Package ‘dtrackr’

July 22, 2025

Title Track your Data Pipelines

Version 0.4.6

Description Track and
document 'dplyr' data pipelines. As you filter, mutate, and join your
way through a data set, 'dtrackr' seamlessly keeps track of your data
flow and makes publication ready documentation of a data pipeline simple.

License MIT + file LICENSE

Language en-GB

Imports dplyr (>= 1.1.0), glue, htmltools, magrittr, rlang, rsvg,
stringr, tibble, tidyr, utils, V8, fs, purrr, base64enc,
pdftools, png, lifecycle

Suggests spelling, here, knitr, rmarkdown, tidyselect, devtools,
testthat (>= 2.1.0), rstudioapi, survival, ggplot2, covr

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.2.9003

Depends R (>= 2.10)

URL `https://terminological.github.io/dtrackr/index.html`,
`https://github.com/terminological/dtrackr`

BugReports `https://github.com/terminological/dtrackr/issues`

NeedsCompilation no

Author Robert Challen [aut, cre] (ORCID:
<`https://orcid.org/0000-0002-5504-7768`>)

Maintainer Robert Challen <`rob.challen@bristol.ac.uk`>

Repository CRAN

Date/Publication 2024-10-21 09:20:02 UTC

Contents

`add_count.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
add_count(x, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`x`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
`...`	<`data-masking`> Variables to group by. Named arguments passed on to `dplyr::add_count` `wt` <`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group.

	`sort` If `TRUE`, will show the largest groups at the top. `name` The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will use `nn`. If there's a column called `n` and `nn`, it'll use `nnn`, and so on, adding `n`s until it gets a new name. `.drop` Handling of factor levels that don't appear in the data, passed on to `group_by()`. For `count()`: if `FALSE` will include counts for empty groups (i.e. for levels of factors that don't exist in the data). **[Deprecated]** For `add_count()`: deprecated since it can't actually affect the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::add_count()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# add_count
# adding in a count or tally column as a new column
iris %>%
  track() %>%
  add_count(Species, name="new_count_total",
            .messages="{.new_cols}",
            # .messages="{.cols}",
            .headline="New columns from add_count:") %>%
  history()

# add_tally
iris %>%
  track() %>%
```

```
  group_by(Species) %>%
  dtrackr::add_tally(wt=Petal.Length, name="new_tally_total",
            .messages="{.new_cols}",
            .headline="New columns from add_tally:") %>%
  history()
```

`add_tally` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
add_tally(x, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`x`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
`...`	<`data-masking`> Variables to group by. Named arguments passed on to `dplyr::add_tally` `wt` <`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group. `sort` If `TRUE`, will show the largest groups at the top. `name` The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will use `nn`. If there's a column called `n` and `nn`, it'll use `nnn`, and so on, adding `n`s until it gets a new name. `.drop` Handling of factor levels that don't appear in the data, passed on to `group_by()`. For `count()`: if `FALSE` will include counts for empty groups (i.e. for levels of factors that don't exist in the data). **[Deprecated]** For `add_count()`: deprecated since it can't actually affect the output.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::add_tally()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# add_count
# adding in a count or tally column as a new column
iris %>%
  track() %>%
  add_count(Species, name="new_count_total",
            .messages="{.new_cols}",
            # .messages="{.cols}",
            .headline="New columns from add_count:") %>%
  history()

# add_tally
iris %>%
  track() %>%
  group_by(Species) %>%
  dtrackr::add_tally(wt=Petal.Length, name="new_tally_total",
            .messages="{.new_cols}",
            .headline="New columns from add_tally:") %>%
  history()
```

`anti_join.trackr_df` *Anti join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::anti_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
anti_join(
  x,
  y,
  ...,
 .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS", "{.count.out} not matched"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::anti_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
```

```
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Anti join
join = lhs %>% anti_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`arrange.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
arrange(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::arrange`

`.by_group` If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

`.locale` The locale to sort character vectors in.

- If `NULL`, the default, uses the `"C"` locale unless the `dplyr.legacy_locale` global option escape hatch is active. See the dplyr-locale help page for more details.
- If a single string from `stringi::stri_locale_list()` is supplied, then this will be used as the locale to sort with. For example, `"en"` will sort with the American English locale. This requires the stringi package.
- If `"C"` is supplied, then character vectors will always be sorted in the C locale. This does not require stringi and is often much faster than supplying a locale identifier.

The C locale is not the same as English locales, such as `"en"`, particularly when it comes to data containing a mix of upper and lower case letters. This is explained in more detail on the locale help page under the `Default locale` section.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::arrange()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# arrange
# In this case we sort the data descending and show the first value
# is the same as the maximum value.
iris %>%
  track() %>%
  arrange(
```

```
    desc(Petal.Width),
    .messages="{.count} items, columns: {.cols}",
    .headline="Reordered dataframe:") %>%
  history()
```

`bind_cols` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
bind_cols(
  ...,
  .messages = "{.count.out} in combined set",
  .headline = "Bind columns"
)
```

Arguments

`...`	a collection of tracked data frames to combine Named arguments passed on to `dplyr::bind_cols` `.name_repair` One of `"unique"`, `"universal"`, or `"check_unique"`. See `vctrs::vec_as_names()` for the meaning of these options.
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::bind_cols()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
```

```
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`bind_rows` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
bind_rows(..., .messages = "{.count.out} in union", .headline = "Union")
```

Arguments

`...`	a collection of tracked data frames to combine Named arguments passed on to `dplyr::bind_rows` `.id` The name of an optional identifier column. Provide a string to create an output column that identifies each input. The column will use names if available, otherwise it will use positions.
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::bind_rows()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")
```

```
lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`capture_exclusions` *Start capturing exclusions on a tracked dataframe.*

Description

Start capturing exclusions on a tracked dataframe.

Usage

```
capture_exclusions(.data, .capture = TRUE)
```

Arguments

`.data`	a tracked dataframe
`.capture`	Should we capture exclusions (things removed from the data set). This is useful for debugging data issues but comes at a significant cost. Defaults to the value of `getOption("dtrackr.exclusions")` or `FALSE`.

Value

the .data dataframe with the exclusions flag set (or cleared if `.capture=FALSE`).

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% filter(Species!="versicolor") %>% history()
```

`comment` *Add a generic comment to the dtrackr history graph*

Description

A comment can be any kind of note and is added once for every current grouping as defined by the `.message` field. It can be made context specific by including variables such as {.count} and {.total} in `.message` which refer to the grouped and ungrouped counts at this current stage of the pipeline respectively. It can also pull in any global variable.

Usage

```
comment(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = (.type == "exclusion"),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`.messages`	a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} of all rows, the {.count} variable which is the count in each group and {.strata} a description of the group
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable (which is `nrow(.data)`) and {.strata} which is a description of the grouping
`.type`	one of "info","...,"exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the same .data dataframe with the history graph updated with the comment

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% comment("hello {.total} rows") %>% history()
```

`count_subgroup` *Add a subgroup count to the dtrackr history graph*

Description

A frequent use case for more detailed description is to have a subgroup count within a flowchart. This works best for factor subgroup columns but other data will be converted to a factor automatically. The count of the items in each subgroup is added as a new stage in the flowchart.

Usage

```
count_subgroup(
  .data,
  .subgroup,
  ...,
  .messages = .defaultCountSubgroup(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL,
  .maxsubgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	a dataframe which may be grouped
`.subgroup`	a column with a small number of levels (e.g. a factor)
`...`	passed to `base::factor(subgroup values, ...)` to allow reordering of levels etc.
`.messages`	a character vector of glue specifications. A glue specification can refer to anything from the calling environment, {.subgroup} for the subgroup column name and {.name} for the subgroup column value, {.count} for the subgroup column count, {.subtotal} for the current stratification grouping count and {.total} for the whole dataset count
`.headline`	a glue specification which can refer to grouping variables of .data, {.subtotal} for the current grouping count, or any variables defined in the calling environment
`.type`	one of "info","exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want to use the summary data from this step in the future then give it a name with .tag.
`.maxsubgroups`	the maximum number of discrete values allowed in .subgroup is configurable with `options("dtrackr.max_supported_groupings"=XX)`. The default is 16. Large values produce unwieldy flow charts.

Value

the same .data dataframe with the history graph updated with a subgroup count as a new stage

Examples

```
library(dplyr)
library(dtrackr)
survival::cgd %>% track() %>% group_by(treat) %>%
  count_subgroup(center) %>% history()
```

`distinct.trackr_df` *Distinct values of data*

Description

Distinct acts in the same way as in `dplyr::distinct`. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting. See `dplyr::distinct()`.

Usage

```
## S3 method for class 'trackr_df'
distinct(
  .data,
  ...,
  .messages = "removing {.count.in-.count.out} duplicates",
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables in the data frame. Named arguments passed on to `dplyr::distinct` `.keep_all` If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.
`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with distinct values and history graph updated.

See Also

dplyr::distinct()

Examples

```
library(dplyr)
library(dtrackr)

tmp = bind_rows(iris %>% track(), iris %>% track() %>% filter(Petal.Length > 5))
tmp %>% group_by(Species) %>% distinct() %>% history()
```

`dot2svg` *Convert* `Graphviz` *dot content to a SVG*

Description

Convert a `graphviz` dot digraph as string to `SVG` as string

Usage

```
dot2svg(dot)
```

Arguments

`dot`	a `graphviz` dot string

Value

the SVG as a string

Examples

```
dot2svg("digraph { A->B }")
```

`excluded` *Get the dtrackr excluded data record*

Description

Get the dtrackr excluded data record

Usage

```
excluded(.data, simplify = TRUE)
```

Arguments

`.data`	a dataframe which may be grouped
`simplify`	return a single summary dataframe of all exclusions.

Value

a new dataframe of the excluded data up to this point in the workflow. This dataframe is by default flattened, but if `.simplify=FALSE` has a nested structure containing records excluded at each part of the pipeline.

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% exclude_all(
   Petal.Length > 5.8 ~ "{.excluded} long ones",
   Petal.Length < 1.3 ~ "{.excluded} short ones",
   .stage = "petal length exclusion"
) %>% excluded()
```

`exclude_all` *Exclude all items matching one or more criteria*

Description

Apply a set of filters and summarise the actions of the filter to the `dtrackr` history graph. Because of the ... filter specification, all parameters MUST BE NAMED. The filters work in an combinatorial manner, i.e. the results EXCLUDE ALL rows that match any of the criteria. If `na.rm = TRUE` they also remove anything that cannot be evaluated by any criteria.

Usage

```
exclude_all(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = FALSE,
  .type = "exclusion",
  .asOffshoot = TRUE,
  .stage = (if (is.null(.tag)) "" else .tag),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	a dplyr filter specification as a set of formulae where the LHS are predicates to test the data set against, items that match any of the predicates will be excluded. The RHS is a glue specification, defining the message, to be entered in the history graph for each predicate. This can refer to grouping variables variables from the environment and {.excluded} and {.matched} or {.missing} (excluded = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	(default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)

`.type`	default "exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = TRUE).
`.stage`	a name for this step in the pathway
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with the history graph updated with the summary of excluded items as a new offshoot stage

Examples

```
library(dplyr)
library(dtrackr)

iris %>% track() %>% capture_exclusions() %>% exclude_all(
      Petal.Length > 5 ~ "{.excluded} long ones",
      Petal.Length < 2 ~ "{.excluded} short ones"
) %>% history()


# simultaneous evaluation of criteria:
data.frame(a = 1:10) %>%
  track() %>%
  exclude_all(
    # These two criteria identify the same value and one item is excluded
    a > 9 ~ "{.excluded} value > 9",
    a == max(a) ~ "{.excluded} max value",
  ) %>%
  status() %>%
  history()

# the behaviour is equivalent to the inverse of dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a <= 9, a != max(a)) %>%
  nrow()

# step-wise evaluation of criteria results in a different output
data.frame(a = 1:10) %>%
  track() %>%
  # Performing the same exclusion sequentially results in 2 items
  # being excluded as the criteria no longer identify the same
  # item.
  exclude_all(a > 9 ~ "{.excluded} value > 9") %>%
  exclude_all(a == max(a) ~ "{.excluded} max value") %>%
  status() %>%
  history()

# the behaviour is equivalent to the inverse of dplyr's filter function:
data.frame(a=1:10) %>%
```

```
  dplyr::filter(a <= 9) %>%
  dplyr::filter(a != max(a)) %>%
  nrow()
```

`filter.trackr_df` *Filtering data*

Description

Filter acts in the same way as in `dplyr` where predicates which evaluate to TRUE act to select items to include, and items for which the predicate cannot be evaluated are excluded. For tracking prior to the filter operation the size of each group is calculated {.count.in} and after the operation the output size of each group {.count.out}. The grouping {.strata} is also available (if grouped) for reporting. See `dplyr::filter()`.

Usage

```
## S3 method for class 'trackr_df'
filter(
  .data,
  ...,
  .messages = "excluded {.excluded} items",
  .headline = .defaultHeadline(),
  .type = "exclusion",
  .asOffshoot = (.type == "exclusion"),
  .stage = (if (is.null(.tag)) "" else .tag),
  .tag = NULL
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Expressions that return a logical value, and are defined in terms of the variables in `.data`. If multiple expressions are included, they are combined with the `&` operator. Only rows for which all conditions evaluate to `TRUE` are kept. Named arguments passed on to `dplyr::filter` `.by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.
`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}

`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	the format type of the action typically an exclusion
`.asOffshoot`	if the type is exclusion, `.asOffshoot` places the information box outside of the main flow, as an exclusion.
`.stage`	a name for this step in the pathway
`.tag`	if you want the summary data from this step in the future then give it a name with `.tag`.

Value

the filtered `.data` dataframe with history graph updated

See Also

dplyr::filter()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species)
tmp %>% filter(Petal.Length > 5) %>% history()
```

`flowchart` *Flowchart output*

Description

Generate a flowchart of the history of the dataframe(s), with all the tracked data pipeline as stages in the flowchart. Multiple dataframes can be plotted together in which case an attempt is made to determine which parts are common.

Usage

```
flowchart(
  .data,
  filename = NULL,
  size = std_size$full,
  maxWidth = size$width,
  maxHeight = size$height,
  formats = c("dot", "png", "pdf", "svg"),
  defaultToHTML = TRUE,
  landscape = size$rot != 0,
  ...
)
```

Arguments

`.data`	the tracked dataframe(s) either as a single dataframe or as a list of dataframes.
`filename`	a file name which will be where the formatted flowcharts are saved. If no extension is specified the output formats are determined by the `formats` parameter.
`size`	a named list with 3 elements, length and width in inches and rotation. A predefined set of standard sizes are available in the std_size object.
`maxWidth`	a width (on the paper) in inches if `size` is not defined
`maxHeight`	a height (on the paper) in inches if `size` is not defined
`formats`	some of `pdf,dot,svg,png,ps`
`defaultToHTML`	if the correct output format is not easy to determine from the context, default providing `HTML` (TRUE) or to embedding the `PNG` (FALSE)
`landscape`	rotate the output by 270 degrees into a landscape format. `maxWidth` and `maxHeight` still apply and refer to the paper width to fit the flowchart into after rotation. (you might need to flip width and height)
`...`	other parameters passed onto either `p_get_as_dot()`, notable ones are `fill` (background colour e.g. `lightgrey`), `fontsize` (in points), `colour` (font colour)

Value

the nature of the flowchart output depends on the context in which the function is called. It will be some form of browse-able html output if called from an interactive session or a `PNG/PDF` link if in `knitr` and knitting latex or word type outputs, if file name is specified the output will also be saved at the given location.

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
tmp %>% group_by(Species) %>% comment(.tag="step2") %>% flowchart()
```

`full_join.trackr_df` *Full join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::full_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
full_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Full join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::full_join`

`by` A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged

into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.

	- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists. `relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::full_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Full join
join = lhs %>% full_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`group_by.trackr_df` *Stratifying your analysis*

Description

Grouping a data set acts in the normal way. When tracking a dataframe sometimes a `group_by()` operation will create a lot of groups. This happens for example if you are doing a `group_by()`, `summarise()` step that is aggregating data on a fine scale, e.g. by day in a time-series. This is generally a terrible idea when tracking a dataframe as the resulting flowchart will have many many branches and be illegible. `dtrackr` will detect this issue and pause tracking the dataframe with a warning. It is up to the user to the `resume()` tracking when the large number of groups have been resolved e.g. using a `dplyr::ungroup()`. This limit is configurable with `options("dtrackr.max_supported_groupings"=X`
The default is 16. See `dplyr::group_by()`.

Usage

```
## S3 method for class 'trackr_df'
group_by(
  .data,
  ...,
  .messages = "stratify by {.cols}",
  .headline = NULL,
  .tag = NULL,
  .maxgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	In `group_by()`, variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate `mutate()` step before the `group_by()`. Computations are not allowed in `nest_by()`. In `ungroup()`, variables to remove from the grouping. Named arguments passed on to `dplyr::group_by` `.add` When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions. `.drop` Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop = FALSE`. See `group_by_drop_default()` for details. `x` A `tbl()`
`.messages`	a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by.
`.headline`	a headline glue spec. The glue code can use any global variable, or {.cols}.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.
`.maxgroups`	the maximum number of subgroups allowed before the tracking is paused.

Value

the .data but grouped.

See Also

dplyr::group_by()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species, .messages="stratify by {.cols}")
tmp %>% comment("{.strata}") %>% history()
```

`group_modify.trackr_df`

Group-wise modification of data and complex operations

Description

Group modifying a data set acts in the normal way. The internal mechanics of the modify function are opaque to the history. This means these can be used to wrap any unsupported operation without losing the history (e.g. `df %>% track() %>% group_modify(function(d,...) { d %>% unsupported_operation() })`) Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::group_modify()`.

Usage

```
## S3 method for class 'trackr_df'
group_modify(
  .data,
  ...,
  .messages = NULL,
  .headline = .defaultHeadline(),
  .type = "modify",
  .tag = NULL
)
```

Arguments

`.data`	A grouped tibble
`...`	Additional arguments passed on to `.f` Named arguments passed on to `dplyr::group_modify` `.f` A function or formula to apply to each group. If a **function**, it is used as is. It should have at least 2 formal arguments. If a **formula**, e.g. `~ head(.x)`, it is converted to a function. In the formula, you can use

	• `.` or `.x` to refer to the subset of rows of `.tbl` for the given group • `.y` to refer to the key, a one row tibble with one column per grouping variable that identifies the group `.keep` are the grouping variables kept in `.x`
`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	default "modify": used to define formatting
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the transformed .data dataframe with the history graph updated.

See Also

dplyr::group_modify()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species)
tmp %>% group_modify(
      function(d,g,...) { return(tibble::tibble(x=runif(10))) },
      .messages="{.count.in} in, {.count.out} out"
) %>% history()
```

`history` *Get the dtrackr history graph*

Description

This provides the raw history graph and is not really intended for mainstream use. The internal structure of the graph is explained below. print and plot S3 methods exist for the dtrackr history graph.

Usage

```
history(.data)
```

Arguments

`.data`	a dataframe which may be grouped

Value

the history graph. This is a list, of class `trackr_graph`, containing the following named items:

- excluded - the data items that have been excluded thus far as a nested dataframe
- tags - a dataframe of tag-value pairs containing the summary of the data at named points in the data flow (see `tagged()`)
- nodes - a dataframe of the nodes of the flow chart
- edges - an edge list (as a dataframe) of the relationships between the nodes in the flow chart
- head - the current most recent nodes added into the graph as a dataframe.

The format of this data may grow over time but these fields are unlikely to be changed.

Examples

```
library(dplyr)
library(dtrackr)
graph = iris %>% track() %>% comment("A comment") %>% history()
print(graph)
```

`include_any` *Include any items matching a criteria*

Description

Apply a set of inclusion criteria and record the actions of the filter to the `dtrackr` history graph. Because of the ... filter specification, all parameters MUST BE NAMED. This function is the opposite of `exclude_all()` and the filtering criteria work to identify rows to include i.e. the results include anything that match any of the criteria. If `na.rm=TRUE` they also keep anything that cannot be evaluated by the criteria.

Usage

```
include_any(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = TRUE,
  .type = "inclusion",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	a dplyr filter specification as a set of formulae where the LHS are predicates to test the data set against, items that match at least one of the predicates will be included. The RHS is a glue specification, defining the message, to be entered in the history graph for each predicate matched. This can refer to grouping variables, variables from the environment and {.included} and {.matched} or {.missing} (included = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	(default TRUE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	default "inclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with the history graph updated with the summary of included items as a new stage

Examples

```
library(dplyr)
library(dtrackr)

iris %>% track() %>% group_by(Species) %>% include_any(
      Petal.Length > 5 ~ "{.included} long ones",
      Petal.Length < 2 ~ "{.included} short ones"
) %>% history()

# simultaneous evaluation of criteria:
data.frame(a = 1:10) %>%
  track() %>%
  include_any(
    # These two criteria identify the same value and one item is excluded
    a > 1 ~ "{.included} value > 1",
    a != min(a) ~ "{.included} everything but the smallest value",
  ) %>%
  status() %>%
  history()

# the behaviour is equivalent to dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a > 1, a != min(a)) %>%
```

```
  nrow()

# step-wise evaluation of criteria results in a different output
data.frame(a = 1:10) %>%
  track() %>%
  # Performing the same exclusion sequentially results in 2 items
  # being excluded as the criteria no longer identify the same
  # item.
  include_any(a > 1 ~ "{.included} value > 1") %>%
  include_any(a != min(a) ~ "{.included} everything but the smallest value") %>%
  status() %>%
  history()

# the behaviour is equivalent to dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a > 1) %>%
  dplyr::filter(a != min(a)) %>%
  nrow()
```

`inner_join.trackr_df` *Inner joins*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::inner_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
inner_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Inner join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::inner_join`

`by` A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::inner_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Inner join
join = lhs %>% inner_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`intersect.trackr_df` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent dplyr operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
## S3 method for class 'trackr_df'
intersect(
  x,
  y,
  ...,
  .messages = "{.count.out} in intersection",
  .headline = "Intersection"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

generics::intersect()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences
```

```
set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`left_join.trackr_df` *Left join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::left_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
left_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Left join by {.keys}"
)
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	Other parameters passed onto methods. Named arguments passed on to `dplyr::left_join`

by A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::left_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets
```

```
# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Left join
join = lhs %>% left_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`mutate.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
mutate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.

Named arguments passed on to `dplyr::mutate`

`.by` **[Experimental]**

<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.keep` Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept.

- `"all"` retains all columns from `.data`. This is the default.
- `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side.
- `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them.
- `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept.

`.before, .after` <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::mutate()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# mutate
# In this example we compare the column names of the input and the
# output to identify the new columns created by the mutate operation as
```

```
# the `.new_cols` variable
iris %>%
  track() %>%
  mutate(extra_col = NA_real_,
         .messages="{.new_cols}",
         .headline="Extra columns from mutate:") %>%
  history()
```

`nest_join.trackr_df` *Nest join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::nest_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
nest_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS", "{.count.out} matched"),
  .headline = "Nest join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::nest_join` `by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

	For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`. `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it. `keep` Should the new list-column contain join keys? The default will preserve the join keys for inequality joins. `name` The name of the list-column created by the join. If `NULL`, the default, the name of `y` is used. `na_matches` Should two `NA` or two `NaN` values match? • `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`. • `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`. `unmatched` How should unmatched keys that would result in dropped rows be handled? • `"drop"` drops unmatched keys from the result. • `"error"` throws an error if unmatched keys are detected. `unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows. • For left joins, it checks `y`. • For right joins, it checks `x`. • For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::nest_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Nest join
join = lhs %>% nest_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`pause` *Pause tracking the data frame.*

Description

Pausing tracking of a data frame may be required if an operation is about to be performed that creates a lot of groupings or that you otherwise don't want to pollute the history graph (e.g. maybe selecting something using an anti-join). Once paused the history is not updated until a `resume()` is called, or when the data frame is ungrouped (if `auto` is enabled).

Usage

```
pause(.data, auto = FALSE)
```

Arguments

`.data`	a tracked dataframe
`auto`	if `TRUE` the tracking will resume automatically when the number of groups has fallen to a sensible level (default is `FALSE`)?

Value

the .data dataframe with history graph tracking paused

Examples

```
iris %>% track() %>% pause() %>% history()
```

`pivot_longer.trackr_df`

Reshaping data using `tidyr::pivot_longer`

Description

A drop in replacement for `tidyr::pivot_longer()` which optionally takes a message and headline to store in the history graph.

Usage

```
## S3 method for class 'trackr_df'
pivot_longer(data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`data`	A data frame to pivot.
`...`	Additional arguments passed on to methods. Named arguments passed on to `tidyr::pivot_longer`

`cols` <`tidy-select`> Columns to pivot into longer format.

`cols_vary` When pivoting `cols` into longer format, how should the output rows be arranged relative to their original row number?

- `"fastest"`, the default, keeps individual rows from `cols` close together in the output. This often produces intuitively ordered output when you have at least one key column from `data` that is not involved in the pivoting process.
- `"slowest"` keeps individual columns from `cols` close together in the output. This often produces intuitively ordered output when you utilize all of the columns from `data` in the pivoting process.

`names_to` A character vector specifying the new column or columns to create from the information stored in the column names of `data` specified by `cols`.

- If length 0, or if `NULL` is supplied, no columns will be created.
- If length 1, a single column will be created which will contain the column names specified by `cols`.
- If length >1, multiple columns will be created. In this case, one of `names_sep` or `names_pattern` must be supplied to specify how the column names should be split. There are also two additional character values you can take advantage of:
  - `NA` will discard the corresponding component of the column name.
  - `".value"` indicates that the corresponding component of the column name defines the name of the output column containing the cell values, overriding `values_to` entirely.

`names_prefix` A regular expression used to remove matching text from the start of each variable name.

`names_sep,names_pattern` If `names_to` contains multiple values, these arguments control how the column name is broken up.

`names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on).

`names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`).

If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.

`names_ptypes,values_ptypes` Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead.

`names_transform,values_transform` Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer.

If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.

`names_repair` What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_to` A string specifying the name of the column to create from the data stored in cell values. If `names_to` is a character containing the special `.value` sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.

`values_drop_na` If `TRUE`, will drop rows that contain only `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in `data` were created by its structure.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the result of the `tidyr::pivot_longer` but with a history graph updated.

See Also

tidyr::pivot_longer()

`pivot_wider.trackr_df` *Reshaping data using* `tidyr::pivot_wider`

Description

A drop in replacement for `tidyr::pivot_wider()` which optionally takes a message and headline to store in the history graph.

Usage

```
## S3 method for class 'trackr_df'
pivot_wider(data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`data`	A data frame to pivot.
`...`	Additional arguments passed on to methods. Named arguments passed on to `tidyr::pivot_wider`

`id_cols` <`tidy-select`> 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`.

`id_expand` Should the values in the `id_cols` columns be expanded by `expand()` before pivoting? This results in more rows, the output will contain a complete expansion of all possible values in `id_cols`. Implicit factor levels that aren't represented in the data will become explicit. Additionally, the row values corresponding to the expanded `id_cols` will be sorted.

`names_from,values_from` <`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`).
If `values_from` contains multiple values, the value will be added to the front of the output column.

`names_prefix` String added to the start of every variable name. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.

`names_sep` If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.

`names_glue` Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns (and special `.value`) to create custom column names.

`names_sort` Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.

`names_vary` When `names_from` identifies a column (or columns) with multiple unique values, and multiple `values_from` columns are provided, in what order should the resulting column names be combined?

- `"fastest"` varies `names_from` values fastest, resulting in a column naming scheme of the form: `value1_name1, value1_name2, value2_name1, value2_name2`. This is the default.
- `"slowest"` varies `names_from` values slowest, resulting in a column naming scheme of the form: `value1_name1, value2_name1, value1_name2, value2_name2`.

`names_expand` Should the values in the `names_from` columns be expanded by `expand()` before pivoting? This results in more columns, the output will contain column names corresponding to a complete expansion of all possible values in `names_from`. Implicit factor levels that aren't represented in the data will become explicit. Additionally, the column names will be sorted, identical to what `names_sort` would produce.

`names_repair` What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_fill` Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing.

This can be a named list if you want to apply different fill values to different value columns.

`values_fn` Optionally, a function applied to the value in each cell in the output. You will typically use this when the combination of `id_cols` and `names_from` columns does not uniquely identify an observation.

This can be a named list if you want to apply different aggregations to different `values_from` columns.

`unused_fn` Optionally, a function applied to summarize the values from the unused columns (i.e. columns not identified by `id_cols`, `names_from`, or `values_from`).

The default drops all unused columns from the result.

This can be a named list if you want to apply different aggregations to different unused columns.

`id_cols` must be supplied for `unused_fn` to be useful, since otherwise all unspecified columns will be considered `id_cols`.

This is similar to grouping by the `id_cols` then summarizing the unused columns using `unused_fn`.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.

`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the data dataframe result of the `tidyr::pivot_wider` function but with a history graph updated with a `.message` if requested.

See Also

tidyr::pivot_wider()

`plot.trackr_graph` *Plots a history graph as html*

Description

Plots a history graph as html

Usage

```
## S3 method for class 'trackr_graph'
plot(x, fill = "lightgrey", fontsize = "8", colour = "black", ...)
```

Arguments

`x`	a dtrackr history graph (e.g. output from `history()`)
`fill`	the default node fill colour
`fontsize`	the default font size
`colour`	the default font colour
`...`	not used

Value

HTML displayed

Examples

```
library(dplyr)
library(dtrackr)
iris %>% comment("hello {.total} rows") %>% history() %>% plot()
```

`print.trackr_graph` *Print a history graph to the console*

Description

Print a history graph to the console

Usage

```
## S3 method for class 'trackr_graph'
print(x, ...)
```

Arguments

`x`	a dtrackr history graph (e.g. output from `p_get()`)
`...`	not used

Value

nothing

Examples

```
library(dplyr)
library(dtrackr)
iris %>% comment("hello {.total} rows") %>% history() %>% print()
```

`p_add_count` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_add_count(x, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`x`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
`...`	<`data-masking`> Variables to group by. Named arguments passed on to `dplyr::add_count` `wt` <`data-masking`> Frequency weights. Can be `NULL` or a variable: • If `NULL` (the default), counts the number of rows in each group. • If a variable, computes `sum(wt)` for each group. `sort` If `TRUE`, will show the largest groups at the top. `name` The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will use `nn`. If there's a column called `n` and `nn`, it'll use `nnn`, and so on, adding `n`s until it gets a new name. `.drop` Handling of factor levels that don't appear in the data, passed on to `group_by()`. For `count()`: if `FALSE` will include counts for empty groups (i.e. for levels of factors that don't exist in the data). **[Deprecated]** For `add_count()`: deprecated since it can't actually affect the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::add_count()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# add_count
# adding in a count or tally column as a new column
iris %>%
```

```
  track() %>%
  add_count(Species, name="new_count_total",
            .messages="{.new_cols}",
            # .messages="{.cols}",
            .headline="New columns from add_count:") %>%
  history()

# add_tally
iris %>%
  track() %>%
  group_by(Species) %>%
  dtrackr::add_tally(wt=Petal.Length, name="new_tally_total",
            .messages="{.new_cols}",
            .headline="New columns from add_tally:") %>%
  history()
```

`p_add_tally` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_add_tally(x, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`x`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
`...`	<`data-masking`> Variables to group by.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::add_tally()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# add_count
# adding in a count or tally column as a new column
iris %>%
  track() %>%
  add_count(Species, name="new_count_total",
            .messages="{.new_cols}",
            # .messages="{.cols}",
            .headline="New columns from add_count:") %>%
  history()

# add_tally
iris %>%
  track() %>%
  group_by(Species) %>%
  dtrackr::add_tally(wt=Petal.Length, name="new_tally_total",
            .messages="{.new_cols}",
            .headline="New columns from add_tally:") %>%
  history()
```

`p_anti_join` *Anti join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::anti_join()` for more details on the underlying functions.

Usage

```
p_anti_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS", "{.count.out} not matched"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::anti_join` `by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`. `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it. `na_matches` Should two `NA` or two `NaN` values match? • `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`. • `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::anti_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Anti join
join = lhs %>% anti_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_arrange` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_arrange(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::arrange` `.by_group` If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only. `.locale` The locale to sort character vectors in. • If `NULL`, the default, uses the `"C"` locale unless the `dplyr.legacy_locale` global option escape hatch is active. See the dplyr-locale help page for more details. • If a single string from `stringi::stri_locale_list()` is supplied, then this will be used as the locale to sort with. For example, `"en"` will sort with the American English locale. This requires the stringi package. • If `"C"` is supplied, then character vectors will always be sorted in the C locale. This does not require stringi and is often much faster than supplying a locale identifier. The C locale is not the same as English locales, such as `"en"`, particularly when it comes to data containing a mix of upper and lower case letters. This is explained in more detail on the locale help page under the `Default locale` section.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::arrange()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# arrange
# In this case we sort the data descending and show the first value
# is the same as the maximum value.
iris %>%
  track() %>%
  arrange(
    desc(Petal.Width),
    .messages="{.count} items, columns: {.cols}",
    .headline="Reordered dataframe:") %>%
  history()
```

`p_bind_cols` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_bind_cols(
  ...,
  .messages = "{.count.out} in combined set",
  .headline = "Bind columns"
)
```

Arguments

`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::bind_cols()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
```

```
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_bind_rows` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_bind_rows(..., .messages = "{.count.out} in union", .headline = "Union")
```

Arguments

`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::bind_rows()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")
```

```
lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_capture_exclusions` *Start capturing exclusions on a tracked dataframe.*

Description

Start capturing exclusions on a tracked dataframe.

Usage

```
p_capture_exclusions(.data, .capture = TRUE)
```

Arguments

`.data`	a tracked dataframe
`.capture`	Should we capture exclusions (things removed from the data set). This is useful for debugging data issues but comes at a significant cost. Defaults to the value of `getOption("dtrackr.exclusions")` or `FALSE`.

Value

the .data dataframe with the exclusions flag set (or cleared if `.capture=FALSE`).

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% filter(Species!="versicolor") %>% history()
```

`p_clear` *Clear the dtrackr history graph*

Description

This is unlikely to be needed directly and is mostly and internal function

Usage

```
p_clear(.data)
```

Arguments

`.data`	a dataframe which may be grouped

Value

the .data dataframe with the history graph removed

Examples

```
library(dplyr)
library(dtrackr)
mtcars %>% track() %>% comment("A comment") %>% p_clear() %>% history()
```

`p_comment` *Add a generic comment to the dtrackr history graph*

Description

A comment can be any kind of note and is added once for every current grouping as defined by the `.message` field. It can be made context specific by including variables such as {.count} and {.total} in `.message` which refer to the grouped and ungrouped counts at this current stage of the pipeline respectively. It can also pull in any global variable.

Usage

```
p_comment(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = (.type == "exclusion"),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`.messages`	a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} of all rows, the {.count} variable which is the count in each group and {.strata} a description of the group
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable (which is `nrow(.data)`) and {.strata} which is a description of the grouping
`.type`	one of "info","...,"exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the same .data dataframe with the history graph updated with the comment

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% comment("hello {.total} rows") %>% history()
```

`p_copy` *Copy the dtrackr history graph from one dataframe to another*

Description

Copy the dtrackr history graph from one dataframe to another

Usage

```
p_copy(.data, from)
```

Arguments

`.data`	a dataframe which may be grouped
`from`	the dataframe to copy the history graph from

Value

the .data dataframe with the history graph of "from"

Examples

```
mtcars %>% p_copy(iris %>% comment("A comment")) %>% history()
```

`p_count_if` *Simple count_if dplyr summary function*

Description

Simple count_if dplyr summary function

Usage

```
p_count_if(..., na.rm = TRUE)
```

Arguments

`...`	expression to be evaluated
`na.rm`	ignore NA values?

Value

a count of the number of times the expression evaluated to true, in the current context

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% dplyr::group_by(Species)
tmp %>% dplyr::summarise(long_ones = p_count_if(Petal.Length > 4))
```

`p_count_subgroup` *Add a subgroup count to the dtrackr history graph*

Description

A frequent use case for more detailed description is to have a subgroup count within a flowchart. This works best for factor subgroup columns but other data will be converted to a factor automatically. The count of the items in each subgroup is added as a new stage in the flowchart.

Usage

```
p_count_subgroup(
  .data,
  .subgroup,
  ...,
  .messages = .defaultCountSubgroup(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL,
  .maxsubgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	a dataframe which may be grouped
`.subgroup`	a column with a small number of levels (e.g. a factor)
`...`	passed to `base::factor(subgroup values, ...)` to allow reordering of levels etc.
`.messages`	a character vector of glue specifications. A glue specification can refer to anything from the calling environment, {.subgroup} for the subgroup column name and {.name} for the subgroup column value, {.count} for the subgroup column count, {.subtotal} for the current stratification grouping count and {.total} for the whole dataset count
`.headline`	a glue specification which can refer to grouping variables of .data, {.subtotal} for the current grouping count, or any variables defined in the calling environment
`.type`	one of "info","exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).

`.tag`	if you want to use the summary data from this step in the future then give it a name with .tag.
`.maxsubgroups`	the maximum number of discrete values allowed in .subgroup is configurable with `options("dtrackr.max_supported_groupings"=XX)`. The default is 16. Large values produce unwieldy flow charts.

Value

the same .data dataframe with the history graph updated with a subgroup count as a new stage

Examples

```
library(dplyr)
library(dtrackr)
survival::cgd %>% track() %>% group_by(treat) %>%
  count_subgroup(center) %>% history()
```

`p_distinct` *Distinct values of data*

Description

Distinct acts in the same way as in `dplyr::distinct`. Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting. See `dplyr::distinct()`.

Usage

```
p_distinct(
  .data,
  ...,
  .messages = "removing {.count.in-.count.out} duplicates",
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables in the data frame. Named arguments passed on to `dplyr::distinct` `.keep_all` If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with distinct values and history graph updated.

See Also

dplyr::distinct()

Examples

```
library(dplyr)
library(dtrackr)

tmp = bind_rows(iris %>% track(), iris %>% track() %>% filter(Petal.Length > 5))
tmp %>% group_by(Species) %>% distinct() %>% history()
```

`p_excluded` *Get the dtrackr excluded data record*

Description

Get the dtrackr excluded data record

Usage

```
p_excluded(.data, simplify = TRUE)
```

Arguments

`.data`	a dataframe which may be grouped
`simplify`	return a single summary dataframe of all exclusions.

Value

a new dataframe of the excluded data up to this point in the workflow. This dataframe is by default flattened, but if `.simplify=FALSE` has a nested structure containing records excluded at each part of the pipeline.

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% capture_exclusions()
tmp %>% exclude_all(
   Petal.Length > 5.8 ~ "{.excluded} long ones",
   Petal.Length < 1.3 ~ "{.excluded} short ones",
   .stage = "petal length exclusion"
) %>% excluded()
```

`p_exclude_all` *Exclude all items matching one or more criteria*

Description

Apply a set of filters and summarise the actions of the filter to the `dtrackr` history graph. Because of the ... filter specification, all parameters MUST BE NAMED. The filters work in an combinatorial manner, i.e. the results EXCLUDE ALL rows that match any of the criteria. If `na.rm = TRUE` they also remove anything that cannot be evaluated by any criteria.

Usage

```
p_exclude_all(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = FALSE,
  .type = "exclusion",
  .asOffshoot = TRUE,
  .stage = (if (is.null(.tag)) "" else .tag),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	a dplyr filter specification as a set of formulae where the LHS are predicates to test the data set against, items that match any of the predicates will be excluded. The RHS is a glue specification, defining the message, to be entered in the history graph for each predicate. This can refer to grouping variables variables from the environment and {.excluded} and {.matched} or {.missing} (excluded = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	(default FALSE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)

`.type`	default "exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = TRUE).
`.stage`	a name for this step in the pathway
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with the history graph updated with the summary of excluded items as a new offshoot stage

Examples

```
library(dplyr)
library(dtrackr)

iris %>% track() %>% capture_exclusions() %>% exclude_all(
      Petal.Length > 5 ~ "{.excluded} long ones",
      Petal.Length < 2 ~ "{.excluded} short ones"
) %>% history()


# simultaneous evaluation of criteria:
data.frame(a = 1:10) %>%
  track() %>%
  exclude_all(
    # These two criteria identify the same value and one item is excluded
    a > 9 ~ "{.excluded} value > 9",
    a == max(a) ~ "{.excluded} max value",
  ) %>%
  status() %>%
  history()

# the behaviour is equivalent to the inverse of dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a <= 9, a != max(a)) %>%
  nrow()

# step-wise evaluation of criteria results in a different output
data.frame(a = 1:10) %>%
  track() %>%
  # Performing the same exclusion sequentially results in 2 items
  # being excluded as the criteria no longer identify the same
  # item.
  exclude_all(a > 9 ~ "{.excluded} value > 9") %>%
  exclude_all(a == max(a) ~ "{.excluded} max value") %>%
  status() %>%
  history()

# the behaviour is equivalent to the inverse of dplyr's filter function:
data.frame(a=1:10) %>%
```

```
dplyr::filter(a <= 9) %>%
dplyr::filter(a != max(a)) %>%
nrow()
```

`p_filter` *Filtering data*

Description

Filter acts in the same way as in `dplyr` where predicates which evaluate to TRUE act to select items to include, and items for which the predicate cannot be evaluated are excluded. For tracking prior to the filter operation the size of each group is calculated {.count.in} and after the operation the output size of each group {.count.out}. The grouping {.strata} is also available (if grouped) for reporting. See `dplyr::filter()`.

Usage

```
p_filter(
  .data,
  ...,
  .messages = "excluded {.excluded} items",
  .headline = .defaultHeadline(),
  .type = "exclusion",
  .asOffshoot = (.type == "exclusion"),
  .stage = (if (is.null(.tag)) "" else .tag),
  .tag = NULL
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Expressions that return a logical value, and are defined in terms of the variables in `.data`. If multiple expressions are included, they are combined with the `&` operator. Only rows for which all conditions evaluate to `TRUE` are kept. Named arguments passed on to `dplyr::filter` `.by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.
`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}

`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	the format type of the action typically an exclusion
`.asOffshoot`	if the type is exclusion, `.asOffshoot` places the information box outside of the main flow, as an exclusion.
`.stage`	a name for this step in the pathway
`.tag`	if you want the summary data from this step in the future then give it a name with `.tag`.

Value

the filtered `.data` dataframe with history graph updated

See Also

dplyr::filter()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species)
tmp %>% filter(Petal.Length > 5) %>% history()
```

`p_flowchart` *Flowchart output*

Description

Generate a flowchart of the history of the dataframe(s), with all the tracked data pipeline as stages in the flowchart. Multiple dataframes can be plotted together in which case an attempt is made to determine which parts are common.

Usage

```
p_flowchart(
  .data,
  filename = NULL,
  size = std_size$full,
  maxWidth = size$width,
  maxHeight = size$height,
  formats = c("dot", "png", "pdf", "svg"),
  defaultToHTML = TRUE,
  landscape = size$rot != 0,
  ...
)
```

Arguments

`.data`	the tracked dataframe(s) either as a single dataframe or as a list of dataframes.
`filename`	a file name which will be where the formatted flowcharts are saved. If no extension is specified the output formats are determined by the `formats` parameter.
`size`	a named list with 3 elements, length and width in inches and rotation. A predefined set of standard sizes are available in the std_size object.
`maxWidth`	a width (on the paper) in inches if `size` is not defined
`maxHeight`	a height (on the paper) in inches if `size` is not defined
`formats`	some of `pdf,dot,svg,png,ps`
`defaultToHTML`	if the correct output format is not easy to determine from the context, default providing `HTML` (TRUE) or to embedding the `PNG` (FALSE)
`landscape`	rotate the output by 270 degrees into a landscape format. `maxWidth` and `maxHeight` still apply and refer to the paper width to fit the flowchart into after rotation. (you might need to flip width and height)
`...`	other parameters passed onto either `p_get_as_dot()`, notable ones are `fill` (background colour e.g. `lightgrey`), `fontsize` (in points), `colour` (font colour)

Value

the nature of the flowchart output depends on the context in which the function is called. It will be some form of browse-able html output if called from an interactive session or a `PNG/PDF` link if in `knitr` and knitting latex or word type outputs, if file name is specified the output will also be saved at the given location.

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
tmp %>% group_by(Species) %>% comment(.tag="step2") %>% flowchart()
```

`p_full_join` *Full join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::full_join()` for more details on the underlying functions.

Usage

```
p_full_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Full join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::full_join`
	`by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.
	`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
	`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
	`keep` Should the join keys from both `x` and `y` be preserved in the output? • If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs. • If `TRUE`, all keys from both inputs are retained. • If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

	`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::full_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Full join
join = lhs %>% full_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_get` *Get the dtrackr history graph*

Description

This provides the raw history graph and is not really intended for mainstream use. The internal structure of the graph is explained below. print and plot S3 methods exist for the dtrackr history graph.

Usage

```
p_get(.data)
```

Arguments

`.data`	a dataframe which may be grouped

Value

the history graph. This is a list, of class `trackr_graph`, containing the following named items:

- excluded - the data items that have been excluded thus far as a nested dataframe
- tags - a dataframe of tag-value pairs containing the summary of the data at named points in the data flow (see `tagged()`)
- nodes - a dataframe of the nodes of the flow chart
- edges - an edge list (as a dataframe) of the relationships between the nodes in the flow chart
- head - the current most recent nodes added into the graph as a dataframe.

The format of this data may grow over time but these fields are unlikely to be changed.

Examples

```
library(dplyr)
library(dtrackr)
graph = iris %>% track() %>% comment("A comment") %>% history()
print(graph)
```

`p_get_as_dot` *DOT output*

Description

(advance usage) outputs a `dtrackr` history graph as a DOT string for rendering with `Graphviz`

Usage

```
p_get_as_dot(.data, fill = "lightgrey", fontsize = "8", colour = "black", ...)
```

Arguments

`.data`	the tracked dataframe
`fill`	the default node fill colour
`fontsize`	the default font size
`colour`	the default font colour
`...`	not used

Value

a representation of the history graph in `Graphviz` dot format.

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% comment(.tag = "step1") %>% filter(Species!="versicolor")
dot = tmp %>% group_by(Species) %>% comment(.tag="step2") %>% p_get_as_dot()
cat(dot)
```

`p_group_by` *Stratifying your analysis*

Description

Grouping a data set acts in the normal way. When tracking a dataframe sometimes a `group_by()` operation will create a lot of groups. This happens for example if you are doing a `group_by()`, `summarise()` step that is aggregating data on a fine scale, e.g. by day in a time-series. This is generally a terrible idea when tracking a dataframe as the resulting flowchart will have many many branches and be illegible. `dtrackr` will detect this issue and pause tracking the dataframe with a warning. It is up to the user to the `resume()` tracking when the large number of groups have been resolved e.g. using a `dplyr::ungroup()`. This limit is configurable with `options("dtrackr.max_supported_groupings"=X` The default is 16. See `dplyr::group_by()`.

Usage

```
p_group_by(
  .data,
  ...,
  .messages = "stratify by {.cols}",
  .headline = NULL,
  .tag = NULL,
  .maxgroups = .defaultMaxSupportedGroupings()
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	In `group_by()`, variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate `mutate()` step before the `group_by()`. Computations are not allowed in `nest_by()`. In `ungroup()`, variables to remove from the grouping. Named arguments passed on to `dplyr::group_by`

	`.add` When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions. `.drop` Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop = FALSE`. See `group_by_drop_default()` for details. `x` A `tbl()`
`.messages`	a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by.
`.headline`	a headline glue spec. The glue code can use any global variable, or {.cols}.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.
`.maxgroups`	the maximum number of subgroups allowed before the tracking is paused.

Value

the .data but grouped.

See Also

dplyr::group_by()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species, .messages="stratify by {.cols}")
tmp %>% comment("{.strata}") %>% history()
```

`p_group_modify` *Group-wise modification of data and complex operations*

Description

Group modifying a data set acts in the normal way. The internal mechanics of the modify function are opaque to the history. This means these can be used to wrap any unsupported operation without losing the history (e.g. `df %>% track() %>% group_modify(function(d,...) { d %>% unsupported_operation() })`) Prior to the operation the size of the group is calculated {.count.in} and after the operation the output size {.count.out} The group {.strata} is also available (if grouped) for reporting See `dplyr::group_modify()`.

Usage

```
p_group_modify(
  .data,
  ...,
  .messages = NULL,
  .headline = .defaultHeadline(),
  .type = "modify",
  .tag = NULL
)
```

Arguments

`.data`	A grouped tibble
`...`	Additional arguments passed on to `.f` Named arguments passed on to `dplyr::group_modify` `.f` A function or formula to apply to each group. If a **function**, it is used as is. It should have at least 2 formal arguments. If a **formula**, e.g. ~ `head(.x)`, it is converted to a function. In the formula, you can use • `.` or `.x` to refer to the subset of rows of `.tbl` for the given group • `.y` to refer to the key, a one row tibble with one column per grouping variable that identifies the group `.keep` are the grouping variables kept in `.x`
`.messages`	a set of glue specs. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.headline`	a headline glue spec. The glue code can use any global variable, or {.strata},{.count.in},and {.count.out}
`.type`	default "modify": used to define formatting
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the transformed .data dataframe with the history graph updated.

See Also

dplyr::group_modify()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% track() %>% group_by(Species)
tmp %>% group_modify(
      function(d,g,...) { return(tibble::tibble(x=runif(10))) },
      .messages="{.count.in} in, {.count.out} out"
) %>% history()
```

`p_include_any` *Include any items matching a criteria*

Description

Apply a set of inclusion criteria and record the actions of the filter to the `dtrackr` history graph. Because of the ... filter specification, all parameters MUST BE NAMED. This function is the opposite of `exclude_all()` and the filtering criteria work to identify rows to include i.e. the results include anything that match any of the criteria. If `na.rm=TRUE` they also keep anything that cannot be evaluated by the criteria.

Usage

```
p_include_any(
  .data,
  ...,
  .headline = .defaultHeadline(),
  na.rm = TRUE,
  .type = "inclusion",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	a dplyr filter specification as a set of formulae where the LHS are predicates to test the data set against, items that match at least one of the predicates will be included. The RHS is a glue specification, defining the message, to be entered in the history graph for each predicate matched. This can refer to grouping variables, variables from the environment and {.included} and {.matched} or {.missing} (included = matched+missing), {.count} and {.total} - group and overall counts respectively, e.g. "excluding {.matched} items and {.missing} with missing values".
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`na.rm`	(default TRUE) if the filter cannot be evaluated for a row count that row as missing and either exclude it (TRUE) or don't exclude it (FALSE)
`.type`	default "inclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the filtered .data dataframe with the history graph updated with the summary of included items as a new stage

Examples

```
library(dplyr)
library(dtrackr)

iris %>% track() %>% group_by(Species) %>% include_any(
      Petal.Length > 5 ~ "{.included} long ones",
      Petal.Length < 2 ~ "{.included} short ones"
) %>% history()

# simultaneous evaluation of criteria:
data.frame(a = 1:10) %>%
  track() %>%
  include_any(
    # These two criteria identify the same value and one item is excluded
    a > 1 ~ "{.included} value > 1",
    a != min(a) ~ "{.included} everything but the smallest value",
  ) %>%
  status() %>%
  history()

# the behaviour is equivalent to dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a > 1, a != min(a)) %>%
  nrow()

# step-wise evaluation of criteria results in a different output
data.frame(a = 1:10) %>%
  track() %>%
  # Performing the same exclusion sequentially results in 2 items
  # being excluded as the criteria no longer identify the same
  # item.
  include_any(a > 1 ~ "{.included} value > 1") %>%
  include_any(a != min(a) ~ "{.included} everything but the smallest value") %>%
  status() %>%
  history()

# the behaviour is equivalent to dplyr's filter function:
data.frame(a=1:10) %>%
  dplyr::filter(a > 1) %>%
  dplyr::filter(a != min(a)) %>%
  nrow()
```

`p_inner_join` *Inner joins*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::inner_join()` for more details on the underlying functions.

Usage

```
p_inner_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Inner join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::inner_join` `by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`. `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it. `suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2. `keep` Should the join keys from both `x` and `y` be preserved in the output? • If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs. • If `TRUE`, all keys from both inputs are retained. • If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

	`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::inner_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Inner join
join = lhs %>% inner_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_intersect` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_intersect(
  x,
  y,
  ...,
  .messages = "{.count.out} in intersection",
  .headline = "Intersection"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

generics::intersect()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
```

```
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_left_join` *Left join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::left_join()` for more details on the underlying functions.

Usage

```
p_left_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
```

```
    .headline = "Left join by {.keys}"
  )
```

Arguments

`x, y` A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

`...` Other parameters passed onto methods. Named arguments passed on to `dplyr::left_join`

`by` A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::left_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Left join
join = lhs %>% left_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_mutate` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_mutate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::mutate` `.by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.keep` Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept. • `"all"` retains all columns from `.data`. This is the default. • `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side. • `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them. • `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept. `.before, .after` <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::mutate()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# mutate
# In this example we compare the column names of the input and the
# output to identify the new columns created by the mutate operation as
# the `.new_cols` variable
iris %>%
  track() %>%
  mutate(extra_col = NA_real_,
         .messages="{.new_cols}",
         .headline="Extra columns from mutate:") %>%
  history()
```

`p_nest_join` *Nest join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::nest_join()` for more details on the underlying functions.

Usage

```
p_nest_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS", "{.count.out} matched"),
  .headline = "Nest join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::nest_join` by A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`keep` Should the new list-column contain join keys? The default will preserve the join keys for inequality joins.

`name` The name of the list-column created by the join. If `NULL`, the default, the name of `y` is used.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::nest_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Nest join
join = lhs %>% nest_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_pause` *Pause tracking the data frame.*

Description

Pausing tracking of a data frame may be required if an operation is about to be performed that creates a lot of groupings or that you otherwise don't want to pollute the history graph (e.g. maybe selecting something using an anti-join). Once paused the history is not updated until a `resume()` is called, or when the data frame is ungrouped (if `auto` is enabled).

Usage

```
p_pause(.data, auto = FALSE)
```

Arguments

`.data`	a tracked dataframe
`auto`	if `TRUE` the tracking will resume automatically when the number of groups has fallen to a sensible level (default is `FALSE`)?

Value

the .data dataframe with history graph tracking paused

Examples

```
iris %>% track() %>% pause() %>% history()
```

`p_pivot_longer` *Reshaping data using* `tidyr::pivot_longer`

Description

A drop in replacement for `tidyr::pivot_longer()` which optionally takes a message and headline to store in the history graph.

Usage

```
p_pivot_longer(data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`data`	A data frame to pivot.
`...`	Additional arguments passed on to methods. Named arguments passed on to `tidyr::pivot_longer` `cols` <`tidy-select`> Columns to pivot into longer format. `cols_vary` When pivoting `cols` into longer format, how should the output rows be arranged relative to their original row number? • `"fastest"`, the default, keeps individual rows from `cols` close together in the output. This often produces intuitively ordered output when you have at least one key column from `data` that is not involved in the pivoting process. • `"slowest"` keeps individual columns from `cols` close together in the output. This often produces intuitively ordered output when you utilize all of the columns from `data` in the pivoting process. `names_to` A character vector specifying the new column or columns to create from the information stored in the column names of `data` specified by `cols`. • If length 0, or if `NULL` is supplied, no columns will be created. • If length 1, a single column will be created which will contain the column names specified by `cols`. • If length >1, multiple columns will be created. In this case, one of `names_sep` or `names_pattern` must be supplied to specify how the column names should be split. There are also two additional character values you can take advantage of: – `NA` will discard the corresponding component of the column name.

  - `".value"` indicates that the corresponding component of the column name defines the name of the output column containing the cell values, overriding `values_to` entirely.

`names_prefix` A regular expression used to remove matching text from the start of each variable name.

`names_sep,names_pattern` If `names_to` contains multiple values, these arguments control how the column name is broken up.

`names_sep` takes the same specification as `separate()`, and can either be a numeric vector (specifying positions to break on), or a single string (specifying a regular expression to split on).

`names_pattern` takes the same specification as `extract()`, a regular expression containing matching groups (`()`).

If these arguments do not give you enough control, use `pivot_longer_spec()` to create a spec object and process manually as needed.

`names_ptypes,values_ptypes` Optionally, a list of column name-prototype pairs. Alternatively, a single empty prototype can be supplied, which will be applied to all columns. A prototype (or ptype for short) is a zero-length vector (like `integer()` or `numeric()`) that defines the type, class, and attributes of a vector. Use these arguments if you want to confirm that the created columns are the types that you expect. Note that if you want to change (instead of confirm) the types of specific columns, you should use `names_transform` or `values_transform` instead.

`names_transform,values_transform` Optionally, a list of column name-function pairs. Alternatively, a single function can be supplied, which will be applied to all columns. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called `week` to an integer.

If not specified, the type of the columns generated from `names_to` will be character, and the type of the variables generated from `values_to` will be the common type of the input columns used to generate them.

`names_repair` What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_to` A string specifying the name of the column to create from the data stored in cell values. If `names_to` is a character containing the special `.value` sentinel, this value will be ignored, and the name of the value column will be derived from part of the existing column names.

`values_drop_na` If `TRUE`, will drop rows that contain only `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in `data` were created by its structure.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.

`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the result of the `tidyr::pivot_longer` but with a history graph updated.

See Also

tidyr::pivot_longer()

`p_pivot_wider` *Reshaping data using* `tidyr::pivot_wider`

Description

A drop in replacement for `tidyr::pivot_wider()` which optionally takes a message and headline to store in the history graph.

Usage

```
p_pivot_wider(data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`data`	A data frame to pivot.
`...`	Additional arguments passed on to methods. Named arguments passed on to `tidyr::pivot_wider` `id_cols` <`tidy-select`> 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`. `id_expand` Should the values in the `id_cols` columns be expanded by `expand()` before pivoting? This results in more rows, the output will contain a complete expansion of all possible values in `id_cols`. Implicit factor levels that aren't represented in the data will become explicit. Additionally, the row values corresponding to the expanded `id_cols` will be sorted. `names_from,values_from` <`tidy-select`> A pair of arguments describing which column (or columns) to get the name of the output column (`names_from`), and which column (or columns) to get the cell values from (`values_from`). If `values_from` contains multiple values, the value will be added to the front of the output column.

`names_prefix` String added to the start of every variable name. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.

`names_sep` If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.

`names_glue` Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns (and special `.value`) to create custom column names.

`names_sort` Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.

`names_vary` When `names_from` identifies a column (or columns) with multiple unique values, and multiple `values_from` columns are provided, in what order should the resulting column names be combined?

- `"fastest"` varies `names_from` values fastest, resulting in a column naming scheme of the form: `value1_name1, value1_name2, value2_name1, value2_name2`. This is the default.
- `"slowest"` varies `names_from` values slowest, resulting in a column naming scheme of the form: `value1_name1, value2_name1, value1_name2, value2_name2`.

`names_expand` Should the values in the `names_from` columns be expanded by `expand()` before pivoting? This results in more columns, the output will contain column names corresponding to a complete expansion of all possible values in `names_from`. Implicit factor levels that aren't represented in the data will become explicit. Additionally, the column names will be sorted, identical to what `names_sort` would produce.

`names_repair` What happens if the output has invalid column names? The default, `"check_unique"` is to error if the columns are duplicated. Use `"minimal"` to allow duplicates in the output, or `"unique"` to de-duplicated by adding numeric suffixes. See `vctrs::vec_as_names()` for more options.

`values_fill` Optionally, a (scalar) value that specifies what each `value` should be filled in with when missing.

This can be a named list if you want to apply different fill values to different value columns.

`values_fn` Optionally, a function applied to the value in each cell in the output. You will typically use this when the combination of `id_cols` and `names_from` columns does not uniquely identify an observation.

This can be a named list if you want to apply different aggregations to different `values_from` columns.

`unused_fn` Optionally, a function applied to summarize the values from the unused columns (i.e. columns not identified by `id_cols`, `names_from`, or `values_from`).

The default drops all unused columns from the result.

This can be a named list if you want to apply different aggregations to different unused columns.

`id_cols` must be supplied for `unused_fn` to be useful, since otherwise all unspecified columns will be considered `id_cols`.

	This is similar to grouping by the `id_cols` then summarizing the unused columns using `unused_fn`.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the data dataframe result of the `tidyr::pivot_wider` function but with a history graph updated with a `.message` if requested.

See Also

tidyr::pivot_wider()

`p_reframe` *Summarise a data set*

Description

Summarising a data set acts in the normal `dplyr` manner to collapse groups to individual rows. Any columns resulting from the summary can be added to the history graph. In the history this also joins any stratified branches and allows you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
p_reframe(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A data frame, to add multiple columns from a single expression. **[Deprecated]** Returning values with size 0 or >1 was deprecated as of 1.1.0. Please use `reframe()` for this instead. Named arguments passed on to `dplyr::reframe`

	`.by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
`.messages`	a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::reframe()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% track()
tmp %>% reframe(tibble(
  param = c("mean","min","max"),
  value = c(mean(Petal.Length), min(Petal.Length), max(Petal.Length))
  ), .messages="length {param}: {value}") %>% history()
```

`p_relocate` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_relocate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::relocate` `.before,.after` <`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::relocate()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# relocate, this shows how the columns can be reordered
iris %>%
  track() %>%
  group_by(Species) %>%
  relocate(
    tidyselect::starts_with("Sepal"),
```

```
    .after=Species,
    .messages="{.cols}",
    .headline="Order of columns from relocate:") %>%
  history()
```

`p_rename` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_rename(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::rename` `.fn` A function used to transform the selected `.cols`. Should return a character vector the same length as the input. `.cols` <`tidy-select`> Columns to rename; defaults to all columns.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::rename()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# rename can show us which columns are new and which have been
# removed (with .dropped_cols)
iris %>%
  track() %>%
  group_by(Species) %>%
  rename(
    Stamen.Width = Sepal.Width,
    Stamen.Length = Sepal.Length,
    .messages=c("added {.new_cols}","dropped {.dropped_cols}"),
    .headline="Renamed columns:") %>%
  history()
```

`p_rename_with` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_rename_with(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::rename_with` `.fn` A function used to transform the selected `.cols`. Should return a character vector the same length as the input. `.cols` <`tidy-select`> Columns to rename; defaults to all columns.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::rename_with()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# rename can show us which columns are new and which have been
# removed (with .dropped_cols)
iris %>%
  track() %>%
  group_by(Species) %>%
  rename(
```

```
    Stamen.Width = Sepal.Width,
    Stamen.Length = Sepal.Length,
    .messages=c("added {.new_cols}","dropped {.dropped_cols}"),
    .headline="Renamed columns:") %>%
  history()
```

`p_resume` *Resume tracking the data frame.*

Description

This may reset the grouping of the tracked data if the grouping structure has changed since the data frame was paused. If you try and resume tracking a data frame with too many groups (as defined by `options("dtrackr.max_supported_groupings"=XX)`) then the resume will fail and the data frame will still be paused. This can be overridden by specifying a value for the `.maxgroups` parameter.

Usage

```
p_resume(.data, ...)
```

Arguments

`.data`	a tracked dataframe
`...`	Named arguments passed on to `p_group_by` `.messages` a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by. `.headline` a headline glue spec. The glue code can use any global variable, or {.cols}. `.tag` if you want the summary data from this step in the future then give it a name with .tag. `.maxgroups` the maximum number of subgroups allowed before the tracking is paused. `...` In `group_by()`, variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate `mutate()` step before the `group_by()`. Computations are not allowed in `nest_by()`. In `ungroup()`, variables to remove from the grouping.

Value

the .data data frame with history graph tracking resumed

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% pause() %>% resume() %>% history()
```

`p_right_join` *Right join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::right_join()` for more details on the underlying functions.

Usage

```
p_right_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in linked set"),
  .headline = "Right join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::right_join` `by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`. `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.

- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::right_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Full join
join = lhs %>% full_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_select` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_select(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::select()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# select
# The output of the select verb (here using tidyselect syntax) can be captured
# and here all column names are being reported with the .cols variable.
iris %>%
  track() %>%
  group_by(Species) %>%
  select(
    tidyselect::starts_with("Sepal"),
    .messages="{.cols}",
    .headline="Output columns from select:") %>%
  history()
```

`p_semi_join` *Semi join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::semi_join()` for more details on the underlying functions.

Usage

```
p_semi_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in intersection"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::semi_join`

`by` A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::semi_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets
```

```
# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Semi join
join = lhs %>% semi_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`p_set` *Set the dtrackr history graph*

Description

This is unlikely to be useful to an end user and is called automatically by many of the other functions here. On the off chance you need to copy history metadata from one dataframe to another

Usage

```
p_set(.data, .graph)
```

Arguments

`.data`	a dataframe which may be grouped
`.graph`	a history graph list (consisting of nodes, edges, and head) see examples

Value

the .data dataframe with the history graph metadata set to the provided value

Examples

```
library(dplyr)
library(dtrackr)
mtcars %>% p_set(iris %>% comment("A comment") %>% p_get()) %>% history()
```

`p_setdiff` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_setdiff(
  x,
  y,
  ...,
  .messages = "{.count.out} items in difference",
  .headline = "Difference"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::setdiff()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
```

```
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_slice` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

	`weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice()

Examples

```
library(dplyr)
library(dtrackr)

# an arbitrary 50 items from the iris dataframe is selected. The
# history is tracked
iris %>% track() %>% slice(51:100) %>% history()
```

`p_slice_head` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice_head(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_head` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_head()

Examples

```
library(dplyr)
library(dtrackr)

# the first 50% of the data frame, is taken and the history tracked
iris %>% track() %>% group_by(Species) %>%
  slice_head(prop=0.5,.messages="{.count.out} / {.count.in}",
             .headline="First {sprintf('%1.0f',prop*100)}%") %>%
  history()

# The last 100 items:
iris %>% track() %>% group_by(Species) %>%
  slice_tail(n=100,.messages="{.count.out} / {.count.in}",
             .headline="Last 100") %>%
  history()
```

`p_slice_max` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice_max(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_max`

	`.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_max()

Examples

```
library(dplyr)
library(dtrackr)


# Subset the data by the maximum of a given value
```

```
iris %>% track() %>% group_by(Species) %>%
  slice_max(prop=0.5, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} = {prop} (with ties)",
            .headline="Widest 50% Sepals") %>%
  history()


# The narrowest 25% of the iris data set by group can be calculated in the
# slice_min() function. Recording this is a matter of tracking and
# using glue specs.
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_min(prop=0.25, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} (with ties)",
            .headline="narrowest {sprintf('%1.0f',prop*100)}% {Species}") %>%
  history()
```

`p_slice_min` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice_min(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_min`

- `.by,by` **[Experimental]**
  <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.
- `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.
- `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows.
  A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows.
- `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble.
- `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.
- `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.
- `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.
- `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_min()

Examples

```
library(dplyr)
library(dtrackr)


# Subset the data by the maximum of a given value
```

```
iris %>% track() %>% group_by(Species) %>%
  slice_max(prop=0.5, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} = {prop} (with ties)",
            .headline="Widest 50% Sepals") %>%
  history()


# The narrowest 25% of the iris data set by group can be calculated in the
# slice_min() function. Recording this is a matter of tracking and
# using glue specs.
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_min(prop=0.25, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} (with ties)",
            .headline="narrowest {sprintf('%1.0f',prop*100)}% {Species}") %>%
  history()
```

`p_slice_sample` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice_sample(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_sample`

`.by,by` **[Experimental]**
<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

`n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows.
A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows.

`order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble.

`with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.

`na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

`weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.

`replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_sample()

Examples

```
library(dplyr)
library(dtrackr)

# In this example the iris dataframe is resampled 100 times with replacement
# within each group and the
```

```
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_sample(n=100, replace=TRUE,
               .messages="{.count.out} / {.count.in} = {n}",
               .headline="100 {Species}") %>%
  history()
```

`p_slice_tail` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
p_slice_tail(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_tail` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

`n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows.

A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows.

`order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble.

`with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.

`na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

`weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.

`replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_tail()

Examples

```
library(dplyr)
library(dtrackr)

# the first 50% of the data frame, is taken and the history tracked
iris %>% track() %>% group_by(Species) %>%
  slice_head(prop=0.5,.messages="{.count.out} / {.count.in}",
             .headline="First {sprintf('%1.0f',prop*100)}%") %>%
  history()

# The last 100 items:
iris %>% track() %>% group_by(Species) %>%
  slice_tail(n=100,.messages="{.count.out} / {.count.in}",
             .headline="Last 100") %>%
```

```
history()
```

---

p_status *Add a summary to the dtrackr history graph*

Description

In the middle of a pipeline you may wish to document something about the data that is more complex than the simple counts. `status` is essentially a `dplyr` summarisation step which is connected to a `glue` specification output, that is recorded in the data frame history. This means you can do an arbitrary interim summarisation and put the result into the flowchart without disrupting the pipeline flow.

Usage

```
p_status(
  .data,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	any normal dplyr::summarise specification, e.g. `count=n()` or `av=mean(x)`, etcetera.
`.messages`	a character vector of glue specifications. A glue specification can refer to the summary outputs, any grouping variables of .data, the {.strata}, or any variables defined in the calling environment
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`.type`	one of "info","exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Details

Because of the ... summary specification parameters MUST BE NAMED.

Value

the same .data dataframe with the history metadata updated with the status inserted as a new stage

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% status(
      long = p_count_if(Petal.Length>5),
      short = p_count_if(Petal.Length<2),
      .messages="{Species}: {long} long ones & {short} short ones"
) %>% history()
```

`p_summarise` *Summarise a data set*

Description

Summarising a data set acts in the normal `dplyr` manner to collapse groups to individual rows. Any columns resulting from the summary can be added to the history graph. In the history this also joins any stratified branches and allows you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
p_summarise(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A data frame, to add multiple columns from a single expression. **[Deprecated]** Returning values with size 0 or >1 was deprecated as of 1.1.0. Please use `reframe()` for this instead. Named arguments passed on to `dplyr::summarise` `.by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.groups` **[Experimental]** Grouping structure of the result. • "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0. • "drop": All levels of grouping are dropped. • "keep": Same grouping structure as `.data`. • "rowwise": Each row is its own group.

When `.groups` is not specified, it is chosen based on the number of rows of the results:

- If all the results have 1 row, you get "drop_last".
- If the number of rows varies, you get "keep" (note that returning a variable number of rows was deprecated in favor of `reframe()`, which also unconditionally drops all levels of grouping).

In addition, a message informs you of that choice, unless the result is ungrouped, the option "dplyr.summarise.inform" is set to `FALSE`, or when `summarise()` is called from a function in a package.

`.messages`	a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::summarise()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% track()
tmp %>% summarise(avg = mean(Petal.Length), .messages="{avg} length") %>% history()
```

`p_tagged` *Retrieve tagged data in the history graph*

Description

Any counts at the individual stages that was stored with a `.tag` option in a pipeline step can be recovered here. The idea here is to provide a quick way to access a single value for the counts or other details tagged in a pipeline into a format that can be reported in text of a document. (e.g. for a results section). For more examples the consort statement vignette has some examples of use.

Usage

```
p_tagged(.data, .tag = NULL, .strata = NULL, .glue = NULL, ...)
```

Arguments

`.data`	the tracked dataframe.
`.tag`	(optional) the tag to retrieve.
`.strata`	(optional) filter the tagged data by the strata. set to "" to filter just the top level ungrouped data.
`.glue`	(optional) a glue specification which will be applied to the tagged content to generate a `.label` for the tagged content.
`...`	(optional) any other named parameters will be passed to `glue::glue` and can be used to generate a label.

Value

various things depending on what is requested.

By default a tibble with a `.tag` column and all associated summary values in a nested `.content` column.

If a `.strata` column is specified the results are filtered to just those that match a given `.strata` grouping (i.e. this will be the grouping label on the flowchart). Ungrouped content will have an empty "" as `.strata`

If `.tag` is specified the result will be for a single tag and `.content` will be automatically un-nested to give a single un-nested dataframe of the content captured at the `.tag` tagged step. This could be single or multiple rows depending on whether the original data was grouped at the point of tagging.

If both the `.tag` and `.glue` is specified a `.label` column will be computed from `.glue` and the tagged content. If the result of this is a single row then just the string value of `.label` is returned.

If just the `.glue` is specified, an un-nested dataframe with `.tag`,`.strata` and `.label` columns with a label for each tag in each strata.

If this seems complex then the best thing is to experiment until you get the output you want, leaving any `.glue` options until you think you know what you are doing. It made sense at the time.

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% comment(.tag = "step1")
tmp = tmp %>% filter(Species!="versicolor") %>% group_by(Species)
tmp %>% comment(.tag="step2") %>% tagged(.glue = "{.count}/{.total}")
```

`p_track` *Start tracking the dtrackr history graph*

Description

Start tracking the dtrackr history graph

Usage

```
p_track(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`.messages`	a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} variable which is the count of all rows, the {.count} variable which is the count of rows in the current group and the {.strata} which describes the current group. Defaults to the value of `getOption("dtrackr.default_message")`.
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is `nrow(.data)`, or {.strata} a summary of the current group. Defaults to the value of `getOption("dtrackr.default_hea`
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with additional history graph metadata, to allow tracking.

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% history()
```

`p_transmute` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
p_transmute(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::transmute()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# In this example we compare the column names of the input and the
# output to identify the new columns created by the transmute operation as
# the `.new_cols` variable
# Here we do the same for a transmute()
iris %>%
  track() %>%
  group_by(Species, .add=TRUE) %>%
```

```
transmute(
  sepal.w = Sepal.Width-1,
  sepal.l = Sepal.Length+1,
  .messages="{.new_cols}",
  .headline="New columns from transmute:") %>%
history()
```

`p_ungroup` *Remove a stratification from a data set*

Description

Un-grouping a data set logically combines the different arms. In the history this joins any stratified branches and acts as a specific type of `status()`, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::ungroup()`.

Usage

```
p_ungroup(
  x,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`x`	A `tbl()`
`...`	variables to remove from the grouping.
`.messages`	a set of glue specs. The glue code can use any any global variable, or {.count}. the default is "total {.count} items"
`.headline`	a headline glue spec. The glue code can use {.count} and {.strata}.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe but ungrouped with the history graph updated showing the ungroup operation as a new stage.

See Also

dplyr::ungroup()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% comment("A test")
tmp %>% ungroup(.messages="{.count} items in combined") %>% history()
```

`p_union` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_union(
  x,
  y,
  ...,
  .messages = "{.count.out} unique items in union",
  .headline = "Distinct union"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

generics::union()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
```

```
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_union_all` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
p_union_all(
  x,
  y,
  ...,
  .messages = "{.count.out} items in union",
  .headline = "Union"
)
```

Arguments

`x, y`	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::union_all()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
```

```
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`p_untrack` *Remove tracking from the dataframe*

Description

Remove tracking from the dataframe

Usage

```
p_untrack(.data)
```

Arguments

`.data`	a tracked dataframe

Value

the .data dataframe with history graph metadata removed.

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% untrack() %>% class()
```

`reframe.trackr_df` *Summarise a data set*

Description

Summarising a data set acts in the normal `dplyr` manner to collapse groups to individual rows. Any columns resulting from the summary can be added to the history graph. In the history this also joins any stratified branches and allows you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
## S3 method for class 'trackr_df'
reframe(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A data frame, to add multiple columns from a single expression. **[Deprecated]** Returning values with size 0 or >1 was deprecated as of 1.1.0. Please use `reframe()` for this instead.
`.messages`	a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::reframe()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% track()
tmp %>% reframe(tibble(
  param = c("mean","min","max"),
  value = c(mean(Petal.Length), min(Petal.Length), max(Petal.Length))
  ), .messages="length {param}: {value}") %>% history()
```

`relocate.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
relocate(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::relocate` `.before`,`.after` <`tidy-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is an error.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::relocate()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# relocate, this shows how the columns can be reordered
iris %>%
  track() %>%
  group_by(Species) %>%
  relocate(
    tidyselect::starts_with("Sepal"),
    .after=Species,
    .messages="{.cols}",
    .headline="Order of columns from relocate:") %>%
  history()
```

`rename.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
rename(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped).

- `NULL`, to remove the column.
- A data frame or tibble, to create multiple columns in the output.

Named arguments passed on to `dplyr::rename`

`.fn`	A function used to transform the selected `.cols`. Should return a character vector the same length as the input.
`.cols`	<`tidy-select`> Columns to rename; defaults to all columns.

`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::rename()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# rename can show us which columns are new and which have been
# removed (with .dropped_cols)
iris %>%
  track() %>%
  group_by(Species) %>%
  rename(
    Stamen.Width = Sepal.Width,
    Stamen.Length = Sepal.Length,
    .messages=c("added {.new_cols}","dropped {.dropped_cols}"),
    .headline="Renamed columns:") %>%
  history()
```

`rename_with.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
rename_with(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output. Named arguments passed on to `dplyr::rename_with` `.fn` A function used to transform the selected `.cols`. Should return a character vector the same length as the input. `.cols` <`tidy-select`> Columns to rename; defaults to all columns.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::rename_with()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# rename can show us which columns are new and which have been
# removed (with .dropped_cols)
iris %>%
  track() %>%
  group_by(Species) %>%
  rename(
    Stamen.Width = Sepal.Width,
    Stamen.Length = Sepal.Length,
    .messages=c("added {.new_cols}","dropped {.dropped_cols}"),
    .headline="Renamed columns:") %>%
  history()
```

`resume` *Resume tracking the data frame.*

Description

This may reset the grouping of the tracked data if the grouping structure has changed since the data frame was paused. If you try and resume tracking a data frame with too many groups (as defined by `options("dtrackr.max_supported_groupings"=XX)`) then the resume will fail and the data frame will still be paused. This can be overridden by specifying a value for the `.maxgroups` parameter.

Usage

```
resume(.data, ...)
```

Arguments

`.data`	a tracked dataframe
`...`	Named arguments passed on to `p_group_by` `.messages` a set of glue specs. The glue code can use any global variable, or {.cols} which is the columns that are being grouped by. `.headline` a headline glue spec. The glue code can use any global variable, or {.cols}.

`.tag` if you want the summary data from this step in the future then give it a name with .tag.

`.maxgroups` the maximum number of subgroups allowed before the tracking is paused.

`...` In `group_by()`, variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate `mutate()` step before the `group_by()`. Computations are not allowed in `nest_by()`. In `ungroup()`, variables to remove from the grouping. Named arguments passed on to `dplyr::group_by`

  `.add` When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`.
  This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions.

  `.drop` Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop = FALSE`. See `group_by_drop_default()` for details.

  `x` A `tbl()`

Value

the .data data frame with history graph tracking resumed

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% pause() %>% resume() %>% history()
```

`right_join.trackr_df` *Right join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::right_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
right_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
```

```
    "{.count.out} in linked set"),
  .headline = "Right join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::right_join`

`by` A join specification created with `join_by()`, or a character vector of variables to join by.

If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly.

To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`suffix` If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`keep` Should the join keys from both `x` and `y` be preserved in the output?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::right_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Full join
join = lhs %>% full_join(rhs, by="name", multiple = "all") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`save_dot` *Save DOT content to a file*

Description

Convert a digraph in dot format to SVG and save it to a range of output file types

Usage

```
save_dot(
  dot,
  filename,
  size = std_size$half,
  maxWidth = size$width,
```

```
  maxHeight = size$height,
  formats = c("dot", "png", "pdf", "svg"),
  landscape = size$rot != 0,
  ...
)
```

Arguments

`dot`	a `graphviz` dot string
`filename`	the full path of the file name (minus extension for multiple formats)
`size`	a named list with 3 elements, length and width in inches and rotation. A predefined set of standard sizes are available in the std_size object.
`maxWidth`	a width (on the paper) in inches if `size` is not defined
`maxHeight`	a height (on the paper) in inches if `size` is not defined
`formats`	some of `pdf,dot,svg,png,ps`
`landscape`	rotate the output by 270 degrees into a landscape format. `maxWidth` and `maxHeight` still apply and refer to the paper width to fit the flowchart into after rotation. (you might need to flip width and height)
`...`	ignored

Value

a list with items `paths` with the absolute paths of the saved files as a named list, and `svg` as the SVG string of the rendered dot file.

Examples

```
save_dot("digraph {A->B}",tempfile())
```

`select.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
select(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::select()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# select
# The output of the select verb (here using tidyselect syntax) can be captured
# and here all column names are being reported with the .cols variable.
iris %>%
  track() %>%
  group_by(Species) %>%
  select(
    tidyselect::starts_with("Sepal"),
    .messages="{.cols}",
    .headline="Output columns from select:") %>%
  history()
```

`semi_join.trackr_df` *Semi join*

Description

Mutating joins behave as `dplyr` joins, except the history graph of the two sides of the joins is merged resulting in a tracked dataframe with the history of both input dataframes. See `dplyr::semi_join()` for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
semi_join(
  x,
  y,
  ...,
  .messages = c("{.count.lhs} on LHS", "{.count.rhs} on RHS",
    "{.count.out} in intersection"),
  .headline = "Semi join by {.keys}"
)
```

Arguments

`x, y`	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Other parameters passed onto methods. Named arguments passed on to `dplyr::semi_join` `by` A join specification created with `join_by()`, or a character vector of variables to join by. If `NULL`, the default, `*_join()` will perform a natural join, using all variables in common across `x` and `y`. A message lists the variables so that you can check they're correct; suppress the message by supplying `by` explicitly. To join on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`. To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`. `join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins. For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`. To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`. `copy` If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.

`na_matches`	Should two `NA` or two `NaN` values match? • `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`. • `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.
`.messages`	a set of glue specs. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively
`.headline`	a glue spec. The glue code can use any global variable, {.keys} for the joining columns, {.count.lhs}, {.count.rhs}, {.count.out} for the input and output dataframes sizes respectively

Value

the join of the two dataframes with the history graph updated.

See Also

dplyr::semi_join()

Examples

```
library(dplyr)
library(dtrackr)
# Joins across data sets

# example data uses the dplyr starways data
people = starwars %>% select(-films, -vehicles, -starships)
films = starwars %>% select(name,films) %>% tidyr::unnest(cols = c(films))

lhs = people %>% track() %>% comment("People df {.total}")
rhs = films %>% track() %>% comment("Films df {.total}") %>%
  comment("a test comment")

# Semi join
join = lhs %>% semi_join(rhs, by="name") %>% comment("joined {.total}")
# See what the history of the graph is:
join %>% history() %>% print()
nrow(join)
# Display the tracked graph (not run in examples)
# join %>% flowchart()
```

`setdiff.trackr_df` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
## S3 method for class 'trackr_df'
setdiff(
  x,
  y,
  ...,
  .messages = "{.count.out} items in difference",
  .headline = "Difference"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::setdiff()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
```

```
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`slice.trackr_df` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.

`na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

`weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.

`replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice()

Examples

```
library(dplyr)
library(dtrackr)

# an arbitrary 50 items from the iris dataframe is selected. The
# history is tracked
iris %>% track() %>% slice(51:100) %>% history()
```

`slice_head.trackr_df` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice_head(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_head` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_head()

Examples

```
library(dplyr)
library(dtrackr)

# the first 50% of the data frame, is taken and the history tracked
iris %>% track() %>% group_by(Species) %>%
  slice_head(prop=0.5,.messages="{.count.out} / {.count.in}",
             .headline="First {sprintf('%1.0f',prop*100)}%") %>%
  history()

# The last 100 items:
iris %>% track() %>% group_by(Species) %>%
  slice_tail(n=100,.messages="{.count.out} / {.count.in}",
             .headline="Last 100") %>%
  history()
```

`slice_max.trackr_df` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice_max(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_max` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_max()

Examples

```
library(dplyr)
library(dtrackr)


# Subset the data by the maximum of a given value
iris %>% track() %>% group_by(Species) %>%
  slice_max(prop=0.5, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} = {prop} (with ties)",
            .headline="Widest 50% Sepals") %>%
  history()


# The narrowest 25% of the iris data set by group can be calculated in the
# slice_min() function. Recording this is a matter of tracking and
# using glue specs.
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_min(prop=0.25, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} (with ties)",
            .headline="narrowest {sprintf('%1.0f',prop*100)}% {Species}") %>%
  history()
```

`slice_min.trackr_df` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice_min(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_min` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_min()

Examples

```
library(dplyr)
library(dtrackr)


# Subset the data by the maximum of a given value
iris %>% track() %>% group_by(Species) %>%
  slice_max(prop=0.5, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} = {prop} (with ties)",
            .headline="Widest 50% Sepals") %>%
  history()


# The narrowest 25% of the iris data set by group can be calculated in the
# slice_min() function. Recording this is a matter of tracking and
# using glue specs.
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_min(prop=0.25, order_by = Sepal.Width,
            .messages="{.count.out} / {.count.in} (with ties)",
            .headline="narrowest {sprintf('%1.0f',prop*100)}% {Species}") %>%
  history()
```

`slice_sample.trackr_df`

Slice operations

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice_sample(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_sample` `.by,by` **[Experimental]** <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by. `.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is. `n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows. A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows. `order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble. `with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows. `na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`. `weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1. `replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.
`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_sample()

Examples

```
library(dplyr)
library(dtrackr)

# In this example the iris dataframe is resampled 100 times with replacement
# within each group and the
iris %>%
  track() %>%
  group_by(Species) %>%
  slice_sample(n=100, replace=TRUE,
               .messages="{.count.out} / {.count.in} = {n}",
               .headline="100 {Species}") %>%
  history()
```

`slice_tail.trackr_df` *Slice operations*

Description

Slice operations behave as in dplyr, except the history graph can be updated with tracked dataframe with the before and after sizes of the dataframe. See `dplyr::slice()`, `dplyr::slice_head()`, `dplyr::slice_tail()`, `dplyr::slice_min()`, `dplyr::slice_max()`, `dplyr::slice_sample()`, for more details on the underlying functions.

Usage

```
## S3 method for class 'trackr_df'
slice_tail(
  .data,
  ...,
  .messages = c("{.count.in} before", "{.count.out} after"),
  .headline = "slice data"
)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods. Named arguments passed on to `dplyr::slice_tail`

`.by,by` **[Experimental]**
<`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.preserve` Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

`n,prop` Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. `prop` will be rounded towards zero to generate an integer number of rows.
A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows.

`order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble.

`with_ties` Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.

`na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

`weight_by` <`data-masking`> Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.

`replace` Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

`.messages`	a set of glue specs. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively and {.excluded} for the difference
`.headline`	a glue spec. The glue code can use any global variable, {.count.in}, {.count.out} for the input and output dataframes sizes respectively.

Value

the sliced dataframe with the history graph updated.

See Also

dplyr::slice_tail()

Examples

```
library(dplyr)
library(dtrackr)

# the first 50% of the data frame, is taken and the history tracked
iris %>% track() %>% group_by(Species) %>%
```

```
  slice_head(prop=0.5,.messages="{.count.out} / {.count.in}",
             .headline="First {sprintf('%1.0f',prop*100)}%") %>%
  history()

# The last 100 items:
iris %>% track() %>% group_by(Species) %>%
  slice_tail(n=100,.messages="{.count.out} / {.count.in}",
             .headline="Last 100") %>%
  history()
```

`status` *Add a summary to the dtrackr history graph*

Description

In the middle of a pipeline you may wish to document something about the data that is more complex than the simple counts. `status` is essentially a `dplyr` summarisation step which is connected to a `glue` specification output, that is recorded in the data frame history. This means you can do an arbitrary interim summarisation and put the result into the flowchart without disrupting the pipeline flow.

Usage

```
status(
  .data,
  ...,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .type = "info",
  .asOffshoot = FALSE,
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`...`	any normal dplyr::summarise specification, e.g. `count=n()` or `av=mean(x)`, etcetera.
`.messages`	a character vector of glue specifications. A glue specification can refer to the summary outputs, any grouping variables of .data, the {.strata}, or any variables defined in the calling environment
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment
`.type`	one of "info","exclusion": used to define formatting
`.asOffshoot`	do you want this comment to be an offshoot of the main flow (default = FALSE).
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Details

Because of the ... summary specification parameters MUST BE NAMED.

Value

the same .data dataframe with the history metadata updated with the status inserted as a new stage

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% group_by(Species)
tmp %>% status(
      long = p_count_if(Petal.Length>5),
      short = p_count_if(Petal.Length<2),
      .messages="{Species}: {long} long ones & {short} short ones"
) %>% history()
```

`std_size` *Standard paper sizes*

Description

A list of standard paper sizes for outputting flowcharts or other dot graphs. These include width and height dimensions in inches and can be used as one way to specify the output size of a dot graph, including flowcharts (see the `size` parameter of `flowchart()`).

Usage

```
std_size
```

Format

An object of class `list` of length 12.

Details

The sizes available are `A4`, `A5`, `full` (fits a portrait A4 with margins), `half` (half an A4 with margins), `third`, `two_third`, `quarter`, `sixth` (all with reference to an A4 page with margins). There are 2 landscape sizes `A4_landscape` and `full_landscape` which fit an A4 page with or without margins. There are also 2 slide dimensions, to fit with standard presentation software dimensions.

This is just a convenience. Similar effects can be achieved by providing `width` and `height` parameters to `flowchart()` directly.

`summarise.trackr_df` *Summarise a data set*

Description

Summarising a data set acts in the normal `dplyr` manner to collapse groups to individual rows. Any columns resulting from the summary can be added to the history graph. In the history this also joins any stratified branches and allows you to generate some summary statistics about the un-grouped data. See `dplyr::summarise()`.

Usage

```
## S3 method for class 'trackr_df'
summarise(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A data frame, to add multiple columns from a single expression. **[Deprecated]** Returning values with size 0 or >1 was deprecated as of 1.1.0. Please use `reframe()` for this instead.
`.messages`	a set of glue specs. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.headline`	a headline glue spec. The glue code can use any summary variable defined in the ... parameter, or any global variable, or {.strata}
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe summarised with the history graph updated showing the summarise operation as a new stage

See Also

dplyr::summarise()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% track()
tmp %>% summarise(avg = mean(Petal.Length), .messages="{avg} length") %>% history()
```

`tagged` *Retrieve tagged data in the history graph*

Description

Any counts at the individual stages that was stored with a `.tag` option in a pipeline step can be recovered here. The idea here is to provide a quick way to access a single value for the counts or other details tagged in a pipeline into a format that can be reported in text of a document. (e.g. for a results section). For more examples the consort statement vignette has some examples of use.

Usage

```
tagged(.data, .tag = NULL, .strata = NULL, .glue = NULL, ...)
```

Arguments

`.data`	the tracked dataframe.
`.tag`	(optional) the tag to retrieve.
`.strata`	(optional) filter the tagged data by the strata. set to "" to filter just the top level ungrouped data.
`.glue`	(optional) a glue specification which will be applied to the tagged content to generate a `.label` for the tagged content.
`...`	(optional) any other named parameters will be passed to `glue::glue` and can be used to generate a label.

Value

various things depending on what is requested.

By default a tibble with a `.tag` column and all associated summary values in a nested `.content` column.

If a `.strata` column is specified the results are filtered to just those that match a given `.strata` grouping (i.e. this will be the grouping label on the flowchart). Ungrouped content will have an empty "" as `.strata`

If `.tag` is specified the result will be for a single tag and `.content` will be automatically un-nested to give a single un-nested dataframe of the content captured at the `.tag` tagged step. This could be single or multiple rows depending on whether the original data was grouped at the point of tagging.

If both the `.tag` and `.glue` is specified a `.label` column will be computed from `.glue` and the tagged content. If the result of this is a single row then just the string value of `.label` is returned.

If just the `.glue` is specified, an un-nested dataframe with `.tag`,`.strata` and `.label` columns with a label for each tag in each strata.

If this seems complex then the best thing is to experiment until you get the output you want, leaving any `.glue` options until you think you know what you are doing. It made sense at the time.

Examples

```
library(dplyr)
library(dtrackr)
tmp = iris %>% track() %>% comment(.tag = "step1")
tmp = tmp %>% filter(Species!="versicolor") %>% group_by(Species)
tmp %>% comment(.tag="step2") %>% tagged(.glue = "{.count}/{.total}")
```

`track` *Start tracking the dtrackr history graph*

Description

Start tracking the dtrackr history graph

Usage

```
track(
  .data,
  .messages = .defaultMessage(),
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`.data`	a dataframe which may be grouped
`.messages`	a character vector of glue specifications. A glue specification can refer to any grouping variables of .data, or any variables defined in the calling environment, the {.total} variable which is the count of all rows, the {.count} variable which is the count of rows in the current group and the {.strata} which describes the current group. Defaults to the value of `getOption("dtrackr.default_message")`.
`.headline`	a glue specification which can refer to grouping variables of .data, or any variables defined in the calling environment, or the {.total} variable which is `nrow(.data)`, or {.strata} a summary of the current group. Defaults to the value of `getOption("dtrackr.default_hea`
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe with additional history graph metadata, to allow tracking.

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% history()
```

`transmute.trackr_df` *dplyr modifying operations*

Description

See `dplyr::mutate()`, `dplyr::add_count()`, `dplyr::add_tally()`, `dplyr::transmute()`, `dplyr::select()`, `dplyr::relocate()`, `dplyr::rename()` `dplyr::rename_with()`, `dplyr::arrange()` for more details on underlying functions. `dtrackr` provides equivalent functions for mutating, selecting and renaming a data set which act in the same way as `dplyr`. `mutate` / `select` / `rename` generally don't add anything in terms of provenance of data so the default behaviour is to miss these out of the `dtrackr` history. This can be overridden with the `.messages`, or `.headline` values in which case they behave just like a `comment()`.

Usage

```
## S3 method for class 'trackr_df'
transmute(.data, ..., .messages = "", .headline = "", .tag = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.messages`	a set of glue specs. The glue code can use any global variable, grouping variable, {.new_cols} or {.dropped_cols} for changes to columns, {.cols} for the output column names, or {.strata}. Defaults to nothing.
`.headline`	a headline glue spec. The glue code can use any global variable, grouping variable, {.new_cols}, {.dropped_cols}, {.cols} or {.strata}. Defaults to nothing.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the `.data` dataframe after being modified by the `dplyr` equivalent function, but with the history graph updated with a new stage if the `.messages` or `.headline` parameter is not empty.

See Also

dplyr::transmute()

Examples

```
library(dplyr)
library(dtrackr)

# mutate and other functions are unitary operations that generally change
# the structure but not size of a dataframe. In dtrackr these are by ignored
# by default but we can change that so that their behaviour is obvious.

# In this example we compare the column names of the input and the
# output to identify the new columns created by the transmute operation as
# the `.new_cols` variable
# Here we do the same for a transmute()
iris %>%
  track() %>%
  group_by(Species, .add=TRUE) %>%
  transmute(
    sepal.w = Sepal.Width-1,
    sepal.l = Sepal.Length+1,
    .messages="{.new_cols}",
    .headline="New columns from transmute:") %>%
  history()
```

`ungroup.trackr_df` *Remove a stratification from a data set*

Description

Un-grouping a data set logically combines the different arms. In the history this joins any stratified branches and acts as a specific type of `status()`, allowing you to generate some summary statistics about the un-grouped data. See `dplyr::ungroup()`.

Usage

```
## S3 method for class 'trackr_df'
ungroup(
  x,
  ...,
  .messages = .defaultMessage(),
```

```
  .headline = .defaultHeadline(),
  .tag = NULL
)
```

Arguments

`x`	A `tbl()`
`...`	variables to remove from the grouping.
`.messages`	a set of glue specs. The glue code can use any any global variable, or {.count}. the default is "total {.count} items"
`.headline`	a headline glue spec. The glue code can use {.count} and {.strata}.
`.tag`	if you want the summary data from this step in the future then give it a name with .tag.

Value

the .data dataframe but ungrouped with the history graph updated showing the ungroup operation as a new stage.

See Also

dplyr::ungroup()

Examples

```
library(dplyr)
library(dtrackr)

tmp = iris %>% group_by(Species) %>% comment("A test")
tmp %>% ungroup(.messages="{.count} items in combined") %>% history()
```

`union.trackr_df` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
## S3 method for class 'trackr_df'
union(
  x,
  y,
  ...,
  .messages = "{.count.out} unique items in union",
  .headline = "Distinct union"
)
```

Arguments

`x, y`	Vectors to combine.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

generics::union()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
```

```
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`union_all.trackr_df` *Set operations*

Description

These perform set operations on tracked dataframes. It merges the history of 2 (or more) dataframes and combines the rows (or columns). It calculates the total number of resulting rows as {.count.out} in other terms it performs exactly the same operation as the equivalent `dplyr` operation. See `dplyr::bind_rows()`, `dplyr::bind_cols()`, `dplyr::intersect()`, `dplyr::union()`, `dplyr::setdiff()`,`dplyr::inte` or `dplyr::union_all()` for the underlying function details.

Usage

```
## S3 method for class 'trackr_df'
union_all(
  x,
```

```
  y,
  ...,
  .messages = "{.count.out} items in union",
  .headline = "Union"
)
```

Arguments

`x, y`	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
`...`	a collection of tracked data frames to combine
`.messages`	a set of glue specs. The glue code can use any global variable, or {.count.out}
`.headline`	a glue spec. The glue code can use any global variable, or {.count.out}

Value

the dplyr output with the history graph updated.

See Also

dplyr::union_all()

Examples

```
library(dplyr)
library(dtrackr)

# Set operations
people = starwars %>% select(-films, -vehicles, -starships)
chrs = people %>% track("start")

lhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Droid" ~ "{.included} droids"
)

# these are different subsets of the same data
rhs = chrs %>% include_any(
  species == "Human" ~ "{.included} humans",
  species == "Gungan" ~ "{.included} gungans"
) %>% comment("{.count} gungans & humans")


# Unions
set = bind_rows(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

```
set = union(lhs,rhs) %>% comment("{.count} human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = union_all(lhs,rhs) %>% comment("{.count} 2*human,droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

# Intersections and differences

set = setdiff(lhs,rhs) %>% comment("{.count} droids and gungans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()

set = intersect(lhs,rhs) %>% comment("{.count} humans")
# display the history of the result:
set %>% history()
nrow(set)
# not run - display the flowchart:
# set %>% flowchart()
```

`untrack` *Remove tracking from the dataframe*

Description

Remove tracking from the dataframe

Usage

```
untrack(.data)
```

Arguments

`.data`	a tracked dataframe

Value

the .data dataframe with history graph metadata removed.

Examples

```
library(dplyr)
library(dtrackr)
iris %>% track() %>% untrack() %>% class()
```

Index