Package 'nestedscales'

November 22, 2026

Title Nested Scales and Multi-Dimensional Indexes for Modeling

Version 0.3.0

URL https://optimal2050.github.io/nestedscales/r/, https://github.com/optimal2050/nestedscales

BugReports https://github.com/optimal2050/nestedscales/issues

Description Provides multi-level discrete representations of data dimensions for modeling, data processing and visualization. A nested scale records its finest units (atoms) and their membership in coarser levels (frames), including nested, cross-cutting and partially covered relationships. Recasts tabular data between levels using explicit, direction-aware rules for sums, means, standard deviations, copying and within-parent shares. Also supports labeling, subsetting, combining scales into multi-dimensional indexes, clustering units and storing data with its scale metadata. Members may carry attributes (a calorific value, an emission factor) and aliases, resolved along the hierarchy and applied row by row, and a scale is exchanged as a language-neutral definition file: a JSON header with inline, 'Arrow' or CSV tables (YAML accepted), with a catalogue of the definitions installed packages ship. Forms the dimension-agnostic core of 'timescales' <https://github.com/optimal2050/timescales> (calendars) and 'geoscales' <https://github.com/optimal2050/geoscales> (regions), designed for optimization and simulation models.

License Apache License (>= 2)

Encoding UTF-8

Depends R (>= 4.2)

Imports dplyr, jsonlite, rlang, S7, stats, utils, vctrs

Suggests arrow, cluster, data.table, dtplyr, ggplot2, knitr, pkgdown, rmarkdown, testthat (>= 3.0.0), tibble, withr, yaml

Config/testthat/edition 3

Config/Needs/website pkgdown, rmarkdown, ggplot2, optimal2050/timescales, optimal2050/geoscales, optimal2050/energypal

VignetteBuilder knitr

Config/roxygen2/version 8.1.1

Collate 'scale-class.R' 'aliases.R' 'product-class.R' 'array.R' 'attributes.R' 'backend.R' 'definition.R' 'catalog.R' 'cluster.R' 'contiguity.R' 'convert.R' 'dataset.R' 'distance.R' 'navigate.R' 'filter.R' 'frames.R' 'join.R' 'labels.R' 'link.R' 'map.R' 'nestedscales-package.R' 'recast.R' 'product-recast.R' 'reconcile.R' 'rules.R' 'scale-construct.R' 'str.R' 'sweep.R' 'tree.R' 'utils.R' 'zzz.R'

NeedsCompilation no

Author Oleg Lugovoy [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-5546-9875>)

Maintainer Oleg Lugovoy <olugovoy@optimalsolution.dev>

Repository CRAN

Date/Publication 2026-11-22 18:10:02 UTC

Contents

as_scale_array 4
as_scale_member 5
attach_aliases_scale 6
attach_attributes_scale 7
attach_labels_scale 8
backend 9
bind_scale 10
clear_scale_maps 12
clear_scale_rules 12
cluster_contiguous 13
cluster_medoids 14
cluster_scale 15
cluster_sweep 17
convert_scale 18
filter_product 19
filter_scale 20
get_scale_rule 21
is_valid_frame 22
join_product 22
join_scale 23
list_scale_rules 24
NestedScale 25
open_scale_dataset 26
product_coverage 27
product_frames 27
product_keys 28
product_leaves 28
product_size 29

product_weights . . . 30
prune_scale . . . 30
recast . . . 31
recast_crosswalk . . . 32
recast_product . . . 34
recast_scale . . . 35
recast_to_leaves . . . 38
reconcile_scale . . . 40
register_scale_distance . . . 42
register_scale_map . . . 43
register_scale_rule . . . 44
ScaleProduct . . . 45
scale_aliases . . . 46
scale_alias_property . . . 46
scale_ancestry . . . 47
scale_attributes . . . 48
SCALE_ATTRIBUTE_KINDS . . . 49
scale_axes . . . 50
scale_catalog . . . 50
scale_class . . . 51
scale_coverage . . . 52
scale_crosses . . . 52
scale_dataset_info . . . 53
scale_distance . . . 54
SCALE_DISTANCES . . . 55
scale_example . . . 55
scale_example2 . . . 56
scale_family . . . 57
scale_frames . . . 57
scale_from_leaftable . . . 58
scale_id . . . 59
scale_is . . . 60
scale_is_uniform . . . 61
scale_label . . . 62
scale_labels . . . 63
scale_leaftable . . . 63
scale_leaf_column . . . 64
scale_leaf_frame . . . 65
scale_leaf_pairs . . . 65
scale_map . . . 66
scale_map_between . . . 67
scale_match . . . 68
scale_members . . . 68
scale_navigate . . . 69
scale_nests . . . 70
scale_payload_collapse . . . 71
scale_payload_io . . . 72
scale_payload_parts . . . 73

scale_payload_slice 73
scale_product 74
scale_rank 75
scale_residuals 75
SCALE_RULES 76
scale_share 77
scale_tree 78
scale_vocab 79
scale_weights 79
str.NestedScale 80
summary.NestedScale 81
summary.ScaleProduct 82
write_scale 82
write_scale_dataset 84
[.NestedScale 85

Index **87**

`as_scale_array` *Data on scales as a dense array*

Description

[Experimental] This feature may change or be removed in a later version.

Reshapes a long table into an array whose dimensions are the scale's axes, in the order the product declares them, with the members as `dimnames`.

Usage

```
as_scale_array(data, x, value = NULL, frames = NULL, fill = NA, limit = 1e+07)

as_scale_table(a, x, value = "value", frames = NULL, drop_na = TRUE)
```

Arguments

`data`	A long table with one column per axis key plus the value column. Lazy inputs are materialised – an array is a dense in-memory object by definition.
`x`	A `NestedScale` or `ScaleProduct`.
`value`	Name of the value column. Defaults to the single numeric column that is not a key.
`frames`	Optional named list giving the frame each axis is keyed at; defaults to each axis's leaf frame.
`fill`	Value for cells the data does not cover.
`limit`	Refuse to allocate more than this many cells.
`a`	An array produced by `as_scale_array()` (or shaped like one).
`drop_na`	Omit cells the array does not cover, rather than emitting them as rows with `NA` values.

Value

An array with one dimension per axis.

`as_scale_table()` returns a `data.frame`: one column per axis key, plus the value column.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
d <- merge(
  data.frame(item = c("U1", "U2")),
  data.frame(period = c("p1", "p2"))
)
d$v <- 1:4
as_scale_array(d, p, value = "v")
as_scale_table(as_scale_array(d, p, value = "v"), p, value = "v")
```

---

as_scale_member *Codes bound to a scale*

Description

[Experimental] This feature may change or be removed in a later version.

`as_scale_member()` marks a vector of codes as belonging to one frame of one scale; it is what `bind_scale()` puts in a column. The result is a character vector of class `scale_member` that carries the scale's name, its `scale_id()` and the frame. `is_scale_member()` tests for it. Experimental, with the rest of the data link; see `bind_scale()`.

Usage

```
as_scale_member(x, scale, frame)

is_scale_member(x)
```

Arguments

`x`	A vector of codes; for `is_scale_member()`, any object.
`scale`	The `NestedScale` the codes belong to.
`frame`	The frame the codes are at.

Details

Combining two such vectors (binding rows, joining) works when they are on the same scale and frame and is an error otherwise. Combined with a plain character vector the result stays bound, the plain codes being taken on trust; the verbs check codes against the scale when they use them.

Value

`as_scale_member()` a `scale_member` vector as long as `x`; `is_scale_member()` a single logical.

See Also

bind_scale(), scale_link()

Examples

```
s <- scale_example()
u <- as_scale_member(c("P", "S", "P"), s, "sector")
u
is_scale_member(u)
as.character(u)
```

---

attach_aliases_scale *Attach aliases to a scale*

Description

Appends rows to the object's alias table: alternative labels by which a member (frame, member) is known, tagged with the source the spelling comes from (a palette, a model's commodity codes, a dataset). The same alias may legitimately point at several members; scale_match() reports that as ambiguous unless the caller names the frame.

Usage

```
attach_aliases_scale(x, aliases, source = NULL)
```

Arguments

x	A NestedScale.
aliases	A data.frame with columns frame, member, alias and, unless source is given, source.
source	Optional single string filled into the source column.

Value

The scale with the rows appended.

Examples

```
s <- attach_aliases_scale(
  scale_example(),
  data.frame(frame = "sector", member = c("P", "S"),
             alias = c("primary", "secondary")),
  source = "labels"
)
scale_match(s, c("Primary", "G1", "nope"))
```

attach_attributes_scale *Attach physical attributes to a scale*

Description

[Experimental] This feature may change or be removed in a later version.

Appends rows to the object's attribute table. A row names the member it belongs to (`frame`, `member`), the attribute (`attribute`, e.g. `"lhv"`), its `kind` (see `SCALE_ATTRIBUTE_KINDS`), its `value`, an optional `lower`/`upper` range, the physical `unit_of_measure` (`"GJ/t"`, `"kg/TJ"`), the `source` the figure was taken from and an optional `note`. Values can be attached at **any** frame: a calorific value for a whole carrier is inherited by every fuel under it that has no value of its own (see `scale_attributes()`).

Usage

```
attach_attributes_scale(x, attributes, source = NULL, kind = NULL)
```

Arguments

`x`	A `NestedScale`.
`attributes`	A `data.frame` with columns `frame`, `member`, `attribute`, `value`, `unit_of_measure`, and optionally `lower`, `upper`, `note`; `kind` and `source` as columns or through the arguments.
`source`	Optional single string filled into the `source` column when `attributes` has none.
`kind`	Optional single string filled into the `kind` column when `attributes` has none.

Details

Existing rows are never overwritten: a row whose `(frame, member, attribute, source)` already exists is an error. Attach a second source under its own `source` name and choose between them with `source=` when reading.

Value

The scale with the rows appended.

Examples

```
s <- scale_example()
s2 <- attach_attributes_scale(
  s,
  data.frame(frame = "class", member = c("G1", "G2", "S1"),
             attribute = "density", value = c(10, 20, 5),
             unit_of_measure = "t/m3"),
  source = "survey", kind = "property"
)
scale_attributes(s2, "density", frame = "item")
```

attach_labels_scale *Attach labels to a scale*

Description

Adds display names to the object's label table. A label belongs to a member (`frame`, `member`) and to a `variant` – a free name such as `"name"`, `"short"` or a language code – so a member may carry several. The table is sparse: frames and members without a row are shown by their code. A row with `member = NA` labels the frame itself.

Usage

```
attach_labels_scale(
  x,
  labels,
  variant = NULL,
  default = NULL,
  overwrite = FALSE
)
```

Arguments

`x`	A `NestedScale`.
`labels`	A `data.frame` with columns `frame`, `member`, `label` and, unless `variant` is given, optionally `variant`.
`variant`	Single string filled into the `variant` column when `labels` has none. Defaults to `"label"`.
`default`	Single string naming the variant used when a lookup names none. `NULL` keeps the current default (initially the first variant attached).
`overwrite`	Replace the label of a (`frame`, `member`, `variant`) already present. `FALSE` (default) makes that an error.

Details

Labels are for display. Alternative spellings to be *matched* belong in the alias table, see `attach_aliases_scale()`.

Value

The scale with the rows added.

See Also

`scale_labels()`, `scale_label()`

Examples

```
s <- attach_labels_scale(
  scale_example(),
  data.frame(frame = "sector", member = c("P", "S", NA),
             label = c("Primary", "Secondary", "Sector"))
)
scale_labels(s)
scale_label(s, c("P", "S", "T"), frame = "sector")
```

---

backend *Backends: the table types the verbs accept*

Description

The conversion verbs run on a `data.frame`, a tibble, a `data.table`, a dtplyr lazy table or an arrow Dataset/Table/query, and return results in the caller's type; lazy inputs stay lazy unless collected. These functions are that contract, exposed so a dimension package built on nestedscales can accept the same inputs as the core verbs.

Usage

```
backend_of(data, arg = "data")

backend_is_lazy(backend)

backend_schema(data, backend = NULL)

backend_lazy(data, backend = NULL)

backend_pull(table)

backend_restore(table, backend, collect = NULL)

backend_bind_rows(
  parts,
  use_dt = requireNamespace("data.table", quietly = TRUE)
)
```

Arguments

`data`	A data object on any supported backend.
`arg`	Argument name used in the error message.
`backend`	A backend name from `backend_of()`; `NULL` resolves it from `data`.
`table`	A query or table.
`collect`	Force materialisation of a lazy result.
`parts`	List of data frames.

use_dt — Use `data.table::rbindlist()`. Defaults to whether data.table is installed; an explicit value makes both branches testable.

Details

- `backend_of()` names the backend of `data`, and errors on an unsupported input – the one place that message is written.
- `backend_is_lazy()` says whether a backend produces queries.
- `backend_schema()` describes the columns without scanning the data.
- `backend_lazy()` lifts `data` into a dplyr-compatible carrier for a pipeline (a `data.table` becomes a dtplyr step when dtplyr is installed).
- `backend_pull()` materialises a query or table as a `data.frame`.
- `backend_restore()` returns a pipeline result in the caller's type.
- `backend_bind_rows()` binds chunk results into one frame.

Value

`backend_of()` a single string naming the backend; `backend_is_lazy()` a logical; `backend_schema()` a zero-row `data.frame`; `backend_lazy()` `data`, wrapped where the backend needs it; `backend_pull()` a `data.frame`; `backend_restore()` the result in the input's class; `backend_bind_rows()` one data frame, or `NULL` when every part is empty.

Examples

```
d <- data.frame(item = c("U1", "U2", "U2"), v = 1:3)
backend_of(d)
backend_is_lazy(backend_of(d))
backend_schema(d)
```

---

bind_scale *Bind a table to a scale*

Description

[Experimental] This feature may change or be removed in a later version.

Records, on the code columns themselves, which scale and which frame their codes are on. A bound column is a `scale_member` vector: it prints and behaves as a character vector, and keeps the link through row subsetting, `mutate()`, grouping, joins and row binding. The verbs (`recast_scale()`, `join_scale()`, the leaf routes) then take the source frame from the link, stop when the data is bound to a different scale, and bind their result.

Usage

```
bind_scale(data, x, cols = NULL)

unbind_scale(data, cols = NULL)

scale_link(data)
```

Arguments

data	A data.frame, tibble or data.table; for scale_link() also a scale_dataset from open_scale_dataset().
x	The NestedScale the codes belong to.
cols	Which columns to bind. NULL (default) binds every column named after a frame of x (or after its leaf column). Otherwise a character vector of frame names, named by column where the column is called something else: c(origin = "sector"). For unbind_scale(), the columns to unbind; NULL for all bound columns.

Details

The link is experimental and may change or be removed. It is an R construct: converting a bound column to a factor or to a plain character vector drops it, and other languages reading the same data do not see it. Data that is not bound is matched to a scale by its codes alone.

Every code is checked against the scale when binding; a code that is not a member is an error. scale_link() reports what a table is bound to and unbind_scale() returns the columns to plain character.

A lazy table (arrow, dtplyr) cannot carry the link: collect it first, or keep the link on disk with write_scale_dataset().

Value

bind_scale() and unbind_scale() return data; scale_link() a data.frame with one row per bound column: column, scale, frame, id.

See Also

scale_id(), as_scale_member()

Examples

```
s <- scale_example()
d <- data.frame(class = c("G1", "G2", "S1"), v = 1:3)
b <- bind_scale(d, s)
scale_link(b)

# the source frame comes from the link, and the result is bound
out <- recast_scale(b, s, target = "sector", rule = "sum")
scale_link(out)

# data bound to one scale is refused by another
try(recast_scale(b, scale_example2(), target = "sector", rule = "sum"))

unbind_scale(b)$class
```

clear_scale_maps *Clear the registered crosswalks*

Description

Mainly useful in tests.

Usage

```
clear_scale_maps(name = NULL)
```

Arguments

name	Registry keys to remove (as listed by list_scale_maps()); NULL (default) clears every registered crosswalk.

Value

Invisibly NULL.

Examples

```
clear_scale_maps()
```

clear_scale_rules *Clear the rule registry*

Description

Mainly useful in tests.

Usage

```
clear_scale_rules(name = NULL, scope = NULL)
```

Arguments

name	Optional character vector of parameter names to remove. NULL (default) clears everything in scope.
scope	Optional dimension namespace.

Value

Invisibly NULL.

Examples

```
register_scale_rule("tmp_param", "sum")
clear_scale_rules("tmp_param")
```

`cluster_contiguous` *Cluster the members of a scale under a contiguity constraint*

Description

Merges only members the constraint allows to be merged, so the resulting groups are contiguous in order (an ordered scale) or connected in space (an adjacency graph).

Usage

```
cluster_contiguous(
  data,
  x,
  k,
  frame = NULL,
  key = NULL,
  value = NULL,
  distance = "euclidean",
  scale_members = FALSE,
  adjacency = "order",
  circular = FALSE,
  linkage = "average",
  new_frame = "cluster",
  labels = "c"
)
```

Arguments

`data`	Long data: member codes, one or more identifier columns saying which observation each row is, and a value column. A member that is not observed over the same set as the others is an error rather than a hole.
`x`	A `NestedScale`.
`k`	Number of clusters.
`frame`	Frame whose members are clustered. Defaults to the leaf frame.
`key, value`	Columns of `data`; inferred when unambiguous.
`distance`	A name from `list_scale_distances()`.
`scale_members`	Standardise each member's profile before measuring; see `scale_distance()`.
`adjacency`	How members may be merged. `"order"` (the default) treats the members at `frame` as a sequence in member order and permits only neighbours – the clusters come out as consecutive blocks. Otherwise a two-column data frame or a square logical/0-1 matrix over the members, giving the pairs that may merge.

`circular`	For `"order"`: treat the sequence as a loop, so the last member neighbours the first. Right for a yearly cycle, wrong for a trajectory.
`linkage`	How the distance between two clusters is measured: `"average"`, `"complete"` or `"single"`.
`new_frame`	Name for the frame the clusters form.
`labels`	How to name the clusters: a prefix (the default `"c"` gives `c01`, `c02`, ...), or `"medoid"` to name each cluster after its representative member, which keeps the codes meaningful.

Details

The constraint can make k unreachable: an adjacency graph with more connected components than k cannot be merged below that number, and the function stops with the count rather than returning something that quietly violates the constraint.

Value

As `cluster_scale()`: the scale with the cluster frame added, and the assignment in the `"clustering"` attribute.

Examples

```
s <- scale_example()
d <- data.frame(
  item = rep(scale_members(s), each = 4),
  t = rep(sprintf("t%d", 1:4), 7), v = as.numeric(seq_len(28))
)
cl <- cluster_contiguous(d, s, k = 3)
attr(cl, "clustering")
```

`cluster_medoids` *The representative member of each cluster*

Description

The medoid is an ACTUAL member, not an average of members, which is what makes it usable as a representative period or a representative region: a model built on medoids is built on things that really happened.

Usage

```
cluster_medoids(x, frame = NULL)
```

Arguments

`x`	A scale returned by `cluster_scale()`.
`frame`	The cluster frame; defaults to the one the clustering added.

Value

A data frame of cluster and its representative member.

Examples

```
s <- scale_example()
d <- data.frame(
  item = rep(scale_members(s), each = 4),
  t = rep(sprintf("t%d", 1:4), 7), v = as.numeric(seq_len(28))
)
cluster_medoids(cluster_scale(d, s, k = 3))
```

---

cluster_scale *Cluster the members of a scale into a new frame*

Description

Groups the members at `frame` by the similarity of their profiles and returns the scale with those groups added as a new, coarser frame.

Usage

```
cluster_scale(
  data,
  x,
  k,
  frame = NULL,
  key = NULL,
  value = NULL,
  method = c("pam", "hclust", "kmeans"),
  distance = "euclidean",
  scale_members = FALSE,
  new_frame = "cluster",
  labels = "c",
  hclust_method = "ward.D2",
  seed = NULL,
  ...
)
```

Arguments

`data`	Long data: member codes, one or more identifier columns saying which observation each row is, and a value column. A member that is not observed over the same set as the others is an error rather than a hole.
`x`	A `NestedScale`.
`k`	Number of clusters.

frame	Frame whose members are clustered. Defaults to the leaf frame.
key, value	Columns of data; inferred when unambiguous.
method	"pam" (partitioning around medoids, via the cluster package), "hclust" (stats::hclust) or "kmeans" (stats::kmeans). "pam" and "hclust" work from the distance matrix, so they honour distance=; "kmeans" works on the raw profiles and ignores it.
distance	A name from list_scale_distances().
scale_members	Standardise each member's profile before measuring; see scale_distance().
new_frame	Name for the frame the clusters form.
labels	How to name the clusters: a prefix (the default "c" gives c01, c02, ...), or "medoid" to name each cluster after its representative member, which keeps the codes meaningful.
hclust_method	Linkage for method = "hclust".
seed	Seed for "kmeans", which starts from a random assignment. NULL leaves the session's stream alone.
...	Passed to the underlying algorithm.

Details

The new frame goes directly above the clustered one, which is the only position that is always true: the clusters are coarser than frame by construction, but they need not nest inside any existing coarser frame, and nestedscales does not require them to. Use scale_nests() to ask whether a given pair happens to nest.

Value

The scale, with new_frame inserted directly above frame. The assignment is also returned as the "clustering" attribute: a data frame of member, cluster and (for medoid-based methods) whether the member is its cluster's medoid.

Examples

```
s <- scale_example()
set.seed(1)
d <- data.frame(
  item = rep(scale_members(s), each = 4),
  t = rep(sprintf("t%d", 1:4), 7),
  v = as.numeric(seq_len(28))
)
cl <- cluster_scale(d, s, k = 3)
scale_frames(cl)
head(attr(cl, "clustering"))
```

`cluster_sweep` *Cluster quality across a range of k*

Description

Cluster quality across a range of k

Usage

```
cluster_sweep(
  data,
  x,
  ks,
  frame = NULL,
  key = NULL,
  value = NULL,
  distance = "euclidean",
  scale_members = FALSE,
  FUN = cluster_scale,
  ...
)
```

Arguments

`data`	Long data: member codes, one or more identifier columns saying which observation each row is, and a value column. A member that is not observed over the same set as the others is an error rather than a hole.
`x`	A `NestedScale`.
`ks`	Values of `k` to try.
`frame`	Frame whose members are clustered. Defaults to the leaf frame.
`key, value`	Columns of `data`; inferred when unambiguous.
`distance`	A name from `list_scale_distances()`.
`scale_members`	Standardise each member's profile before measuring; see `scale_distance()`.
`FUN`	The clustering function to sweep: `cluster_scale()` (default) or `cluster_contiguous()`.
`...`	Passed to `FUN`.

Value

A data frame with one row per `k`: `k`, `within` (total within-cluster sum of distances to the cluster medoid), `silhouette` (average, `NA` at `k = 1`), and `smallest` (the size of the smallest cluster, which is often what rules a k out).

Examples

```
s <- scale_example()
d <- data.frame(
  item = rep(scale_members(s), each = 4),
  t = rep(sprintf("t%d", 1:4), 7), v = as.numeric(seq_len(28))
)
cluster_sweep(d, s, ks = 2:4)
```

`convert_scale` *Convert member-keyed quantities with a member attribute*

Description

[Experimental] This feature may change or be removed in a later version.

Multiplies (or divides) the numeric value columns of `data` by the value of `attribute` resolved for each row's member – a net calorific value to turn mass into energy, an emission factor to turn energy into CO2, a density to turn volume into mass. Values are resolved along the hierarchy exactly as `scale_attributes()` does, so a factor stored on a coarser member applies to every leaf under it that has no factor of its own.

Usage

```
convert_scale(
  data,
  x,
  attribute,
  from_unit,
  to_unit,
  key = NULL,
  frame = NULL,
  values = NULL,
  source = NULL,
  na_action = c("error", "keep"),
  collect = NULL
)
```

Arguments

`data`	The data, in any supported backend, with a column of codes at `frame`.
`x`	A `NestedScale` carrying `attribute`.
`attribute`	Name of the attribute to convert with.
`from_unit, to_unit`	The physical units of the values before and after, matching the attribute's unit of measure.
`key`	Name of the code column in `data`; defaults to `frame`.

frame	Frame the codes belong to. Inferred when exactly one of the scale's frame names (or its leaf column) is a column of data.
values	Value columns to convert; NULL takes every numeric column that is not the key.
source	Restrict the attribute to one source; NULL errors on a conflict between sources.
na_action	What to do with rows whose member has no value: "error" (default) or "keep" (an NA result).
collect	For lazy inputs: materialise (TRUE) or return the query.

Details

Direction comes from the attribute's unit of measure, read as "<numerator>/<denominator>": with lhv in "GJ/t", from_unit = "t", to_unit = "GJ" multiplies, and from_unit = "GJ", to_unit = "t" divides. The inverse is allowed only when the attribute's kind is "property"; a "factor" is one-way. No unit-prefix arithmetic is done: a column in kilotonnes must be turned into tonnes before converting with a GJ/t factor.

A member for which no value resolves is an **error** by default – a silently dropped row is a wrong total – unless na_action = "keep", which leaves NA in its rows.

Value

data with the value columns converted, plus two columns named after the attribute: <attribute> (the factor applied) and <attribute>.origin_frame (the frame the factor was taken from), in the input's class.

Examples

```
s <- attach_attributes_scale(
  scale_example(),
  data.frame(frame = c("sector", "item"), member = c("P", "U5"),
             attribute = "pack_size", value = c(20, 30),
             unit_of_measure = "pieces/box"),
  source = "example", kind = "property"
)
boxes <- data.frame(item = c("U1", "U2", "U5"), n = c(100, 200, 50))
pieces <- convert_scale(boxes, s, attribute = "pack_size",
                        from_unit = "box", to_unit = "pieces")
pieces
# back to boxes: a property is invertible
convert_scale(pieces[c("item", "n")], s, attribute = "pack_size",
              from_unit = "pieces", to_unit = "box")
```

---

filter_product *Subset a product componentwise*

Description

[Experimental] This feature may change or be removed in a later version.

Applies filter_scale() to one axis and returns a new product.

Usage

```
filter_product(x, axis, frame, member, ...)
```

Arguments

x	A ScaleProduct.
axis	Axis to filter.
frame, member	Passed to filter_scale().
...	Passed to filter_scale().

Value

A ScaleProduct.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
filter_product(p, "a", "sector", "P")
```

---

filter_scale *Subset a NestedScale by member*

Description

Keeps only the leaves belonging to member at frame, and rebuilds the member vocabularies accordingly.

Usage

```
filter_scale(x, frame, member, drop_empty_frames = FALSE)
```

Arguments

x	A NestedScale.
frame	Frame that member belongs to.
member	Character vector of codes to keep.
drop_empty_frames	Drop frames left with no codes at all.

Details

A genuine subset is a SAMPLE and is book-kept as one: meta$coverage records, per weight column, the kept fraction of the ROOT parent's total (so filters compose against the original object), meta$parent_totals stores those root totals (making the coverage claim verifiable by the validator), meta$parent_name records the parent, and meta$name is mangled to "parent[frame:codes]" so a sample never impersonates its parent in the crosswalk registry or in join_scale() column names. A filter that keeps every leaf is a true no-op. Read the fraction back with scale_coverage().

Value

A NestedScale.

Examples

```
s <- scale_example()
p <- filter_scale(s, "sector", "P")
p
scale_coverage(p)
```

---

get_scale_rule *Look up a registered rule*

Description

A scoped lookup prefers the entry registered for that scope and falls back to an unscoped one.

Usage

```
get_scale_rule(param, scope = NULL)
```

Arguments

param	Name of the value column.
scope	Optional dimension namespace.

Value

A list with elements rule, weight and scope, or NULL if param has not been registered.

Examples

```
register_scale_rule("demand", "sum")
get_scale_rule("demand")
get_scale_rule("not_registered")
clear_scale_rules("demand")
```

is_valid_frame *Validate frame names*

Description

A frame name must be a non-empty, non-NA string that is a syntactically valid R name (so it can be used as a data.frame column without quoting).

Usage

```
is_valid_frame(x)
```

Arguments

x	Character vector of candidate frame names.

Value

A logical vector the same length as x.

Examples

```
is_valid_frame(c("industry", "reg32", "", "2bad"))
```

join_product *Attach every axis's labels to product-indexed data*

Description

[Experimental] This feature may change or be removed in a later version.

join_scale() applied to each axis in turn, keyed by that axis's column. Because every attached column is named after its scale, the result carries all the axes' labels side by side.

Usage

```
join_product(data, x, axes = NULL, ...)
```

Arguments

data	The data, carrying one key column per axis.
x	A ScaleProduct.
axes	Axis names to attach; NULL (default) attaches all of them.
...	Passed to join_scale() (attach, meta, as_factor, ...).

Value

`data` with each axis's labels attached, in its own class.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
d <- data.frame(item = c("U1", "U3"), period = c("p1", "p2"), v = 1:2)
join_product(d, p, attach = TRUE)
```

`join_scale` *Attach scale labels and memberships to a dataset*

Description

Adds a code column named after the scale, and optionally the codes of coarser frames (`"<name>.<frame>"`), the display names of the codes (`"<name>.label"`) and the member's share and weight (`"<name>.share"`, `"<name>.weight"`).

Usage

```
join_scale(
  data,
  x,
  key = NULL,
  frame = NULL,
  attach = NULL,
  meta = FALSE,
  weight = NULL,
  as_factor = TRUE,
  collect = NULL,
  diagnostics = c("auto", "on", "off"),
  labels = NULL
)
```

Arguments

`data`	The data, in any supported backend, with a column of member codes.
`x`	The `NestedScale` to attach.
`key`	Name of the code column in `data`. Inferred from the scale's name, the keyed frame, or the scale's leaf column.
`frame`	The frame the codes in `key` are at. `NULL` (default) is inferred as the single frame name appearing among the data's columns.
`attach`	Coarser frames whose codes to attach as new columns: `TRUE` for all of them, a character vector to choose, `NULL`/`FALSE` for none.
`meta`	Attach the member's `share` and `weight` at the keyed frame.

weight	Weight column used for meta. NULL uses the default.
as_factor	Return attached membership columns as factors with the frame's full vocabulary as levels (so empty groups still plot).
collect	For lazy inputs: materialise (TRUE) or return the query.
diagnostics	Whether to run the checks that read the data (no matching codes; codes that are not members). "auto" (default) runs them on eager inputs and skips them on arrow/dtplyr, which makes this verb zero-scan there; "on"/"off" force the choice.
labels	Attach the display names of the keyed codes as "<name>.label": TRUE for the default variant, a string to name one, NULL/FALSE (default) for none. See attach_labels_scale().

Details

Existing columns are never overwritten – a collision is an error.

Value

data with the attached columns, in its own class.

Examples

```
s <- scale_example()
d <- data.frame(class = c("G1", "G2", "S1"), v = 1:3)
join_scale(d, s, attach = TRUE)
join_scale(d, s, attach = "sector")
join_scale(d, s, meta = TRUE)
```

---

list_scale_rules *List registered rules*

Description

List registered rules

Usage

```
list_scale_rules(scope = NULL)
```

Arguments

scope	Optional dimension namespace; NULL lists every entry.

Value

A data.frame with columns param, rule, weight and scope.

Examples

```
register_scale_rule("invcost", "weighted_mean", weight = "size")
list_scale_rules()
clear_scale_rules("invcost")
```

`NestedScale` *NestedScale (S7 class)*

Description

A nested discrete partition: a flat table of leaves plus the ordered hierarchy of frames that groups them. The dimension is not named – time and space are the subclasses `Calendar` (timescales) and `Geoscale` (geoscales); anything else (industries, income brackets, temperature regimes, technology vintages) is declared directly.

Usage

```
NestedScale(
  leaftable,
  frames,
  members,
  leaf = "leaf",
  meta = list(),
  attributes = NULL,
  aliases = NULL,
  labels = NULL
)
```

Arguments

`leaftable`	`data.frame` with a unique leaf column, one column per frame in `frames`, and zero or more numeric weight columns.
`frames`	Ordered character vector of frame names (coarsest first); each must appear as a column of `leaftable`.
`members`	Named list; `members[[f]]` is the full ordered set of codes present at frame `f`. Must equal the non-`NA` values of `leaftable[[f]]` as a set.
`leaf`	Name of the leaf column (a single string, default `"leaf"`).
`meta`	Named list of attributes (`name`, `desc`, `weights`, `default_weight`, `residuals` (see `scale_residuals()`), `default_label`, `source`, sample bookkeeping).
`attributes`	Optional table of per-member physical attributes keyed by `(frame, member)`; see `attach_attributes_scale()` for the columns.
`aliases`	Optional table of alternative labels keyed by `(frame, member)`; see `attach_aliases_scale()` for the columns.
`labels`	Optional table of display names keyed by `(frame, member, variant)`; see `attach_labels_scale()` for the columns.

Details

Construct with scale_from_leaftable().

Value

A NestedScale object.

See Also

scale_from_leaftable(), scale_frames(), scale_members()

Examples

```
scale_example()
```

---

open_scale_dataset *Open a scale-aware dataset*

Description

[Experimental] This feature may change or be removed in a later version.

Usage

```
open_scale_dataset(path)

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

Arguments

path	A folder written by write_scale_dataset().
x	A scale_dataset.
...	Ignored.

Value

A scale_dataset: a list with data (a lazily-read arrow Dataset) and scale (the NestedScale or ScaleProduct it is indexed by). Feed them to the verbs as recast_scale(ds$data, ds$scale, ...). scale_link() reports which columns of the stored data are on which scale and frame.

Examples

```
# see write_scale_dataset()
```

product_coverage *Coverage of each axis*

Description

[Experimental] This feature may change or be removed in a later version.

Sampling is per-axis and independent, so the coverage of the product is the product of its axes' coverages – `prod(product_coverage(p))`.

Usage

```
product_coverage(x, weight = NULL)
```

Arguments

`x`	A `ScaleProduct`.
`weight`	Optional weight name, applied to every axis. `NULL` uses each axis's own default weight.

Value

A named numeric, one entry per axis.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
product_coverage(p)
prod(product_coverage(p))
```

product_frames *Frames of each axis*

Description

[Experimental] This feature may change or be removed in a later version.

Usage

```
product_frames(x)
```

Arguments

`x`	A `ScaleProduct`.

Value

A named list of character vectors, each coarsest-first.

Examples

```
product_frames(scale_product(a = scale_example(), b = scale_example2()))
```

`product_keys` *The data key column of each axis*

Description

[Experimental] This feature may change or be removed in a later version.

Data on a product carries one of these columns per axis.

Usage

```
product_keys(x)
```

Arguments

`x`	A `ScaleProduct`.

Value

A named character vector, axis -> column name.

Examples

```
product_keys(scale_product(a = scale_example(), b = scale_example2()))
```

`product_leaves` *Realise the leaves of a product*

Description

[Experimental] This feature may change or be removed in a later version.

The cross product of the component leaftables: one row per tuple of component leaves, with each axis's key column named by `product_keys()` and its frame columns prefixed `"<axis>."`.

Usage

```
product_leaves(x, limit = 1e+06)
```

Arguments

x	A ScaleProduct.
limit	Maximum number of rows to build. Exceeding it is an error that reports the size, rather than a very long wait.

Details

This is guarded on purpose. An hourly calendar crossed with NUTS3 is ~12.9M rows; the whole point of a product is that its leaves exist implicitly, and conversion never needs them realised (see recast_scale()). Ask for them only when you actually want the grid.

Value

A data.frame with product_size(x)$total rows.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
head(product_leaves(p))
```

---

product_size *Size of a product*

Description

[Experimental] This feature may change or be removed in a later version.

The per-axis leaf counts and their product – the number of rows a fully dense realisation would have.

Usage

```
product_size(x)
```

Arguments

x	A ScaleProduct.

Value

A list with axes (named integer, leaves per axis) and total (double, their product).

Examples

```
product_size(scale_product(a = scale_example(), b = scale_example2()))
```

product_weights *Weight columns of each axis*

Description

[Experimental] This feature may change or be removed in a later version.

A product has no weights of its own: the weight of a product leaf is the product of its components' weights, which is derived when needed and never stored.

Usage

```
product_weights(x)
```

Arguments

x	A ScaleProduct.

Value

A named list of character vectors.

Examples

```
product_weights(scale_product(a = scale_example(), b = scale_example2()))
```

prune_scale *Collapse a NestedScale to a coarser frame*

Description

Returns a new NestedScale whose leaf layer is frame, dropping every finer frame. Weights are summed over the collapsed leaves.

Usage

```
prune_scale(x, frame)
```

Arguments

x	A NestedScale.
frame	The frame to become the new leaf layer.

Details

The result is renamed `"name@frame"` with the parent recorded in `meta$parent_name`; every other meta field is preserved. Leaves with no code at `frame` are dropped, and that loss is reflected in `meta$coverage`. The concrete class is kept: a subclass comes back as itself, its attribute and alias rows at the surviving frames kept, and a positional payload merged through `scale_payload_collapse()`.

Value

A `NestedScale`.

Examples

```
prune_scale(scale_example(), "class")
```

---

recast *Recast data through a scale*

Description

The bare pipeline verb, dispatching on the scale so one entry point serves a `NestedScale`, a `ScaleProduct`, and the dimension subclasses in `timescales` and `geoscales`, which register their own methods against this generic. `recast_scale()` and `recast_product()` are the explicit workers.

Usage

```
recast(x, scale, ...)
```

Arguments

`x`	The data to recast.
`scale`	The scale to recast through.
`...`	Passed to the dispatched method: for a `NestedScale` the arguments of `recast_scale()`, with `target` the target frame and `source` the source frame (inferred when omitted); for a `ScaleProduct` those of `recast_product()`.

Details

The scale is `scale`; the source frame is `source` and the target `target`, as in `recast_scale()`, so a call reads the same through the generic and through the worker.

Value

The recast data, in the input's class.

Examples

```
s <- scale_example()
d <- data.frame(
  item = c("U1", "U2", "U3", "U4", "U5", "U6"),
  capacity = c(1, 2, 3, 4, 5, 6)
)
recast(d, s, target = "sector", rule = "sum")
```

`recast_crosswalk` *Recast values through a given crosswalk*

Description

The engine behind `recast_scale()`, for a crosswalk built elsewhere: by a dimension package whose conversion cannot be read off one scale's leaftable, such as timescales mapping one calendar onto another through a datetime grid. The crosswalk says how much of each source code falls in each target code; the rules then aggregate exactly as in `recast_scale()`, on any supported backend.

Usage

```
recast_crosswalk(
  data,
  map,
  source,
  target,
  key = source,
  values = NULL,
  rule = NULL,
  ids = NULL,
  by = character(),
  targets = NULL,
  missing_sources = c("na", "ignore"),
  member = "member",
  collect = NULL
)
```

Arguments

`data`	The data, in any supported backend, with a code column and one or more numeric value columns.
`map`	The crosswalk, a `data.frame`.
`source, target`	Names of the source and target code columns of `map`.
`key`	Name of the code column of `data`. Defaults to `source`.
`values`	Value columns. Default: every numeric column that is neither `key` nor in `ids`.

rule	One of SCALE_RULES other than the share rules, for every value column, or a named vector with one entry per column. NULL looks each column up with get_scale_rule().
ids	Identifier columns, preserved as groups. Default: every column that is neither key nor a value column. Other columns are dropped.
by	Columns of map matched against the identically named identifier columns of data, for a crosswalk that differs between identifier values (a calendar crosswalk per year, say).
targets	Optional full target vocabulary. A materialised result is completed to every target per identifier combination, in this order, with NA where nothing landed; an NA entry adds an explicit NA row.
missing_sources	"na" (default) or "ignore"; see recast_scale().
member	The word for a target code in error messages.
collect	For lazy inputs: materialise (TRUE) or return the query.

Details

The crosswalk is a data frame with one row per (source, target) pair:

- the source and target codes, in the columns named by source and target;
- n_from, the number of leaves (grid points) in the source code, and n_overlap, the number of them that fall in the target;
- optionally w, the weight of the pair for "weighted_mean" – by default n_overlap;
- optionally w_from, the source's total weight. With it, "sum" splits a source across its targets by w / w_from; without it, by n_overlap / n_from, an equal split over leaves;
- any columns named in by.

A target whose sources are only partly present in the data, per identifier combination, comes back NA unless missing_sources = "ignore".

Value

The recast data in its own class, with columns c(to, ids, values). Lazy in, lazy out.

Examples

```
# Two months onto one quarter, with the months' day counts as leaves
map <- data.frame(
  month = c("m01", "m02"), quarter = "Q1",
  n_from = c(31, 28), n_overlap = c(31, 28)
)
d <- data.frame(
  month = c("m01", "m02"), energy = c(310, 280),
  price = c(10, 20)
)
recast_crosswalk(d, map,
  source = "month", target = "quarter",
```

```
  rule = c(energy = "sum", price = "weighted_mean")
)
```

`recast_product` *Recast data indexed by a product of scales*

Description

[Experimental] This feature may change or be removed in a later version.

Converts one or more axes of a product in a single call. Axes named in `target` are converted; axes left out pass through untouched.

Usage

```
recast_product(
  data,
  x,
  target,
  values = NULL,
  rules = NULL,
  weights = NULL,
  na_action = c("drop", "error", "keep"),
  collect = NULL
)
```

Arguments

`data`	The data, in any supported backend, carrying one key column per axis (see `product_keys()`) plus the value columns.
`x`	A `ScaleProduct`.
`target`	Named list or character vector, one entry per axis to convert: either a frame name of that axis's scale, or another `NestedScale` to convert into. Axes omitted pass through unchanged.
`values`	Value columns to convert. Default: all numeric columns that are not key columns. Numeric identifiers (a year) must be excluded explicitly.
`rules`	Rule per value column and axis: a scalar for everything, or a named list whose entries are scalars or vectors named by axis, e.g. `list(load = c(time = "weighted_mean", space = "sum"))`. `NULL` (default) resolves each column on each axis through that dimension's own scope in the rule registry – which is where an intensive/extensive asymmetry between dimensions belongs.
`weights`	Weight column per axis, e.g. `c(space = "pop")`; scalar applies to every axis.
`na_action`	Passed to each axis's pass.
`collect`	For lazy inputs: materialise (`TRUE`) or return the query.

Details

Execution is a sequence of ordinary single-axis `recast_scale()` calls – see the file comment in `R/product-recast.R` for why that is exactly equivalent to a joint recast for every supported rule, and why the joint crosswalk is never built. The passes are ordered so the axis that reduces the data most runs first.

A per-axis `"sd"` is refused because it is order-dependent; a scalar `"sd"` means the pooled sd over all product leaves in a target block, which is well defined and is computed without materialising the product. `"share"` is not defined across a product, because it changes which axis the result is keyed by.

Value

The recast data in the input's class, keyed by the converted axes' target members.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
d <- merge(
  data.frame(item = c("U1", "U2", "U3", "U4", "U5", "U6")),
  data.frame(period = c("p1", "p2", "p3", "p4"))
)
d$cap <- seq_len(nrow(d))

# aggregate both axes at once
recast_product(d, p, target = list(a = "sector", b = "era"), rules = "sum")

# different rules per axis
recast_product(d, p,
  target = list(a = "sector", b = "era"),
  rules = list(cap = c(a = "sum", b = "weighted_mean"))
)
```

`recast_scale` *Recast values from one resolution to another*

Description

The central conversion verb: takes a table keyed by member code at frame `source` and returns one keyed at `target` – a frame name of the same `NestedScale`, or ANOTHER NestedScale (whose leaf layer is the target, matched on shared leaf codes). Handles both aggregation (fine to coarse) and disaggregation (coarse to fine) with one rule per value column; frames that cross-cut work too, because the route always goes through the leaf layer. A crosswalk registered with `register_scale_map()` short-circuits the derivation.

Usage

```
recast_scale(
  data,
  x,
  source = NULL,
  target,
  key = NULL,
  values = NULL,
  rule = NULL,
  weight = NULL,
  na_action = c("drop", "error", "keep"),
  parent = NULL,
  collect = NULL,
  diagnostics = c("auto", "on", "off"),
  missing_sources = c("na", "ignore"),
  batch = NULL,
  batch_by = NULL,
  ...
)
```

Arguments

data	The data to recast, in any supported backend, with a column named by key plus one or more numeric value columns; other columns are preserved as identifiers.
x	The NestedScale the codes in data belong to.
source	Frame name the codes belong to. NULL (default) is inferred: key when it is a frame name, else the single frame name appearing among the data's columns.
target	Target frame name of x, or another (named) NestedScale – then the target is that object's leaf layer, matched on shared leaf codes.
key	Name of the code column in data. Defaults to source when that column exists, otherwise the name of the scale's leaf column.
values	Character vector of value columns to convert. Default: all numeric columns other than the key and the scale's frame columns. Numeric identifiers (e.g. year) must be excluded explicitly.
rule	One of SCALE_RULES applied to every value column, or a named vector with one entry per column; NULL (default) looks each column up with get_scale_rule(). A column with neither an explicit rule nor a registry entry is an ERROR – there is deliberately no fallback (a silently guessed rule is a silent unit error).
weight	Weight column used by sum (splitting) and weighted_mean; scalar or named per column. NULL uses each column's registered weight, falling back to the object's default weight; when the object declares no weights at all, leaves weigh 1 (an equal split, warned about when it is material).
na_action	What to do with leaves that have no code at source or target: "drop" (default, with a warning – the affected source share is genuinely lost), "error", or "keep" (retain an explicit NA row so totals conserve).

parent — `rule = "share"` only: the frame defining the groups the shares are taken within. `NULL` (default) uses `target` when it differs from `source`, else the frame immediately above `source`.

collect — For lazy inputs: materialise the result (`TRUE`) or return the uncollected query (default).

diagnostics — Whether to run the checks that need a scan of the data: unknown source codes, dropped shares, and source members missing from the data. `"auto"` (default) runs them on eager inputs and skips them on arrow/dtplyr, where a scan would defeat the point of returning a query; `"on"` and `"off"` force the choice. Errors that do not depend on the data – unknown frames, reserved column names, a missing rule – are raised regardless.

missing_sources — What to do when a target member's sources are only partially present in the data. `"na"` (default) makes the result `NA` – a partial answer is not an answer. `"ignore"` aggregates whatever is there, which is faster but silently turns incomplete data into a plausible number, so it is opt-in.

batch, batch_by — Process the job in chunks of at most `batch` distinct values of the identifier column `batch_by` (default: the first identifier column), binding the pieces. Chunking by an identifier partitions the aggregation groups exactly, so the numbers are unchanged; any other column is refused. Applies only when materialising, and turns off per-chunk diagnostics, which would otherwise report sources as missing merely because they fall in another chunk.

... — Passed to `scale_map()` (e.g. `year` for a calendar).

Details

Columns of the data that are neither the key nor a value column are treated as identifiers (panel columns – a year, a technology) and preserved as grouping columns, so panel data recasts correctly in one call. This is what makes mixed pipelines across two dimensions work: a recast in one dimension carries the other dimension's key through untouched. Columns named like the scale's own frames are treated as member attributes and dropped.

`"sum"` splits each source value across its member's leaves proportionally to the weight before summing up, so totals are conserved. `"weighted_mean"` weights by the leaf weights; `"mean"` is the plain leaf-count mean – the two differ exactly when leaf weights differ. `"copy"` requires a constant value per target member; `"sd"` is aggregation-only.

`"share"` inverts the output contract: the result stays keyed at `source`, and each value becomes that member's share of the total over its parent group. `source` must nest within the parent. It cannot be combined with other rules in one call, and `weight=` is ignored – the observed values themselves are the weights.

Value

The recast table in the input's class, with columns `c(to, identifiers, values)`.

Backends

The data may be a `data.frame`, tibble, `data.table`, `dtplyr` lazy table, or an arrow Dataset/Table/query. The result comes back in the input's class; lazy inputs return the uncollected query unless `collect`

= TRUE. Lazy results contain the observed target members only – the full-vocabulary completion (and its NA rows) applies when the result is materialised.

Examples

```
s <- scale_example()

# Extensive quantity, fine -> coarse: totals are preserved
d <- data.frame(
  item = c("U1", "U2", "U3", "U4", "U5", "U6"),
  capacity = c(1, 2, 3, 4, 5, 6)
)
recast_scale(d, s, source = "item", target = "sector", rule = "sum")

# Coarse -> fine: split proportionally to size
y <- data.frame(sector = c("P", "S"), capacity = c(10, 20))
recast_scale(y, s,
  source = "sector", target = "class",
  rule = "sum", weight = "size"
)

# Share within parent: result stays at the leaves, sums to 1 per sector
recast_scale(
  d, s,
  source = "item", target = "sector",
  rule = "share",
  na_action = "keep"
)
```

---

recast_to_leaves *Recast member data down to the leaf layer, and back*

Description

The two public halves of the from -> leaves -> to route: recast_to_leaves() projects member-keyed data DOWN to the leaf layer (one row per leaf), and recast_from_leaves() aggregates leaf-keyed data UP into a frame's members. Their composition is recast_scale(), and because the leaf rows are keyed by the leaf codes, recast_from_leaves(recast_to_leaves(data, a), b, target) recasts across two scales that share leaf codes.

Usage

```
recast_to_leaves(
  data,
  x,
  source = NULL,
  key = NULL,
  values = NULL,
  rule = NULL,
```

```
  weight = NULL,
  attach_weight = TRUE,
  collect = NULL
)

recast_from_leaves(
  data,
  x,
  target,
  key = NULL,
  values = NULL,
  rule = NULL,
  weight = NULL,
  na_rm = FALSE,
  na_action = c("drop", "error", "keep"),
  collect = NULL
)
```

Arguments

data	The data: for `recast_to_leaves()` keyed by member code at frame `source`; for `recast_from_leaves()` keyed by leaf codes.
x	The `NestedScale` the data is keyed in, or aggregated into.
source	`to_leaves` only: frame name the codes belong to. `NULL` (default) is inferred as in `recast_scale()`.
key	The key column; defaults as in `recast_scale()`.
values, rule	As in `recast_scale()`.
weight	`to_leaves`: a declared weight column of `x`. `from_leaves`: the name of a column of `data` to weight by, so the weight may vary by identifier. `NULL` uses a `weight` column of the data if present, else the scale's declared weight.
attach_weight	`to_leaves` only: attach the `weight` column (default `TRUE`).
collect	For lazy inputs: materialise (`TRUE`) or return the query.
target	`from_leaves` only: target frame name.
na_rm	`from_leaves` only: read an `NA` value as "this member says nothing" rather than as an unknown that makes the whole group `NA`.
na_action	`from_leaves` only: what to do with leaves that have no code at `target`.

Details

Going down, extensive columns (rule `"sum"`) are split across a member's leaves proportionally to the chosen weight so totals conserve; intensive columns are repeated. A `weight` column is attached by default so the return trip's `"weighted_mean"` reproduces the source weighting exactly.

Value

`recast_to_leaves()`: one row per (leaf x identifier combination). `recast_from_leaves()`: one row per (member x identifier combination). Both in the input's class; lazy in, lazy out.

Examples

```
s <- scale_example()
d <- data.frame(sector = c("P", "S"), capacity = c(10, 20))
leaves <- recast_to_leaves(d, s,
  source = "sector", rule = "sum",
  weight = "size"
)
head(leaves)
recast_from_leaves(
  leaves, s,
  target = "class",
  rule = "sum",
  na_action = "keep"
)
```

`reconcile_scale` *Check fine data against an independent parent total*

Description

[Experimental] This feature may change or be removed in a later version.

Aggregates `data` from `source` to `target` and compares the result with `totals`, the parent figure from another source. Reports the gap by default; can also close it.

Usage

```
reconcile_scale(
  data,
  x,
  source,
  target,
  totals,
  key = NULL,
  values = NULL,
  rule = "sum",
  weight = NULL,
  balance = c("none", "residual", "proportional"),
  tolerance = 1e-09
)
```

Arguments

`data`	The fine data, keyed at `source`.
`x`	A `NestedScale`.
`source, target`	Frame names: where the data is, and the coarser frame whose totals it is checked against.

totals	The independent parent figures: a table keyed at target carrying the same value columns as data, plus any identifier columns (which are matched, so a per-year total works).
key	Column of data holding the codes at source; defaults as in recast_scale().
values	Value columns to check. Default: the numeric columns common to data and totals.
rule	How the value aggregates. "sum" (default) is the question "does it add up"; an intensive quantity reconciles as "weighted_mean" instead.
weight	Weight column for rule = "weighted_mean".
balance	What to do about the gap: • "none" (default) – report it and change nothing. • "residual" – return the data with each group's gap written onto its declared residual member at source. Needs exactly one residual per affected group, and rule = "sum". • "proportional" – return the data scaled by target / aggregated within each group, so every group hits its total.
tolerance	Relative size below which a gap counts as rounding; sets the ok column, and gaps within it are left alone when balancing.

Value

With balance = "none", a long gap table: the target column, any identifiers, value (which column), aggregated, target, gap (target - aggregated), rel_gap and ok. With a balancing method, the adjusted data keyed as the input was, carrying that same table as its "reconciliation" attribute.

Examples

```
lf <- data.frame(
  country = c("DE", "DE", "DE"),
  item = c("DE1", "DE2", "DE_XR"),
  pop = c(40, 30, 0), stringsAsFactors = FALSE
)
s <- scale_from_leaftable(lf,
  frames = c("country", "item"), leaf = "item",
  weights = "pop", name = "geo",
  residuals = list(item = "DE_XR")
)
fine <- data.frame(item = c("DE1", "DE2", "DE_XR"), gdp = c(400, 300, 0))
published <- data.frame(country = "DE", gdp = 750)

reconcile_scale(fine, s, source = "item", target = "country", published)

reconcile_scale(fine, s,
  source = "item", target = "country", published,
  balance = "residual"
)
```

`register_scale_distance` *Register a distance*

Description

[Experimental] This feature may change or be removed in a later version.

The built-ins cover level and shape; anything else – dynamic time warping, a domain-specific dissimilarity, a precomputed matrix – is registered here and then usable by name everywhere in the package.

Usage

```
register_scale_distance(name, fn)

get_scale_distance(name)

list_scale_distances()

clear_scale_distances(name = NULL)
```

Arguments

`name`	Distances to remove; `NULL` clears every registered one. Built-ins cannot be removed.
`fn`	A function of the member-by-feature matrix returning a stats::dist object (or a square matrix) over its rows.

Value

Invisibly, `name`.

Examples

```
register_scale_distance("first_diff", function(m) {
  stats::dist(t(apply(m, 1, diff)))
})
"first_diff" %in% list_scale_distances()
clear_scale_distances("first_diff")
```

register_scale_map *Register / look up a direct crosswalk*

Description

A registered map short-circuits the leaf-layer derivation in `scale_map()` or `scale_map_between()` (and thereby `recast_scale()`) for one pair of resolutions – for cases where the exact correspondence is known (hand-audited crosswalks, official concordance tables).

Usage

```
register_scale_map(x, source, target, map)

register_scale_map_between(source, target, map)

get_scale_map(x, source, target)

get_scale_map_between(source, target)

list_scale_maps()
```

Arguments

`x`	The `NestedScale` the frames belong to, or its name.
`source, target`	For the within-scale functions, frame names of `x`. For the `_between` functions, two `NestedScale` objects or their names.
`map`	A `data.frame` shaped like a `scale_map()` result: the two label columns named after the frames (or scales), plus `n_from`, `n_overlap`, `w` and `w_from`. `NULL` removes a previously registered map.

Details

`register_scale_map()` and `get_scale_map()` handle a pair of frames of one scale; the map is scoped to that scale, so the same frame pair in two different scales does not collide. `register_scale_map_between()` and `get_scale_map_between()` handle a pair of scales.

Value

Invisibly, the registry key. The `get_` functions return the registered map (or `NULL`); `list_scale_maps()` a `data.frame` of registry keys.

Examples

```
s <- scale_example()
fake <- data.frame(
  class = "G1", group = "GC", n_from = 1L,
  n_overlap = 1L, w = 1, w_from = 1
```

```
)
register_scale_map(s, "class", "group", fake)
list_scale_maps()
get_scale_map(s, "class", "group")
register_scale_map(s, "class", "group", NULL) # remove
clear_scale_maps()
```

`register_scale_rule` *Register how a parameter should be recast*

Description

Records the rule (and optionally the weight) to use for a named value column, so callers of `recast_scale()` need not repeat it. Downstream packages register their own parameter maps at load time.

Usage

```
register_scale_rule(param, rule, weight = NULL, scope = NULL)
```

Arguments

`param`	Name of the value column.
`rule`	One of `SCALE_RULES`.
`weight`	Optional weight column name used by `sum` (down) and `weighted_mean` (up). `NULL` means the scale's default weight.
`scope`	Optional dimension namespace (e.g. `"calendar"`, `"geoscale"`). `NULL` registers the rule for every dimension.

Value

Invisibly, the registered entry.

Examples

```
register_scale_rule("capacity", "sum")
register_scale_rule("eff", "weighted_mean", weight = "count")
get_scale_rule("eff")
clear_scale_rules(c("capacity", "eff"))
```

ScaleProduct *A product of several scales (S7 class)*

Description

[Experimental] This feature may change or be removed in a later version.

Combines two or more NestedScale objects into one multi-dimensional index – time x space, time x space x industry, and so on. The leaves of the product are tuples of the component leaves, and they stay LAZY: nothing is materialised at construction, and product_leaves() is the guarded way to realise them.

Usage

```
ScaleProduct(axes, keys, joint_weights = NULL, meta = list())
```

Arguments

axes	Named list of NestedScale objects; the names are the axis names.
keys	Named character vector, one entry per axis, giving the data column that carries that axis's member codes.
joint_weights	Reserved for a weight that varies jointly across axes (an hourly profile used as a spatial split weight). NULL in this version; a supplied table is validated but not yet consumed.
meta	Named list of attributes (name, desc).

Details

Construct with scale_product().

Value

A ScaleProduct object.

See Also

scale_product(), product_leaves(), product_size()

Examples

```
scale_product(a = scale_example(), b = scale_example2())
```

scale_aliases *Read the alias table*

Description

Read the alias table

Usage

```
scale_aliases(x, frame = NULL, source = NULL)
```

Arguments

x	A NestedScale.
frame	Restrict to one frame; NULL for all.
source	Restrict to one source (or several); NULL for all.

Value

A data.frame with columns frame, member, alias, source.

Examples

```
s <- attach_aliases_scale(
  scale_example(),
  data.frame(frame = "sector", member = "P", alias = "primary"),
  source = "labels"
)
scale_aliases(s)
scale_aliases(s, frame = "class")
```

scale_alias_property *An alias property for a subclass*

Description

Builds a getter/setter property that reads and writes an inherited property under a different name – how timescales' Calendar keeps @timeframes and geoscales' Geoscale keeps @geoframes working over the inherited @frames, so no existing code has to change when they become subclasses.

Usage

```
scale_alias_property(name)
```

Arguments

name	Name of the inherited property to alias.

Details

The setter is a no-op on `NULL`. S7's generated constructor writes every dynamic property once with `NULL` before the real values are set, so a setter that forwards `NULL` makes the class impossible to construct.

Value

An S7 property.

Examples

```
Aliased <- S7::new_class("Aliased",
  parent = scale_class(),
  properties = list(levels = scale_alias_property("frames"))
)
a <- Aliased(
  leaftable = scale_leaftable(scale_example()),
  frames = scale_frames(scale_example()),
  members = S7::prop(scale_example(), "members"),
  leaf = scale_leaf_column(scale_example())
)
a@levels
```

---

scale_ancestry *Ancestry between all frame pairs*

Description

Every `(coarser, finer)` code pair that shares at least one leaf, for all frame pairs.

Usage

```
scale_ancestry(x)
```

Arguments

`x`	A `NestedScale`.

Details

Computed **leaf-mediated**, directly from the leaftable – deliberately not as a transitive closure of `scale_family()`. Closing a hierarchy that cross-cuts manufactures false relationships: in the example scale, group `GB` straddles both sectors, so closing `sector -> class -> group -> item` would wrongly report sector `P` as an ancestor of member `U5`, which lies in sector `S`.

For frames that do not nest this relation is *overlap*, not containment – test a given pair with `scale_nests()`.

Frame columns are retained because codes are not unique across frames: in the example, `"G1"` exists at both `class` and `group`, so a bare `(parent, child)` pair would read as a self-loop.

Value

A `data.frame` with columns `parent_frame`, `parent`, `child_frame`, `child`.

Examples

```
head(scale_ancestry(scale_example()))
```

`scale_attributes` *Read attributes, resolved along the hierarchy*

Description

[Experimental] This feature may change or be removed in a later version.

With `frame = NULL`, returns the stored attribute table (filtered by `attribute` and `source`). With a `frame`, returns one row per member of that frame per attribute, **resolved**: a member's own value when it has one, otherwise the value of its nearest ancestor that has one, walking coarser frames one at a time. The row reports where the value came from in `origin_frame`/`origin_member`; a member with no value anywhere gets an `NA` row.

Usage

```
scale_attributes(
  x,
  attribute = NULL,
  frame = NULL,
  source = NULL,
  inherit = TRUE
)
```

Arguments

`x`	A `NestedScale`.
`attribute`	Character vector of attribute names; `NULL` for all.
`frame`	Frame whose members to resolve for; `NULL` returns the stored table unresolved.
`source`	Restrict to one source (or several). `NULL` uses whatever is stored and errors on a conflict.
`inherit`	Walk to coarser frames for members without a value of their own (default `TRUE`). `FALSE` reports stored values only.

Details

Two situations are errors rather than guesses:

- the walk reaches a frame where the member has **several** ancestors (the frames cross-cut) and at least one of them holds the attribute – "inheritance is ambiguous";
- several `sources` publish the attribute for the same member and `source` is `NULL` – pass `source=` to choose.

Value

A `data.frame` with the attribute columns (`frame`, `member`, `attribute`, `kind`, `value`, `lower`, `upper`, `unit_of_measure`, `source`, `note`), plus `origin_frame` and `origin_member` when `frame` is given.

Examples

```
s <- attach_attributes_scale(
  scale_example(),
  data.frame(frame = c("sector", "class"), member = c("P", "S1"),
             attribute = "density", value = c(10, 5),
             unit_of_measure = "t/m3"),
  source = "survey", kind = "property"
)
scale_attributes(s)                            # stored rows
scale_attributes(s, "density", frame = "item") # resolved per leaf
```

SCALE_ATTRIBUTE_KINDS *Kinds of attribute*

Description

[Experimental] This feature may change or be removed in a later version.

Every attribute name carries a kind, which `convert_scale()` consults:

Usage

```
SCALE_ATTRIBUTE_KINDS
```

Format

A character vector of length 2.

Details

`property` A physical property of the member, an invertible ratio: a calorific value in GJ/t converts tonnes to gigajoules and gigajoules back to tonnes; a density, a yield.

`factor` A one-way multiplier: an emission factor in kg/TJ turns energy into CO2, and there is no physical sense in which CO2 turns back into energy; a loss factor, a tax rate.

Value

A character vector of the two attribute kinds accepted in the `kind` column of `attach_attributes_scale()`: `"property"` and `"factor"`.

Examples

```
SCALE_ATTRIBUTE_KINDS
```

scale_axes *Axes of a product*

Description

[Experimental] This feature may change or be removed in a later version.

Usage

```
scale_axes(x, axis = NULL)
```

Arguments

x	A ScaleProduct.
axis	Optional single axis name, to get just that NestedScale.

Value

A named list of NestedScale objects, or one NestedScale when axis is given.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
names(scale_axes(p))
scale_axes(p, "a")
```

scale_catalog *Shipped scale definitions*

Description

Packages ship scale definitions as files (see write_scale()) and register the directory holding them; scale_catalog() lists what every registered directory holds and read_scale() takes an id from it. nestedscales itself ships no definitions.

Usage

```
register_scale_dir(path, package = NULL)

list_scale_dirs()

clear_scale_dirs()

scale_catalog()
```

Arguments

path	A directory of definition files or bundles.
package	The package registering it; NULL for a directory registered by hand (listed under package = NA).

Details

register_scale_dir() is what a dimension package calls in its .onLoad(): register_scale_dir(system.file("scales
package = pkgname), package = pkgname).

Value

scale_catalog() a data.frame with one row per definition: id, name, kind (the class, without its package prefix), leaf (the name of the leaf column), frames (joined by /), n_leaves, package, path. register_scale_dir() the registry key, invisibly; list_scale_dirs() a data.frame with package and path; clear_scale_dirs() NULL, invisibly.

Examples

```
scale_catalog()
```

---

scale_class *The scale classes, for subclassing and dispatch*

Description

These return the classes NestedScale and ScaleProduct, which are also exported under their own names. Needed to build a subclass (S7::new_class(..., parent = scale_class()), which is what scale_alias_property() is for) or to register an S7 method against one. To merely ask what an object is, use scale_is().

Usage

```
scale_class()

scale_product_class()
```

Value

The S7 class object.

Examples

```
scale_class()
S7::S7_inherits(scale_example(), scale_class())
```

scale_coverage *Coverage of a sampled NestedScale*

Description

The fraction of the root parent's weight that survives in this object. A scale that was never subset covers all of itself, so every weight reports 1.

Usage

```
scale_coverage(x, weight = NULL)
```

Arguments

x	A NestedScale.
weight	A single weight name, or NULL for all declared weights.

Value

A named numeric over the declared weights, or a single unnamed numeric when weight is given.

Examples

```
scale_coverage(scale_example())
```

scale_crosses *Do two frames form a product?*

Description

TRUE when every member of a combines with the same set of members of b, as the months and hours of a MONTH x HOUR calendar do: the frames overlay rather than nest, and the leaves are their combinations. The complement of scale_nests() for the other common shape; a pair can be neither.

Usage

```
scale_crosses(x, a, b)
```

Arguments

x	A NestedScale.
a, b	Frame names.

Value

A logical scalar. When FALSE, the attribute "offenders" names the members of a whose set of b members differs from the first one's.

Examples

```
s <- scale_example()
scale_crosses(s, "class", "group")

grid <- expand.grid(m = c("m1", "m2"), h = c("h1", "h2", "h3"),
                    stringsAsFactors = FALSE)
grid$item <- paste(grid$m, grid$h, sep = "_")
mh <- scale_from_leaftable(grid, frames = c("m", "h"), leaf = "item")
scale_crosses(mh, "m", "h")
scale_nests(mh, "m", "h")
```

---

scale_dataset_info *Describe a stored dataset without opening it fully*

Description

[Experimental] This feature may change or be removed in a later version.

Usage

```
scale_dataset_info(path)
```

Arguments

path	A folder written by write_scale_dataset().

Value

A list with kind, axes, keys, files and bytes.

Examples

```
# see write_scale_dataset()
```

`scale_distance` *Distance between the members of a scale*

Description

[Experimental] This feature may change or be removed in a later version.

Reshapes long, member-keyed data into a member-by-observation matrix and measures the distance between its rows.

Usage

```
scale_distance(
  data,
  x,
  frame = NULL,
  key = NULL,
  value = NULL,
  method = "euclidean",
  scale_members = FALSE
)
```

Arguments

`data`	Long data: a column of member codes, one or more identifier columns saying which observation each row is, and a value column.
`x`	A `NestedScale`.
`frame`	Frame whose members are compared. Defaults to the leaf frame.
`key`	Column of `data` holding the member codes. Defaults to `frame` when present, else the scale's leaf column.
`value`	Value column; inferred when unambiguous.
`method`	A name from `list_scale_distances()`.
`scale_members`	Standardise each member's profile to mean 0, sd 1 before measuring. Turns a level distance into a shape distance; has no effect on `"correlation"`, which already ignores level.

Value

A stats::dist object over the frame's members, in member order.

Examples

```
s <- scale_example()
d <- data.frame(
  item = rep(scale_members(s), each = 3),
  t = rep(sprintf("t%d", 1:3), 7), v = as.numeric(seq_len(21))
)
round(scale_distance(d, s, method = "correlation"), 2)
```

SCALE_DISTANCES *Built-in distances*

Description

[Experimental] This feature may change or be removed in a later version.

`euclidean`, `manhattan`, `maximum` Straight to `stats::dist()`. Compare members by the LEVEL of their profiles.

`correlation` `1 - cor(t(x))`, Pearson. Compares SHAPE: two members that rise and fall together are close however different their magnitudes, which is what you usually want when grouping load or generation profiles.

`spearman` The same on ranks – shape, robust to outliers.

`cosine` `1 - cosine similarity`. Shape again, but sensitive to sign and not centred, so a member that is uniformly positive is close to every other such member.

Usage

```
SCALE_DISTANCES
```

Format

A character vector.

Value

A character vector of the six built-in distance names accepted by the `method` argument of `scale_distance()`; `list_scale_distances()` adds any registered with `register_scale_distance()`.

Examples

```
SCALE_DISTANCES
```

scale_example *A small example NestedScale*

Description

A synthetic 3-frame hierarchy used in examples and tests. It deliberately reproduces three awkward features of real classification tables:

Usage

```
scale_example()
```

Details

- a code ("G1") reused at more than one frame, so bare codes are ambiguous and every lookup must name its frame;
- a non-nesting pair of frames – group "GB" draws members from two different classes, so class and group do not form a tree;
- a member ("OTH") with no code at any coarser frame (partial coverage).

Value

A NestedScale with 7 leaves and frames sector/class/group/item.

Examples

```
s <- scale_example()
s
scale_nests(s, "class", "group") # FALSE - they cross-cut
```

---

scale_example2 *A second small example NestedScale*

Description

A different dimension from scale_example(), for the examples that need two axes. Two frames, four leaves, its own leaf column (period) and a weight – deliberately well behaved, so the awkwardness in an example stays on the other axis.

Usage

```
scale_example2()
```

Value

A NestedScale with 4 leaves and frames era/period.

Examples

```
scale_example2()
scale_product(a = scale_example(), b = scale_example2())
```

scale_family *Immediate parent-child table between two frames*

Description

Immediate parent-child table between two frames

Usage

```
scale_family(x, parent = NULL, child = NULL)
```

Arguments

x	A NestedScale.
parent, child	Frame names. Defaults to every adjacent pair in x@frames.

Value

A data.frame with columns parent_frame, parent, child_frame, child. Leaves unassigned at either frame are omitted.

Examples

```
s <- scale_example()
scale_family(s, "group", "item")
```

scale_frames *Frames of a NestedScale*

Description

The hierarchy names, ordered coarsest first. The last entry is the leaf frame whenever one frame enumerates the leaves; when none does – a Calendar's leaves are combinations of its frames – the leaf column identifies them instead.

Usage

```
scale_frames(x, finest = FALSE)

## S3 method for class 'NestedScale'
names(x)
```

Arguments

x	A NestedScale.
finest	Return only the finest (leaf) frame.

Value

A character vector of frame names, or a single name when `finest = TRUE`.

Examples

```
s <- scale_example()
scale_frames(s)
scale_frames(s, finest = TRUE)
```

`scale_from_leaftable` *Build a NestedScale from a flat table of leaves*

Description

The general constructor. Takes a wide `data.frame` with one row per leaf (the finest member) and one column per frame, and returns a `NestedScale`.

Usage

```
scale_from_leaftable(
  leaftable,
  frames,
  leaf = NULL,
  weights = NULL,
  default_weight = NULL,
  members = NULL,
  name = "",
  desc = "",
  attributes = NULL,
  aliases = NULL,
  labels = NULL,
  ...
)
```

Arguments

`leaftable`	`data.frame` with one row per leaf, one column per name in `frames`, and optionally numeric weight columns.
`frames`	Ordered character vector of frame names, **coarsest first**. Each must be a column of `leaftable`.
`leaf`	Name of the leaf column: the column that identifies the leaves, one unique code per row. Defaults to a column called `"leaf"` if present, otherwise to the finest frame (the last entry of `frames`).
`weights`	Character vector naming the weight columns. Defaults to all numeric columns that are neither frames nor reserved names.

default_weight	The weight used when a caller does not name one. Defaults to the first entry of weights.
members	Optional named list giving the ordered code vocabulary per frame. Derived from leaftable when NULL (first-appearance order).
name, desc	Short name and description.
attributes	Optional table of per-member physical attributes keyed by (frame, member); see attach_attributes_scale() for the columns.
aliases	Optional table of alternative labels keyed by (frame, member); see attach_aliases_scale() for the columns.
labels	Optional table of display names keyed by (frame, member, variant); see attach_labels_scale() for the columns.
...	Further named entries merged into meta (e.g. source).

Details

Blank strings ("") in frame columns are normalised to NA, meaning "this leaf has no code at this frame" – partial coverage is normal in real classification tables.

Value

A NestedScale.

Examples

```
df <- data.frame(
  section = c("A", "A", "A", "B"),
  group   = c("A1", "A1", "A2", "B1"),
  item    = c("A11", "A12", "A21", "B11"),
  gva     = c(100, 200, 300, 400)
)
scale_from_leaftable(df, frames = c("section", "group", "item"))
```

---

scale_id *The identity of a scale*

Description

A short string that says which scale an object is: its name and a hash of its structure, "<name>@<hash>". The hash covers the leaf column, the frames and the leaf-to-frame memberships; weights, attributes, aliases, labels and the order of rows or members do not enter, so adding a weight keeps data bound to the scale valid, while a subset or a re-grouping is a different scale. A data link compares the hash alone: two objects with the same structure under different names are the same scale to it, and the name is kept for messages.

Usage

```
scale_id(x)
```

Arguments

x	A NestedScale.

Details

The hash is the first eight hexadecimal digits of the MD5 of a canonical UTF-8 text: the line key=<leaf column name> (the prefix key= is part of the hashed text and is kept so that identities do not change), the line frames= with the frame names tab-separated, then one tab-separated line per leaf (its code in the leaf column first, then its code at each frame, empty where unassigned), leaves sorted by the leaf column in the C locale, lines joined by "\n".

Value

A single string.

See Also

bind_scale()

Examples

```
scale_id(scale_example())
```

---

scale_is *Is this a scale, or a product of scales?*

Description

These are how calling code asks what an object is. Build a scale with scale_from_leaftable() and a product with scale_product().

Usage

```
scale_is(x)

scale_is_product(x)
```

Arguments

x	Any object.

Details

Named with the object prefix rather than is_scale(), which ggplot2 already exports.

Value

A single TRUE or FALSE. A product is NOT a scale: it holds several and materialises no leaves of its own.

Examples

```
scale_is(scale_example())
scale_is(data.frame(item = "U1"))
scale_is_product(scale_product(a = scale_example(), b = scale_example2()))
```

`scale_is_uniform` *Are the members of a frame the same size?*

Description

`TRUE` when every member of `frame` holds the same number of leaves, or the same total of a weight column: the dimension-free form of a "regular" calendar (equal durations) or an equal-area regionalisation. Leaves with no code at `frame` are ignored.

Usage

```
scale_is_uniform(x, frame, weight = NULL, tolerance = 1e-09)
```

Arguments

`x`	A `NestedScale`.
`frame`	A frame name.
`weight`	`NULL` compares leaf counts; a weight column name compares its per-member totals.
`tolerance`	Relative tolerance for weight totals.

Value

A logical scalar. When `FALSE`, the attribute `"sizes"` holds the per-member counts or totals.

Examples

```
s <- scale_example()
scale_is_uniform(s, "class")                 # two leaves each
scale_is_uniform(s, "class", weight = "size") # but not the same size
```

scale_label *Look up the labels of members*

Description

Returns one display name per code. A code without a label of the chosen variant is returned unchanged, so the result is always usable as an axis or legend text. `NA` in `members` stands for the frame itself and returns the frame's label, or its name when it has none.

Usage

```
scale_label(x, members, frame, variant = NULL)
```

Arguments

`x`	A `NestedScale`.
`members`	Character vector of member codes at `frame`.
`frame`	The frame the codes are at.
`variant`	The label variant. `NULL` uses the scale's default, see `attach_labels_scale()`.

Value

A character vector as long as `members`.

See Also

`attach_labels_scale()`, `scale_labels()`

Examples

```
s <- attach_labels_scale(
  scale_example(),
  data.frame(frame = "sector", member = c("P", "S", NA),
             label = c("Primary", "Secondary", "Sector"))
)
scale_label(s, c("S", "P", "T"), frame = "sector")
scale_label(s, NA, frame = "sector")
scale_label(s, c("G1", "G2"), frame = "class")   # no labels: the codes
```

scale_labels *Read the label table*

Description

Read the label table

Usage

```
scale_labels(x, frame = NULL, variant = NULL)
```

Arguments

x	A NestedScale.
frame	Restrict to one frame; NULL for all.
variant	Restrict to one variant (or several); NULL for all.

Value

A data.frame with columns frame, member, variant, label. A row with member = NA is the label of the frame itself.

See Also

attach_labels_scale(), scale_label()

Examples

```
s <- attach_labels_scale(
  scale_example(),
  data.frame(frame = "sector", member = c("P", "S"),
             label = c("Primary", "Secondary"))
)
scale_labels(s)
scale_labels(s, frame = "class")
```

scale_leaftable *The leaftable of a NestedScale*

Description

The one-row-per-leaf table the scale is built on, as a plain data.frame – the exported accessor to prefer over reaching for x@leaftable. as.data.frame() and ggplot2::fortify() on a NestedScale are equivalent, so ggplot(s) + geom_*() pipelines work directly.

Usage

```
scale_leaftable(x)

## S3 method for class 'NestedScale'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class '`nestedscales::NestedScale`'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	A NestedScale.
row.names, optional	Ignored (S3 signature compatibility).
...	Ignored.

Value

A data.frame: one row per leaf, with the frame columns plus any weight columns.

Examples

```
head(scale_leaftable(scale_example()))
head(as.data.frame(scale_example()))
```

---

scale_leaf_column *The name of the leaf column*

Description

The column of the leaftable that identifies the leaves.

Usage

```
scale_leaf_column(x)
```

Arguments

x	A NestedScale.

Value

A single string.

Examples

```
scale_leaf_column(scale_example())
```

scale_leaf_frame *The frame whose members are the leaves*

Description

The finest frame when one code of it sits on each leaftable row, and the leaf column otherwise. A `NestedScale` built from a nested hierarchy is the first kind; a timescales `Calendar` is the second, because its timeslices are combinations of its frames rather than codes of any one of them.

Usage

```
scale_leaf_frame(x)
```

Arguments

x	A `NestedScale`.

Details

Needed by the dimension packages: it is the `frame` to pass to `join_scale()` or `recast_scale()` for data keyed at the leaves.

Value

A single string: the name of the frame whose members are the leaves, or of the leaf column when no frame is.

Examples

```
scale_leaf_frame(scale_example())
```

scale_leaf_pairs *Leaf-level correspondence between two frames*

Description

The seam a dimension package overrides when its leaf layer is generated rather than enumerated. Returns one row per leaf (or per leaf and extra identifier, e.g. per grid instant and year).

Usage

```
scale_leaf_pairs(x, ...)
```

Arguments

x	A NestedScale.
...	Method arguments: source and target (frame names of x), weight (a weight column, or NULL for the scale's default), and whatever else the dimension needs to generate its leaves – year for a calendar.

Value

A data.frame with columns source, target, w, plus any extra identifier columns the dimension carries. The extra columns must be named in the by argument of the caller.

Examples

```
head(scale_leaf_pairs(scale_example(), "class", "group"))
```

---

scale_map *Crosswalk between two frames of a scale*

Description

Materialises the from -> leaves -> to route as a table: one row per pair of overlapping members with

Usage

```
scale_map(x, source, target, weight = NULL, by = character(), ...)
```

Arguments

x	The NestedScale the frames belong to.
source, target	Frame names of x.
weight	Weight column for w. NULL uses the default weight; when the object declares no weights at all, every leaf gets weight 1 (an equal split).
by	Extra identifier columns carried through from scale_leaf_pairs() (e.g. "year").
...	Passed to scale_leaf_pairs().

Details

- n_from – leaves in the source member (its full set, before any target coverage is considered),
- n_overlap – leaves the pair shares,
- w – the weight of the overlap (summed leaf weights, chosen weight column), the quantity "weighted_mean" aggregation uses,
- w_from – the full weight of the source member; w / w_from is the split share "sum" disaggregation uses.

The two label columns are named by the frames; rows with an NA target label are leaves target does not cover. A crosswalk registered with register_scale_map() is returned as-is instead of being derived.

Value

A data.frame with columns <from>, <to> (NA = uncovered by target), n_from, n_overlap, w, w_from, plus any by columns.

See Also

scale_map_between() for the map between two scales.

Examples

```
s <- scale_example()
scale_map(s, "class", "group")
scale_map(s, "sector", "class", weight = "count")
```

---

scale_map_between *Crosswalk between two scales through their shared leaves*

Description

The counterpart of scale_map() for two scales: leaves are matched on their codes, so a member of source overlaps a member of target where they contain the same leaves. Same columns as scale_map(), with the two label columns named after the scales. Leaves of source absent from target get an NA target (with a warning). A crosswalk registered with register_scale_map_between() is returned as-is instead of being derived.

Usage

```
scale_map_between(source, target, weight = NULL)
```

Arguments

source, target	Two named NestedScale objects whose leaf codes overlap.
weight	Weight column of source for w; NULL uses its default weight, or weight 1 per leaf when it declares none.

Value

A data.frame with columns <from name>, <to name>, n_from, n_overlap, w, w_from.

Examples

```
a <- scale_example()
b <- scale_from_leaftable(
  data.frame(
    big = c("X", "X", "Y", "Y", "Y", "Y", "Z"),
    item = c("U1", "U2", "U3", "U4", "U5", "U6", "OTH")
  ),
  frames = c("big", "item"), name = "other"
)
scale_map_between(a, b)
```

scale_match *Match labels to members, strictly*

Description

Resolves each label to a `(frame, member)` member in two stages, stopping at the first that hits: an **exact** member name (case-insensitive), then an **alias** (case-insensitive). A label that hits several members – the same name at two frames, or an alias shared by two members – is reported as `"ambiguous"` with the candidates, unless `frame=` narrows the search to one frame. A label that hits nothing is `"none"`. Nothing is guessed: no fuzzy matching, no substring matching.

Usage

```
scale_match(x, labels, frame = NULL, source = NULL)
```

Arguments

`x`	A `NestedScale`.
`labels`	Character vector of labels to resolve.
`frame`	Restrict matching to one frame; `NULL` searches all frames and the leaf column.
`source`	Restrict the alias stage to one source (or several).

Value

A `data.frame` with one row per label: `label`, `frame`, `member`, `method` (`"exact"`, `"alias"`, `"ambiguous"`, `"none"`) and `candidates` (the `frame/member` pairs of an ambiguous hit, joined by `"; "`, else `NA`).

Examples

```
s <- scale_example()
scale_match(s, c("P", "G1", "u1", "nope"))   # G1 is a class AND a group
scale_match(s, "G1", frame = "group")
```

scale_members *Members of a NestedScale*

Description

The code vocabulary at one frame, in canonical order.

Usage

```
scale_members(x, frame = NULL)
```

Arguments

x	A NestedScale.
frame	A single frame name; defaults to the finest frame (the leaves).

Value

A character vector of member codes.

Examples

```
s <- scale_example()
scale_members(s)
scale_members(s, "group")
```

---

scale_navigate *Navigate a NestedScale's hierarchy*

Description

Codes related to member at another frame, found through the leaves. scale_children()/scale_parents() step one frame down/up by default; scale_descendants()/scale_ancestors() report every finer/coarser frame as a frame-tagged table.

Usage

```
scale_children(x, frame, member, target = NULL)

scale_parents(x, frame, member, target = NULL)

scale_descendants(x, frame, member, target = NULL)

scale_ancestors(x, frame, member, target = NULL)
```

Arguments

x	A NestedScale.
frame	The frame member belongs to.
member	Character vector of codes.
target	Target frame. NULL uses the adjacent frame (children/parents) or every finer/coarser frame (descendants/ancestors).

Value

A character vector (children/parents) or a data.frame with columns frame and member (descendants/ancestors).

Examples

```
s <- scale_example()
scale_children(s, "sector", "P")
scale_parents(s, "item", "U5")
scale_descendants(s, "sector", "P")
scale_ancestors(s, "item", "U5")
```

`scale_nests` *Do two frames nest?*

Description

Tests whether every code at the finer frame falls entirely within a single code at the coarser frame. Real hierarchies often fail this: a spatial code can straddle two parents, and a reporting classification can assign one detailed class to several aggregates.

Usage

```
scale_nests(x, parent, child)
```

Arguments

`x`	A `NestedScale`.
`parent`, `child`	Frame names.

Details

Nesting is *not* required by `NestedScale` – conversion routes through the leaf layer and works either way. This function is a diagnostic.

Value

`TRUE` or `FALSE`. When `FALSE`, the offending child codes are attached as the `"offenders"` attribute.

Examples

```
s <- scale_example()
scale_nests(s, "class", "group") # FALSE - they cross-cut
```

scale_payload_collapse *Collapse a subclass's per-leaf payload*

Description

The companion of `scale_payload_slice()` for the rebuilds that MERGE leaves rather than keep a subset of them – `prune_scale()`, which makes a coarser frame the new leaf layer. Called with `map`, an integer vector with one entry per row of the OLD leaftable giving the index of the NEW row that leaf was merged into, or `NA` for a leaf that was dropped.

Usage

```
scale_payload_collapse(x, ...)
```

Arguments

`x`	A `NestedScale`.
`...`	Method arguments: `map`, the integer vector described above.

Details

The `NestedScale` default returns `x` unchanged, which is right for a payload keyed by member codes (the core's own attribute and alias tables are re-keyed by the engine itself). A subclass whose payload is positional – one geometry per leaftable row – must override it and merge its payload by `map`; if it does not, its own validator rejects the length mismatch, so the failure is loud rather than a silently misaligned payload.

Value

`x`, with any payload merged to the new rows.

Examples

```
identical(
  scale_payload_collapse(scale_example(), c(1L, 1L, 2L, 2L, 3L, 3L, NA)),
  scale_example()
)
```

scale_payload_io *Write and read a subclass's payload in a definition file*

Description

The seam a dimension package uses to carry what the core does not know about – geoscales' geometry – in a scale definition written by `write_scale()` and read by `read_scale()`.

Usage

```
scale_write_payload(x, ...)

scale_read_payload(x, ...)

register_scale_class(class)

list_scale_classes()
```

Arguments

`x`	A `NestedScale` (the subclass instance when writing; the core object when reading).
`...`	Method arguments: `dir`, the bundle directory or `NULL`, and `header`, the header list.
`class`	An S7 class that inherits from `scale_class()`.

Details

`scale_write_payload(x, dir, header)` is called after the core tables are written, with the bundle directory (`NULL` for a single-file definition, which can hold inline data only) and the header so far; it writes its files and returns the header with its entries added under `payload`. `scale_read_payload(x, dir, header)` is called with the CORE scale rebuilt from the header and returns the subclass object. The `NestedScale` defaults return the header and `x` unchanged.

`read_scale()` finds the method through the class recorded in the header, so a dimension package registers its class once, in `.onLoad()`, with `register_scale_class()`.

Value

`scale_write_payload()` the header; `scale_read_payload()` a scale; `register_scale_class()` the registry key, invisibly; `list_scale_classes()` a character vector of registered class names.

Examples

```
list_scale_classes()
```

scale_payload_parts *Describe the parts a subclass adds to a scale*

Description

The seam a dimension package overrides so that `scale_tree()` can mark the properties it adds and name its own verbs. The method returns a data.frame with a `part` column and any of `parent`, `kind`, `shape`, `mark`, `verb` and `note`. A row whose `part` and `parent` match a part the core already lists replaces the fields it gives; any other row is added. A top-level part is a property of the object; `parent` names the property a part belongs to. `mark` is one of `"fixed"`, `"derived"` or `"mutable"`. The `NestedScale` default adds nothing.

Usage

```
scale_payload_parts(x, ...)
```

Arguments

`x`	A `NestedScale`.
`...`	Not used.

Value

A data.frame of parts, or `NULL`.

See Also

`scale_tree()`

Examples

```
scale_payload_parts(scale_example())
```

scale_payload_slice *Subset a subclass's per-leaf payload*

Description

The seam a dimension package overrides when it carries per-leaf data alongside the leaftable – geoscales' `Geoscale` and its geometry. Called whenever the engine rebuilds an object from a row subset, with the kept row indices. The `NestedScale` default has no payload and returns `x` unchanged.

Usage

```
scale_payload_slice(x, ...)
```

Arguments

x	A NestedScale.
...	Method arguments: i, the integer vector of kept leaftable row indices.

Value

x, with any payload subset to the kept rows.

Examples

```
identical(
  scale_payload_slice(scale_example(), 1:3),
  scale_example()
)
```

---

scale_product *Combine scales into a product index*

Description

[Experimental] This feature may change or be removed in a later version.

Usage

```
scale_product(..., name = "", desc = "", keys = NULL, joint_weights = NULL)
```

Arguments

...	Two or more named NestedScale objects; the argument names become the axis names.
name, desc	Short name and description of the product.
keys	Optional named character vector overriding the data key column per axis. By default each axis uses the name of its scale's leaf column (scale_leaf_column()); if that would make two axes share a column name, the AXIS NAMES are used instead.
joint_weights	Reserved; see ScaleProduct.

Value

A ScaleProduct.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
p
product_size(p)
```

scale_rank *Frame rank*

Description

Position of a frame in the hierarchy: 1 is the coarsest.

Usage

```
scale_rank(x, frame)
```

Arguments

x	A NestedScale.
frame	Character vector of frame names.

Value

An integer vector of ranks; NA for names that are not frames of x.

Examples

```
s <- scale_example()
scale_rank(s, c("group", "item"))
```

scale_residuals *Members that are residuals*

Description

A residual carries a real value but is not a destination for allocation: non-regionalized GDP, taxes that belong to no industry, an "n.e.c." bucket. It aggregates upward like any other member – that is the point, it is what makes the parent total reconcile – but recast_scale() never splits a coarse figure INTO one.

Usage

```
scale_residuals(x, frame = NULL)
```

Arguments

x	A NestedScale.
frame	A single frame name for just that frame's residuals, or NULL for the whole named list.

Details

Declare them when building the scale, per frame, because codes are not unique across frames: `scale_from_leaftable(..., residuals = list(item = "DE_XR"))`.

Value

A character vector of member codes, or the named list over frames. Empty when the scale declares none.

Examples

```
s <- scale_example()
scale_residuals(s) # none declared

lf <- scale_leaftable(s)
r <- scale_from_leaftable(
  lf,
  frames = scale_frames(s), leaf = "item",
  weights = scale_weights(s), name = "with_residual",
  residuals = list(item = "OTH")
)
scale_residuals(r, "item")
```

SCALE_RULES *Supported aggregation rules*

Description

Each rule defines behaviour in **both** directions. Direction is taken from the frame ranks, so aggregation and disaggregation are one operation:

Usage

```
SCALE_RULES
```

Format

A character vector of length 7.

Details

`sum` Up: sum. Down: split proportionally to the weight. For extensive quantities (capacity, demand, area, population).

`weighted_mean` Up: weight-weighted mean. Down: copy unchanged. For intensive quantities (efficiency, price, capacity factor).

`mean` Up: unweighted mean. Down: copy unchanged.

`copy` Up: the common value, erroring if it is not constant. Down: copy unchanged. For scale-invariant scalars.

`sd` Up: standard deviation over the leaves (aggregation only; going down it degenerates to `NA` for single-leaf groups).

`share` Share within parent: each source member's value divided by the total over its parent group. Unlike every other rule the result stays keyed at the **source** frame – the recast target (or `parent=`) names the parent – so it cannot be mixed with other rules in one call (the two share rules mix freely, being one computation). For building distribution keys and normalised profiles; requires `from` to nest within the parent.

`logshare` The same computation as `share` – the values ARE shares – but figures draw it on a fixed log10 percent scale (0.01%..100%), where `share` gets a fixed linear 0..1 scale.

Value

A character vector of the seven rule names accepted by the `rule` argument of `recast_scale()` and `register_scale_rule()`.

Examples

```
SCALE_RULES
```

`scale_share` *Weight shares within a frame*

Description

Normalised weights, either of the whole object or within each parent group.

Usage

```
scale_share(x, frame, weight = NULL, within = NULL)
```

Arguments

`x`	A `NestedScale`.
`frame`	Frame to report shares for.
`weight`	Weight column. `NULL` uses the default.
`within`	Optional coarser frame to normalise within. `NULL` normalises over the whole object.

Value

A `data.frame` with a code column named `frame`, the weight, and `share`. When `within` is given, a column of that name carries the parent code.

Examples

```
s <- scale_example()
scale_share(s, "class", weight = "size")
scale_share(s, "class", weight = "size", within = "sector")
```

scale_tree *The parts of a scale and which of them may change*

Description

Lists the parts of a scale as a tree: its properties, the columns of the leaftable, and whatever a subclass adds. Each part carries one of three marks, defined by the identity of the scale (scale_id()), which is computed from the leaf column, the frame names and the frame codes:

Usage

```
scale_tree(x)

## S3 method for class 'scale_tree'
print(x, ascii = !isTRUE(l10n_info()[["UTF-8"]]), ...)
```

Arguments

x	A NestedScale or a ScaleProduct; in print(), the table scale_tree() returned.
ascii	Draw the branches with +, - and |. The default does so outside a UTF-8 session.
...	Ignored.

Details

- fixed: the part enters the identity. Changing it gives another scale, which data bound to the first one no longer matches; the route is to build a new scale.
- derived: the part is computed from another one and is never set.
- mutable: the part can change and the scale stays the same scale. The function that changes it is named where there is one.

The marks describe the object; they are not enforced on assignment. str() on a scale prints the same tree.

Value

A data.frame of class scale_tree, one row per part: part, the parent property it belongs to ("" for a property), its kind, shape, mark, the verb that changes it and a note. Printing it draws the tree.

See Also

scale_id(), scale_payload_parts()

Examples

```
scale_tree(scale_example())
tr <- scale_tree(scale_example())
tr$part[tr$mark == "fixed"]
scale_tree(scale_product(a = scale_example(), b = scale_example2()))
```

`scale_vocab` *The dimension vocabulary of a scale*

Description

Engine messages are written once and interpolated with the words of the dimension at hand, so a `Calendar` flowing through the shared engine still errors in the language of time ("timeframe", "timeslice") and a `Geoscale` in the language of space ("geoframe", "region"). Subclasses supply their own method; the `NestedScale` default is the neutral vocabulary.

Usage

```
scale_vocab(x, ...)
```

Arguments

`x`	A `NestedScale`.
`...`	Passed to methods.

Value

A named list with elements `object`, `frame`, `frames`, `member`, `members`, `leaves`, and optionally `register_rule` and `register_map`: the calls that error messages suggest for registering a value column's rule or an explicit crosswalk, when the dimension package offers its own (`"register_geoscale_rule()"`).

Examples

```
scale_vocab(scale_example())
```

`scale_weights` *Weight columns of a NestedScale*

Description

Weight columns of a NestedScale

Usage

```
scale_weights(x)
```

Arguments

x	A NestedScale.

Value

A character vector of weight column names (possibly empty).

Examples

```
scale_weights(scale_example())
```

---

str.NestedScale *Show the structure of a scale*

Description

Prints the tree of scale_tree(): the properties of the object, the columns of its leaftable and what a subclass adds, each with its size and whether it is fixed, derived or mutable. The properties themselves are read with @ or S7::prop(); S7::prop_names() gives their names.

Usage

```
## S3 method for class 'NestedScale'
str(object, ...)

## S3 method for class '`nestedscales::NestedScale`'
str(object, ...)

## S3 method for class '`nestedscales::ScaleProduct`'
str(object, ...)
```

Arguments

object	A NestedScale or a ScaleProduct.
...	Passed to the print method of scale_tree().

Value

object, invisibly.

See Also

scale_tree()

Examples

```
str(scale_example())
S7::prop_names(scale_example())
```

summary.NestedScale *Summarize a NestedScale*

Description

Complements print() with the quantitative view: per-weight totals and coverage, and the adjacent-frame nesting table. Returns a "summary_Scale" object (a list) with its own print method.

Usage

```
## S3 method for class 'NestedScale'
summary(object, ...)

## S3 method for class '`nestedscales::NestedScale`'
summary(object, ...)

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

Arguments

object	A NestedScale.
...	Ignored.
x	A "summary_Scale" object (the print method's argument).

Value

summary() returns a list of class "summary_Scale": name, desc, frames (named member counts), unassigned (named NA-leaf counts), n_leaves, weights, weight_totals, default_weight, coverage (see scale_coverage()), sampled, parent_name, nesting (adjacent-pair table with offender counts, see scale_nests()), attributes (a data.frame of row counts by attribute, frame and source), aliases (counts by source), labels (counts by frame and variant), source, vocab.

Examples

```
summary(scale_example())
```

summary.ScaleProduct *Summarize a product*

Description

Summarize a product

Usage

```
## S3 method for class 'ScaleProduct'
summary(object, ...)

## S3 method for class '`nestedscales::ScaleProduct`'
summary(object, ...)

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

Arguments

object	A ScaleProduct.
...	Ignored.
x	A "summary_ScaleProduct" object (the print method's argument).

Value

A list of class "summary_ScaleProduct": name, desc, axes (per-axis summary rows), total, coverage, joint_coverage, joint_weights.

Examples

```
summary(scale_product(a = scale_example(), b = scale_example2()))
```

write_scale *Write a scale as a definition file, and read one back*

Description

A definition is a header (JSON, the stored form; YAML on request) plus tables. A small scale fits in **one file** with every table inline; a large one is a **bundle** directory holding scale.json and its tables as Arrow IPC files (or CSV when arrow is not installed). Either form is language-neutral: the header is plain JSON and the tables are Arrow or CSV with their column types recorded.

Usage

```
write_scale(
  x,
  path,
  format = c("json", "yaml"),
  tables = c("auto", "inline", "arrow", "csv"),
  overwrite = FALSE
)

read_scale(path, package = NULL, class = c("stored", "core"))
```

Arguments

x	A NestedScale.
path	A file (.json, .yaml, .yml) or a bundle directory; for read_scale() also a catalogue id.
format	Header format: "json" (default) or "yaml". A single-file path fixes it by its extension.
tables	Where the tables go: "auto" (inline when the tables hold at most 1000 rows in all, else "arrow" when arrow is installed, else "csv"), "inline", "arrow" or "csv". A single file can only inline.
overwrite	Replace a definition already at path.
package	For a catalogue id: the package whose definition to take when several packages ship the same id.
class	"stored" (default) returns the class recorded in the header; "core" the plain NestedScale.

Details

path ending in .json, .yaml or .yml writes a single file; any other path is a bundle directory. A bundle is only ever written into an empty directory or over a bundle written before (overwrite = TRUE); nothing else in a directory is touched.

read_scale() takes a file, a bundle directory, or the id of a shipped definition (see scale_catalog()). The scale comes back as the class it was written from when that class is registered (see register_scale_class()); class = "core" asks for the plain NestedScale instead, dropping any payload.

Value

write_scale() returns path, invisibly. read_scale() returns the scale.

Round trips

An arrow bundle reads back identical. A JSON or YAML header carries numbers at 15 significant digits, so an inline table reads back equal (all.equal()), not necessarily identical, for arbitrary doubles.

Examples

```
s <- scale_example()
f <- file.path(tempdir(), "example.json")
write_scale(s, f, overwrite = TRUE)
back <- read_scale(f)
identical(scale_frames(back), scale_frames(s))
all.equal(scale_leaftable(back), scale_leaftable(s))

# a bundle, with the tables as files
if (requireNamespace("arrow", quietly = TRUE)) {
  d <- file.path(tempdir(), "example.scale")
  write_scale(s, d, tables = "arrow", overwrite = TRUE)
  list.files(d)
  identical(scale_leaftable(read_scale(d)), scale_leaftable(s))
}
```

---

write_scale_dataset *Write data and the scales that index it to one folder*

Description

[Experimental] This feature may change or be removed in a later version.

The data is written as a hive-partitioned Arrow IPC dataset and the scales travel with it, so `open_scale_dataset()` gives back both and the result can be recast without re-declaring anything. Each scale is stored as the definition `write_scale()` writes, so its attributes and aliases travel too.

Usage

```
write_scale_dataset(
  data,
  x,
  path,
  partitioning = NULL,
  compression = "zstd",
  overwrite = FALSE
)
```

Arguments

`data`	The table to write, in any supported backend.
`x`	The `NestedScale` or `ScaleProduct` the data is indexed by.
`path`	Directory to create. It must not already contain a store unless `overwrite = TRUE`.
`partitioning`	Columns to partition the data by (hive style). `NULL` (default) picks nothing; a good choice is a low-cardinality identifier such as a year, which lets a filtered read skip whole files.

compression	Codec for the IPC files, e.g. "zstd" (default), "lz4" or "uncompressed". If the codec is unavailable in the installed Arrow build, the files are written uncompressed with a warning.
overwrite	Replace an existing store at path.

Value

path, invisibly.

Examples

```
p <- scale_product(a = scale_example(), b = scale_example2())
d <- merge(
  data.frame(item = c("U1", "U2")),
  data.frame(period = c("p1", "p2"))
)
d$v <- 1:4
dir <- file.path(tempdir(), "store")
if (requireNamespace("arrow", quietly = TRUE)) {
  write_scale_dataset(d, p, dir, overwrite = TRUE)
  ds <- open_scale_dataset(dir)
  ds
}
```

[.NestedScale *Subset a NestedScale*

Description

x[frame, member] is filter_scale().

Usage

```
## S3 method for class 'NestedScale'
x[i, j, ...]

## S3 method for class '`nestedscales::NestedScale`'
x[i, j, ...]
```

Arguments

x	A NestedScale.
i	Frame name.
j	Member codes to keep.
...	Unused.

Value

A NestedScale.

Examples

```
scale_example()["sector", "P"]
```

Index

[.NestedScale, 85
[.nestedscales::NestedScale
(.NestedScale), 85

as.data.frame.NestedScale
(scale_leaftable), 63
as.data.frame.nestedscales::NestedScale
(scale_leaftable), 63
as_scale_array, 4
as_scale_array(), *4*
as_scale_member, 5
as_scale_member(), *11*
as_scale_table (as_scale_array), 4
attach_aliases_scale, 6
attach_aliases_scale(), *8*, *25*, *59*
attach_attributes_scale, 7
attach_attributes_scale(), *25*, *49*, *59*
attach_labels_scale, 8
attach_labels_scale(), *24, 25*, *59*, *62, 63*

backend, 9
backend_bind_rows (backend), 9
backend_is_lazy (backend), 9
backend_lazy (backend), 9
backend_of (backend), 9
backend_pull (backend), 9
backend_restore (backend), 9
backend_schema (backend), 9
bind_scale, 10
bind_scale(), *5, 6*, *60*

clear_scale_dirs (scale_catalog), 50
clear_scale_distances
(register_scale_distance), 42
clear_scale_maps, 12
clear_scale_rules, 12
cluster_contiguous, 13
cluster_contiguous(), *17*
cluster_medoids, 14
cluster_scale, 15
cluster_scale(), *14*, *17*
cluster_sweep, 17
convert_scale, 18
convert_scale(), *49*

filter_product, 19
filter_scale, 20
filter_scale(), *19, 20*, *85*

get_scale_distance
(register_scale_distance), 42
get_scale_map (register_scale_map), 43
get_scale_map_between
(register_scale_map), 43
get_scale_rule, 21
get_scale_rule(), *33*, *36*

is_scale_member (as_scale_member), 5
is_valid_frame, 22

join_product, 22
join_scale, 23
join_scale(), *10*, *20*, *22*, *65*

list_scale_classes (scale_payload_io),
72
list_scale_dirs (scale_catalog), 50
list_scale_distances
(register_scale_distance), 42
list_scale_distances(), *13*, *16, 17*, *54, 55*
list_scale_maps (register_scale_map), 43
list_scale_maps(), *12*
list_scale_rules, 24

names.NestedScale (scale_frames), 57
NestedScale, *4–8*, *11*, *18*, *20, 21*, *23*, 25, *26*,
30, 31, *34–36*, *39, 40*, *43*, *45–48*,
50–52, *56–75*, *77–81*, *83–86*

open_scale_dataset, 26
open_scale_dataset(), *11*, *84*

print(), *81*
print.scale_dataset
(open_scale_dataset), 26
print.scale_tree (scale_tree), 78
print.summary_Scale
(summary.NestedScale), 81
print.summary_ScaleProduct
(summary.ScaleProduct), 82
product_coverage, 27
product_frames, 27
product_keys, 28
product_keys(), *28*, *34*
product_leaves, 28
product_leaves(), *45*
product_size, 29
product_size(), *45*
product_weights, 30
prune_scale, 30
prune_scale(), *71*

read_scale (write_scale), 82
read_scale(), *50*, *72*
recast, 31
recast_crosswalk, 32
recast_from_leaves (recast_to_leaves),
38
recast_product, 34
recast_product(), *31*
recast_scale, 35
recast_scale(), *10*, *29*, *31–33*, *35*, *38, 39*,
41, *43, 44*, *65*, *75*, *77*
recast_to_leaves, 38
reconcile_scale, 40
register_scale_class
(scale_payload_io), 72
register_scale_class(), *83*
register_scale_dir (scale_catalog), 50
register_scale_distance, 42
register_scale_distance(), *55*
register_scale_map, 43
register_scale_map(), *35*, *66*
register_scale_map_between
(register_scale_map), 43
register_scale_map_between(), *67*
register_scale_rule, 44
register_scale_rule(), *77*

scale_alias_property, 46
scale_alias_property(), *51*
scale_aliases, 46
scale_ancestors (scale_navigate), 69
scale_ancestry, 47
SCALE_ATTRIBUTE_KINDS, *7*, 49
scale_attributes, 48
scale_attributes(), *7*, *18*
scale_axes, 50
scale_catalog, 50
scale_catalog(), *83*
scale_children (scale_navigate), 69
scale_class, 51
scale_class(), *72*
scale_coverage, 52
scale_coverage(), *20*, *81*
scale_crosses, 52
scale_dataset_info, 53
scale_descendants (scale_navigate), 69
scale_distance, 54
scale_distance(), *13*, *16, 17*, *55*
SCALE_DISTANCES, 55
scale_example, 55
scale_example(), *56*
scale_example2, 56
scale_family, 57
scale_family(), *47*
scale_frames, 57
scale_frames(), *26*
scale_from_leaftable, 58
scale_from_leaftable(), *26*, *60*
scale_id, 59
scale_id(), *5*, *11*, *78*
scale_is, 60
scale_is(), *51*
scale_is_product (scale_is), 60
scale_is_uniform, 61
scale_label, 62
scale_label(), *8*, *63*
scale_labels, 63
scale_labels(), *8*, *62*
scale_leaf_column, 64
scale_leaf_column(), *74*
scale_leaf_frame, 65
scale_leaf_pairs, 65
scale_leaf_pairs(), *66*
scale_leaftable, 63
scale_link (bind_scale), 10
scale_link(), *6*, *26*
scale_map, 66

scale_map(), *37*, *43*, *67*
scale_map_between, 67
scale_map_between(), *43*, *67*
scale_match, 68
scale_match(), *6*
scale_member, *10*
scale_members, 68
scale_members(), *26*
scale_navigate, 69
scale_nests, 70
scale_nests(), *47*, *52*, *81*
scale_parents *(*scale_navigate*)*, 69
scale_payload_collapse, 71
scale_payload_collapse(), *31*
scale_payload_io, 72
scale_payload_parts, 73
scale_payload_parts(), *78*
scale_payload_slice, 73
scale_payload_slice(), *71*
scale_product, 74
scale_product(), *45*, *60*
scale_product_class *(*scale_class*)*, 51
scale_rank, 75
scale_read_payload *(*scale_payload_io*)*, 72
scale_residuals, 75
scale_residuals(), *25*
SCALE_RULES, *33*, *36*, *44*, 76
scale_share, 77
scale_tree, 78
scale_tree(), *73*, *80*
scale_vocab, 79
scale_weights, 79
scale_write_payload *(*scale_payload_io*)*, 72
ScaleProduct, *4*, *20*, *22*, *26–31*, *34*, 45, *50*, *51*, *74*, *78*, *80*, *82*, *84*
stats::dist, *42*, *54*
stats::dist(), *55*
stats::hclust, *16*
stats::kmeans, *16*
str.NestedScale, 80
str.nestedscales::NestedScale *(*str.NestedScale*)*, 80
str.nestedscales::ScaleProduct *(*str.NestedScale*)*, 80
summary.NestedScale, 81
summary.nestedscales::NestedScale *(*summary.NestedScale*)*, 81
summary.nestedscales::ScaleProduct *(*summary.ScaleProduct*)*, 82
summary.ScaleProduct, 82

unbind_scale *(*bind_scale*)*, 10

write_scale, 82
write_scale(), *50*, *72*, *84*
write_scale_dataset, 84
write_scale_dataset(), *11*, *26*, *53*