Package 'tidySpatialExperiment'

November 4, 2025

Type Package

Title SpatialExperiment with tidy principles

Version 1.7.1

Description tidySpatialExperiment provides a bridge between the SpatialExperiment package and the tidyverse ecosystem. It creates an invisible layer that allows you to interact with a SpatialExperiment object as if it were a tibble; enabling the use of functions from dplyr, tidyr, ggplot2 and plotly. But, underneath, your data remains a SpatialExperiment object.

License GPL (>= 3)

Depends R (>= 4.3.0), SpatialExperiment, tidySingleCellExperiment, ttservice

Imports SummarizedExperiment, SingleCellExperiment, BiocGenerics, S4Vectors, methods, utils, pkgconfig, tibble, dplyr, tidyr, ggplot2 (>= 4.0.0), plotly, rlang, purrr, stringr, vctrs, tidyselect, pillar, cli, fansi, lifecycle, magick, tidygate (>= 1.0.13), shiny

Suggests BiocStyle, testthat, knitr, markdown, scater, igraph, cowplot, DropletUtils, tidySummarizedExperiment

VignetteBuilder knitr

Biarch true

biocViews Infrastructure, RNASeq, GeneExpression, Sequencing, Spatial, Transcriptomics, SingleCell

Encoding UTF-8

RoxygenNote 7.3.3

Roxygen list(markdown = TRUE)

URL https://github.com/william-hutchison/tidySpatialExperiment, https://william-hutchison.github.io/tidySpatialExperiment/

BugReports https://github.com/william-hutchison/tidySpatialExperiment/issues

LazyData true

git_url https://git.bioconductor.org/packages/tidySpatialExperiment

git_branch devel

git_last_commit 9871264

git_last_commit_date 2025-11-02

Repository Bioconductor 3.23

Date/Publication 2025-11-03

Author William Hutchison [aut, cre] (ORCID:
<https://orcid.org/0009-0001-6242-4269>),
Stefano Mangiola [aut]

Maintainer William Hutchison <hutchison.w@wehi.edu.au>

Contents

`add_class` *Add class to abject*

Description

Add class to abject

Usage

```
add_class(var, name)
```

Arguments

`var`	A tibble
`name`	A character name of the attribute

Value

A tibble with an additional attribute

`add_count.SpatialExperiment` *Count the observations in each group*

Description

`count()` lets you quickly count the unique values of one or more variables: `df %>% count(a, b)` is roughly equivalent to `df %>% group_by(a, b) %>% summarise(n = n())`. `count()` is paired with `tally()`, a lower-level helper that is equivalent to `df %>% summarise(n = n())`. Supply `wt` to perform weighted counts, switching the summary from `n = n()` to `n = sum(wt)`.

`add_count()` and `add_tally()` are equivalents to `count()` and `tally()` but use `mutate()` instead of `summarise()` so that they add a new column with group-wise counts.

Usage

```
## S3 method for class 'SpatialExperiment'
add_count(x, ..., wt = NULL, sort = FALSE, name = 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.
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 ns until it gets a new name.

Value

An object of the same type as .data. count() and add_count() group transiently, so the output has the same groups as the input.

Examples

```
example(read10xVisium)
spe |>
    count()
spe |>
    add_count()
```

---

aggregate_cells *Aggregate cells*

Description

Combine cells into groups based on shared variables and aggregate feature counts.

Arguments

.data	A tidySpatialExperiment object
.sample	A vector of variables by which cells are aggregated
slot	The slot to which the function is applied
assays	The assay to which the function is applied
aggregation_function	The method of cell-feature value aggregation

Value

A SummarizedExperiment object

Examples

```
example(read10xVisium)
spe |>
    aggregate_cells(sample_id, assays = "counts")
```

`arrange` *Order rows using column values*

Description

`arrange()` orders the rows of a data frame by the values of selected columns.

Unlike other dplyr verbs, `arrange()` largely ignores grouping; you need to explicitly mention grouping variables (or use `.by_group = TRUE`) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

Details

Missing values:

Unlike base sorting with `sort()`, `NA` are:

- always sorted to the end for local data, even when wrapped with `desc()`.
- treated differently for remote data, depending on the backend.

Value

An object of the same type as `.data`. The output has the following properties:

- All rows appear in the output, but (usually) in a different place.
- Columns are not modified.
- Groups are not modified.
- Data frame attributes are preserved.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `mutate()`, `rename()`, `slice()`, `summarise()`

Examples

```
example(read10xVisium)

spe |>
    arrange(array_row)
```

---

as_tibble *Coerce lists, matrices, and more to data frames*

Description

`as_tibble()` turns an existing object, such as a data frame or matrix, into a so-called tibble, a data frame with class `tbl_df`. This is in contrast with `tibble()`, which builds a tibble from individual columns. `as_tibble()` is to `tibble()` as `base::as.data.frame()` is to `base::data.frame()`.

`as_tibble()` is an S3 generic, with methods for:

- `data.frame`: Thin wrapper around the `list` method that implements tibble's treatment of rownames.
- `matrix`, `poly`, `ts`, `table`
- Default: Other inputs are first coerced with `base::as.data.frame()`.

`as_tibble_row()` converts a vector to a tibble with one row. If the input is a list, all elements must have size one.

`as_tibble_col()` converts a vector to a tibble with one column.

Usage

```
## S3 method for class 'SpatialExperiment'
as_tibble(
  x,
  ...,
  .name_repair = c("check_unique", "unique", "universal", "minimal"),
  rownames = pkgconfig::get_config("tibble::rownames", NULL)
)
```

Arguments

`x`	A data frame, list, matrix, or other object that could reasonably be coerced to a tibble.
`...`	Unused, for extensibility.
`.name_repair`	Treatment of problematic column names: • `"minimal"`: No name repair or checks, beyond basic existence, • `"unique"`: Make sure names are unique and not empty, • `"check_unique"`: (default value), no name repair, but check they are `unique`, • `"universal"`: Make the names `unique` and syntactic

- "unique_quiet": Same as "unique", but "quiet"
- "universal_quiet": Same as "universal", but "quiet"
- a function: apply custom name repair (e.g., .name_repair = make.names for names in the style of base R).
- A purrr-style anonymous function, see rlang::as_function()

This argument is passed on as repair to vctrs::vec_as_names(). See there for more details on these terms and the strategies used to enforce them.

rownames How to treat existing row names of a data frame or matrix:

- NULL: remove row names. This is the default.
- NA: keep row names.
- A string: the name of a new column. Existing rownames are transferred into this column and the row.names attribute is deleted. No name repair is applied to the new column name, even if x already contains a column of that name. Use as_tibble(rownames_to_column(...)) to safeguard against this case.

Read more in rownames.

Value

tibble

Row names

The default behavior is to silently remove row names.

New code should explicitly convert row names to a new column using the rownames argument.

For existing code that relies on the retention of row names, call pkgconfig::set_config("tibble::rownames" = NA) in your script or in your package's .onLoad() function.

Life cycle

Using as_tibble() for vectors is superseded as of version 3.0.0, prefer the more expressive as_tibble_row() and as_tibble_col() variants for new code.

See Also

tibble() constructs a tibble from individual columns. enframe() converts a named vector to a tibble with a column of names and column of values. Name repair is implemented using vctrs::vec_as_names().

Examples

```
example(read10xVisium)
spe |>
    as_tibble()
```

bind_cols *Efficiently bind multiple data frames by row and column*

Description

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

Details

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

Value

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

Examples

```
# Note: "dplyr" does not provide a generic function for `bind_cols`. Therefore, the generic
# function `bind_cols` located in "ttservice" should be called explicitly with
# `ttservice::bind_cols` to avoid conflicts.

example(read10xVisium)
spe |>
    ttservice::bind_cols(1:99)
```

bind_rows *Efficiently bind multiple data frames by row and column*

Description

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

Details

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

Value

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

Examples

```
# Note: "dplyr" does not provide a generic function for `bind_rows`. Therefore, the generic
# function `bind_rows` located in "ttservice" should be called explicitly with
# `ttservice::bind_rows` to avoid conflicts.

example(read10xVisium)
spe |>
    ttservice::bind_rows(spe)
```

---

demo_brush_data *Demo brush data*

Description

Demo brush data

Usage

```
demo_brush_data
```

Format

An object of class `spec_tbl_df` (inherits from `tbl_df`, `tbl`, `data.frame`) with 30 rows and 3 columns.

---

demo_select_data *Demo select data*

Description

Demo select data

Usage

```
demo_select_data
```

Format

An object of class `spec_tbl_df` (inherits from `tbl_df`, `tbl`, `data.frame`) with 5 rows and 4 columns.

distinct *Keep distinct/unique rows*

Description

Keep only unique/distinct rows from a data frame. This is similar to `unique.data.frame()` but considerably faster.

Value

An object of the same type as `.data`. The output has the following properties:

- Rows are a subset of the input but appear in the same order.
- Columns are not modified if `...` is empty or `.keep_all` is `TRUE`. Otherwise, `distinct()` first calls `mutate()` to create new columns.
- Groups are not modified.
- Data frame attributes are preserved.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

```
example(read10xVisium)
spe |>
   distinct(sample_id)
```

drop_class *Remove class to abject*

Description

Remove class to abject

Usage

```
drop_class(var, name)
```

Arguments

`var`	A tibble
`name`	A character name of the class

Value

A tibble with an additional attribute

`ellipse` *Ellipse Gating Function*

Description

Function to create an ellipse gate in a SpatialExperiment object

Usage

```
ellipse(spatial_coord1, spatial_coord2, center, axes_lengths)
```

Arguments

`spatial_coord1`	Numeric vector for x-coordinates
`spatial_coord2`	Numeric vector for y-coordinates
`center`	Numeric vector (length 2) for ellipse center (x, y)
`axes_lengths`	Numeric vector (length 2) for the lengths of the major and minor axes of the ellipse

Value

Logical vector indicating points within the ellipse

Examples

```
example(read10xVisium)
spe |>
    mutate(in_ellipse = ellipse(
        array_col, array_row, center = c(50, 50), axes_lengths = c(20, 10))
    )
```

extract *Extract a character column into multiple columns using regular expression groups*

Description

[Superseded]

`extract()` has been superseded in favour of `separate_wider_regex()` because it has a more polished API and better handling of problems. Superseded functions will not go away, but will only receive critical bug fixes.

Given a regular expression with capturing groups, `extract()` turns each group into a new column. If the groups don't match, or the input is NA, the output will be NA.

Usage

```
## S3 method for class 'SpatialExperiment'
extract(
  data,
  col,
  into,
  regex = "([[:alnum:]]+)",
  remove = TRUE,
  convert = FALSE,
  ...
)
```

Arguments

`data`	A data frame.
`col`	<`tidy-select`> Column to expand.
`into`	Names of new variables to create as character vector. Use `NA` to omit the variable in the output.
`regex`	A string representing a regular expression used to extract the desired values. There should be one group (defined by `()`) for each element of `into`.
`remove`	If `TRUE`, remove input column from output data frame.
`convert`	If `TRUE`, will run `type.convert()` with `as.is = TRUE` on new columns. This is useful if the component columns are integer, numeric or logical. NB: this will cause string `"NA"`s to be converted to `NA`s.
`...`	Additional arguments passed on to methods.

Value

`tidySpatialExperiment`

See Also

separate() to split up by a separator.

Examples

```
example(read10xVisium)
spe |>
    extract(col = array_row, into = "A", regex = "([[:digit:]]3)")
```

---

filter *Keep rows that match a condition*

Description

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

Usage

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

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.
.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.

Details

The filter() function is used to subset the rows of .data, applying the expressions in ... to the column values to determine which rows should be retained. It can be applied to both grouped and ungrouped data (see group_by() and ungroup()). However, dplyr is not yet smart enough to optimise the filtering operation on grouped datasets that do not need grouped calculations. For this reason, filtering is often considerably faster on ungrouped data.

Value

An object of the same type as `.data`. The output has the following properties:

- Rows are a subset of the input, but appear in the same order.
- Columns are not modified.
- The number of groups may be reduced (if `.preserve` is not `TRUE`).
- Data frame attributes are preserved.

Useful filter functions

There are many functions and operators that are useful when constructing the expressions used to filter the data:

- `==`, `>`, `>=` etc
- `&`, `|`, `!`, `xor()`
- `is.na()`
- `between()`, `near()`

Grouped tibbles

Because filtering expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped filtering:

```
starwars %>% filter(mass > mean(mass, na.rm = TRUE))
```

With the grouped equivalent:

```
starwars %>% group_by(gender) %>% filter(mass > mean(mass, na.rm = TRUE))
```

In the ungrouped version, `filter()` compares the value of `mass` in each row to the global average (taken over the whole data set), keeping only the rows with `mass` greater than this global average. In contrast, the grouped version calculates the average mass separately for each `gender` group, and keeps rows with `mass` greater than the relevant within-gender average.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `arrange()`, `mutate()`, `reframe()`, `rename()`, `select()`, `slice()`, `summarise()`

Examples

```
example(read10xVisium)
spe |>
    filter(in_tissue == TRUE)
```

`formatting` *Printing tibbles*

Description

One of the main features of the `tbl_df` class is the printing:

- Tibbles only print as many rows and columns as fit on one screen, supplemented by a summary of the remaining rows and columns.
- Tibble reveals the type of each column, which keeps the user informed about whether a variable is, e.g., `<chr>` or `<fct>` (character versus factor). See `vignette("types")` for an overview of common type abbreviations.

Printing can be tweaked for a one-off call by calling `print()` explicitly and setting arguments like `n` and `width`. More persistent control is available by setting the options described in pillar::pillar_options. See also `vignette("digits")` for a comparison to base options, and `vignette("numbers")` that showcases `num()` and `char()` for creating columns with custom formatting options.

As of tibble 3.1.0, printing is handled entirely by the **pillar** package. If you implement a package that extends tibble, the printed output can be customized in various ways. See `vignette("extending", package = "pillar")` for details, and pillar::pillar_options for options that control the display in the console.

Usage

```
## S3 method for class 'SpatialExperiment'
print(x, ..., n = NULL, width = NULL)
```

Arguments

`x`	Object to format or print.
`...`	These dots are for future extensions and must be empty.
`n`	Number of rows to show. If `NULL`, the default, will print all rows if less than the `print_max` option. Otherwise, will print as many rows as specified by the `print_min` option.
`width`	Width of text output to generate. This defaults to `NULL`, which means use the `width` option.

Value

Prints a message to the console describing the contents of the `tidySpatialExperiment`.

Examples

```
example(read10xVisium)
spe |>
    print()
```

---

gate *Interactively gate cells by spatial coordinates*

Description

Gate cells based on their X and Y coordinates. By default, this function launches an interactive scatter plot with image data overlaid. Colour, shape, size and alpha can be defined as constant values, or can be controlled by the values of a specified column.

If previously drawn gates are supplied to the `programmatic_gates` argument, cells will be gated programmatically. This feature allows the reproduction of previously drawn interactive gates. Programmatic gating is based on the package gatepoints by Wajid Jawaid.

Usage

```
gate(
  spe,
  image_index = 1,
  colour = NULL,
  shape = NULL,
  alpha = 1,
  size = 2,
  hide_points = FALSE,
  programmatic_gates = NULL
)
```

Arguments

`spe`	A SpatialExperiment object.
`image_index`	The image to display if multiple are stored within the provided SpatialExperiment object.
`colour`	A single colour string compatible with ggplot2. Or, a vector representing the point colour.
`shape`	A single ggplot2 shape numeric ranging from 0 to 127. Or, a vector representing the point shape, coercible to a factor of 6 or less levels.
`alpha`	A single ggplot2 alpha numeric ranging from 0 to 1.
`size`	A single ggplot2 size numeric ranging from 0 to 20.
`hide_points`	A logical. If TRUE, points are hidden during interactive gating. This can greatly improve performance with large SpatialExperiment objects.

programmatic_gates
: A data.frame of the gate brush data, as saved in tidygate_env$gates. The column x records X coordinates, the column y records Y coordinates and the column .gate records the gate number. When this argument is supplied, gates will be drawn programmatically.

Value

A vector of strings, of the gates each X and Y coordinate pair is within. If gates are drawn interactively, they are temporarily saved to tidygate_env$gates.

Examples

```
example(read10xVisium)
data(demo_brush_data, package = "tidySpatialExperiment")

# Gate points interactively
if(interactive()) {
    spe |>
        gate(colour = "blue", shape = "in_tissue")
}

# Gate points programmatically
spe |>
  gate(programmatic_gates = demo_brush_data)
```

---

gate_interactive *Gate interactive*

Description

Interactively gate points by their location in space, with image data overlaid.

Usage

```
gate_interactive(spe, image_index, colour, shape, alpha, size, hide_points)
```

Arguments

spe	A SpatialExperiment object.
image_index	The image to display if multiple are stored within the provided SpatialExperiment object.
colour	A single colour string compatible with ggplot2. Or, a vector representing the point colour.
shape	A single ggplot2 shape numeric ranging from 0 to 127. Or, a vector representing the point shape, coercible to a factor of 6 or less levels.
alpha	A single ggplot2 alpha numeric ranging from 0 to 1.

size	A single ggplot2 size numeric ranging from 0 to 20.
hide_points	A logical. If TRUE, points are hidden during interactive gating. This can greatly improve performance with large SpatialExperiment objects.

Value

The input SpatialExperiment object with a new column .gated, recording the gates each X and Y coordinate pair is within. If gates are drawn interactively, they are temporarily saved to tidygate_env$gates

Examples

```
example(read10xVisium)
data(demo_brush_data, package = "tidySpatialExperiment")

if(interactive()) {
    spe |>
        gate(colour = "blue", shape = "in_tissue")
}
```

---

gate_programmatic *Gate spatial data with pre-recorded lasso selection coordinates*

Description

A helpful way to repeat previous interactive lasso selections to enable reproducibility. Programmatic gating is based on the package gatepoints by Wajid Jawaid.

Usage

```
gate_programmatic(spe, programmatic_gates)
```

Arguments

spe	A SpatialExperiment object
programmatic_gates	A data.frame recording the gate brush data, as output by tidygate_env$gates. The column x records X coordinates, the column y records Y coordinates and the column .gated records the gate.

Value

The input SpatialExperiment object with a new column .gated, recording the gates each X and Y coordinate pair is within.

Examples

```
example(read10xVisium)
data(demo_brush_data, package = "tidySpatialExperiment")

spe |>
  gate(programmatic_gates = demo_brush_data)
```

`ggplot` *Create a new* `ggplot` *from a* `tidySpatialExperiment`

Description

`ggplot()` initializes a ggplot object. It can be used to declare the input data frame for a graphic and to specify the set of plot aesthetics intended to be common throughout all subsequent layers unless specifically overridden.

Details

`ggplot()` is used to construct the initial plot object, and is almost always followed by a plus sign (+) to add components to the plot.

There are three common patterns used to invoke `ggplot()`:

- `ggplot(data = df, mapping = aes(x, y, other aesthetics))`
- `ggplot(data = df)`
- `ggplot()`

The first pattern is recommended if all layers use the same data and the same set of aesthetics, although this method can also be used when adding a layer using data from another data frame.

The second pattern specifies the default data frame to use for the plot, but no aesthetics are defined up front. This is useful when one data frame is used predominantly for the plot, but the aesthetics vary from one layer to another.

The third pattern initializes a skeleton `ggplot` object, which is fleshed out as layers are added. This is useful when multiple data frames are used to produce different layers, as is often the case in complex graphics.

The `data =` and `mapping =` specifications in the arguments are optional (and are often omitted in practice), so long as the data and the mapping values are passed into the function in the right order. In the examples below, however, they are left in place for clarity.

Value

`ggplot`

See Also

The first steps chapter of the online ggplot2 book.

Examples

```
example(read10xVisium)
spe |>
    ggplot(ggplot2::aes(x = .cell, y = array_row)) +
    ggplot2::geom_point()
```

`glimpse` *Get a glimpse of your data*

Description

`glimpse()` is like a transposed version of `print()`: columns run down the page, and data runs across. This makes it possible to see every column in a data frame. It's a little like `str()` applied to a data frame but it tries to show you as much data as possible. (And it always shows the underlying data, even when applied to a remote data source.)

See `format_glimpse()` for details on the formatting.

Value

x original x is (invisibly) returned, allowing `glimpse()` to be used within a data pipe line.

S3 methods

`glimpse` is an S3 generic with a customised method for `tbls` and `data.frames`, and a default method that calls `str()`.

Examples

```
example(read10xVisium)
spe |>
    glimpse()
```

`group_by` *Group by one or more variables*

Description

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

Value

A grouped data frame with class `grouped_df`, unless the combination of `...` and add yields a empty set of grouping columns, in which case a tibble will be returned.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `group_by()`: no methods found.
- `ungroup()`: no methods found.

Ordering

Currently, `group_by()` internally orders the groups in ascending order. This results in ordered output from functions that aggregate groups, such as `summarise()`.

When used as grouping columns, character vectors are ordered in the C locale for performance and reproducibility across R sessions. If the resulting ordering of your grouped operation matters and is dependent on the locale, you should follow up the grouped operation with an explicit call to `arrange()` and set the `.locale` argument. For example:

```
data %>%
  group_by(chr) %>%
  summarise(avg = mean(x)) %>%
  arrange(chr, .locale = "en")
```

This is often useful as a preliminary step before generating content intended for humans, such as an HTML table.

Legacy behavior:

Prior to dplyr 1.1.0, character vector grouping columns were ordered in the system locale. If you need to temporarily revert to this behavior, you can set the global option `dplyr.legacy_locale` to `TRUE`, but this should be used sparingly and you should expect this option to be removed in a future version of dplyr. It is better to update existing code to explicitly call `arrange(.locale = )` instead. Note that setting `dplyr.legacy_locale` will also force calls to `arrange()` to use the system locale.

See Also

Other grouping functions: `group_map()`, `group_nest()`, `group_split()`, `group_trim()`

Examples

```
example(read10xVisium)
spe |>
    group_by(sample_id)
```

`inner_join` *Mutating joins*

Description

Mutating joins add columns from `y` to `x`, matching observations based on the keys. There are four mutating joins: the inner join, and the three outer joins.

Inner join:

An `inner_join()` only keeps observations from `x` that have a matching key in `y`.

The most important property of an inner join is that unmatched rows in either input are not included in the result. This means that generally inner joins are not appropriate in most analyses, because it is too easy to lose observations.

Outer joins:

The three outer joins keep observations that appear in at least one of the data frames:

- A `left_join()` keeps all observations in `x`.
- A `right_join()` keeps all observations in `y`.
- A `full_join()` keeps all observations in `x` and `y`.

Usage

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

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.
`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.
...	Other parameters passed onto methods.

Value

An object of the same type as x (including the same groups). The order of the rows and columns of x is preserved as much as possible. The output has the following properties:

- The rows are affect by the join type.
  - inner_join() returns matched x rows.
  - left_join() returns all x rows.
  - right_join() returns matched of x rows, followed by unmatched y rows.
  - full_join() returns all x rows, followed by unmatched y rows.
- Output columns include all columns from x and all non-key columns from y. If keep = TRUE, the key columns from y are included as well.
- If non-key columns in x and y have the same name, suffixes are added to disambiguate. If keep = TRUE and key columns in x and y have the same name, suffixes are added to disambiguate these as well.
- If keep = FALSE, output columns included in by are coerced to their common type between x and y.

Many-to-many relationships

By default, dplyr guards against many-to-many relationships in equality joins by throwing a warning. These occur when both of the following are true:

- A row in x matches multiple rows in y.
- A row in y matches multiple rows in x.

This is typically surprising, as most joins involve a relationship of one-to-one, one-to-many, or many-to-one, and is often the result of an improperly specified join. Many-to-many relationships are particularly problematic because they can result in a Cartesian explosion of the number of rows returned from the join.

If a many-to-many relationship is expected, silence this warning by explicitly setting relationship = "many-to-many".

In production code, it is best to preemptively set relationship to whatever relationship you expect to exist between the keys of x and y, as this forces an error to occur immediately if the data doesn't align with your expectations.

Inequality joins typically result in many-to-many relationships by nature, so they don't warn on them by default, but you should still take extra care when specifying an inequality join, because they also have the capability to return a large number of rows.

Rolling joins don't warn on many-to-many relationships either, but many rolling joins follow a many-to-one relationship, so it is often useful to set `relationship = "many-to-one"` to enforce this.

Note that in SQL, most database providers won't let you specify a many-to-many relationship between two tables, instead requiring that you create a third *junction table* that results in two one-to-many relationships instead.

Methods

These functions are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `inner_join()`: no methods found.
- `left_join()`: no methods found.
- `right_join()`: no methods found.
- `full_join()`: no methods found.

See Also

Other joins: `cross_join()`, `filter-joins`, `nest_join()`

Examples

```
example(read10xVisium)
spe |>
    inner_join(
        spe |>
            filter(in_tissue == TRUE) |>
            mutate(new_column = 1)
        )
```

`join_features` *Extract and join information for features.*

Description

join_features() extracts and joins information for specified features

Arguments

.data	A SpatialExperiment object
features	A vector of feature identifiers to join
all	If TRUE return all
exclude_zeros	If TRUE exclude zero values
shape	Format of the returned table "long" or "wide"
...	Parameters to pass to join wide, i.e. assay name to extract feature abundance from and gene prefix, for shape="wide"

Details

This function extracts information for specified features and returns the information in either long or wide format.

Value

An object containing the information.for the specified features

Examples

```
example(read10xVisium)
spe |>
    join_features(features = "ENSMUSG00000025900")
```

---

left_join *Mutating joins*

Description

Mutating joins add columns from `y` to `x`, matching observations based on the keys. There are four mutating joins: the inner join, and the three outer joins.

Inner join:

An `inner_join()` only keeps observations from `x` that have a matching key in `y`.

The most important property of an inner join is that unmatched rows in either input are not included in the result. This means that generally inner joins are not appropriate in most analyses, because it is too easy to lose observations.

Outer joins:

The three outer joins keep observations that appear in at least one of the data frames:

- A `left_join()` keeps all observations in `x`.
- A `right_join()` keeps all observations in `y`.
- A `full_join()` keeps all observations in `x` and `y`.

Usage

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

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.
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.
...	Other parameters passed onto methods.

Value

An object of the same type as x (including the same groups). The order of the rows and columns of x is preserved as much as possible. The output has the following properties:

- The rows are affect by the join type.
  - inner_join() returns matched x rows.
  - left_join() returns all x rows.
  - right_join() returns matched of x rows, followed by unmatched y rows.
  - full_join() returns all x rows, followed by unmatched y rows.
- Output columns include all columns from x and all non-key columns from y. If keep = TRUE, the key columns from y are included as well.

- If non-key columns in `x` and `y` have the same name, `suffixes` are added to disambiguate. If `keep = TRUE` and key columns in `x` and `y` have the same name, `suffixes` are added to disambiguate these as well.
- If `keep = FALSE`, output columns included in `by` are coerced to their common type between `x` and `y`.

Many-to-many relationships

By default, dplyr guards against many-to-many relationships in equality joins by throwing a warning. These occur when both of the following are true:

- A row in `x` matches multiple rows in `y`.
- A row in `y` matches multiple rows in `x`.

This is typically surprising, as most joins involve a relationship of one-to-one, one-to-many, or many-to-one, and is often the result of an improperly specified join. Many-to-many relationships are particularly problematic because they can result in a Cartesian explosion of the number of rows returned from the join.

If a many-to-many relationship is expected, silence this warning by explicitly setting `relationship = "many-to-many"`.

In production code, it is best to preemptively set `relationship` to whatever relationship you expect to exist between the keys of `x` and `y`, as this forces an error to occur immediately if the data doesn't align with your expectations.

Inequality joins typically result in many-to-many relationships by nature, so they don't warn on them by default, but you should still take extra care when specifying an inequality join, because they also have the capability to return a large number of rows.

Rolling joins don't warn on many-to-many relationships either, but many rolling joins follow a many-to-one relationship, so it is often useful to set `relationship = "many-to-one"` to enforce this.

Note that in SQL, most database providers won't let you specify a many-to-many relationship between two tables, instead requiring that you create a third *junction table* that results in two one-to-many relationships instead.

Methods

These functions are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `inner_join()`: no methods found.
- `left_join()`: no methods found.
- `right_join()`: no methods found.
- `full_join()`: no methods found.

See Also

Other joins: `cross_join()`, `filter-joins`, `nest_join()`

Examples

```
example(read10xVisium)
spe |>
    left_join(
        spe |>
            filter(in_tissue == TRUE) |>
            mutate(new_column = 1)
        )
```

`mutate` *Create, modify, and delete columns*

Description

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

Usage

```
## S3 method for class 'SpatialExperiment'
mutate(.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.
`...`	<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.

Value

An object of the same type as `.data`. The output has the following properties:

- Columns from `.data` will be preserved according to the `.keep` argument.
- Existing columns that are modified by `...` will always be returned in their original location.
- New columns created through `...` will be placed according to the `.before` and `.after` arguments.
- The number of rows is not affected.
- Columns given the value `NULL` will be removed.
- Groups will be recomputed if a grouping variable is mutated.
- Data frame attributes are preserved.

Useful mutate functions

- +, -, log(), etc., for their usual mathematical meanings
- lead(), lag()
- dense_rank(), min_rank(), percent_rank(), row_number(), cume_dist(), ntile()
- cumsum(), cummean(), cummin(), cummax(), cumany(), cumall()
- na_if(), coalesce()
- if_else(), recode(), case_when()

Grouped tibbles

Because mutating expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped mutate:

```
starwars %>%
  select(name, mass, species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))
```

With the grouped equivalent:

```
starwars %>%
  select(name, mass, species) %>%
  group_by(species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))
```

The former normalises `mass` by the global average whereas the latter normalises by the averages within species levels.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages: no methods found.

See Also

Other single table verbs: arrange(), rename(), slice(), summarise()

Examples

```
example(read10xVisium)
spe |>
    mutate(array_col = 1)
```

nest *Nest rows into a list-column of data frames*

Description

Nesting creates a list-column of data frames; unnesting flattens it back out into regular columns. Nesting is implicitly a summarising operation: you get one row for each group defined by the non-nested columns. This is useful in conjunction with other summaries that work with whole datasets, most notably models.

Learn more in `vignette("nest")`.

Usage

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

Arguments

`.data`	A data frame.
`...`	<tidy-select> Columns to nest; these will appear in the inner data frames. Specified using name-variable pairs of the form `new_col = c(col1, col2, col3)`. The right hand side can be any valid tidyselect expression. If not supplied, then `...` is derived as all columns *not* selected by `.by`, and will use the column name from `.key`. **[Deprecated]**: previously you could write `df %>% nest(x, y, z)`. Convert to `df %>% nest(data = c(x, y, z))`.
`.names_sep`	If `NULL`, the default, the inner names will come from the former outer names. If a string, the new inner names will use the outer names with `names_sep` automatically stripped. This makes `names_sep` roughly symmetric between nesting and unnesting.

Details

If neither `...` nor `.by` are supplied, `nest()` will nest all variables, and will use the column name supplied through `.key`.

Value

`tidySpatialExperiment_nested`

New syntax

tidyr 1.0.0 introduced a new syntax for `nest()` and `unnest()` that's designed to be more similar to other functions. Converting to the new syntax should be straightforward (guided by the message you'll receive) but if you just need to run an old analysis, you can easily revert to the previous behaviour using `nest_legacy()` and `unnest_legacy()` as follows:

```
library(tidyr)
nest <- nest_legacy
unnest <- unnest_legacy
```

Grouped data frames

`df %>% nest(data = c(x, y))` specifies the columns to be nested; i.e. the columns that will appear in the inner data frame. `df %>% nest(.by = c(x, y))` specifies the columns to nest *by*; i.e. the columns that will remain in the outer data frame. An alternative way to achieve the latter is to `nest()` a grouped data frame created by `dplyr::group_by()`. The grouping variables remain in the outer data frame and the others are nested. The result preserves the grouping of the input.

Variables supplied to `nest()` will override grouping variables so that `df %>% group_by(x, y) %>% nest(data = !z)` will be equivalent to `df %>% nest(data = !z)`.

You can't supply `.by` with a grouped data frame, as the groups already represent what you are nesting by.

Examples

```
example(read10xVisium)
spe |>
    nest(data = -sample_id)
```

`pivot_longer` *Pivot data from wide to long*

Description

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `pivot_wider()`

Learn more in `vignette("pivot")`.

Details

`pivot_longer()` is an updated approach to `gather()`, designed to be both simpler to use and to handle more use cases. We recommend you use `pivot_longer()` for new code; `gather()` isn't going away but is no longer under active development.

Value

`tidySingleCellExperiment`

Examples

```
example(read10xVisium)
spe |>
   pivot_longer(c(array_row, array_col), names_to = "dimension", values_to = "location")
```

plot_ly *Initiate a plotly visualization*

Description

This function maps R objects to plotly.js, an (MIT licensed) web-based interactive charting library. It provides abstractions for doing common things (e.g. mapping data values to fill colors (via `color`) or creating animations (via `frame`)) and sets some different defaults to make the interface feel more 'R-like' (i.e., closer to `plot()` and `ggplot2::qplot()`).

Usage

```
## S3 method for class 'SpatialExperiment'
plot_ly(
  data = data.frame(),
  ...,
  type = NULL,
  name = NULL,
  color = NULL,
  colors = NULL,
  alpha = NULL,
  stroke = NULL,
  strokes = NULL,
  alpha_stroke = 1,
  size = NULL,
  sizes = c(10, 100),
  span = NULL,
  spans = c(1, 20),
  symbol = NULL,
  symbols = NULL,
  linetype = NULL,
  linetypes = NULL,
  split = NULL,
  frame = NULL,
  width = NULL,
  height = NULL,
  source = "A"
)
```

Arguments

`data`	A data frame (optional) or crosstalk::SharedData object.
`...`	Arguments (i.e., attributes) passed along to the trace `type`. See `schema()` for a list of acceptable attributes for a given trace `type` (by going to `traces -> type -> attributes`). Note that attributes provided at this level may override other arguments (e.g. `plot_ly(x = 1:10, y = 1:10, color = I("red"), marker = list(color = "blue"))`).

type	A character string specifying the trace type (e.g. "scatter", "bar", "box", etc). If specified, it *always* creates a trace, otherwise
name	Values mapped to the trace's name attribute. Since a trace can only have one name, this argument acts very much like split in that it creates one trace for every unique value.
color	Values mapped to relevant 'fill-color' attribute(s) (e.g. fillcolor, marker.color, textfont.color, etc.). The mapping from data values to color codes may be controlled using colors and alpha, or avoided altogether via I() (e.g., color = I("red")). Any color understood by grDevices::col2rgb() may be used in this way.
colors	Either a colorbrewer2.org palette name (e.g. "YlOrRd" or "Blues"), or a vector of colors to interpolate in hexadecimal "#RRGGBB" format, or a color interpolation function like colorRamp().
alpha	A number between 0 and 1 specifying the alpha channel applied to color. Defaults to 0.5 when mapping to fillcolor and 1 otherwise.
stroke	Similar to color, but values are mapped to relevant 'stroke-color' attribute(s) (e.g., marker.line.color and line.color for filled polygons). If not specified, stroke inherits from color.
strokes	Similar to colors, but controls the stroke mapping.
alpha_stroke	Similar to alpha, but applied to stroke.
size	(Numeric) values mapped to relevant 'fill-size' attribute(s) (e.g., marker.size, textfont.size, and error_x.width). The mapping from data values to symbols may be controlled using sizes, or avoided altogether via I() (e.g., size = I(30)).
sizes	A numeric vector of length 2 used to scale size to pixels.
span	(Numeric) values mapped to relevant 'stroke-size' attribute(s) (e.g., marker.line.width, line.width for filled polygons, and error_x.thickness) The mapping from data values to symbols may be controlled using spans, or avoided altogether via I() (e.g., span = I(30)).
spans	A numeric vector of length 2 used to scale span to pixels.
symbol	(Discrete) values mapped to marker.symbol. The mapping from data values to symbols may be controlled using symbols, or avoided altogether via I() (e.g., symbol = I("pentagon")). Any pch value or symbol name may be used in this way.
symbols	A character vector of pch values or symbol names.
linetype	(Discrete) values mapped to line.dash. The mapping from data values to symbols may be controlled using linetypes, or avoided altogether via I() (e.g., linetype = I("dash")). Any lty (see par) value or dash name may be used in this way.
linetypes	A character vector of lty values or dash names
split	(Discrete) values used to create multiple traces (one trace per value).
frame	(Discrete) values used to create animation frames.
width	Width in pixels (optional, defaults to automatic sizing).
height	Height in pixels (optional, defaults to automatic sizing).

source — a character string of length 1. Match the value of this string with the source argument in event_data() to retrieve the event data corresponding to a specific plot (shiny apps can have multiple plots).

Details

Unless type is specified, this function just initiates a plotly object with 'global' attributes that are passed onto downstream uses of add_trace() (or similar). A formula must always be used when referencing column name(s) in data (e.g. plot_ly(mtcars, x = ~wt)). Formulas are optional when supplying values directly, but they do help inform default axis/scale titles (e.g., plot_ly(x = mtcars$wt) vs plot_ly(x = ~mtcars$wt))

Value

plotly

Author(s)

Carson Sievert

References

https://plotly-r.com/overview.html

See Also

- For initializing a plotly-geo object: plot_geo()
- For initializing a plotly-mapbox object: plot_mapbox()
- For translating a ggplot2 object to a plotly object: ggplotly()
- For modifying any plotly object: layout(), add_trace(), style()
- For linked brushing: highlight()
- For arranging multiple plots: subplot(), crosstalk::bscols()
- For inspecting plotly objects: plotly_json()
- For quick, accurate, and searchable plotly.js reference: schema()

Examples

```
example(read10xVisium)
spe |>
    plot_ly(x = ~ array_col, y = ~ array_row)
```

pull *Extract a single column*

Description

`pull()` is similar to `$`. It's mostly useful because it looks a little nicer in pipes, it also works with remote data frames, and it can optionally name the output.

Value

A vector the same size as `.data`.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

```
example(read10xVisium)
spe |>
    pull(in_tissue)
```

quo_names *Convert array of quosure (e.g. c(col_a, col_b)) into character vector*

Description

Convert array of quosure (e.g. c(col_a, col_b)) into character vector

Usage

```
quo_names(v)
```

Arguments

v	A array of quosures (e.g. c(col_a, col_b))

Value

A character vector

rectangle *Rectangle Gating Function*

Description

Determines whether points specified by spatial coordinates are within a defined rectangle.

Usage

```
rectangle(spatial_coord1, spatial_coord2, center, height, width)
```

Arguments

spatial_coord1	Numeric vector for x-coordinates (e.g., array_col)
spatial_coord2	Numeric vector for y-coordinates (e.g., array_row)
center	Numeric vector of length 2 specifying the center of the rectangle (x, y)
height	The height of the rectangle
width	The width of the rectangle

Value

Logical vector indicating points within the rectangle

Examples

```
example(read10xVisium)
spe |>
    mutate(in_rectangle = rectangle(
      array_col, array_row, center = c(50, 50), height = 20, width = 10)
      )
```

rename *Rename columns*

Description

`rename()` changes the names of individual variables using `new_name = old_name` syntax; `rename_with()` renames columns using a function.

Value

An object of the same type as `.data`. The output has the following properties:

- Rows are not affected.
- Column names are changed; column order is preserved.
- Data frame attributes are preserved.
- Groups are updated to reflect new names.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `arrange()`, `mutate()`, `slice()`, `summarise()`

Examples

```
example(read10xVisium)
spe |>
    rename(in_liver = in_tissue)
```

---

right_join *Mutating joins*

Description

Mutating joins add columns from `y` to `x`, matching observations based on the keys. There are four mutating joins: the inner join, and the three outer joins.

Inner join:

An `inner_join()` only keeps observations from `x` that have a matching key in `y`.

The most important property of an inner join is that unmatched rows in either input are not included in the result. This means that generally inner joins are not appropriate in most analyses, because it is too easy to lose observations.

Outer joins:

The three outer joins keep observations that appear in at least one of the data frames:

- A `left_join()` keeps all observations in `x`.
- A `right_join()` keeps all observations in `y`.
- A `full_join()` keeps all observations in `x` and `y`.

Usage

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

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.
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.
...	Other parameters passed onto methods.

Value

An object of the same type as `x` (including the same groups). The order of the rows and columns of `x` is preserved as much as possible. The output has the following properties:

- The rows are affect by the join type.
  - `inner_join()` returns matched `x` rows.
  - `left_join()` returns all `x` rows.
  - `right_join()` returns matched of `x` rows, followed by unmatched `y` rows.
  - `full_join()` returns all `x` rows, followed by unmatched `y` rows.
- Output columns include all columns from `x` and all non-key columns from `y`. If `keep = TRUE`, the key columns from `y` are included as well.
- If non-key columns in `x` and `y` have the same name, `suffixes` are added to disambiguate. If `keep = TRUE` and key columns in `x` and `y` have the same name, `suffixes` are added to disambiguate these as well.
- If `keep = FALSE`, output columns included in `by` are coerced to their common type between `x` and `y`.

Many-to-many relationships

By default, dplyr guards against many-to-many relationships in equality joins by throwing a warning. These occur when both of the following are true:

- A row in `x` matches multiple rows in `y`.
- A row in `y` matches multiple rows in `x`.

This is typically surprising, as most joins involve a relationship of one-to-one, one-to-many, or many-to-one, and is often the result of an improperly specified join. Many-to-many relationships are particularly problematic because they can result in a Cartesian explosion of the number of rows returned from the join.

If a many-to-many relationship is expected, silence this warning by explicitly setting `relationship = "many-to-many"`.

In production code, it is best to preemptively set `relationship` to whatever relationship you expect to exist between the keys of `x` and `y`, as this forces an error to occur immediately if the data doesn't align with your expectations.

Inequality joins typically result in many-to-many relationships by nature, so they don't warn on them by default, but you should still take extra care when specifying an inequality join, because they also have the capability to return a large number of rows.

Rolling joins don't warn on many-to-many relationships either, but many rolling joins follow a many-to-one relationship, so it is often useful to set `relationship = "many-to-one"` to enforce this.

Note that in SQL, most database providers won't let you specify a many-to-many relationship between two tables, instead requiring that you create a third *junction table* that results in two one-to-many relationships instead.

Methods

These functions are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `inner_join()`: no methods found.
- `left_join()`: no methods found.
- `right_join()`: no methods found.
- `full_join()`: no methods found.

See Also

Other joins: `cross_join()`, `filter-joins`, `nest_join()`

Examples

```
example(read10xVisium)

spe |>
    right_join(
        spe |>
            filter(in_tissue == TRUE) |>
            mutate(new_column = 1)
        )
```

`rowwise` *Group input by rows*

Description

`rowwise()` allows you to compute on a data frame a row-at-a-time. This is most useful when a vectorised function doesn't exist.

Most dplyr verbs preserve row-wise grouping. The exception is `summarise()`, which return a grouped_df. You can explicitly ungroup with `ungroup()` or `as_tibble()`, or convert to a grouped_df with `group_by()`.

Value

A row-wise data frame with class `rowwise_df`. Note that a `rowwise_df` is implicitly grouped by row, but is not a `grouped_df`.

List-columns

Because a rowwise has exactly one row per group it offers a small convenience for working with list-columns. Normally, `summarise()` and `mutate()` extract a groups worth of data with `[`. But when you index a list in this way, you get back another list. When you're working with a `rowwise` tibble, then dplyr will use `[[` instead of `[` to make your life a little easier.

See Also

`nest_by()` for a convenient way of creating rowwise data frames with nested data.

Examples

```
example(read10xVisium)
spe |>
    rowwise()
```

sample_n *Sample n rows from a table*

Description

[Superseded] `sample_n()` and `sample_frac()` have been superseded in favour of `slice_sample()`. While they will not be deprecated in the near future, retirement means that we will only perform critical bug fixes, so we recommend moving to the newer alternative.

These functions were superseded because we realised it was more convenient to have two mutually exclusive arguments to one function, rather than two separate functions. This also made it to clean up a few other smaller design issues with `sample_n()`/`sample_frac`:

- The connection to `slice()` was not obvious.
- The name of the first argument, `tbl`, is inconsistent with other single table verbs which use `.data`.
- The `size` argument uses tidy evaluation, which is surprising and undocumented.
- It was easier to remove the deprecated `.env` argument.
- `...` was in a suboptimal position.

Usage

```
## S3 method for class 'SpatialExperiment'
sample_n(tbl, size, replace = FALSE, weight = NULL, .env = NULL, ...)

## S3 method for class 'SpatialExperiment'
sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = NULL, ...)
```

Arguments

`tbl`	A data.frame.
`size`	<tidy-select> For `sample_n()`, the number of rows to select. For `sample_frac()`, the fraction of rows to select. If `tbl` is grouped, `size` applies to each group.
`replace`	Sample with or without replacement?
`weight`	<tidy-select> 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.
`.env`	DEPRECATED.
`...`	ignored

Value

`tidySpatialExperiment`

Examples

```
example(read10xVisium)
spe |>
    sample_n(10)
spe |>
    sample_frac(0.1)
```

`select` *Keep or drop columns using their names and types*

Description

Select (and optionally rename) variables in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. `a:f` selects all columns from `a` on the left to `f` on the right) or type (e.g. `where(is.numeric)` selects all numeric columns).

Overview of selection features:

Tidyverse selections implement a dialect of R where operators make it easy to select variables:

- `:` for selecting a range of consecutive variables.
- `!` for taking the complement of a set of variables.
- `&` and `|` for selecting the intersection or the union of two sets of variables.
- `c()` for combining selections.

In addition, you can use **selection helpers**. Some helpers select specific columns:

- `everything()`: Matches all variables.
- `last_col()`: Select last variable, possibly with an offset.
- `group_cols()`: Select all grouping columns.

Other helpers select variables by matching patterns in their names:

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like x01, x02, x03.

Or from variables stored in a character vector:

- `all_of()`: Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of()`: Same as `all_of()`, except that no error is thrown for names that don't exist.

Or using a predicate function:

- `where()`: Applies a function to all variables and selects those for which the function returns `TRUE`.

Usage

```
## S3 method for class 'SpatialExperiment'
select(.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.
`...`	<tidy-select> One or more unquoted expressions separated by commas. Variable names can be used as if they were positions in the data frame, so expressions like `x:y` can be used to select a range of variables.

Value

An object of the same type as `.data`. The output has the following properties:

- Rows are not affected.
- Output columns are a subset of input columns, potentially with a different order. Columns will be renamed if `new_name = old_name` form is used.
- Data frame attributes are preserved.
- Groups are maintained; you can't select off grouping variables.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

Here we show the usage for the basic selection operators. See the specific help pages to learn about helpers like `starts_with()`.

The selection language can be used in functions like `dplyr::select()` or `tidyr::pivot_longer()`. Let's first attach the tidyverse:

```
library(tidyverse)

# For better printing
iris <- as_tibble(iris)
```

Select variables by name:

```
starwars %>% select(height)
#> # A tibble: 87 x 1
#>   height
#>    <int>
```

```
#> 1    172
#> 2    167
#> 3     96
#> 4    202
#> # i 83 more rows

iris %>% pivot_longer(Sepal.Length)
#> # A tibble: 150 x 6
#>   Sepal.Width Petal.Length Petal.Width Species name         value
#>         <dbl>        <dbl>       <dbl> <fct>   <chr>        <dbl>
#> 1         3.5          1.4         0.2 setosa  Sepal.Length   5.1
#> 2         3            1.4         0.2 setosa  Sepal.Length   4.9
#> 3         3.2          1.3         0.2 setosa  Sepal.Length   4.7
#> 4         3.1          1.5         0.2 setosa  Sepal.Length   4.6
#> # i 146 more rows
```

Select multiple variables by separating them with commas. Note how the order of columns is determined by the order of inputs:

```
starwars %>% select(homeworld, height, mass)
#> # A tibble: 87 x 3
#>   homeworld height  mass
#>   <chr>      <int> <dbl>
#> 1 Tatooine     172    77
#> 2 Tatooine     167    75
#> 3 Naboo         96    32
#> 4 Tatooine     202   136
#> # i 83 more rows
```

Functions like `tidyr::pivot_longer()` don't take variables with dots. In this case use `c()` to select multiple variables:

```
iris %>% pivot_longer(c(Sepal.Length, Petal.Length))
#> # A tibble: 300 x 5
#>   Sepal.Width Petal.Width Species name         value
#>         <dbl>       <dbl> <fct>   <chr>        <dbl>
#> 1         3.5         0.2 setosa  Sepal.Length   5.1
#> 2         3.5         0.2 setosa  Petal.Length   1.4
#> 3         3           0.2 setosa  Sepal.Length   4.9
#> 4         3           0.2 setosa  Petal.Length   1.4
#> # i 296 more rows
```

Operators::

The : operator selects a range of consecutive variables:

```
starwars %>% select(name:mass)
#> # A tibble: 87 x 3
#>   name           height  mass
#>   <chr>           <int> <dbl>
```

```
#> 1 Luke Skywalker    172    77
#> 2 C-3PO             167    75
#> 3 R2-D2              96    32
#> 4 Darth Vader       202   136
#> # i 83 more rows
```

The ! operator negates a selection:

```
starwars %>% select(!(name:mass))
#> # A tibble: 87 x 11
#>   hair_color skin_color  eye_color birth_year sex   gender   homeworld species
#>   <chr>      <chr>       <chr>          <dbl> <chr> <chr>    <chr>     <chr>
#> 1 blond      fair        blue            19   male  masculine Tatooine  Human
#> 2 <NA>       gold        yellow         112   none  masculine Tatooine  Droid
#> 3 <NA>       white, blue red             33   none  masculine Naboo     Droid
#> 4 none       white       yellow          41.9 male  masculine Tatooine  Human
#> # i 83 more rows
#> # i 3 more variables: films <list>, vehicles <list>, starships <list>

iris %>% select(!c(Sepal.Length, Petal.Length))
#> # A tibble: 150 x 3
#>   Sepal.Width Petal.Width Species
#>         <dbl>       <dbl> <fct>
#> 1         3.5         0.2 setosa
#> 2         3           0.2 setosa
#> 3         3.2         0.2 setosa
#> 4         3.1         0.2 setosa
#> # i 146 more rows

iris %>% select(!ends_with("Width"))
#> # A tibble: 150 x 3
#>   Sepal.Length Petal.Length Species
#>          <dbl>        <dbl> <fct>
#> 1          5.1          1.4 setosa
#> 2          4.9          1.4 setosa
#> 3          4.7          1.3 setosa
#> 4          4.6          1.5 setosa
#> # i 146 more rows
```

& and | take the intersection or the union of two selections:

```
iris %>% select(starts_with("Petal") & ends_with("Width"))
#> # A tibble: 150 x 1
#>   Petal.Width
#>         <dbl>
#> 1         0.2
#> 2         0.2
#> 3         0.2
#> 4         0.2
#> # i 146 more rows
```

```
iris %>% select(starts_with("Petal") | ends_with("Width"))
#> # A tibble: 150 x 3
#>   Petal.Length Petal.Width Sepal.Width
#>          <dbl>       <dbl>       <dbl>
#> 1          1.4         0.2         3.5
#> 2          1.4         0.2         3
#> 3          1.3         0.2         3.2
#> 4          1.5         0.2         3.1
#> # i 146 more rows
```

To take the difference between two selections, combine the & and ! operators:

```
iris %>% select(starts_with("Petal") & !ends_with("Width"))
#> # A tibble: 150 x 1
#>   Petal.Length
#>          <dbl>
#> 1          1.4
#> 2          1.4
#> 3          1.3
#> 4          1.5
#> # i 146 more rows
```

See Also

Other single table verbs: `arrange()`, `filter()`, `mutate()`, `reframe()`, `rename()`, `slice()`, `summarise()`

Examples

```
example(read10xVisium)
spe |>
    select(in_tissue)
```

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

Description

[Superseded]

`separate()` has been superseded in favour of `separate_wider_position()` and `separate_wider_delim()` because the two functions make the two uses more obvious, the API is more polished, and the handling of problems is better. Superseded functions will not go away, but will only receive critical bug fixes.

Given either a regular expression or a vector of character positions, `separate()` turns a single character column into multiple columns.

Usage

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

Arguments

data	A data frame.
col	<tidy-select> Column to expand.
into	Names of new variables to create as character vector. Use NA to omit the variable in the output.
sep	Separator between columns. If character, sep is interpreted as a regular expression. The default value is a regular expression that matches any sequence of non-alphanumeric values. If numeric, sep is interpreted as character positions to split at. Positive values start at 1 at the far-left of the string; negative value start at -1 at the far-right of the string. The length of sep should be one less than into.
remove	If TRUE, remove input column from output data frame.
convert	If TRUE, will run type.convert() with as.is = TRUE on new columns. This is useful if the component columns are integer, numeric or logical. NB: this will cause string "NA"s to be converted to NAs.
extra	If sep is a character vector, this controls what happens when there are too many pieces. There are three valid options: • "warn" (the default): emit a warning and drop extra values. • "drop": drop any extra values without a warning. • "merge": only splits at most length(into) times
fill	If sep is a character vector, this controls what happens when there are not enough pieces. There are three valid options: • "warn" (the default): emit a warning and fill from the right • "right": fill with missing values on the right • "left": fill with missing values on the left
...	Additional arguments passed on to methods.

Value

tidySpatialExperiment

See Also

unite(), the complement, extract() which uses regular expression capturing groups.

Examples

```
example(read10xVisium)
spe |>
    separate(col = sample_id, into = c("A", "B"), sep = "[[:alnum:]]n")
```

---

slice *Subset rows using their positions*

Description

slice() lets you index rows by their (integer) locations. It allows you to select, remove, and duplicate rows. It is accompanied by a number of helpers for common use cases:

- slice_head() and slice_tail() select the first or last rows.
- slice_sample() randomly selects rows.
- slice_min() and slice_max() select rows with the smallest or largest values of a variable.

If .data is a grouped_df, the operation will be performed on each group, so that (e.g.) slice_head(df, n = 5) will select the first five rows in each group.

Details

Slice does not work with relational databases because they have no intrinsic notion of row order. If you want to perform the equivalent operation, use filter() and row_number().

Value

An object of the same type as .data. The output has the following properties:

- Each row may appear 0, 1, or many times in the output.
- Columns are not modified.
- Groups are not modified.
- Data frame attributes are preserved.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- slice(): no methods found.

- `slice_head()`: no methods found.
- `slice_tail()`: no methods found.
- `slice_min()`: no methods found.
- `slice_max()`: no methods found.
- `slice_sample()`: no methods found.

See Also

Other single table verbs: `arrange()`, `mutate()`, `rename()`, `summarise()`

Examples

```
example(read10xVisium)
spe |>
   slice(1)
```

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

Description

`summarise()` creates a new data frame. It returns one row for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

`summarise()` and `summarize()` are synonyms.

Value

An object *usually* of the same type as `.data`.

- The rows come from the underlying `group_keys()`.
- The columns are a combination of the grouping keys and the summary expressions that you provide.
- The grouping structure is controlled by the `.groups=` argument, the output may be another grouped_df, a tibble or a rowwise data frame.
- Data frame attributes are **not** preserved, because `summarise()` fundamentally creates a new data frame.

Useful functions

- Center: `mean()`, `median()`
- Spread: `sd()`, `IQR()`, `mad()`
- Range: `min()`, `max()`,
- Position: `first()`, `last()`, `nth()`,
- Count: `n()`, `n_distinct()`
- Logical: `any()`, `all()`

Backend variations

The data frame backend supports creating a variable and using it in the same summary. This means that previously created summary variables can be further transformed or combined within the summary, as in `mutate()`. However, it also means that summary variables with the same names as previous variables overwrite them, making those variables unavailable to later summary variables.

This behaviour may not be supported in other backends. To avoid unexpected results, consider using new names for your summary variables, especially when creating multiple summaries.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `arrange()`, `mutate()`, `rename()`, `slice()`

Examples

```
example(read10xVisium)
spe |>
    summarise(mean(array_row))
```

`tbl_format_header` *Format the header of a tibble*

Description

[Experimental]

For easier customization, the formatting of a tibble is split into three components: header, body, and footer. The `tbl_format_header()` method is responsible for formatting the header of a tibble.

Override this method if you need to change the appearance of the entire header. If you only need to change or extend the components shown in the header, override or extend `tbl_sum()` for your class which is called by the default method.

Usage

```
## S3 method for class 'tidySpatialExperiment'
tbl_format_header(x, setup, ...)
```

Arguments

`x`	A tibble-like object.
`setup`	A setup object returned from `tbl_format_setup()`.
`...`	These dots are for future extensions and must be empty.

Value

A character vector.

Examples

```
# TODO
```

`unite` *Unite multiple columns into one by pasting strings together*

Description

Convenience function to paste together multiple columns into one.

Usage

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

Arguments

`data`	A data frame.
`col`	The name of the new column, as a string or symbol. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::ensym()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
`...`	<`tidy-select`> Columns to unite
`sep`	Separator to use between values.
`remove`	If `TRUE`, remove input columns from output data frame.
`na.rm`	If `TRUE`, missing values will be removed prior to uniting each value.

Value

`tidySpatialExperiment`

See Also

`separate()`, the complement.

Examples

```
example(read10xVisium)
spe |>
    unite("A", array_row:array_col)
```

unnest *Unnest a list-column of data frames into rows and columns*

Description

Unnest expands a list-column containing data frames into rows and columns.

Usage

```
## S3 method for class 'tidySpatialExperiment_nested'
unnest(
  data,
  cols,
  ...,
  keep_empty = FALSE,
  ptype = NULL,
  names_sep = NULL,
  names_repair = "check_unique",
  .drop,
  .id,
  .sep,
  .preserve
)
```

Arguments

data	A data frame.
cols	<tidy-select> List-columns to unnest. When selecting multiple columns, values from the same row will be recycled to their common size.
...	**[Deprecated]**: previously you could write `df %>% unnest(x, y, z)`. Convert to `df %>% unnest(c(x, y, z))`. If you previously created a new variable in `unnest()` you'll now need to do it explicitly with `mutate()`. Convert `df %>% unnest(y = fun(x, y, z))` to `df %>% mutate(y = fun(x, y, z)) %>% unnest(y)`.
keep_empty	By default, you get one row of output for each element of the list that you are unchopping/unnesting. This means that if there's a size-0 element (like `NULL` or an empty data frame or vector), then that entire row will be dropped from the output. If you want to preserve all rows, use `keep_empty = TRUE` to replace size-0 elements with a single row of missing values.
ptype	Optionally, a named list of column name-prototype pairs to coerce `cols` to, overriding the default that will be guessed from combining the individual values. Alternatively, a single empty ptype can be supplied, which will be applied to all `cols`.
names_sep	If `NULL`, the default, the outer names will come from the inner names. If a string, the outer names will be formed by pasting together the outer and the inner column names, separated by `names_sep`.

`names_repair` Used to check that output data frame has valid names. Must be one of the following options:

- `"minimal"`: no name repair or checks, beyond basic existence,
- `"unique"`: make sure names are unique and not empty,
- `"check_unique"`: (the default), no name repair, but check they are unique,
- `"universal"`: make the names unique and syntactic
- a function: apply custom name repair.
- tidyr_legacy: use the name repair from tidyr 0.8.
- a formula: a purrr-style anonymous function (see `rlang::as_function()`)

See `vctrs::vec_as_names()` for more details on these terms and the strategies used to enforce them.

`.drop, .preserve`

[Deprecated]: all list-columns are now preserved; If there are any that you don't want in the output use `select()` to remove them prior to unnesting.

`.id` **[Deprecated]**: convert `df %>% unnest(x, .id = "id")` to `df %>% mutate(id = names(x)) %>% unnest(`

`.sep` **[Deprecated]**: use `names_sep` instead.

Value

`tidySpatialExperiment`

New syntax

tidyr 1.0.0 introduced a new syntax for `nest()` and `unnest()` that's designed to be more similar to other functions. Converting to the new syntax should be straightforward (guided by the message you'll receive) but if you just need to run an old analysis, you can easily revert to the previous behaviour using `nest_legacy()` and `unnest_legacy()` as follows:

```
library(tidyr)
nest <- nest_legacy
unnest <- unnest_legacy
```

See Also

Other rectangling: `hoist()`, `unnest_longer()`, `unnest_wider()`

Examples

```
example(read10xVisium)
spe |>
    nest(data = -sample_id) |>
    unnest(data)
```

Index