frame.
...	Variables to group by.
.add	Should groups be added to existing groups? Default is FALSE.
.order	Should groups be ordered? If FALSE groups will be ordered based on first-appearance. Typically, setting order to FALSE is faster.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidyselect.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.drop	Should unused factor levels be dropped? Default is TRUE.

Details

f_group_by() works almost exactly like the 'dplyr' equivalent. An attribute "ordered" (TRUE or FALSE) is added to the group data to signify if the groups are sorted or not.

Ordered vs Sorted:

The distinction between ordered and sorted is somewhat subtle. Functions in fastplyr that use a sort argument generally refer to the top-level dataset being sorted in some way, either by sorting the group columns like in f_expand() or f_distinct(), or some other columns, like the count column in f_count().

The .order argument, when set to TRUE (the default), is used to mean that the group data will be calculated using a sort-based algorithm, leading to sorted group data. When .order is FALSE, the group data will be returned based on the order-of-first appearance of the groups in the data. This order-of-first appearance may still naturally be sorted depending on the data. For example, group_id(1:3, order = T) results in the same group IDs as group_id(1:3, order = F) because 1, 2, and 3 appear in the data in ascending sequence whereas group_id(3:1, order = T) does not equal group_id(3:1, order = F)

Part of the reason for the distinction is that internally fastplyr can in theory calculate group data using the sort-based algorithm and still return unsorted groups, though this combination is only available to the user in limited places like f_distinct(.order = TRUE, .sort = FALSE).

The other reason is to prevent confusion in the meaning of sort and order so that order always refers to the algorithm specified, resulting in sorted groups, and sort implies a physical sorting of the returned data. It's also worth mentioning that in most functions, sort will implicitly utilise the sort-based algorithm specified via order = TRUE.

Using the order-of-first appearance algorithm for speed:

In many situations (not all) it can be faster to use the order-of-first appearance algorithm, specified via .order = FALSE.

This can generally be accessed by first calling f_group_by(data, ..., .order = FALSE) and then performing your calculations.

To utilise this algorithm more globally and package-wide, set the '.fastplyr.order.groups' option to FALSE using the code: options(.fastplyr.order.groups = FALSE).

Value

`f_group_by()` returns a `grouped_df` that can be used for further for grouped calculations.

`group_ordered()` returns `TRUE` if the group data are sorted, i.e if `attr(attr(data, "groups"), "ordered") == TRUE`. If sorted, which is usually the default, this leads to summary calculations like `f_summarise()` or `dplyr::summarise()` producing sorted groups. If `FALSE` they are returned based on order-of-first appearance in the data.

---

f_group_split *Alternative to* `dplyr::group_split`

Description

Alternative to `dplyr::group_split`

Usage

```
f_group_split(
  .data,
  ...,
  .add = FALSE,
  .order = group_by_order_default(.data),
  .by = NULL,
  .cols = NULL,
  .drop = df_group_by_drop_default(.data),
  .group_names = FALSE
)
```

Arguments

`.data`	data frame.
`...`	Variables to group by.
`.add`	Should groups be added to existing groups? Default is `FALSE`.
`.order`	Should groups be ordered? If `FALSE` groups will be ordered based on first-appearance. Typically, setting order to `FALSE` is faster.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using `tidyselect`.
`.cols`	(Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
`.drop`	Should unused factor levels be dropped? Default is `TRUE`.
`.group_names`	Should group names be added? Default is `FALSE`.

Value

A list of data frames split by group.

f_left_join *Fast SQL joins*

Description

Mostly a wrapper around `collapse::join()` that behaves more like dplyr's joins. List columns, lubridate intervals and vctrs rcrds work here too.

Usage

```
f_left_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
  ...
)

f_right_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
  ...
)

f_inner_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
  ...
)

f_full_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
```

```
  ...
)

f_anti_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
  ...
)

f_semi_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  multiple = TRUE,
  keep = FALSE,
  ...
)

f_cross_join(x, y, suffix = c(".x", ".y"), ...)

f_union_all(x, y, ...)

f_union(x, y, ...)
```

Arguments

x	Left data frame.
y	Right data frame.
by	character(1) - Columns to join on.
suffix	character(2) - Suffix to paste onto common cols between x and y in the joined output.
multiple	logical(1) - Should multiple matches be returned? If FALSE the first match in y is used. Default is TRUE.
keep	logical(1) - Should join columns from both data frames be kept? Default is FALSE.
...	Additional arguments passed to collapse::join().

Value

A joined data frame, joined on the columns specified with by, using an equality join.

f_cross_join() returns all possible combinations between the two data frames.

`f_mutate` *A faster* `mutate()` *with per-group optimisations*

Description

A faster `mutate()` with per-group optimisations

Usage

```
f_mutate(
  .data,
  ...,
  .by = NULL,
  .order = group_by_order_default(.data),
  .keep = "all"
)
```

Arguments

`.data`	A data frame.
`...`	Name-value pairs of summary functions. Expressions with `across()` are also accepted.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.order`	Should the groups be returned in sorted order? If `FALSE`, this will return the groups in order of first appearance, and in many cases is faster.
`.keep`	Which columns to keep. Options are 'all', 'used', 'unused' and 'none'.

Value

A data frame with added columns.

Details

fastplyr data-masking functions like `f_mutate` and `f_summarise` operate very similarly to their dplyr counterparts but with some crucial differences. Optimisations for by-group operations kick in for common statistical functions which are detailed below. A message will be printed which one can disable by running `options(fastplyr.inform = FALSE)`. When this happens, the expressions which become optimised no longer obey data-masking rules pertaining to sequential and dependent expression execution. For example, the pseudo code `f_summarise(data, mean = mean(x), mean2 = round(mean), .by = g)` when optimised will not work because the named col `mean` will not be visible in later expressions.

One can disable fastplyr optimisations globally by running `options(fastplyr.optimise = F)`.

Optimised statistical functions:

Some functions are internally optimised using 'collapse' fast statistical functions. This makes execution on many groups very fast.

For fast quantiles (percentiles) by group, see tidy_quantiles

List of currently optimised functions

`dplyr::n` -> <custom_expression>
`dplyr::row_number` -> <custom_expression> (only for `f_mutate`)
`dplyr::cur_group` -> <custom_expression>
`dplyr::cur_group_id` -> <custom_expression>
`dplyr::cur_group_rows` -> <custom_expression> (only for `f_mutate`)
`dplyr::lag` -> <custom_expression> (only for `f_mutate`)
`dplyr::lead` -> <custom_expression> (only for `f_mutate`)
`base::sum` -> `collapse::fsum`
`base::prod` -> `collapse::fprod`
`base::min` -> `collapse::fmin`
`base::max` -> `collapse::fmax`
`stats::mean` -> `collapse::fmean`
`stats::median` -> `collapse::fmedian`
`stats::sd` -> `collapse::fsd`
`stats::var` -> `collapse::fvar`
`dplyr::first` -> `collapse::ffirst`
`dplyr::last` -> `collapse::flast`
`dplyr::n_distinct` -> `collapse::fndistinct`

`f_nest_by` *Create a subset of data for each group*

Description

A faster `nest_by()`.

Usage

```
f_nest_by(
  .data,
  ...,
  .add = FALSE,
  .order = group_by_order_default(.data),
  .by = NULL,
  .cols = NULL,
  .drop = df_group_by_drop_default(.data)
)
```

Arguments

.data	data frame.
...	Variables to group by.
.add	Should groups be added to existing groups? Default is FALSE.
.order	Should groups be ordered? If FALSE groups will be ordered based on first-appearance. Typically, setting order to FALSE is faster.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidyselect.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.drop	Should unused factor levels be dropped? Default is TRUE.

Value

A row-wise grouped_df of the corresponding data of each group.

Examples

```
library(dplyr)
library(fastplyr)

# Stratified linear-model example

models <- iris |>
  f_nest_by(Species) |>
  mutate(model = list(lm(Sepal.Length ~ Petal.Width + Petal.Length, data = first(data))),
         summary = list(summary(first(model))),
         r_sq = first(summary)$r.squared)
models
models$summary

# dplyr's `nest_by()` is admittedly more convenient
# as it performs a double bracket subset `[[` on list elements for you
# which we have emulated by using `first()`

# `f_nest_by()` is faster when many groups are involved

models <- iris |>
  nest_by(Species) |>
  mutate(model = list(lm(Sepal.Length ~ Petal.Width + Petal.Length, data = data)),
         summary = list(summary(model)),
         r_sq = summary$r.squared)
models$summary

models$summary[[1]]
```

`f_reframe` *A faster* `reframe()` *with per-group optimisations*

Description

A faster `reframe()` with per-group optimisations

Usage

```
f_reframe(.data, ..., .by = NULL, .order = group_by_order_default(.data))
```

Arguments

`.data`	A data frame.
`...`	Name-value pairs of summary functions. Expressions with `across()` are also accepted.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.order`	Should the groups be returned in sorted order? If `FALSE`, this will return the groups in order of first appearance, and in many cases is faster.

Value

A data frame of specified results.

Details

fastplyr data-masking functions like `f_mutate` and `f_summarise` operate very similarly to their dplyr counterparts but with some crucial differences. Optimisations for by-group operations kick in for common statistical functions which are detailed below. A message will be printed which one can disable by running `options(fastplyr.inform = FALSE)`. When this happens, the expressions which become optimised no longer obey data-masking rules pertaining to sequential and dependent expression execution. For example, the pseudo code `f_summarise(data, mean = mean(x), mean2 = round(mean), .by = g)` when optimised will not work because the named col `mean` will not be visible in later expressions.

One can disable fastplyr optimisations globally by running `options(fastplyr.optimise = F)`.

Optimised statistical functions:

Some functions are internally optimised using 'collapse' fast statistical functions. This makes execution on many groups very fast.

For fast quantiles (percentiles) by group, see tidy_quantiles

List of currently optimised functions

`dplyr::n` -> <custom_expression>
`dplyr::row_number` -> <custom_expression> (only for `f_mutate`)
`dplyr::cur_group` -> <custom_expression>
`dplyr::cur_group_id` -> <custom_expression>

dplyr::cur_group_rows -> <custom_expression> (only for f_mutate)
dplyr::lag -> <custom_expression> (only for f_mutate)
dplyr::lead -> <custom_expression> (only for f_mutate)
base::sum -> collapse::fsum
base::prod -> collapse::fprod
base::min -> collapse::fmin
base::max -> collapse::fmax
stats::mean -> collapse::fmean
stats::median -> collapse::fmedian
stats::sd -> collapse::fsd
stats::var -> collapse::fvar
dplyr::first -> collapse::ffirst
dplyr::last -> collapse::flast
dplyr::n_distinct -> collapse::fndistinct

---

f_rowwise *A convenience function to group by every row*

Description

fastplyr currently cannot handle rowwise_df objects created through dplyr::rowwise() and so this is a convenience function to allow you to perform row-wise operations. For common efficient row-wise functions, see the 'kit' package.

Usage

```
f_rowwise(.data, ..., .ascending = TRUE, .cols = NULL, .name = ".row_id")
```

Arguments

.data	data frame.
...	Variables to group by using tidyselect.
.ascending	Should data be grouped in ascending row-wise order? Default is TRUE.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.name	Name of row-id column to be added.

Value

A row-wise grouped_df.

f_select *Fast 'dplyr'* select()/rename()/pull()

Description

f_select() operates the exact same way as dplyr::select() and can be used naturally with tidy-select helpers. It uses collapse to perform the actual selecting of variables and is considerably faster than dplyr for selecting exact columns, and even more so when supplying the .cols argument.

Usage

```
f_select(data, ..., .cols = NULL)

f_rename(data, ..., .cols = NULL)

f_pull(data, ..., .cols = NULL)

nothing()
```

Arguments

data	A data frame.
...	Variables to select using tidy-select. See ?dplyr::select for more info.
.cols	(Optional) faster alternative to ... that accepts a named character vector or numeric vector. No checks on duplicates column names are done when using .cols. If speed is an expensive resource, it is recommended to use this.

Value

A data.frame of selected columns.

f_slice *Faster* dplyr::slice()

Description

When there are lots of groups, the f_slice() functions are much faster.

Usage

```
f_slice(
  .data,
  i = 0L,
  ...,
  .by = NULL,
  .order = group_by_order_default(.data),
  keep_order = FALSE
)

f_slice_head(
  .data,
  n,
  prop,
  .by = NULL,
  .order = group_by_order_default(.data),
  keep_order = FALSE
)

f_slice_tail(
  .data,
  n,
  prop,
  .by = NULL,
  .order = group_by_order_default(.data),
  keep_order = FALSE
)

f_slice_min(
  .data,
  order_by,
  n,
  prop,
  .by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
  .order = group_by_order_default(.data),
  keep_order = FALSE
)

f_slice_max(
  .data,
  order_by,
  n,
  prop,
  .by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
```

```
  .order = group_by_order_default(.data),
  keep_order = FALSE
)

f_slice_sample(
  .data,
  n,
  replace = FALSE,
  prop,
  .by = NULL,
  .order = group_by_order_default(.data),
  keep_order = FALSE,
  weights = NULL
)
```

Arguments

.data	A data frame.
i	An integer vector of slice locations. Please see the details below on how i works as it only accepts simple integer vectors.
...	A temporary argument to give the user an error if dots are used.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
.order	Should the groups be returned in sorted order? If FALSE, this will return the groups in order of first appearance, and in many cases is faster.
keep_order	Should the sliced data frame be returned in its original order? The default is FALSE.
n	Number of rows.
prop	Proportion of rows.
order_by	Variables to order by.
with_ties	Should ties be kept together? The default is TRUE.
na_rm	Should missing values in f_slice_max() and f_slice_min() be removed? The default is FALSE.
replace	Should f_slice_sample() sample with or without replacement? Default is FALSE, without replacement.
weights	Probability weights used in f_slice_sample().

Details

Important note about the i argument in f_slice:

i is first evaluated on an un-grouped basis and then searches for those locations in each group. Thus if you supply an expression of slice locations that vary by-group, this will not be respected nor checked. For example,
do f_slice(data, 10:20, .by = group)

not `f_slice(data, sample(1:10), .by = group)`.

The former results in slice locations that do not vary by group but the latter will result in different within-group slice locations which `f_slice` cannot correctly compute.

To do the the latter type of by-group slicing, use `f_filter`, e.g.
`f_filter(data, row_number() %in% slices, .by = groups)` or even faster:
`library(cheapr)`
`f_filter(data, row_number() %in_% slices, .by = groups)`

`f_slice_sample`**:**

The arguments of `f_slice_sample()` align more closely with `base::sample()` and thus by default re-samples each entire group without replacement.

Value

A `data.frame` filtered on the specified row indices.

`f_summarise` *Summarise each group down to one row*

Description

Like `dplyr::summarise()` but with some internal optimisations for common statistical functions.

Usage

```
f_summarise(.data, ..., .by = NULL, .order = group_by_order_default(.data))

f_summarize(.data, ..., .by = NULL, .order = group_by_order_default(.data))
```

Arguments

`.data`	A data frame.
`...`	Name-value pairs of summary functions. Expressions with `across()` are also accepted.
`.by`	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
`.order`	Should the groups be returned in sorted order? If `FALSE`, this will return the groups in order of first appearance, and in many cases is faster.

Value

An un-grouped data frame of summaries by group.

Details

fastplyr data-masking functions like `f_mutate` and `f_summarise` operate very similarly to their dplyr counterparts but with some crucial differences. Optimisations for by-group operations kick in for common statistical functions which are detailed below. A message will be printed which one can disable by running `options(fastplyr.inform = FALSE)`. When this happens, the expressions which become optimised no longer obey data-masking rules pertaining to sequential and dependent expression execution. For example, the pseudo code `f_summarise(data, mean = mean(x), mean2 = round(mean), .by = g)` when optimised will not work because the named col `mean` will not be visible in later expressions.

One can disable fastplyr optimisations globally by running `options(fastplyr.optimise = F)`.

Optimised statistical functions:

Some functions are internally optimised using 'collapse' fast statistical functions. This makes execution on many groups very fast.

For fast quantiles (percentiles) by group, see tidy_quantiles

List of currently optimised functions

`dplyr::n` -> <custom_expression>
`dplyr::row_number` -> <custom_expression> (only for `f_mutate`)
`dplyr::cur_group` -> <custom_expression>
`dplyr::cur_group_id` -> <custom_expression>
`dplyr::cur_group_rows` -> <custom_expression> (only for `f_mutate`)
`dplyr::lag` -> <custom_expression> (only for `f_mutate`)
`dplyr::lead` -> <custom_expression> (only for `f_mutate`)
`base::sum` -> `collapse::fsum`
`base::prod` -> `collapse::fprod`
`base::min` -> `collapse::fmin`
`base::max` -> `collapse::fmax`
`stats::mean` -> `collapse::fmean`
`stats::median` -> `collapse::fmedian`
`stats::sd` -> `collapse::fsd`
`stats::var` -> `collapse::fvar`
`dplyr::first` -> `collapse::ffirst`
`dplyr::last` -> `collapse::flast`
`dplyr::n_distinct` -> `collapse::fndistinct`

See Also

tidy_quantiles

Examples

```
library(fastplyr)
library(nycflights13)
library(dplyr)
options(fastplyr.inform = FALSE)
# Number of flights per month, including first and last day
flights |>
```

```
  f_group_by(year, month) |>
  f_summarise(first_day = first(day),
              last_day = last(day),
              num_flights = n())

## Fast mean summary using `across()`

flights |>
  f_summarise(
    across(where(is.numeric), mean),
    .by = tailnum
  )

flights |>
  f_group_by(.cols = "tailnum") |>
  f_summarise(
    across(where(is.numeric), mean)
  )
```

---

f_ungroup *Un-group* grouped_df

Description

Un-group grouped_df

Usage

```
f_ungroup(data)

group_ordered(data)
```

Arguments

data	A data frame.

Value

An un-grouped data frame.

get_group_unaware_fns *Get list of current group-unaware functions*

Description

Get list of current group-unaware functions

Usage

```
get_group_unaware_fns()
```

Value

A named list of functions marked as group-unaware in fastplyr.

Examples

```
library(fastplyr)

fns <- get_group_unaware_fns()

names(fns)
fns$round
```

group_by_order_default *Default value for ordering of groups*

Description

A default value, TRUE or FALSE that controls which algorithm to use for calculating groups. See f_group_by for more details.

Usage

```
group_by_order_default(x)
```

Arguments

x	A data frame.

Value

A logical of length 1, either TRUE or FALSE.

group_data *Fast group metadata*

Description

Fast group metadata

Usage

```
f_group_data(x)

f_group_keys(x)

f_group_rows(x)

f_group_indices(x)

f_group_vars(x)

f_group_size(x)

f_n_groups(x)
```

Arguments

x	A `data.frame` or `grouped_df`.

Value

Requested group metadata.

group_id *Fast group and row IDs*

Description

These are tidy-based functions for calculating group IDs and row IDs.

- `group_id()` returns an integer vector of group IDs the same size as the `x`.
- `row_id()` returns an integer vector of row IDs.
- `f_consecutive_id()` returns an integer vector of consecutive run IDs.

The add_ variants add a column of group IDs/row IDs.

Usage

```
group_id(x, order = TRUE, ascending = TRUE, as_qg = FALSE)

row_id(x, ascending = TRUE)

f_consecutive_id(x)
```

Arguments

x	A vector or data frame.
order	Should the groups be ordered? When order is TRUE (the default) the group IDs will be ordered but not sorted. If FALSE the order of the group IDs will be based on first appearance.
ascending	Should the order be ascending or descending? The default is TRUE. For row_id() this determines if the row IDs are in increasing or decreasing order.
as_qg	Should the group IDs be returned as a collapse "qG" class? The default (FALSE) always returns an integer vector.

Details

Note - When working with data frames it is highly recommended to use the add_ variants of these functions. Not only are they more intuitive to use, they also have optimisations for large numbers of groups.

group_id**:**
This assigns an integer value to unique elements of a vector or unique rows of a data frame. It is an extremely useful function for analysis as you can compress a lot of information into a single column, using that for further operations.

row_id**:**
This assigns a row number to each group. To assign plain row numbers to a data frame one can use add_row_id(). This function can be used in rolling calculations, finding duplicates and more.

consecutive_id**:**
An alternative to dplyr::consecutive_id(), f_consecutive_id() also creates an integer vector with values in the range [1, n] where n is the length of the vector or number of rows of the data frame. The ID increments every time x[i] != x[i - 1] thus giving information on when there is a change in value. f_consecutive_id has a very small overhead in terms of calling the function, making it suitable for repeated calls.

Value

An integer vector.

See Also

add_group_id add_row_id add_consecutive_id

list_tidy *Alternative to* rlang::list2

Description

Evaluates arguments dynamically like rlang::list2 but objects created in list_tidy have precedence over environment objects.

Usage

```
list_tidy(..., .keep_null = TRUE, .named = FALSE)
```

Arguments

...	Dynamic name-value pairs.
.keep_null	[logical(1)] - Should NULL elements be kept? Default is TRUE.
.named	[logical(1)] - Should all list elements be named? Default is FALSE.

new_tbl *Fast 'tibble' alternatives*

Description

Fast 'tibble' alternatives

Usage

```
new_tbl(..., .nrows = NULL, .recycle = TRUE, .name_repair = TRUE)

f_enframe(x, name = "name", value = "value")

f_deframe(x)

as_tbl(x)
```

Arguments

...	Dynamic name-value pairs.
.nrows	integer(1) (Optional) number of rows. Commonly used to initialise a 0-column data frame with rows.
.recycle	logical(1) Should arguments be recycled? Default is FALSE.
.name_repair	logical(1) Should duplicate names be made unique? Default is TRUE.
x	A data frame or vector.
name	character(1) Name to use for column of names.
value	character(1) Name to use for column of values.

Details

new_tbl and as_tbl are alternatives to tibble and as_tibble respectively.

f_enframe(x) where x is a data.frame converts x into a tibble of column names and list-values.

Value

A tibble or vector.

---

remove_rows_if_any_na *Fast remove rows with* NA *values*

Description

Fast remove rows with NA values

Usage

```
remove_rows_if_any_na(.data, ..., .cols = NULL)

remove_rows_if_all_na(.data, ..., .cols = NULL)
```

Arguments

.data	A data frame.
...	Cols to fill NA values specified through tidyselect notation. If left empty all cols are used by default.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.

Value

A data frame with removed rows containing either any or all NA values.

---

tidy_quantiles *Fast grouped sample quantiles*

Description

Fast grouped sample quantiles

Usage

```
tidy_quantiles(
  data,
  ...,
  probs = seq(0, 1, 0.25),
  type = 7,
  pivot = c("long", "wide"),
  na.rm = TRUE,
  .by = NULL,
  .cols = NULL,
  .order = group_by_order_default(data),
  .drop_groups = deprecated()
)
```

Arguments

data	A data frame.
...	<data-masking> Variables to calculate quantiles for.
probs	numeric(n) - Quantile probabilities.
type	integer(1) - Quantile type, see ?collapse::fquantile
pivot	character(1) - Pivot result wide or long? Default is "wide".
na.rm	logical(1) Should NA values be ignored? Default is TRUE.
.by	(Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.
.cols	(Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this.
.order	Should the groups be returned in sorted order? If FALSE, this will return the groups in order of first appearance, and in many cases is faster.
.drop_groups	lifecycle::badge("deprecated")

Value

A data frame of sample quantiles.

Examples

```
library(fastplyr)
library(dplyr)
groups <- 1 * 2^(0:10)

# Normal distributed samples by group using the group value as the mean
# and sqrt(groups) as the sd

samples <- tibble(groups) |>
  reframe(x = rnorm(100, mean = groups, sd = sqrt(groups)), .by = groups) |>
  f_group_by(groups)
```

```
# Fast means and quantiles by group

quantiles <- samples |>
  tidy_quantiles(x, pivot = "wide")

means <- samples |>
  f_summarise(mean = mean(x))

means |>
  f_left_join(quantiles)
```

Index