Package 'vectra'

June 29, 2026

Title Columnar Query Engine for Larger-than-RAM Data

Version 0.9.7

Description A minimal columnar query engine with lazy execution on datasets larger than RAM. Provides 'dplyr'-like verbs (filter(), select(), mutate(), group_by(), summarise(), joins, window functions) and common aggregations (n(), sum(), mean(), min(), max(), sd(), first(), last()) backed by a pure C11 pull-based execution engine and a custom on-disk format ('.vtr'). Reads and writes 'GeoTIFF' (including tiled and 'BigTIFF' layouts) and a tiled raster format ('.vec') with overview pyramids and time cubes for larger-than-RAM raster data. Streams vector operations (spatial transforms, point-in-polygon and nearest-feature joins including a two-sided grid-partitioned join, select-by-location, clip, erase, dissolve, 'rasterization', 'polygonization', and contouring) through 'sf', and runs raster operations (zonal statistics, focal windows, terrain derivatives, resample or 'reproject' warp, polygon masking, map algebra, and 'mosaicking') in native C or over the tiled '.vec' format, one batch or tile at a time for data larger than RAM.

License MIT + file LICENSE

Depends R (>= 4.1.0)

SystemRequirements GNU make

Encoding UTF-8

Imports tidyselect, rlang, libgeos, parallel

LinkingTo libgeos

Suggests biglm, bit64, igraph, knitr, openxlsx2, rmarkdown, sf, terra, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

URL https://gillescolling.com/vectra/, https://github.com/gcol33/vectra

BugReports https://github.com/gcol33/vectra/issues

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Gilles Colling [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-3070-6066>)

Maintainer Gilles Colling <gilles.colling051@gmail.com>

Repository CRAN

Date/Publication 2026-06-29 13:00:02 UTC

Contents

`across` *Apply a function across multiple columns*

Description

Used inside `mutate()` or `summarise()` to apply a function to multiple columns selected with tidyselect. Returns a named list of expressions.

Usage

```
across(.cols, .fns, ..., .names = NULL)
```

Arguments

`.cols`	Column selection (tidyselect).
`.fns`	A function, formula, or named list of functions.
`...`	Additional arguments passed to `.fns`.
`.names`	A glue-style naming pattern. Uses `{.col}` and `{.fn}`. Default: `"{.col}"` if `.fns` is a single function, `"{.col}_{.fn}"` if `.fns` is a named list.

Value

A named list used internally by mutate/summarise.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
# In summarise (conceptual; across is expanded to individual expressions)
unlink(f)
```

---

append_vtr *Append rows to an existing .vtr file*

Description

Appends one or more new row groups to the end of an existing `.vtr` file without touching or recompressing existing row groups. The schema of `x` must exactly match the schema of the target file (same column names and types, in the same order).

Usage

```
append_vtr(x, path, ...)
```

Arguments

`x`	A `vectra_node` (lazy query) or a `data.frame`.
`path`	File path of an existing `.vtr` file to append to.
`...`	Additional arguments passed to methods.

Details

The operation is not fully atomic: if the process is interrupted after new row groups are written but before the header is patched, the file will be in a corrupted state. Use `write_vtr()` for safety-critical write-once workloads.

Value

Invisible `NULL`.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars[1:10, ], f)
append_vtr(mtcars[11:20, ], f)
result <- tbl(f) |> collect()
stopifnot(nrow(result) == 20L)
unlink(f)
```

arrange *Sort rows by column values*

Description

Sort rows by column values

Usage

```
arrange(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Column names (unquoted). Wrap in desc() for descending order.

Details

Uses an external merge sort with a 1 GB memory budget. When data exceeds this limit, sorted runs are spilled to temporary .vtr files and merged via a k-way min-heap. NAs sort last in ascending order.

This is a materializing operation.

Value

A new vectra_node with sorted rows.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> arrange(desc(mpg)) |> collect() |> head()
unlink(f)
```

bind_rows *Bind rows or columns from multiple vectra tables*

Description

Bind rows or columns from multiple vectra tables

Usage

```
bind_rows(..., .id = NULL)

bind_cols(...)
```

Arguments

...	vectra_node objects or data.frames to combine.
.id	Optional column name for a source identifier.

Details

When all inputs are vectra_node objects with identical column names and types and no .id is requested, bind_rows creates a streaming ConcatNode that iterates children sequentially without materializing.

Otherwise, inputs are collected and combined in R. Missing columns are filled with NA.

bind_cols requires the same number of rows in each input.

Value

A vectra_node (streaming) when all inputs are vectra_node with identical schemas and .id is NULL. Otherwise a data.frame.

Examples

```
f1 <- tempfile(fileext = ".vtr")
f2 <- tempfile(fileext = ".vtr")
write_vtr(data.frame(x = 1:3, y = 4:6), f1)
write_vtr(data.frame(x = 7:9, y = 10:12), f2)
bind_rows(tbl(f1), tbl(f2)) |> collect()
bind_cols(tbl(f1), tbl(f2))
unlink(c(f1, f2))
```

---

block_fuzzy_lookup *Fuzzy-match query keys against a materialized block*

Description

Computes string distances between query keys and a string column in a materialized block. Optionally uses exact-match blocking on a second column (e.g., genus) to reduce the search space.

Usage

```
block_fuzzy_lookup(
  block,
  column,
  keys,
  method = "dl",
  max_dist = 0.2,
  block_col = NULL,
  block_keys = NULL,
  n_threads = 4L
)
```

Arguments

block	A vectra_block from materialize().
column	Character scalar. Name of the string column to fuzzy-match against.
keys	Character vector. Query strings to match.
method	Character. Distance method: "dl" (Damerau-Levenshtein, default), "levenshtein", or "jw" (Jaro-Winkler).
max_dist	Numeric. Maximum normalized distance (default 0.2).
block_col	Optional character scalar. Column name for exact-match blocking (e.g., genus). When provided, only rows where block_col matches the corresponding block_keys value are compared.
block_keys	Optional character vector (same length as keys). Exact-match values for blocking. Required when block_col is provided.
n_threads	Integer. Number of OpenMP threads (default 4L).

Value

A data.frame with columns query_idx (1-based position in keys), fuzzy_dist (normalized distance), plus all columns from the block.

---

block_lookup *Probe a materialized block by column value*

Description

Performs a hash lookup on a string column of a materialized block. Returns all rows where the column value matches one of the query keys. Hash indices are built lazily on first use and cached for subsequent calls.

Usage

```
block_lookup(block, column, keys, ci = FALSE)
```

Arguments

block	A vectra_block from materialize().
column	Character scalar. Name of the string column to match against.
keys	Character vector. Query values to look up.
ci	Logical. Case-insensitive matching (default FALSE).

Value

A data.frame with column query_idx (1-based position in keys) plus all columns from the block, for each (query, block_row) match pair.

Examples

```
f <- tempfile(fileext = ".vtr")
df <- data.frame(taxonID = 1:2,
                 canonicalName = c("Quercus robur", "Pinus sylvestris"))
write_vtr(df, f)
blk <- materialize(tbl(f))
hits <- block_lookup(blk, "canonicalName", c("Quercus robur"))
ci_hits <- block_lookup(blk, "canonicalName", c("quercus robur"), ci = TRUE)
unlink(f)
```

---

chunk_feeder *Turn a query into a resettable chunk generator*

Description

Wraps a query so a pull-based consumer can read it one chunk at a time and re-read it from the start as many times as needed. The returned closure follows the `data(reset)` protocol that `biglm::bigglm()` expects: called with `reset = TRUE` it rewinds to the beginning of the data, and called with `reset = FALSE` it returns the next chunk as a data.frame, or `NULL` once the data is exhausted. This lets `bigglm()` fit a generalized linear model on a dataset larger than RAM, streaming each iteratively reweighted pass through the engine without ever holding the full design matrix.

Usage

```
chunk_feeder(.source)
```

Arguments

`.source`	Either a function of no arguments returning a fresh `vectra_node` each time it is called (e.g. `function() tbl_csv("occ.csv") |> select(presence, bio1, bio12)`), or an offloaded node from `offload()`. Every chunk must contain all variables the consumer's formula references.

Details

Because a vectra node is consumed as it streams, re-reading requires a fresh node on each pass. `chunk_feeder()` accepts either form: a *factory*, a function of no arguments that returns a new node each time it is called; or an offloaded node from `offload()`, which is backed by a file and replays from disk directly. On every `reset = TRUE` a fresh stream is started, so the same query is replayed on each pass.

Prefer feeding an `offload()` of the prepared query: the pipeline (scan, joins, mutate) runs once into the spill, and every reweighted pass is then a disk scan of the prepared columns rather than a re-run of the pipeline.

Value

A function `function(reset = FALSE)`. With `reset = TRUE` it rewinds and returns `invisible(NULL)`; with `reset = FALSE` it returns the next chunk as a data.frame, or `NULL` at end of stream.

See Also

`offload()` for the replay cache, and `collect_chunked()` for single-pass reductions that vectra drives.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)

feed <- chunk_feeder(function() tbl(f) |> select(mpg, wt, hp))
feed(reset = TRUE)       # rewind to the start of the stream
first <- feed()          # first chunk as a data.frame
head(first)


# Out-of-core GLM: prepare once with offload(), then bigglm() replays it.
if (requireNamespace("biglm", quietly = TRUE)) {
  s <- offload(tbl(f) |> select(mpg, wt, hp))
  fit <- biglm::bigglm(mpg ~ wt + hp, data = chunk_feeder(s),
                       family = gaussian())
  coef(fit)
}

unlink(f)
```

---

collect *Execute a lazy query and return a data.frame*

Description

Pulls all batches from the execution plan and materializes the result as an R data.frame.

Usage

```
collect(x, ...)
```

Arguments

`x`	A `vectra_node` object.
`...`	Ignored.

Value

A data.frame with the query results.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
result <- tbl(f) |> collect()
head(result)
unlink(f)
```

`collect_chunked` *Fold a function over a query, one batch at a time*

Description

Streams a lazy query through R in bounded pieces and reduces them with `f`, instead of materializing the whole result the way `collect()` does. The engine pulls one batch (a data.frame of up to a few hundred thousand rows) at a time; `f` is called as `f(acc, chunk)` and its return value becomes the accumulator for the next batch. Peak memory is one batch plus whatever the accumulator holds, so a result far larger than RAM can be reduced to a small summary in a single pass.

Usage

```
collect_chunked(x, f, .init = NULL, combine = NULL, commutative = FALSE)

## Default S3 method:
collect_chunked(x, f, .init = NULL, combine = NULL, commutative = FALSE)

## S3 method for class 'vectra_node'
collect_chunked(x, f, .init = NULL, combine = NULL, commutative = FALSE)

## S3 method for class 'vectra_partition'
collect_chunked(x, f, .init = NULL, combine = NULL, commutative = FALSE)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_csv()`, `tbl_tiff()`, ... and any chain of verbs). It is consumed: after `collect_chunked()` returns, the stream is drained and `x` cannot be collected again.
`f`	A function of two arguments `function(acc, chunk)` returning the updated accumulator. `chunk` is a data.frame holding the next batch of rows.
`.init`	Initial accumulator value. Passed to `f` with the first batch and returned unchanged if the query yields no rows. When `combine` is supplied this is also the monoid identity (the value `combine` leaves unchanged).

combine	Optional function function(acc, acc) that merges two accumulators. Supplying it declares the reduction a monoid with .init as identity, which is what lets the fold run over the independent shards of a partition (offload(x, by = ...)) and have the partial results merged correctly. For a plain node the stream is a single sequence, so combine is not needed and is ignored.
commutative	Logical; declare that combine does not depend on the order of its arguments. Lets a partitioned fold merge shards in any order (no stable sort required). Default FALSE.

Details

This is the streaming counterpart to a fold (Reduce()): use it when the query returns more rows than fit in memory but the *reduction* is small. A running count, per-group sufficient statistics, the cross-products X'X and X'y behind a linear fit, an online mean or histogram - all accumulate in bounded space across the stream. When you instead need the model-fitting consumer to drive the iteration (and to re-read the data on each pass, as an iteratively reweighted GLM does), use chunk_feeder().

Value

The final accumulator. For a node: f applied left-to-right across every batch, seeded with .init. For a partition: each shard folded with f/.init, then those per-shard accumulators merged with combine.

See Also

chunk_feeder() for pull-based consumers such as biglm::bigglm(), offload() for the replay cache and the partitioned monoidal reduce, group_map() and group_modify() for per-shard application, and collect() to materialize the full result.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)

# Row count without materializing the result.
collect_chunked(tbl(f), function(acc, chunk) acc + nrow(chunk), .init = 0L)

# Accumulate the normal-equation pieces X'X and X'y for an exact OLS fit
# of mpg ~ wt + hp, in one streaming pass.
acc <- collect_chunked(
  tbl(f) |> select(mpg, wt, hp),
  function(acc, chunk) {
    X <- cbind(1, chunk$wt, chunk$hp)
    y <- chunk$mpg
    list(XtX = acc$XtX + crossprod(X), Xty = acc$Xty + crossprod(X, y))
  },
  .init = list(XtX = matrix(0, 3, 3), Xty = matrix(0, 3, 1))
)
solve(acc$XtX, acc$Xty)            # same as coef(lm(mpg ~ wt + hp, mtcars))
unlink(f)
```

collect_sf *Materialize a spatial query as an sf object*

Description

Collects a vectra_node (typically the result of spatial_map() or spatial_join()) into memory and rebuilds an sf object from its hex-WKB geometry column. The CRS defaults to the one carried on the node.

Usage

```
collect_sf(x, geom = "geometry", crs = NULL)
```

Arguments

x	A vectra_node with a hex-WKB / WKT geometry column, or a data.frame already collected from one.
geom	Name of the geometry column. Default "geometry".
crs	Override the coordinate reference system. Defaults to the CRS the node carries, or unknown.

Details

This is the spatial counterpart to collect(): use it when the final result fits in memory as sf. For a result still larger than RAM, keep it as a node and write it out with write_vtr() (the geometry stays as a WKB string column) or reduce it with collect_chunked().

Value

An sf object.

See Also

spatial_map(), spatial_join(), collect().

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  NAME = nc$NAME,
  geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
), f)
result <- tbl(f) |> spatial_map(~ sf::st_centroid(.x), crs = sf::st_crs(nc))
collect_sf(result)
unlink(f)
```

contours *Extract contour iso-lines from a streamed raster*

Description

Traces contour lines at one or more levels from a `.vec` raster with marching squares, reading the raster one tile-row strip at a time (each strip expanded by one row so a cell straddling the strip boundary is traced once). Each strip contributes line segments, which are accumulated into a lazy `vectra_node` carrying a `level` column and hex-WKB geometry. With `merge = TRUE` the segments of each level are joined into continuous lines.

Usage

```
contours(x, levels, band = 1L, merge = TRUE, crs = NA, flush_rows = NULL)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster.
`levels`	Numeric vector of contour levels to trace.
`band`	Band to contour (1-based). Default 1.
`merge`	If `TRUE` (default) join each level's segments into continuous lines with `sf::st_line_merge()`; if `FALSE` return the raw per-cell segments.
`crs`	Coordinate reference system recorded on the node. Defaults to the raster's EPSG, else unknown.
`flush_rows`	Rows buffered before a spill flush. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Extraction is the *sort / partition* tier of the spatial toolbox: bounded to one haloed strip at a time. The optional final merge collects the segment set, which is small relative to the raster, and joins it per level; this is the small all-to-all step on the output, not on the grid. Geometry assembly and the merge are delegated to **sf** (an optional dependency).

Value

A `vectra_node` with a `level` column and a hex-WKB `geometry` column, materialise it with `collect_sf()`.

See Also

`polygonize()` for area features, `terrain()` for the DEM derivatives contours often accompany, `collect_sf()` to materialise as `sf`.

Examples

```
z <- outer(1:20, 1:20, function(r, c) r + c)
f <- tempfile(fileext = ".vec")
vec_write_raster(z, f, dtype = "f64", extent = c(0, 0, 20, 20))

iso <- contours(f, levels = c(15, 25, 35))
collect_sf(iso)
unlink(f)
```

---

count *Count observations by group*

Description

Count observations by group

Usage

```
count(x, ..., wt = NULL, sort = FALSE, name = NULL)

tally(x, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

x	A vectra_node object.
...	Grouping columns (unquoted).
wt	Column to weight by (unquoted). If NULL, counts rows.
sort	If TRUE, sort output in descending order of n.
name	Name of the count column (default "n").

Details

Equivalent to group_by(...) |> summarise(n = n()). When wt is provided, uses sum(wt) instead of n(). When sort = TRUE, results are sorted in descending order of the count column.

Value

A vectra_node with group columns and a count column.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> count(cyl) |> collect()
unlink(f)
```

`create_index` *Create a hash index on a .vtr file column*

Description

Builds a persistent hash index stored as a `.vtri` sidecar file alongside the `.vtr` file. The index maps key hashes to row group indices, enabling O(1) row group identification for equality predicates (`filter(col == value)`).

Usage

```
create_index(path, column, ci = FALSE)
```

Arguments

`path`	Path to a `.vtr` file.
`column`	Character vector. Name(s) of column(s) to index.
`ci`	Logical. Build a case-insensitive index? Default `FALSE`.

Details

For composite indexes on multiple columns, pass a character vector. Composite indexes accelerate AND-combined equality predicates (e.g., `filter(col1 == "a", col2 == "b")`).

The index is automatically loaded by `tbl()` when present. It composes with zone-map pruning and binary search on sorted columns.

Value

Invisible `NULL`. The index is written as a `.vtri` sidecar file.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = letters, val = 1:26, stringsAsFactors = FALSE), f)
create_index(f, "id")
tbl(f) |> filter(id == "m") |> collect()
unlink(c(f, paste0(f, ".id.vtri")))
```

cross_join *Cross join two vectra tables*

Description

Returns every combination of rows from `x` and `y` (Cartesian product). Both tables are collected before joining.

Usage

```
cross_join(x, y, suffix = c(".x", ".y"), ...)
```

Arguments

`x`	A `vectra_node` object or data.frame.
`y`	A `vectra_node` object or data.frame.
`suffix`	Suffixes for disambiguating column names (default `c(".x", ".y")`).
`...`	Ignored.

Value

A data.frame with `nrow(x) * nrow(y)` rows.

Examples

```
f1 <- tempfile(fileext = ".vtr")
f2 <- tempfile(fileext = ".vtr")
write_vtr(data.frame(a = 1:2), f1)
write_vtr(data.frame(b = c("x", "y", "z"), stringsAsFactors = FALSE), f2)
cross_join(tbl(f1), tbl(f2))
unlink(c(f1, f2))
```

delete_vtr *Logically delete rows from a .vtr file*

Description

Marks the specified 0-based physical row indices as deleted by writing (or updating) a tombstone side file (`<path>.del`). The original `.vtr` file is never modified. The next call to `tbl()` on the same path will automatically exclude the deleted rows.

Usage

```
delete_vtr(path, row_ids)
```

Arguments

path	File path of the .vtr file to delete rows from.
row_ids	A numeric vector of **0-based** physical row indices to delete. Out-of-range indices are silently ignored on read (they will never match a real row).

Details

Tombstone files are cumulative: calling delete_vtr() multiple times on the same file merges all deletions (union, deduplicated). To undo deletions, remove the .del file manually with unlink(paste0(path, ".del")).

Value

Invisible NULL.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)

# Delete the first and third rows (0-based indices 0 and 2)
delete_vtr(f, c(0, 2))

result <- tbl(f) |> collect()
stopifnot(nrow(result) == nrow(mtcars) - 2L)

unlink(c(f, paste0(f, ".del")))
```

---

desc *Mark a column for descending sort order*

Description

Used inside arrange() to sort a column in descending order.

Usage

```
desc(x)
```

Arguments

x	A column name.

Value

A marker used by arrange().

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> arrange(desc(mpg)) |> collect() |> head()
unlink(f)
```

`diff_vtr` *Compute the logical diff between two .vtr files*

Description

Streams both files and computes a set-level diff keyed on `key_col`. Returns a list with two elements:

Usage

```
diff_vtr(old_path, new_path, key_col)
```

Arguments

`old_path`	Path to the older `.vtr` file.
`new_path`	Path to the newer `.vtr` file.
`key_col`	Name of the column to use as the row key (must exist in both files with the same type).

Details

- `added`: a `vectra_node` (lazy `tbl()`) of rows present in `new_path` but not `old_path` (matched on `key_col`). Call `collect()` to materialise. The underlying temp file is deleted when the node is garbage-collected **or** when the calling R session ends via `on.exit()`.
- `deleted`: a vector of key values present in `old_path` but not `new_path`.

This is a **logical diff** (key-based set difference), not a binary file diff. Rows with the same key that have changed values are not reported as modified — use `added` and `deleted` together to detect updates (a key that appears in both means a row was replaced).

Value

A named list with elements `added` (a `vectra_node`) and `deleted` (a vector of key values).

Examples

```
f1 <- tempfile(fileext = ".vtr")
f2 <- tempfile(fileext = ".vtr")
df1 <- data.frame(id = 1:5, val = letters[1:5], stringsAsFactors = FALSE)
df2 <- data.frame(id = c(3L, 4L, 5L, 6L, 7L),
                  val = c("C", "d", "e", "f", "g"),
                  stringsAsFactors = FALSE)
```

```
write_vtr(df1, f1)
write_vtr(df2, f2)

d <- diff_vtr(f1, f2, "id")
# Rows 1 and 2 deleted; rows 6 and 7 added
stopifnot(all(d$deleted %in% c(1, 2)))
stopifnot(all(collect(d$added)$id %in% c(6, 7)))

unlink(c(f1, f2))
```

`distinct` *Keep distinct/unique rows*

Description

Keep distinct/unique rows

Usage

```
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

`.data`	A `vectra_node` object.
`...`	Column names (unquoted). If empty, uses all columns.
`.keep_all`	If `TRUE`, keep all columns (not just those in `...`).

Details

Uses hash-based grouping with zero aggregations. When `.keep_all = TRUE` with a column subset, falls back to R's `duplicated()` with a message.

This is a materializing operation.

Value

A `vectra_node` with unique rows.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> distinct(cyl) |> collect()
unlink(f)
```

explain *Print the execution plan for a vectra query*

Description

Shows the node types, column schemas, and structure of the lazy query plan.

Usage

```
explain(x, ...)
```

Arguments

x	A vectra_node object.
...	Ignored.

Value

Invisible x.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> filter(cyl > 4) |> select(mpg, cyl) |> explain()
unlink(f)
```

filter *Filter rows of a vectra query*

Description

Filter rows of a vectra query

Usage

```
filter(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Filter expressions (combined with &).

Details

Filter uses zero-copy selection vectors: matching rows are indexed without copying data. Multiple conditions are combined with `&`. Supported expression types: arithmetic (`+`, `-`, `*`, `/`, `%%`), comparison (`==`, `!=`, `<`, `<=`, `>`, `>=`), boolean (`&`, `|`, `!`), `is.na()`, and string functions (`nchar()`, `substr()`, `grepl()` with fixed patterns).

NA comparisons return NA (SQL semantics). Use `is.na()` to filter NAs explicitly.

This is a streaming operation (constant memory per batch).

Value

A new `vectra_node` with the filter applied.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> filter(cyl > 4) |> collect() |> head()
unlink(f)
```

`focal` *Moving-window (focal) statistics over a streamed raster*

Description

Applies a moving window to a `.vec` raster, reading the input one tile-row strip at a time – each strip expanded by the kernel radius (a halo read) so window neighbours are available without ever holding the whole grid resident. The per-window statistic is computed in C. When `path` is given the output is streamed straight back to a new `.vec` one tile-row at a time, so neither the input nor the output band is ever fully in memory; this is the raster op that runs out of core where an in-memory engine needs the whole raster at once.

Usage

```
focal(
  x,
  w = matrix(1, 3, 3),
  fun = c("mean", "sum", "min", "max", "sd", "median"),
  na.rm = TRUE,
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

x	A vectra_raster (from vec_open_raster()) or a path to a .vec raster.
w	A numeric weight matrix with odd dimensions, or a single positive odd integer k for a k x k window of ones. Default matrix(1, 3, 3).
fun	Window statistic: one of "sum", "mean", "min", "max", "sd", "median". Default "mean".
na.rm	Skip nodata cells inside the window (TRUE, default) or let them propagate NA (FALSE).
band	Band to read (1-based). Default 1.
path	Optional output .vec path. When given the result is streamed to disk and the opened vec_open_raster() handle is returned invisibly; when NULL the result is returned as an in-memory matrix.
dtype	Storage dtype for .vec output (see vec_write_raster()). Default "f32".
compression	Compression effort for .vec output. Default "fast".

Details

This is the *sort / partition* tier of the spatial toolbox: bounded to one haloed strip at a time, exploiting tile locality.

The window w is a numeric weight matrix with odd dimensions (or a single odd integer k, shorthand for a k x k matrix of ones). NA weights mark cells outside the window. For fun = "sum"/"mean" the weights scale the values (sum is sum(w * x), mean is sum(w * x) / sum(w)); for the other statistics a finite weight only marks membership. With na.rm = TRUE (the default) nodata cells inside the window are skipped; with na.rm = FALSE any nodata cell – including a window that runs off the raster edge – makes the result NA, matching the resident behaviour.

Value

When path is NULL, a numeric matrix (row 1 northmost) carrying gt, extent, crs, and fun attributes. When path is given, the written vectra_raster handle (invisibly).

See Also

terrain() for DEM derivatives built on the same strip pass, zonal() for per-zone summaries.

Examples

```
m <- matrix(1:36, 6, 6, byrow = TRUE)
f <- tempfile(fileext = ".vec")
vec_write_raster(m, f, dtype = "f64", extent = c(0, 0, 6, 6))

# 3x3 mean smoother; edge cells see off-raster neighbours.
focal(f, w = matrix(1, 3, 3), fun = "mean")
unlink(f)
```

fuzzy_join *Fuzzy join two vectra tables by string distance*

Description

Joins two tables using approximate string matching on key columns. Optionally blocks by a second column (e.g., genus) for performance — only rows sharing the same blocking key are compared.

Usage

```
fuzzy_join(
  x,
  y,
  by,
  method = "dl",
  max_dist = 0.2,
  block_by = NULL,
  n_threads = 4L,
  suffix = ".y"
)
```

Arguments

x	A vectra_node object (probe / query side).
y	A vectra_node object (build / reference side).
by	A named character vector of length 1: c("probe_col" = "build_col"). The columns to compute string distance on.
method	Character. Distance algorithm: "dl" (Damerau-Levenshtein, default), "levenshtein", or "jw" (Jaro-Winkler).
max_dist	Numeric. Maximum normalized distance (0-1) to keep a match. Default 0.2.
block_by	Optional named character vector of length 1: c("probe_col" = "build_col"). Rows must match exactly on these columns before distance is computed. Dramatically reduces comparisons.
n_threads	Integer. Number of OpenMP threads for parallel distance computation over partitions. Default 4L.
suffix	Character. Suffix appended to build-side column names that collide with probe-side names. Default ".y".

Value

A vectra_node with all probe columns, all build columns (suffixed on collision), and a fuzzy_dist column (double).

geom_expressions *Geometry functions inside mutate(), filter(), and summarise()*

Description

vectra recognizes a family of `st_*` geometry functions directly inside expression verbs. They run on the GEOS C library straight off the hex-WKB geometry column, one row at a time, with no per-batch round-trip through **sf**: `tbl(f) |> filter(st_area(geometry) > 1e6)` prunes the stream in C, and `mutate(centroid = st_centroid(geometry))` adds a new hex-WKB geometry column. The computation is planar, in the geometry's own coordinate units, exactly as the streaming spatial verbs are.

Details

These names are interpreted by the expression engine; they are not exported R functions and are not called as such. They are available only inside `mutate()`, `transmute()`, `filter()`, and a grouped `summarise()` over a `vectra_node`. The geometry argument is the hex-WKB column (named `geometry` by convention). A function that returns a geometry produces another hex-WKB column; materialize it with `collect_sf()` (point it at the column with `geom =`), or write it with `write_tiff()`/`sf::st_write()`.

Measures (return a number)

`st_area(g)` Area of polygonal geometry (0 for lines and points).

`st_length(g)`, `st_perimeter(g)` Length of a line, or the perimeter (boundary length) of a polygon. The two names are aliases.

`st_x(g)`, `st_y(g)` Coordinate of a point geometry; `NA` for a non-point.

`st_npoints(g)` Number of coordinates in the geometry.

`st_ngeometries(g)` Number of sub-geometries in a collection or multi-geometry (1 for a single geometry).

`st_distance(a, b)` Shortest planar distance between two geometries (see the binary second argument below).

Predicates (return TRUE / FALSE)

Unary: `st_is_valid(g)`, `st_is_empty(g)`, `st_is_simple(g)`.

Binary topological relations, each taking a second geometry: `st_intersects`, `st_within`, `st_contains`, `st_overlaps`, `st_touches`, `st_crosses`, `st_equals`, `st_disjoint`, `st_covers`, `st_covered_by`. Used in `filter()` they keep the rows where the relation holds.

Type (returns a string)

`st_geometry_type(g)` gives the GEOS geometry type name (`"Point"`, `"Polygon"`, `"MultiPolygon"`, ...).

Transforms (return a geometry)

`st_centroid(g)` Area (or length, or vertex) centroid.

`st_point_on_surface(g)` A point guaranteed to lie on the geometry.

`st_boundary(g)` The topological boundary.

`st_envelope(g)` The axis-aligned bounding rectangle.

`st_convex_hull(g)` The convex hull.

`st_make_valid(g)` Repair an invalid geometry.

`st_buffer(g, dist)` Buffer by `dist` (round joins, 8 segments per quadrant).

`st_simplify(g, tol)` Topology-preserving Douglas-Peucker simplification with tolerance `tol`.

The second geometry of a binary op

For `st_distance` and the binary predicates, the second argument can be another geometry column (compared row by row), a constant `sf`/`sfc` object (a multi-feature object is unioned to one geometry), or a hex-WKB string. A constant is parsed once and reused across every row, so testing a whole stream against one area of interest stays cheap.

Missing geometry

A missing (`NA`) or unparseable geometry, or an operation that has no answer (a coordinate of a non-point, distance to a missing geometry), yields `NA` for that row rather than an error.

See Also

`mutate()`, `filter()`, `collect_sf()` to materialize a geometry result as `sf`; `spatial_map()` for an arbitrary per-feature **sf** transform; `spatial_filter()` and `spatial_join()` for relating a stream to a resident reference layer.

Examples

```
if (requireNamespace("sf", quietly = TRUE)) {
  nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
  f <- tempfile(fileext = ".vtr")
  write_vtr(data.frame(
    NAME     = nc$NAME,
    geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
  ), f)

  # measures: add area and perimeter columns
  tbl(f) |>
    mutate(area = st_area(geometry), perim = st_perimeter(geometry)) |>
    select(NAME, area, perim) |>
    collect() |>
    head()

  # predicate: keep counties intersecting an area of interest
  aoi <- sf::st_as_sfc(sf::st_bbox(
    c(xmin = -81.5, ymin = 36.2, xmax = -80.5, ymax = 36.6)))
```

```
  tbl(f) |> filter(st_intersects(geometry, aoi)) |> collect() |> nrow()

  # transform: replace each county with its centroid, materialize as sf
  tbl(f) |>
    mutate(geometry = st_centroid(geometry)) |>
    select(NAME, geometry) |>
    collect_sf()

  unlink(f)
}
```

`glimpse` *Get a glimpse of a vectra table*

Description

Shows column names, types, and a preview of the first few values without collecting the full result.

Usage

```
glimpse(x, width = 5L, ...)
```

Arguments

`x`	A `vectra_node` object.
`width`	Maximum number of preview rows to fetch (default 5).
`...`	Ignored.

Value

Invisible `x`.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> glimpse()
unlink(f)
```

grid — *Define a uniform grid for a partitioned spatial join*

Description

Describes the regular grid that `spatial_join()` uses to partition two streamed layers for the both-sides-larger-than-RAM case. Cell `(cx, cy)` covers `[origin_x + cx * cellsize_x, origin_x + (cx + 1) * cellsize_x)` and likewise in y, so a coordinate maps to the cell `floor((coord - origin) / cellsize)`. Pick a `cellsize` comparable to the scale of the join (large enough that most cells hold a workable shard, small enough that one cell's features fit in memory); for an extended-on-extended join choose it larger than the left features.

Usage

```
grid(cellsize, origin = c(0, 0))
```

Arguments

`cellsize`	Cell size: a single number for square cells, or `c(cellsize_x, cellsize_y)`.
`origin`	Grid origin `c(x0, y0)` (a cell corner). Default `c(0, 0)`.

Value

A `vectra_grid` specification to pass as `spatial_join(partition =)`.

See Also

`spatial_join()` for the join it partitions.

Examples

```
grid(1000)
grid(c(0.5, 0.25), origin = c(-180, -90))
```

group_by — *Group a vectra query by columns*

Description

Group a vectra query by columns

Usage

```
group_by(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Grouping column names (unquoted).

Value

A vectra_node with grouping information stored.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> group_by(cyl) |> summarise(avg = mean(mpg)) |> collect()
unlink(f)
```

---

group_map *Apply a function to each shard of a partition*

Description

Run a function once per shard of a partition (offload(x, by = ...)) and gather the results. Each shard is read into memory as a data.frame and passed to .f together with its key, so a model that couples rows within a group becomes a set of independent per-shard fits. This is the per-group counterpart to collect_chunked(), which instead merges every shard into a single accumulator.

Usage

```
group_map(.data, .f, ...)

## S3 method for class 'vectra_partition'
group_map(.data, .f, ...)

group_modify(.data, .f, ...)

## S3 method for class 'vectra_partition'
group_modify(.data, .f, ...)
```

Arguments

.data	A vectra_partition from offload() with a by key.
.f	A function applied to each shard. It receives the shard as a data.frame and the shard key (a string) as its first two arguments; any further arguments in ... follow. A purrr-style formula such as ~ lm(y ~ x, .x) also works, with .x the shard data and .y the key. For group_modify(), .f must return a data.frame.
...	Additional arguments passed on to .f.

Details

`group_map()` returns a named list, one element per shard keyed by the shard key, and places no constraint on what `.f` returns. Use it for per-group results that do not rebind into a table, such as fitted models.

`group_modify()` expects `.f` to return a data.frame for each shard and binds those frames into one. When a shard's result does not already carry the partition key column, the key is added as a leading column (named after the partition's by), so every row records the shard it came from. Use it for per-group summaries that recombine into a single table.

Each shard is materialized in full before `.f` sees it, so partition the query on a key whose groups fit in memory. For a reduction that stays bounded without ever holding a whole group, fold the partition with `collect_chunked()` instead.

Value

`group_map()` returns a named list with one element per shard. `group_modify()` returns a single data.frame: the per-shard results row-bound, with the shard key restored as a column when `.f` dropped it.

See Also

`offload()` to build a partition, and `collect_chunked()` for the partitioned monoidal reduce.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
p <- offload(tbl(f), by = "cyl")

# One fit per shard, returned as a named list keyed by cyl.
fits <- group_map(p, function(d, cyl) coef(lm(mpg ~ wt, data = d)))
fits

# Per-shard summaries recombined into one table, key restored as a column.
group_modify(p, function(d, cyl)
  data.frame(n = nrow(d), mean_mpg = mean(d$mpg)))
unlink(f)
```

---

has_index *Check if a hash index exists for a .vtr column*

Description

Check if a hash index exists for a .vtr column

Usage

```
has_index(path, column)
```

Arguments

path	Path to a .vtr file.
column	Character vector. Name(s) of column(s).

Value

Logical scalar: TRUE if a .vtri index file exists.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = letters, val = 1:26, stringsAsFactors = FALSE), f)
has_index(f, "id")   # FALSE
create_index(f, "id")
has_index(f, "id")   # TRUE
unlink(c(f, paste0(f, ".id.vtri")))
```

---

head.vectra_node *Limit results to first n rows*

Description

Limit results to first n rows

Usage

```
## S3 method for class 'vectra_node'
head(x, n = 6L, ...)
```

Arguments

x	A vectra_node object.
n	Number of rows to return.
...	Ignored.

Value

A data.frame with the first n rows.

left_join *Join two vectra tables*

Description

Join two vectra tables

Usage

```
left_join(x, y, by = NULL, suffix = c(".x", ".y"), ...)

inner_join(x, y, by = NULL, suffix = c(".x", ".y"), ...)

right_join(x, y, by = NULL, suffix = c(".x", ".y"), ...)

full_join(x, y, by = NULL, suffix = c(".x", ".y"), ...)

semi_join(x, y, by = NULL, ...)

anti_join(x, y, by = NULL, ...)
```

Arguments

x	A vectra_node object (left table).
y	A vectra_node object (right table).
by	A character vector of column names to join by, or a named vector like c("a" = "b"). NULL for natural join (common columns).
suffix	A character vector of length 2 for disambiguating non-key columns with the same name (default c(".x", ".y")).
...	Ignored.

Details

All joins use a build-right, probe-left hash join. The entire right-side table is materialized into a hash table; left-side batches stream through. Memory cost is proportional to the right-side table size.

NA keys never match (SQL NULL semantics). Key types are auto-coerced following the bool < int64 < double hierarchy. Joining string against numeric keys is an error.

Value

A vectra_node with the joined result.

Examples

```
f1 <- tempfile(fileext = ".vtr")
f2 <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = c(1, 2, 3), x = c(10, 20, 30)), f1)
write_vtr(data.frame(id = c(1, 2, 4), y = c(100, 200, 400)), f2)
left_join(tbl(f1), tbl(f2), by = "id") |> collect()
unlink(c(f1, f2))
```

---

link *Define a link between a fact table and a dimension table*

Description

Creates a link descriptor that specifies how to join a dimension table to a fact table via one or more key columns.

Usage

```
link(key, node)
```

Arguments

key	A character vector or named character vector specifying join keys. Unnamed: same column name in both tables. Named: `c("fact_col" = "dim_col")`.
node	A `vectra_node` object (the dimension table). Must be file-backed (created via `tbl()`, `tbl_csv()`, or `tbl_sqlite()`).

Value

A `vectra_link` object.

Examples

```
f_obs <- tempfile(fileext = ".vtr")
f_sp  <- tempfile(fileext = ".vtr")
write_vtr(data.frame(sp_id = 1:3, value = c(10, 20, 30)), f_obs)
write_vtr(data.frame(sp_id = 1:3, name = c("A", "B", "C")), f_sp)
lnk <- link("sp_id", tbl(f_sp))
unlink(c(f_obs, f_sp))
```

lookup *Look up columns from linked dimension tables*

Description

Resolves columns from dimension tables registered in a `vtr_schema()`, automatically building the necessary join tree. Reports unmatched keys as a diagnostic message.

Usage

```
lookup(.schema, ..., .join = "left", .report = TRUE)
```

Arguments

`.schema`	A `vectra_schema` object.
`...`	Column references: bare names for fact columns, or `dimension$column` for dimension columns.
`.join`	Join type: `"left"` (default, keeps all fact rows) or `"inner"` (drops unmatched fact rows).
`.report`	Logical. If `TRUE` (default), print a message with the number of unmatched keys per dimension.

Details

Column references use `dimension$column` syntax (e.g., `species$name`). Columns from the fact table can be referenced by name directly.

When `.report = TRUE`, each needed dimension is checked for unmatched keys by opening fresh scans of the fact and dimension tables. This adds one extra read pass per dimension but does not affect the lazy result node.

Only dimensions referenced in `...` are joined. Unreferenced dimensions are never scanned.

Value

A `vectra_node` with the selected columns.

Examples

```
f_obs <- tempfile(fileext = ".vtr")
f_sp  <- tempfile(fileext = ".vtr")
f_ct  <- tempfile(fileext = ".vtr")
write_vtr(data.frame(sp_id = 1:4, ct_code = c("AT", "DE", "FR", "XX"),
                     value = 10:13), f_obs)
write_vtr(data.frame(sp_id = 1:3,
                     name = c("Oak", "Beech", "Pine")), f_sp)
write_vtr(data.frame(ct_code = c("AT", "DE", "FR"),
                     gdp = c(400, 3800, 2700)), f_ct)
```

```
s <- vtr_schema(
  fact    = tbl(f_obs),
  species = link("sp_id", tbl(f_sp)),
  country = link("ct_code", tbl(f_ct))
)

# Pull columns from any linked dimension
result <- lookup(s, value, species$name, country$gdp)
collect(result)

unlink(c(f_obs, f_sp, f_ct))
```

`mask` *Mask a streamed raster to a polygon layer*

Description

Keeps the pixels of a `.vec` raster whose cell centre falls inside a resident polygon layer and sets the rest to `background`, reading the raster one tile-row strip at a time so the whole grid is never resident. It is the raster counterpart of `spatial_clip()`: the streamed side is the (large) raster and the small `mask` layer stays in memory. With `inverse = TRUE` the inside is cleared and the outside kept.

Usage

```
mask(
  x,
  mask,
  inverse = FALSE,
  band = NULL,
  background = NA_real_,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster.
`mask`	An `sf` or `sfc` polygon layer to clip against. When it carries no CRS it inherits the raster's EPSG.
`inverse`	If `FALSE` (default) keep pixels inside `mask`; if `TRUE` keep the pixels outside it.
`band`	Band(s) to mask (1-based). Default `NULL` masks every band.
`background`	Value written to cleared pixels. Default `NA_real_`.

path	Optional output .vec path. When given the result is streamed to disk and the opened vec_open_raster() handle is returned invisibly; when NULL the result is returned in memory (a matrix for one band, a list of matrices for several).
dtype	Storage dtype for .vec output (see vec_write_raster()). Default "f32".
compression	Compression effort for .vec output. Default "fast".

Details

This is the *monoid fold* tier of the spatial toolbox: bounded to one strip plus the resident mask, a single streaming pass, no spill. A pixel is tested against the mask only when its centre falls in the mask bounding box, so the point-in-polygon work stays proportional to the overlap rather than the whole grid. Point-in-polygon is delegated to **sf** (an optional dependency); topology and CRS handling are **sf**'s.

Value

When path is NULL, a numeric matrix (one band) or a list of matrices (several), each carrying gt, extent, and crs attributes (row 1 northmost). When path is given, the written vectra_raster handle (invisibly).

See Also

spatial_clip() for the vector analogue, zonal() for per-zone summaries over the same pixel-in-polygon assignment.

Examples

```
vals <- matrix(1:100, 10, 10, byrow = TRUE)
f <- tempfile(fileext = ".vec")
vec_write_raster(vals, f, dtype = "f64", extent = c(0, 0, 10, 10))

disc <- sf::st_buffer(sf::st_sfc(sf::st_point(c(5, 5))), 3)
inside <- mask(f, disc)
sum(!is.na(inside))
unlink(f)
```

---

materialize *Materialize a vectra node into a reusable in-memory block*

Description

Consumes a vectra node (pulling all batches) and stores the result as a persistent columnar block in memory. Unlike nodes, blocks can be probed repeatedly via block_lookup() without re-scanning.

Usage

```
materialize(.data)
```

Arguments

.data	A vectra_node (consumed; cannot be used after this call).

Value

A vectra_block object (external pointer to C-level ColumnBlock).

Examples

```
f <- tempfile(fileext = ".vtr")
df <- data.frame(taxonID = 1:3,
                 canonicalName = c("Quercus robur", "Pinus sylvestris",
                                   "Fagus sylvatica"))
write_vtr(df, f)
blk <- materialize(tbl(f) |> select(taxonID, canonicalName))
hits <- block_lookup(blk, "canonicalName",
                     c("Quercus robur", "Pinus sylvestris"))
unlink(f)
```

---

mosaic *Merge aligned rasters onto a common grid*

Description

Combines several .vec rasters that share a resolution and cell grid into one raster spanning their union, resolving overlap with fun. The output is walked one tile-row strip at a time and each input contributes only the window overlapping the current strip, so neither the inputs nor the output are held whole in memory.

Usage

```
mosaic(
  rasters,
  fun = c("first", "last", "mean", "sum", "min", "max"),
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

rasters	A list of vectra_raster handles or .vec paths sharing resolution and grid alignment.

fun	Overlap rule where inputs cover the same cell: "first" (the earliest input in rasters, default), "last", "mean", "sum", "min", or "max". Cells covered by no input come back NA.
band	Band read from every input (1-based). Default 1.
path	Optional output .vec path. When given the result is streamed to disk and the opened vec_open_raster() handle is returned invisibly; when NULL the result is returned as an in-memory matrix.
dtype	Storage dtype for .vec output (see vec_write_raster()). Default "f32".
compression	Compression effort for .vec output. Default "fast".

Details

This is the *monoid fold* tier of the spatial toolbox: each output strip folds the overlapping input windows, bounded memory, no spill. The inputs must share resolution and lie on a common cell grid; warp() them onto a shared grid first if they do not. No **sf** is needed.

Value

When path is NULL, a numeric matrix on the union grid (row 1 northmost) carrying gt, extent, and crs attributes. When path is given, the written vectra_raster handle (invisibly).

See Also

warp() to bring rasters onto a shared grid, rast_calc() for cellwise combination of already-aligned rasters.

Examples

```
a <- matrix(1, 4, 4); b <- matrix(2, 4, 4)
fa <- tempfile(fileext = ".vec"); fb <- tempfile(fileext = ".vec")
vec_write_raster(a, fa, dtype = "f64", extent = c(0, 0, 4, 4))
vec_write_raster(b, fb, dtype = "f64", extent = c(2, 2, 6, 6))

m <- mosaic(list(fa, fb), fun = "mean")
dim(m)
unlink(c(fa, fb))
```

---

mutate *Add or transform columns*

Description

Add or transform columns

Usage

```
mutate(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Named expressions for new or transformed columns.

Details

Supported expression types: arithmetic (+, -, *, /, %%), comparison, boolean, is.na(), nchar(), substr(), grepl() (fixed match only). Window functions (row_number(), rank(), dense_rank(), lag(), lead(), cumsum(), cummean(), cummin(), cummax()) are detected automatically and routed to a dedicated window node.

When grouped, window functions respect partition boundaries.

This is a streaming operation for regular expressions; window functions materialize all rows within each partition.

Value

A new vectra_node with mutated columns.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> mutate(kpl = mpg * 0.425144) |> collect() |> head()
unlink(f)
```

---

offload *Spill a query to disk and stream it back (the offload functor)*

Description

Materializes a query once to disk and returns a stream that holds the same rows, so every later pass is a disk scan instead of a re-run of the upstream pipeline. The materialization streams batch by batch, so peak memory stays at one batch regardless of result size. This is the bridge from the bounded single-pass world of collect_chunked() to out-of-core fits.

Usage

```
offload(
  x,
  by = NULL,
  n = NULL,
  method = c("auto", "level", "range", "hash"),
  path = NULL,
  compress = c("fast", "small", "none")
)
```

Arguments

x	A vectra_node to materialize.
by	Optional name (string) of a partition key column. When supplied, the result is a partition rather than a single node.
n	Number of buckets for method = "range" or "hash". Ignored for a one-shard-per-value partition.
method	Partition strategy: "auto" (default; one shard per value for a discrete key, n ranges for a numeric key), "level" (one shard per distinct value), "range" (n equal-width value ranges), or "hash" (n buckets by a stable hash of the key, co-locating each key).
path	Optional file path for a durable replay-cache spill (used only when by is NULL). When NULL a temporary file is used and removed when the returned node is garbage-collected.
compress	Compression for spill files, passed to write_vtr(): "fast" (default), "small", or "none".

Details

With no by, offload() returns a **replay cache**: a vectra_node backed by one .vtr file. Feed it to a pull-based consumer such as biglm::bigglm() through chunk_feeder(), which accepts an offloaded node directly, so each iteratively reweighted pass reads the prepared columns from disk rather than rebuilding them. Bake the selects and mutates into the query you offload, and replay does no further work.

With by, offload() returns a **partition**: the rows split into disjoint shards, one per key value (discrete key) or per value range (method = "range", or any numeric key), written in a single streaming pass. A partition prints as a list of shards and behaves like one: length(), names() (the keys), p[["key"]] (a shard node), and lapply(p, ...) all work. Fold it with collect_chunked() (supplying combine). The union of the shards reproduces the input; row totals are checked.

Value

A vectra_node (no by) or a vectra_partition (with by), each carrying a cost grade shown by print() and explain().

See Also

chunk_feeder() (accepts an offloaded node), collect_chunked() for the partitioned monoidal reduce, and arrange() for the external-sort instance.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)

# Replay cache: same rows, now on disk.
s <- offload(tbl(f) |> filter(cyl > 4) |> select(mpg, wt, hp))
nrow(collect(s))
```

```
# Partition by a key: a list of per-shard nodes.
p <- offload(tbl(f), by = "cyl")
names(p)
length(p)
nrow(collect(p[[1]]))
unlink(f)
```

`polygonize` *Vectorise a raster into polygons*

Description

Converts a `.vec` raster into polygon features, the inverse of `rasterize()`. The raster is read one tile-row strip at a time; within each strip equal-valued cells along a row collapse to a rectangle, and (with `dissolve = TRUE`) the rectangles are then merged by value through `spatial_dissolve()` so each distinct value becomes a single polygon spanning the whole raster. The result is a lazy `vectra_node` carrying a value column and hex-WKB geometry.

Usage

```
polygonize(
  x,
  band = 1L,
  dissolve = TRUE,
  na_rm = TRUE,
  values = "value",
  crs = NA,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster.
`band`	Band to vectorise (1-based). Default 1.
`dissolve`	If `TRUE` (default) merge cells of equal value into one polygon per value; if `FALSE` emit one square polygon per cell.
`na_rm`	Drop nodata cells (`TRUE`, default) or vectorise them as a value.
`values`	Name of the output value column. Default `"value"`.
`crs`	Coordinate reference system recorded on the node. Defaults to the raster's EPSG, else unknown.
`flush_rows`	Rows buffered before a spill flush. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Extraction is the *monoid fold* tier (one strip at a time); the by-value dissolve rides the *sort / partition* tier of `spatial_dissolve()`, localising each value to a shard before the union. Geometry assembly is delegated to **sf** (an optional dependency).

Value

A `vectra_node` with the value column and a hex-WKB `geometry` column, materialise it with `collect_sf()`.

See Also

`rasterize()` for the inverse, `contours()` for iso-lines, `collect_sf()` to materialise as `sf`.

Examples

```
m <- matrix(c(1, 1, 2, 2, 1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3), 4, 4,
            byrow = TRUE)
f <- tempfile(fileext = ".vec")
vec_write_raster(m, f, dtype = "f64", extent = c(0, 0, 4, 4))

polys <- polygonize(f)
collect_sf(polys)
unlink(f)
```

---

print.vectra_node *Print a vectra query node*

Description

Print a vectra query node

Usage

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

Arguments

`x`	A `vectra_node` object.
`...`	Ignored.

Value

Invisible `x`.

proximity *Euclidean distance to the nearest feature (proximity)*

Description

Computes, for every cell of a `.vec` raster, the straight-line Euclidean distance to the nearest feature cell, in CRS units. Feature cells are the non-NA cells by default, or the cells whose value is in `target`. This is the raster proximity / Euclidean-distance staple, the distance companion to `rasterize()`.

Usage

```
proximity(
  x,
  target = NULL,
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster.
`target`	Optional numeric vector of feature values. When `NULL` (default) every non-NA cell is a feature; otherwise a cell is a feature when its value is in `target`.
`band`	Band to read (1-based). Default 1.
`path`	Optional output `.vec` path. When given the result is streamed to disk and the opened `vec_open_raster()` handle is returned invisibly; when `NULL` the result is returned as an in-memory matrix.
`dtype`	Storage dtype for `.vec` output (see `vec_write_raster()`). Default `"f32"`.
`compression`	Compression effort for `.vec` output. Default `"fast"`.

Details

The exact Euclidean distance transform is separable (Felzenszwalb and Huttenlocher 2012): a one-dimensional lower-envelope-of-parabolas transform along the rows, then the same transform along the columns, each linear in the line length and each line independent. vectra runs it as four streamed passes over tile-row strips, with an out-of-core transpose between the row pass and the column pass, so the whole grid is never resident. The row pass scales squared distances by the x resolution and the column pass by the y resolution, so the result is exact on anisotropic (non-square) cells. This places proximity on the sort / partition tier of the spatial toolbox.

Distances are straight-line Euclidean in the raster CRS units. Cost-distance, which accumulates a per-cell friction along the path, is a global shortest-path problem and stays resident: `collect()` the raster and run a resident solver for that.

Value

When `path` is `NULL`, a numeric matrix (row 1 northmost) carrying `gt`, `extent`, and `crs` attributes, with distance in CRS units and `NA` where the raster holds no feature anywhere. When `path` is given, the written `vectra_raster` handle (invisibly).

See Also

`rasterize()` to build a raster from streamed points, `mask()` to clip a raster to a polygon layer.

Examples

```
m <- matrix(NA_real_, 12, 12)
m[3, 4] <- 1; m[9, 10] <- 1
f <- tempfile(fileext = ".vec")
vec_write_raster(m, f, dtype = "f64", extent = c(0, 0, 12, 12))

d <- proximity(f)
round(d[1:3, 1:3], 2)
unlink(f)
```

---

pull *Extract a single column as a vector*

Description

Extract a single column as a vector

Usage

```
pull(.data, var = -1)
```

Arguments

`.data`	A `vectra_node` object.
`var`	Column name (unquoted) or positive integer position.

Value

A vector.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> pull(mpg) |> head()
unlink(f)
```

`rasterize` *Rasterize a streamed point layer onto a fixed grid*

Description

Folds a larger-than-RAM stream of points into a fixed raster grid one batch at a time. The grid (`template`) is held resident in memory while the points flow past the engine, so peak memory is the grid plus one batch regardless of how many points there are – the streaming counterpart to running `terra::rasterize()` on a point set that has to fit in RAM. Each point's coordinate is mapped to its grid cell through the raster geotransform and the per-cell value is accumulated in C.

Usage

```
rasterize(
  x,
  template = NULL,
  field = NULL,
  fun = c("count", "sum", "mean", "min", "max"),
  extent = NULL,
  res = NULL,
  dims = NULL,
  coords = c("x", "y"),
  geom = NULL,
  crs = NA,
  background = NA_real_,
  path = NULL,
  dtype = "f32"
)
```

Arguments

`x`	A `vectra_node` streaming the points (from `tbl()`, `tbl_csv()`, any verb chain). It is consumed by the stream.
`template`	Optional grid to borrow geometry and CRS from: a `vectra_raster` (from `vec_open_raster()`), or a numeric `c(xmin, ymin, xmax, ymax)` extent. When omitted, supply `extent` with `res` or `dims`.
`field`	Name of a numeric column to aggregate. Required for every `fun` except `"count"` (which ignores it).
`fun`	Reduction over the points in each cell: one of `"count"`, `"sum"`, `"mean"`, `"min"`, `"max"`. `NA` values in `field` are skipped.
`extent`	Numeric `c(xmin, ymin, xmax, ymax)` defining the grid extent when no `template` is given.
`res`	Cell size: a single number for square cells, or `c(xres, yres)`. The cell counts are rounded to fit `extent` exactly. Supply `res` or `dims`.
`dims`	Grid shape `c(nrow, ncol)`, an alternative to `res`.

coords	Length-2 character vector naming the x and y coordinate columns. Default c("x", "y"). Ignored when geom is supplied.
geom	Name of a hex-WKB point-geometry column to rasterize instead of coordinate columns. Requires **sf**.
crs	Coordinate reference system recorded on the output, in any form sf::st_crs() accepts or a bare EPSG integer. Defaults to the template's, then the node's, else unknown.
background	Value for cells that receive no point. Default NA_real_.
path	Optional output path. When given, the grid is written to a .vec raster via vec_write_raster() and the opened vec_open_raster() handle is returned invisibly. When NULL, the grid is returned in memory.
dtype	Storage dtype for the .vec output (see vec_write_raster()). Default "f32".

Details

The reduction fun is a monoid over the points falling in each cell: "count" tallies points (no field needed); "sum", "mean", "min", "max" aggregate a numeric field. Cells that receive no point take the background value (NA by default). This is the *monoid fold* tier of the spatial toolbox: bounded memory, a single streaming pass, no spill.

Points arrive either as two numeric coordinate columns (coords, the default and fully **sf**-free path – the headline larger-than-RAM case) or decoded from a hex-WKB point-geometry column (geom, which needs **sf**). Geometry input is expected to be points (one coordinate per row); line and polygon coverage rasterization is out of scope here.

Value

When path is NULL, a numeric matrix with nrow grid rows (row 1 northmost) and ncol grid columns, carrying gt, extent, res, crs, and fun attributes. When path is given, the written vectra_raster handle (invisibly).

See Also

vec_write_raster() and vec_to_tiff() for raster output, spatial_join() to instead tag points with polygon attributes.

Examples

```
set.seed(1)
n <- 1e4
pts <- data.frame(x = runif(n, 0, 10), y = runif(n, 0, 10), z = rnorm(n))
f <- tempfile(fileext = ".vtr")
write_vtr(pts, f)

# Point density on a 10x10 grid, streamed: the grid is resident, the
# points are not.
counts <- tbl(f) |> rasterize(extent = c(0, 0, 10, 10), dims = c(10, 10))
counts

# Mean of z per cell.
```

```
zmean <- tbl(f) |>
  rasterize(extent = c(0, 0, 10, 10), dims = c(10, 10),
            field = "z", fun = "mean")
unlink(f)
```

`rast_calc` *Cellwise calculation over aligned rasters (map algebra)*

Description

Evaluates an expression cell by cell across one or more `.vec` rasters that share a grid, reading every input one tile-row strip at a time so no whole band is ever resident. Inside `expr` each name in `rasters` refers to that raster's strip as a numeric vector, so ordinary vectorised R expresses the calculation: a band index `(nir - red) / (nir + red)`, a reclassification `cut(dem, breaks)`, a threshold `ifelse(slope > 30, 1L, 0L)`, or arithmetic across layers. The result is written one strip at a time to a single-band output.

Usage

```
rast_calc(
  rasters,
  expr,
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

`rasters`	A named list of `vectra_raster` handles or `.vec` paths sharing a grid. The names are the variables available inside `expr`.
`expr`	An expression in those names producing one value per cell (or a scalar, recycled). Evaluated against each strip with the caller's environment as the enclosing scope.
`band`	Band read from every input (1-based). Default 1.
`path`	Optional output `.vec` path. When given the result is streamed to disk and the opened `vec_open_raster()` handle is returned invisibly; when `NULL` the result is returned as an in-memory matrix.
`dtype`	Storage dtype for `.vec` output (see `vec_write_raster()`). Default `"f32"`.
`compression`	Compression effort for `.vec` output. Default `"fast"`.

Details

This is the *monoid fold* tier of the spatial toolbox: bounded to one strip per input, a single streaming pass, no spill. The rasters must share dimensions and geotransform; `warp()` them onto a common grid first if they do not. No **sf** is needed.

Value

When `path` is `NULL`, a numeric matrix (row 1 northmost) carrying `gt`, `extent`, and `crs` attributes. When `path` is given, the written `vectra_raster` handle (invisibly).

See Also

`warp()` to align rasters onto a shared grid first, `focal()` for neighbourhood rather than cellwise calculation.

Examples

```
nir <- matrix(c(40, 50, 60, 70), 2, 2)
red <- matrix(c(10, 20, 30, 40), 2, 2)
fn <- tempfile(fileext = ".vec"); fr <- tempfile(fileext = ".vec")
vec_write_raster(nir, fn, dtype = "f64", extent = c(0, 0, 2, 2))
vec_write_raster(red, fr, dtype = "f64", extent = c(0, 0, 2, 2))

ndvi <- rast_calc(list(nir = fn, red = fr), (nir - red) / (nir + red))
round(ndvi, 3)
unlink(c(fn, fr))
```

---

reframe *Summarise with variable-length output per group*

Description

Like `summarise()` but allows expressions that return more than one row per group. Currently implemented via `collect()` fallback.

Usage

```
reframe(.data, ...)
```

Arguments

`.data`	A `vectra_node` object.
`...`	Named expressions.

Value

A data.frame (not a lazy node).

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(g = c("a", "a", "b"), x = c(1, 2, 3)), f)
tbl(f) |> group_by(g) |> reframe(range_x = range(x))
unlink(f)
```

relocate *Relocate columns*

Description

Relocate columns

Usage

```
relocate(.data, ..., .before = NULL, .after = NULL)
```

Arguments

.data	A vectra_node object.
...	Column names to move.
.before	Column name to place before (unquoted).
.after	Column name to place after (unquoted).

Value

A new vectra_node with reordered columns.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> relocate(hp, wt, .before = cyl) |> collect() |> head()
unlink(f)
```

rename *Rename columns*

Description

Rename columns

Usage

```
rename(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Rename pairs: new_name = old_name.

Value

A new vectra_node with renamed columns.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> rename(miles_per_gallon = mpg) |> collect() |> head()
unlink(f)
```

---

select *Select columns from a vectra query*

Description

Select columns from a vectra query

Usage

```
select(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Column names (unquoted).

Value

A new vectra_node with only the selected columns.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> select(mpg, cyl) |> collect() |> head()
unlink(f)
```

slice *Select rows by position*

Description

Select rows by position

Usage

```
slice(.data, ...)
```

Arguments

.data	A vectra_node object.
...	Integer row indices (positive or negative).

Value

A data.frame with the selected rows.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> slice(1, 3, 5)
unlink(f)
```

slice_head *Select first or last rows*

Description

Select first or last rows

Usage

```
slice_head(.data, n = 1L)

slice_tail(.data, n = 1L)

slice_min(.data, order_by, n = 1L, with_ties = TRUE)

slice_max(.data, order_by, n = 1L, with_ties = TRUE)
```

Arguments

.data	A vectra_node object.
n	Number of rows to select.
order_by	Column to order by (for slice_min/slice_max).
with_ties	If TRUE (default), includes all rows that tie with the nth value. If FALSE, returns exactly n rows.

Details

When slice_min()/slice_max() follow group_by(), the n smallest/largest rows are taken within each group and the whole winning row is kept (every column, including geometry carried as a string). with_ties = FALSE returns exactly n rows per group; with_ties = TRUE keeps rows tied at the nth value via min-rank. The n = 1, with_ties = FALSE case streams: it holds only the running winner per group, so memory scales with the number of groups (the result size), not the input. Other grouped cases buffer their input.

Value

A vectra_node for slice_head(), for grouped slice_min()/slice_max(), and for ungrouped slice_min/max(..., with_ties = FALSE). A data.frame for slice_tail() and ungrouped slice_min/max(..., with_ties = TRUE) (the default), since these must materialize all rows.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> slice_head(n = 3) |> collect()
tbl(f) |> slice_min(order_by = mpg, n = 3) |> collect()
tbl(f) |> slice_max(order_by = mpg, n = 3) |> collect()
# earliest row per group, geometry/attrs preserved:
tbl(f) |> group_by(cyl) |> slice_min(mpg, n = 1, with_ties = FALSE) |> collect()
unlink(f)
```

---

spatial_centerline *Trace the centerline (medial axis) of streamed polygons*

Description

Approximates the centerline of every polygon in a streamed layer, one batch at a time – the medial axis a per-feature transform such as a buffer cannot produce. Each polygon's boundary is densified and its Voronoi diagram taken; the Voronoi edges that fall inside the polygon trace the points equidistant from two stretches of boundary, which is its skeleton, and they are merged into maximal lines. This is the usual approximation for river or road centerlines from a filled shape; prune drops the short branches that the skeleton grows toward convex corners. A non-polygon geometry passes through unchanged.

Usage

```
spatial_centerline(
  x,
  density = NULL,
  prune = 0,
  geom = "geometry",
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
density	Boundary sampling spacing in CRS units: the polygon outline is densified to at most this vertex spacing before the Voronoi diagram is built. NULL (default) uses one-hundredth of each polygon's bounding-box diagonal.
prune	Drop centerline branches shorter than this length (CRS units), removing the short spurs the skeleton grows toward convex corners. 0 (default) keeps every branch.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

The centerline is an approximation whose detail is set by density (the boundary sampling spacing): finer sampling traces the axis more closely at more cost. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; the Voronoi construction is **sf**/GEOS and expects projected or unprojected planar data. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node of the centerlines, each carrying its source polygon's attributes (replicated if the centerline is several lines) and the input CRS, backed by temporary .vtr spills removed when the node is garbage-collected.

See Also

spatial_construct() with kind = "pole" for the single deepest interior point, spatial_simplify() to simplify a coverage, collect_sf() to materialize as sf.

Examples

```
road <- sf::st_polygon(list(rbind(
  c(0, 0), c(10, 0), c(10, 2), c(0, 2), c(0, 0))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  geometry = sf::st_as_binary(sf::st_sfc(road), hex = TRUE)
), f)

# The centerline runs down the middle of the strip.
tbl(f) |> spatial_centerline(density = 0.25, prune = 0.5) |> collect_sf()
unlink(f)
```

---

spatial_clip *Clip or erase a streamed layer against a resident mask*

Description

Streams a large layer x through the engine and cuts each batch's geometry against a small resident mask (a study boundary, a buffer, a set of patches). By default this clips – the intersection with the mask, the GIS "Clip" tool – keeping only the parts of x that fall inside mask. With erase = TRUE it instead erases – the difference, the "Erase"/"Difference" tool – keeping the parts of x outside mask. The mask is dissolved to a single geometry once and held resident while the billion-row left stream flows past one batch at a time.

Usage

```
spatial_clip(
  x,
  mask,
  erase = FALSE,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.

mask	An sf or sfc object whose dissolved geometry clips (or, with erase = TRUE, erases) the stream.
erase	If TRUE, keep the parts of x *outside* mask (difference) rather than inside (intersection). Default FALSE.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
coords	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. c("x", "y")), for inputs such as tiff_extract_points() output. The coordinate columns are retained.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use collect_sf() to materialize. On projected or unprojected planar data the cut runs natively on the GEOS C API straight off the hex-WKB column (the mask parsed once); geographic coordinates with spherical geometry on (sf::sf_use_s2()) and coordinate-assembled (coords) input cut through **sf** instead. When mask carries no CRS it inherits the stream's.

Value

A vectra_node of the cut geometry with x's attributes, backed by temporary .vtr spills and carrying the input CRS.

See Also

spatial_filter() to keep whole features by location without cutting them, spatial_map() for per-feature transforms, collect_sf().

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
mask <- sf::st_union(nc[nc$NAME %in% c("Ashe", "Alleghany"), ])

f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  NAME = nc$NAME,
  geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
), f)

# Clip every county polygon to the two-county mask, streaming.
clipped <- tbl(f) |> spatial_clip(mask, crs = sf::st_crs(nc))
collect_sf(clipped)
```

```
unlink(f)
```

`spatial_construct` *Build a set-wise geometry construction, optionally per group*

Description

Constructs one geometry (or a tessellation) from a whole set of features – the constructions a per-feature `spatial_map()` cannot express because they need every feature in scope at once. Like `spatial_dissolve()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per by group in a single bounded pass, then each shard's geometry is combined and the construction built with **sf**. With no by, the whole layer yields one construction. Peak memory is the routing budget during the pass, then one group's geometry while it is built – partition on a key whose groups fit in memory.

Usage

```
spatial_construct(
  x,
  kind = .CONSTRUCT_KINDS,
  by = NULL,
  geom = "geometry",
  crs = NA,
  ratio = 0.3,
  allow_holes = FALSE,
  tolerance = 0,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`kind`	The construction to build; one of the values above.
`by`	Character vector of attribute columns to construct within: one construction (or tessellation) per distinct combination of their values. `NULL` (default) builds a single construction from the whole layer.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`ratio`	For `kind = "concave_hull"`, the concaveness in `[0, 1]` (1 is the convex hull). Default `0.3`.

allow_holes	For kind = "concave_hull", whether the hull may contain holes. Default FALSE.
tolerance	Distance tolerance for the kinds that take one ("inscribed_circle", "pole", "voronoi", "delaunay"). 0 (default) lets the inscribed-circle kinds derive a tolerance from the extent and the tessellations use the GEOS default.
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

kind selects the construction:

"convex_hull" the convex hull of the set.

"concave_hull" the concave hull (ratio, allow_holes).

"envelope" the axis-aligned bounding rectangle.

"oriented_box" the minimum-area rotated bounding rectangle.

"enclosing_circle" the minimum bounding circle.

"inscribed_circle" the maximum inscribed circle (largest circle that fits inside the set's union).

"pole" the pole of inaccessibility – the centre of the maximum inscribed circle, the point inside the shape farthest from its edges.

"voronoi" the Voronoi tessellation, one polygon per cell.

"delaunay" the Delaunay triangulation, one polygon per triangle.

The enclosing kinds and pole emit one feature per group; voronoi and delaunay emit one feature per cell, each carrying the group's by values.

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use collect_sf() to materialize. Topology is **sf**/GEOS throughout (an optional dependency, Suggests); some constructions need projected coordinates.

Value

A vectra_node of the construction – one row per group for the enclosing kinds, one row per cell for the tessellations – carrying the by columns and the input CRS, backed by temporary .vtr spills removed when the node is garbage-collected.

See Also

spatial_dissolve() to merge a group into one feature, spatial_map() for per-feature transforms, collect_sf() to materialize.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
nc$band <- nc$SID74 > 5
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  band = nc$band,
  geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
```

```
), f)

# One convex hull per band.
tbl(f) |>
  spatial_construct("convex_hull", by = "band", crs = sf::st_crs(nc)) |>
  collect_sf()
unlink(f)
```

`spatial_dissolve` *Dissolve geometries by group*

Description

Unions the geometries within each by group into a single feature (the GIS "Dissolve" tool), optionally summarising attributes. Unlike the streamed per-batch verbs, dissolve needs every geometry of a group together to union them, so it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per group in a single bounded pass, then each shard is read in and unioned with **sf**. Peak memory is the routing budget during the pass, then one group's geometries while it is unioned – partition the input on a key whose groups fit in memory. With no by, the whole layer dissolves into one feature.

Usage

```
spatial_dissolve(
  x,
  by = NULL,
  ...,
  geom = "geometry",
  crs = NA,
  .fun = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`by`	Character vector of attribute columns to dissolve within: one output feature per distinct combination of their values. `NULL` (default) dissolves the entire layer into a single feature.
`...`	Further arguments passed to `sf::st_union()` (e.g. `is_coverage = TRUE`).
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.

`.fun`	Optional named list of attribute summaries. Each element is a function taking the group's data.frame and returning a length-1 value; the list name becomes the output column (e.g. `.fun = list(total = function(d) sum(d$pop))`). Default `NULL` keeps only the by columns and the dissolved geometry.
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use `collect_sf()` to materialize. On projected or unprojected planar data each group is unioned natively on the GEOS C API straight off the hex-WKB column; geographic coordinates with spherical geometry on (`sf::sf_use_s2()`), or any extra `sf::st_union()` arguments (e.g. `is_coverage = TRUE`), union through **sf** instead. The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of one row per group – the by columns, any `.fun` summaries, and the dissolved geometry – backed by temporary `.vtr` spills removed when the node is garbage-collected, carrying the input CRS for `collect_sf()`.

See Also

`spatial_overlay()` to split overlaps apart rather than merge them, `offload()` for the partition tier this rides on, `collect_sf()`.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
nc$band <- nc$SID74 > 5            # an attribute to dissolve within
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  band = nc$band, BIR74 = nc$BIR74,
  geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
), f)

# Merge the counties into two features by `band`, summing births.
merged <- tbl(f) |>
  spatial_dissolve(by = "band", crs = sf::st_crs(nc),
                   .fun = list(births = function(d) sum(d$BIR74)))
collect_sf(merged)
unlink(f)
```

`spatial_eliminate` *Merge sliver polygons into a neighbour*

Description

Cleans a polygon coverage by absorbing every feature whose area is below `max_area` into an adjacent feature (the QGIS "Eliminate Selected Polygons"): the sliver removal a per-feature transform cannot do, because the target a sliver merges into is one of its neighbours, not the sliver itself. Each small feature is joined to the neighbour it shares the longest border with (or the largest-area neighbour, with `into = "largest_area"`); chains of slivers collapse so a connected run of small features flows to its single largest member, whose attribute row survives. A small feature with no neighbour is kept unchanged, so nothing vanishes. Like `spatial_dissolve()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per `by` group in a single bounded pass, and each group is cleaned as an independent coverage. Peak memory is the routing budget during the pass, then one group's geometry while its slivers are merged – partition on a key whose groups fit in memory. With no `by`, the whole layer is one coverage.

Usage

```
spatial_eliminate(
  x,
  max_area,
  by = NULL,
  into = c("longest_border", "largest_area"),
  geom = "geometry",
  crs = NA,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`max_area`	Area threshold in CRS units squared: a feature smaller than this is a sliver and is merged into a neighbour. Larger values absorb more.
`by`	Character vector of attribute columns whose groups are each cleaned as an independent coverage. `NULL` (default) treats the whole layer as one coverage.
`into`	How to pick the neighbour a sliver merges into: `"longest_border"` (default) the neighbour sharing the longest boundary, or `"largest_area"` the neighbour with the greatest area.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Adjacency and shared-border length are **sf**/GEOS (sf::st_intersects, sf::st_boundary, sf::st_intersection) and expect projected or unprojected planar data; `max_area` is in CRS units squared. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node. The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of the cleaned coverage – one row per surviving feature, each carrying its (largest member's) attributes and the input CRS, backed by temporary `.vtr` spills removed when the node is garbage-collected.

See Also

`spatial_dissolve()` to merge geometries by attribute, `spatial_simplify()` for coverage-preserving simplification, `spatial_topology()` for the shared-edge adjacency, `collect_sf()` to materialize as `sf`.

Examples

```
big    <- sf::st_polygon(list(rbind(
  c(0, 0), c(10, 0), c(10, 10), c(0, 10), c(0, 0))))
sliver <- sf::st_polygon(list(rbind(
  c(10, 0), c(10.3, 0), c(10.3, 10), c(10, 10), c(10, 0))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = c("keep", "sliver"),
  geometry = sf::st_as_binary(sf::st_sfc(big, sliver), hex = TRUE)
), f)

# The thin sliver is absorbed into the square it borders.
tbl(f) |> spatial_eliminate(max_area = 5) |> collect_sf()
unlink(f)
```

---

spatial_explode *Explode multipart geometries into single-part features*

Description

Streams a lazy vectra query and splits every multipart geometry into its component single-part geometries: a `MULTIPOLYGON` becomes one row per polygon, a `MULTILINESTRING` one row per linestring, a `MULTIPOINT` one row per point, and a `GEOMETRYCOLLECTION` one row per member (recursively). The attributes of the source feature are copied onto each part. Already single-part geometries pass through unchanged, as does an empty geometry (kept as one row). This is the streaming counterpart of the QGIS "multipart to singleparts" tool and of `sf::st_cast()` to a single-part type.

Usage

```
spatial_explode(
  x,
  geom = "geometry",
  crs = NA,
  out_geom = NULL,
  part = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
part	Optional name of an integer column numbering the parts within each source feature, 1-based. Default NULL adds no such column.
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

One batch is decoded, exploded, and spilled at a time, so peak memory tracks one batch and its parts, not the whole layer.

Value

A vectra_node of single-part features, backed by temporary .vtr spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

spatial_map() for per-feature transforms, spatial_dissolve() to merge features the other way, collect_sf() to materialize as sf.

Examples

```
mp <- sf::st_multipolygon(list(
  list(rbind(c(0, 0), c(1, 0), c(1, 1), c(0, 1), c(0, 0))),
  list(rbind(c(2, 2), c(3, 2), c(3, 3), c(2, 3), c(2, 2)))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
```

```
  id = 1L,
  geometry = sf::st_as_binary(sf::st_sfc(mp), hex = TRUE)
), f)

# One row per polygon, attributes copied, parts numbered.
tbl(f) |> spatial_explode(part = "part_id") |> collect_sf()
unlink(f)
```

`spatial_filter` *Keep streamed rows by their spatial relation to a resident layer*

Description

Streams a large left side `x` through the engine and keeps each row whose geometry satisfies an **sf** binary predicate against a small resident layer `y` (select by location). This is the spatial counterpart to a `semi_join()`: rows are filtered, never duplicated, and no columns are added, so the output carries `x`'s schema unchanged. With `negate = TRUE` it keeps the rows that do *not* match (select by location, inverted). The billion-row left stream never materializes; `y` (a study region, habitat patches, a coastline buffer, ...) stays resident.

Usage

```
spatial_filter(
  x,
  y,
  predicate = NULL,
  negate = FALSE,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  flush_rows = NULL,
  ...
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`y`	An `sf` or `sfc` object: the resident locator layer to test against.
`predicate`	An **sf** binary predicate function, e.g. sf::st_intersects (default), sf::st_within, sf::st_covered_by, sf::st_is_within_distance. A left row is kept when the predicate reports at least one match against `y`.
`negate`	If `TRUE`, keep the rows with no match instead (the inverted select-by-location). Default `FALSE`.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.

coords	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. c("x", "y")), for inputs such as tiff_extract_points() output. The coordinate columns are retained.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).
...	Further arguments passed to predicate, e.g. dist = for sf::st_is_within_distance.

Details

For the recognised predicates – the topological ones (intersects, within, contains, overlaps, covers, covered by, touches, crosses), equals, disjoint, and within-distance (sf::st_is_within_distance, whose radius is passed as dist =) – on projected or unprojected planar data, the test runs natively on the GEOS C API straight off the hex-WKB column: y is parsed once into a spatial index and each batch is tested in C, with no per-batch round-trip through **sf**. Coordinate-assembled (coords) point input runs natively too, building each point in C rather than through **sf**; disjoint is the one exception there (its matches are the bounding boxes the index prunes away) and keeps the **sf** loop, as it does for the join. Geographic coordinates with spherical geometry on (sf::sf_use_s2()) and any other predicate use **sf** instead, preserving its semantics. When y carries no CRS it inherits the stream's so the predicate does not reject on a mismatch.

Value

A vectra_node of the kept rows with x's schema, backed by temporary .vtr spills and carrying the input CRS.

See Also

spatial_join() to tag rows with y's attributes, spatial_clip() to cut geometry against a mask, filter() for attribute predicates.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
region <- nc[nc$NAME %in% c("Ashe", "Alleghany", "Surry"), "NAME"]

set.seed(1)
pts <- sf::st_coordinates(sf::st_sample(nc, 300))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = seq_len(nrow(pts)), x = pts[, 1], y = pts[, 2]), f)

# Keep only the points that fall inside the three-county region, streaming.
inside <- tbl(f) |>
  spatial_filter(region, coords = c("x", "y"), crs = sf::st_crs(nc))
nrow(collect(inside))
unlink(f)
```

`spatial_join` *Spatial join a streamed query against a resident sf object*

Description

Streams a large left side `x` through the engine and joins each batch against a small right side `y` held resident in memory, using an **sf** binary predicate (`st_intersects` by default). This is the spatial analogue of a hash join with the small side on the build side: the billion-row left stream never materializes, while `y` (admin polygons, habitat patches, ...) stays in RAM. The dominant real workload it serves is tagging huge point sets with the polygon they fall in.

Usage

```
spatial_join(
  x,
  y,
  join = NULL,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  left = TRUE,
  suffix = c(".x", ".y"),
  partition = NULL,
  y_geom = NULL,
  y_coords = NULL,
  out_geom = NULL,
  flush_rows = NULL,
  ...
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`y`	The right side of the join: an `sf` object held resident (the default), or – when `partition` is given – a streamed `vectra_node`.
`join`	An **sf** binary predicate function, e.g. sf::st_intersects (default), sf::st_within, sf::st_contains, sf::st_nearest_feature.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`coords`	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. `c("x", "y")`), for inputs such as `tiff_extract_points()` output. The coordinate columns are retained.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.

left	If TRUE (default) keep every left row (left join); if FALSE keep only matches (inner join).
suffix	Length-2 character vector disambiguating columns present on both sides. Default c(".x", ".y").
partition	Optional grid() specification enabling the two-sided streamed path, in which y is itself a vectra_node. Default NULL keeps the resident-y path.
y_geom, y_coords	Geometry transport for a streamed y under partition: the name of y's hex-WKB geometry column (y_geom, default the left geom), or a length-2 character vector of y's coordinate columns (y_coords). Ignored without partition.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).
...	Further arguments passed to sf::st_join().

Details

For the recognised predicates – the topological ones (intersects, within, contains, overlaps, covers, covered by, touches, crosses), equals, within-distance (sf::st_is_within_distance, radius passed as dist =), and nearest feature (sf::st_nearest_feature) – on projected or unprojected planar data, the match runs natively on the GEOS C API straight off the hex-WKB column: y is parsed once into a spatial index, each batch's matches come back from C, and y's attributes are attached in R without decoding the left side to **sf**. Coordinate-assembled (coords) point input runs natively too, building each point in C (the emitted point geometry is built in C as well). Geographic coordinates with spherical geometry on (sf::sf_use_s2()), a disjoint join (whose matches are the bounding-box complement an index cannot prune), and other extra sf::st_join() arguments use **sf** instead, preserving its semantics.

When both sides are larger than RAM, pass partition = grid(cellsize) and a streamed vectra_node as y: both inputs are binned to a uniform spatial grid, then joined one shard at a time. Each left feature is assigned to the single grid cell of its reference point while each right feature is replicated to every cell its bounding box overlaps, so a left row is emitted exactly once and the result equals the resident join. This is exact for point left geometries (the dominant case – tagging a huge point set with the polygon it falls in) and finds, for an extended left feature, the matches whose right bounding box overlaps the left reference cell; choose a cellsize larger than the left features for an extended-on-extended join. The partition path serves topological predicates (intersects, within, contains, overlaps, covers, covered by). It also serves sf::st_nearest_feature: because nearest is not local to one cell, each left feature then searches its own cell and the eight around it, so the true nearest is found when it lies within one cell of the left reference cell (pick a cellsize at least the largest expected nearest distance). Topology and CRS handling are **sf**'s; vectra supplies the stream and the grid partition.

Value

A vectra_node of the joined stream, backed by temporary .vtr spills and carrying the left CRS.

See Also

spatial_map() for per-feature transforms, collect_sf() to materialize as sf, offload() to partition both-sides-huge joins.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)

# A stream of points, stored with x/y coordinate columns.
set.seed(1)
pts <- sf::st_coordinates(sf::st_sample(nc, 200))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = seq_len(nrow(pts)), x = pts[, 1], y = pts[, 2]), f)

# Tag each point with the county it falls in, streaming.
tagged <- tbl(f) |>
  spatial_join(nc["NAME"], join = sf::st_intersects,
               coords = c("x", "y"), crs = sf::st_crs(nc))
head(collect(tagged))

# Both sides streamed: bin to a grid and join per shard. Here y is a
# vectra_node rather than a resident sf object.
g <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  NAME = nc$NAME,
  geometry = sf::st_as_binary(sf::st_geometry(nc), hex = TRUE)
), g)
tagged2 <- tbl(f) |>
  spatial_join(tbl(g), coords = c("x", "y"), crs = sf::st_crs(nc),
               partition = grid(0.5))
head(collect(tagged2))
unlink(c(f, g))
```

---

spatial_knn *k nearest neighbours of a streamed layer, with distances*

Description

Streams a large left side x through the engine and, for each feature, finds the k nearest features of a small resident layer y, returning one row per (left, neighbour) pair with the neighbour's rank, identifier, and distance. Where spatial_join() with sf::st_nearest_feature attaches only the single nearest match, this returns the top k and the distances themselves – the nearest-k query and the building block of a distance matrix. The billion-row left stream never materializes; y (the candidate neighbours) stays resident.

Usage

```
spatial_knn(
  x,
  y,
  k = 1L,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  y_id = NULL,
  id_col = "neighbor",
  dist_col = "distance",
  rank_col = "rank",
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
y	An sf or sfc object: the resident candidate-neighbour layer.
k	Number of nearest neighbours to return per left feature (capped at the number of y features). Default 1.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
coords	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. c("x", "y")), for inputs such as tiff_extract_points() output. The coordinate columns are retained.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
y_id	Optional name of a column in y whose value identifies each neighbour in the output. Default NULL uses y's 1-based row index.
id_col, dist_col, rank_col	Names of the output columns holding the neighbour identifier, the distance, and the 1-based rank (1 = nearest). Defaults "neighbor", "distance", "rank".
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Distances are **sf**'s sf::st_distance(): planar (CRS units) on projected or unprojected planar data, great-circle (metres) on geographic coordinates with spherical geometry on (sf::sf_use_s2()). Each batch forms its left-by-y distance matrix, so y should be the small side; when y carries no CRS

it inherits the stream's. The left geometry rides through unchanged (replicated once per neighbour). The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of one row per (left, neighbour) pair – `x`'s columns (geometry included) plus the rank, neighbour identifier, and distance – backed by temporary `.vtr` spills (removed when the node is garbage- collected) and carrying the input CRS.

See Also

`spatial_join()` for a nearest-feature attribute join, `collect_sf()` to materialize as `sf`.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
towns <- sf::st_centroid(sf::st_geometry(nc))[1:5]
towns <- sf::st_sf(town = nc$NAME[1:5], geometry = towns)

set.seed(1)
pts <- sf::st_coordinates(sf::st_sample(nc, 100))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = seq_len(nrow(pts)), x = pts[, 1], y = pts[, 2]), f)

# The two nearest towns to each point, with distances.
tbl(f) |>
  spatial_knn(towns, k = 2, coords = c("x", "y"), crs = sf::st_crs(nc),
              y_id = "town") |>
  collect() |> head()
unlink(f)
```

---

spatial_line_merge *Merge contiguous line segments into maximal lines*

Description

Sews the line segments of each group into the longest possible linestrings (`sf::st_line_merge`, the line counterpart of a dissolve): segments that meet end to end become one chain, and each chain is emitted as its own row. Where a plain union of lines returns a single multilinestring of all the parts, this joins the parts through their shared endpoints; at a crossing where more than two segments meet the merge is ambiguous and the segments stay separate. Like `spatial_dissolve()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per `by` group in a single bounded pass, then each group's segments are merged together. Peak memory is the routing budget during the pass, then one group's geometry while it is merged. With no `by`, the whole layer is merged at once.

Usage

```
spatial_line_merge(
  x,
  by = NULL,
  geom = "geometry",
  crs = NA,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
by	Character vector of attribute columns to merge within: one set of maximal lines per distinct combination of their values. NULL (default) merges the whole layer at once.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Each merged line is new geometry built from the whole group, so it carries the by columns only, not the attributes of any single source segment. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node of one row per maximal merged line, carrying the by columns and the input CRS, backed by temporary .vtr spills removed when the node is garbage-collected.

See Also

spatial_dissolve() to union geometries by group, spatial_explode() for the opposite direction (multipart to single part), collect_sf() to materialize as sf.

Examples

```
seg <- sf::st_sfc(
  sf::st_linestring(rbind(c(0, 0), c(1, 0))),
  sf::st_linestring(rbind(c(1, 0), c(2, 0))),
  sf::st_linestring(rbind(c(2, 0), c(3, 0))))
f <- tempfile(fileext = ".vtr")
```

```
write_vtr(data.frame(
  geometry = sf::st_as_binary(seg, hex = TRUE)
), f)

# The three end-to-end segments become one line.
tbl(f) |> spatial_line_merge() |> collect_sf()
unlink(f)
```

`spatial_locate` *Locate streamed points along a resident line layer*

Description

Streams a large point layer `x` through the engine and, for each point, finds the nearest line of a small resident `line` layer and where the point falls along it – linear referencing (`sf::st_line_project`). Each point gets the identifier of its nearest line, the **measure** (distance along that line from its start to the point's projection), and the perpendicular distance to the line. With `snap = TRUE` the point geometry is moved onto the line at that measure. This is the two-layer companion to a per-feature `sf::st_line_interpolate`, which goes the other way (a measure back to a point); the billion-row point stream never materializes, while `line` (the reference network) stays resident.

Usage

```
spatial_locate(
  x,
  line,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  y_id = NULL,
  id_col = "line",
  measure_col = "measure",
  dist_col = "distance",
  snap = FALSE,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`line`	An `sf` or `sfc` object of the reference lines (the resident layer).
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.

coords — Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. `c("x", "y")`), for inputs such as `tiff_extract_points()` output. The coordinate columns are retained.

crs — Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.

y_id — Optional name of a column in `line` whose value identifies the matched line in the output. Default `NULL` uses `line`'s 1-based row index.

id_col, measure_col, dist_col — Names of the output columns holding the matched-line identifier, the measure along the line, and the perpendicular distance. Defaults `"line"`, `"measure"`, `"distance"`.

snap — If `TRUE`, replace each point's geometry with its projection onto the nearest line. Default `FALSE` keeps the original points.

out_geom — Name of the output geometry column. Defaults to `geom` (or `"geometry"` when `coords` is used).

flush_rows — Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Nearest line and distance are **sf**'s sf::st_nearest_feature and sf::st_distance: planar (CRS units) on projected or unprojected planar data, great-circle (metres) on geographic coordinates with spherical geometry on (`sf::sf_use_s2()`). Points arrive either as a hex-WKB geometry column (`geom`) or as two coordinate columns (`coords`). The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of `x`'s rows – geometry included (or snapped onto the line) – plus the matched-line identifier, the measure, and the perpendicular distance, backed by temporary `.vtr` spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

`spatial_knn()` for nearest neighbours with distances, `spatial_join()` for a nearest-feature attribute join, `spatial_map()` with `~ sf::st_line_interpolate(line, .x$m)` for the inverse, `collect_sf()` to materialize as `sf`.

Examples

```
line <- sf::st_sfc(
  sf::st_linestring(rbind(c(0, 0), c(10, 0))),
  sf::st_linestring(rbind(c(0, 5), c(0, 15))))
line <- sf::st_sf(road = c("main", "side"), geometry = line)
pts  <- data.frame(id = 1:2, x = c(3, 1), y = c(1, 9))
f <- tempfile(fileext = ".vtr")
write_vtr(pts, f)
```

```
# Each point's position along its nearest road.
tbl(f) |>
  spatial_locate(line, coords = c("x", "y"), y_id = "road") |>
  collect()
unlink(f)
```

`spatial_map` *Stream a query through an sf transform*

Description

Applies a per-feature **sf** operation (buffer, centroid, area, CRS transform, simplify, ...) to a lazy vectra query one batch at a time and returns a new lazy node. The engine pulls one batch, hands it to `fn` as an `sf` object, encodes the result back into the stream, and spills to disk, so peak memory is one batch regardless of result size. This is the streaming, larger-than-RAM counterpart to running the same `sf` call on a whole in-memory table.

Usage

```
spatial_map(
  x,
  fn,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`fn`	A function (or purrr-style formula such as `~ sf::st_buffer(.x, 1000)`) taking one `sf` batch and returning an `sf` object, `sfc`, or plain data.frame. The active geometry of the return becomes the output geometry.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`coords`	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. `c("x", "y")`), for inputs such as `tiff_extract_points()` output. The coordinate columns are retained.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.

out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Geometry travels through the engine as hex-encoded WKB in an ordinary string column (vectra has no native geometry type), and the coordinate reference system is carried on the returned node rather than in the .vtr file. Use collect_sf() to materialize the result as an sf object, or collect() to get the underlying data.frame with the WKB string column.

Topology is delegated entirely to **sf**/GEOS; vectra only supplies the streaming. The sf package is an optional dependency (Suggests).

Value

A vectra_node backed by temporary .vtr spills (removed when the node is garbage-collected), carrying the output CRS for collect_sf().

See Also

spatial_join() to join a streamed side against a resident sf object, collect_sf() to materialize as sf.

Examples

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  NAME = nc$NAME,
  geometry = sf::st_as_binary(sf::st_centroid(sf::st_geometry(nc)),
                              hex = TRUE)
), f)

# Buffer every county centroid by 0.1 degree, streaming.
buffered <- tbl(f) |>
  spatial_map(~ sf::st_buffer(.x, 0.1), crs = sf::st_crs(nc))
collect_sf(buffered)
unlink(f)
```

---

spatial_network *Build a routable network graph from a line layer*

Description

Builds the node-edge graph a shortest-path query needs: nodes at line endpoints, edges weighted by geometry length or a cost column. The graph is held resident in an external pointer and passed to `spatial_route()` and `spatial_service_area()`, which stream origin (and destination) batches past it. This is the network counterpart of a resident `sf` `y` in `spatial_knn()`: the graph is the resident budget (bounded by the network size), while the query side scales by streaming.

Usage

```
spatial_network(
  lines,
  weight = NULL,
  directed = FALSE,
  direction = NULL,
  weight_to = NULL,
  tolerance = 0,
  geom = "geometry",
  crs = NA
)
```

Arguments

`lines`	An `sf` or `sfc` object of (multi)linestrings, or a `vectra_node` carrying a hex-WKB line geometry column (it is materialised).
`weight`	Name of a column in `lines` holding each edge's traversal cost. `NULL` (default) uses the geometry length (`sf::st_length()`).
`directed`	If `TRUE`, edges are one-way (by `direction`, or the digitised from->to direction). `FALSE` (default) makes every edge two-way.
`direction`	Name of a column of one-way codes, used when `directed`: `"B"` two-way, `"FT"` from->to only, `"TF"` to->from only, `"N"` closed (the `+/-/0` sign convention is also accepted). `NULL` uses `"FT"` for every line, or `"B"` when `weight_to` is given.
`weight_to`	Name of a column holding the reverse-direction cost on a two-way edge of a directed graph. `NULL` reuses `weight`.
`tolerance`	Endpoints within this distance (CRS units) are snapped to one node. `0` (default) joins only exactly-coincident endpoints.
`geom`, `crs`	Geometry column name and CRS, as in `spatial_map()`. The CRS defaults to the one `lines` carries.

Details

Endpoints within `tolerance` of each other are snapped to a single node, so a layer whose touching lines do not share exactly-equal endpoint coordinates still connects. The input must be (multi)linestrings; a `MULTILINESTRING` is split into its parts, each becoming one edge with the source attributes. Lines are treated as already split at their junctions (the usual road-network convention); two lines that merely cross in their interiors are not connected unless they share an endpoint. The graph and the Dijkstra solver are native C; **sf** (an optional dependency, Suggests) supplies only the geometry.

Value

A `vectra_network` object: the resident graph (an external pointer plus the node coordinates, edge table, and source-line geometry needed to snap queries and rebuild routes). Print it to see the node, edge, and connected-component counts.

See Also

`spatial_route()` for shortest paths and origin-destination costs, `spatial_service_area()` for reachability and isochrones.

Examples

```
# A small grid of streets.
mk <- function(x1, y1, x2, y2)
  sf::st_linestring(rbind(c(x1, y1), c(x2, y2)))
streets <- sf::st_sfc(
  mk(0, 0, 1, 0), mk(1, 0, 2, 0), mk(0, 0, 0, 1),
  mk(0, 1, 1, 1), mk(1, 0, 1, 1), mk(1, 1, 2, 1), mk(2, 0, 2, 1))
net <- spatial_network(streets)
net
```

---

spatial_overlay *Self-overlay a polygon layer into disjoint pieces (QGIS-style Union)*

Description

Splits a polygon layer along all its own overlaps into disjoint pieces and returns a lazy node with one row per piece per covering polygon: where k polygons overlap, that piece appears k times, each row carrying one source polygon's attributes. This is the union overlay GIS tools expose as "Union (single layer)", with the overlap retained once per contributing feature rather than dissolved. Resolve the duplicates with a grouped `slice_min()` / `slice_max()` – for example earliest designation year wins: `group_by(piece_id) |> slice_min(year)`.

Usage

```
spatial_overlay(
  x,
  y = NULL,
  vars = NULL,
  vars_y = NULL,
  how = c("intersection", "union", "identity", "symdiff"),
  piece = "piece_id",
  geom = "geometry",
  grid = NULL,
  precision = NULL,
  dedup = TRUE,
  flush_rows = NULL,
```

```
  mem_limit = NULL,
  threads = NULL,
  quiet = TRUE,
  layer = NULL,
  query = NULL,
  layer_y = NULL,
  query_y = NULL,
  read_chunk = NULL
)
```

Arguments

x	An sf object with polygon or multipolygon geometry, or a single path to a vector file (e.g. a GeoPackage). A path is read in feature batches via layer / query, so the whole layer is never held in memory at once – peak memory then tracks the cleaned geometry, not the source size, which lets a larger-than-RAM layer overlay on a modest machine.
y	Optional second layer to overlay x against, in the same forms x accepts (an sf object or a file path read via layer_y / query_y). It must share the CRS of x. NULL (the default) self-unions x.
vars	Character vector of attribute columns of x to carry onto each piece. Default NULL keeps them all; name a subset to keep the streamed output narrow.
vars_y	Character vector of attribute columns of y to carry onto each piece (two-layer overlay only). Default NULL keeps them all. A name shared with an x column is disambiguated with a .x / .y suffix in the output.
how	For a two-layer overlay, which pieces to keep: "intersection" (covered by both layers; the default), "union" (every piece of either, the absent side's attributes filled with NA), "identity" (all of x, split by y, with y's attributes where it covers and NA elsewhere), or "symdiff" (pieces in exactly one layer). Ignored when y = NULL.
piece	Name of the integer piece-id column added to the output (the key you group by to resolve overlaps). Default "piece_id".
geom	Name of the output hex-WKB geometry column. Default "geometry".
grid	Fixed-precision snapping grid size in CRS units. Coordinates are snapped to this grid before noding so near-duplicate shared boundaries merge into one. NULL (the default) derives it from coordinate magnitude (max(abs(st_bbox(x))) * 3e-8), which suits projected layers. Pass a number to override when that default is too coarse for fine geometry (or too coarse because an outlier coordinate inflated the magnitude), or 0 to disable snapping entirely.
precision	Fixed-precision grid size, in CRS units, for noding the boundary linework. Noding on a fixed grid is deterministic and avoids the floating noder's repair-and-retry on dense overlapping linework, which is what makes a large dense layer feasible to overlay. It is far finer than grid so intersection points are not collapsed. NULL (the default) derives it from coordinate magnitude (max(abs(st_bbox(x))) * 1e-13); pass a number to override, or 0 to node in floating precision.

dedup — Overlay one representative per group of byte-identical cleaned geometries and fan the per-record attributes back onto its pieces afterwards. Duplicates add no faces, so the result is identical; this only removes the redundant noding when many records are stacked over one site (common in WDPA-style data). TRUE by default; set FALSE to overlay every record.

flush_rows — Exploded rows buffered before a spill flush. Defaults to getOption("vectra.spatial_flush", 5e5).

mem_limit — Approximate peak working-set budget in bytes, bounding the per-tile size (tile_bytes = mem_limit / (threads * 24)). It is a throughput knob with an interior optimum, not "bigger is faster": too small replicates features across many tiles, too large nodes too much linework per tile (a superlinear cost), and a budget of tens of GB runs slower than the default on a dense layer. Lower it for tighter memory. Defaults via getOption("vectra.overlay_mem_limit", ...) to a value that scales with threads to hold the per-tile size near its measured optimum.

threads — Number of OpenMP threads for the per-component overlay within a chunk. 0 (the default, via getOption("vectra.overlay_threads", 0)) uses all available cores.

quiet — If FALSE, show a text progress bar over the overlay chunks.

layer — When x is a file path, the name of the layer to read. Ignored for an sf x. Supply this or query.

query — When x is a file path, a SQL statement selecting the features to overlay (read in batches via LIMIT/OFFSET); use it instead of layer for a subset or join. With query and no layer, pass grid explicitly, since the layer extent cannot be read from the file metadata.

layer_y, query_y — The layer / query equivalents for a file-path y.

read_chunk — Features per read/parse batch. NULL (default) sizes it from available RAM. Smaller batches lower peak memory; larger ones do fewer round trips.

Details

The topology is done once with **sf**/GEOS and tiled over connected overlap clusters (disjoint clusters never share a piece, so the tiling is exact and bounded in memory), then the exploded pieces are streamed to a .vtr and handed back as a lazy node. Geometry rides through the engine as hex-encoded WKB in a string column; the CRS is carried on the node for collect_sf().

The overlay runs on a fixed-precision model: coordinates are snapped to a grid derived from their own magnitude so the pieces come out disjoint and their areas reconstruct the union of the inputs, instead of drifting by the fraction of a percent that floating-point sliver artefacts on invalid input otherwise introduce. Inputs are also passed through sf::st_make_valid().

With a second layer y, the same machinery overlays two layers instead of self-unioning one: both layers are noded together into one planar partition, and each piece carries the attributes of the x record and the y record that cover it. how selects which pieces to keep – the intersection (pieces covered by both), the union (every piece of either), x split by y ("identity"), or the parts in exactly one layer ("symdiff"). With y = NULL (the default) the function self-unions x and how is ignored.

Value

A `vectra_node` over the exploded overlay, backed by temporary `.vtr` spills removed when the node is garbage-collected, carrying the CRS of `x` for `collect_sf()`. For a self-union it is one row per piece per covering polygon; for a two-layer overlay one row per piece per covering `x`-record / `y`-record pair, with the columns of both layers.

See Also

`slice_min()` / `slice_max()` to resolve each piece to one winner, `collect_sf()` to materialize as `sf`.

Examples

```
# Two overlapping squares designated in different years.
sq <- function(a, b) sf::st_polygon(list(rbind(
  c(a, 0), c(b, 0), c(b, 1), c(a, 1), c(a, 0))))
polys <- sf::st_sf(year = c(1990L, 2010L),
                   geometry = sf::st_sfc(sq(0, 2), sq(1, 3)))

# Split into disjoint pieces; earliest year wins where they overlap.
first <- spatial_overlay(polys) |>
  group_by(piece_id) |>
  slice_min(year, n = 1, with_ties = FALSE) |>
  collect_sf()
first

# Two-layer overlay: intersect the squares with a zone layer, keeping both
# sets of attributes on each overlapping piece.
zones <- sf::st_sf(zone = c("A", "B"),
                   geometry = sf::st_sfc(sq(0, 1.5), sq(1.5, 3)))
inter <- spatial_overlay(polys, zones, how = "intersection") |> collect_sf()
inter
```

---

spatial_polygonize *Build polygonal faces from a line network*

Description

Forms the polygons enclosed by a set of lines (the QGIS "Polygonize", GEOS `Polygonize`): the inverse of taking polygon boundaries. The lines of each group are unioned and noded so every crossing becomes a shared vertex, then the faces of that planar arrangement are returned, one per row. A pile of lines that does not close any area yields no faces. Like `spatial_dissolve()` and `spatial_construct()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per by group in a single bounded pass, then each group's lines are polygonized together. Peak memory is the routing budget during the pass, then one group's geometry while its faces are built – partition on a key whose groups fit in memory. With no `by`, the whole layer yields one set of faces.

Usage

```
spatial_polygonize(
  x,
  by = NULL,
  geom = "geometry",
  crs = NA,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
by	Character vector of attribute columns to polygonize within: one set of faces per distinct combination of their values. NULL (default) polygonizes the whole layer at once.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Each face is new geometry built from the whole group, so it carries the by columns only, not the attributes of any single source line. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; the noding is **sf**/GEOS and expects projected or unprojected planar data. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node of one row per face, carrying the by columns and the input CRS, backed by temporary .vtr spills removed when the node is garbage-collected.

See Also

spatial_split() to cut existing polygons by a blade, spatial_construct() for hulls and tessellations, spatial_dissolve() to merge geometries by group, collect_sf() to materialize as sf.

Examples

```
grid <- sf::st_sfc(
  sf::st_linestring(rbind(c(0, 0), c(2, 0))),
  sf::st_linestring(rbind(c(0, 1), c(2, 1))),
  sf::st_linestring(rbind(c(0, 2), c(2, 2))),
```

```
  sf::st_linestring(rbind(c(0, 0), c(0, 2))),
  sf::st_linestring(rbind(c(1, 0), c(1, 2))),
  sf::st_linestring(rbind(c(2, 0), c(2, 2))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  geometry = sf::st_as_binary(grid, hex = TRUE)
), f)

# The four unit cells enclosed by the grid of lines.
tbl(f) |> spatial_polygonize() |> collect_sf()
unlink(f)
```

`spatial_route` *Shortest paths and origin-destination costs over a network*

Description

Streams a layer of origins `x` past a resident `spatial_network()` and, for each origin, finds the shortest path to one or more destinations `to`. Each origin and destination is snapped to its nearest graph node; the solver (native-C Dijkstra) returns one row per (origin, destination) pair with the total cost and, by default, the route geometry. With `geometry = FALSE` only the cost is returned, so a destination set per origin yields the origin-destination cost matrix in long form. The billion-origin stream never materialises; the graph stays resident.

Usage

```
spatial_route(
  x,
  network,
  to,
  to_id = NULL,
  geometry = TRUE,
  cost_col = "cost",
  dest_col = "destination",
  geom = "geometry",
  coords = NULL,
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` of origin features (point geometry, or `coords`).
`network`	A `vectra_network` from `spatial_network()`.
`to`	An `sf`/`sfc` of destination points. One destination routes every origin to it; several produce one row per (origin, destination).

`to_id`	Optional column in `to` identifying each destination in the output. `NULL` (default) uses the 1-based destination index.
`geometry`	If `TRUE` (default) each row carries the route line; if `FALSE` only the cost column (the cost-matrix form, no geometry).
`cost_col, dest_col`	Names of the output cost and destination-identifier columns. Defaults `"cost"`, `"destination"`.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`coords`	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. `c("x", "y")`), for inputs such as `tiff_extract_points()` output. The coordinate columns are retained.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`out_geom`	Name of the output geometry column. Defaults to `geom` (or `"geometry"` when `coords` is used).
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

An unreachable destination returns an infinite cost and an empty geometry rather than dropping the row, so a cost matrix stays rectangular. Snapping is to the nearest node, so place origins and destinations on or near the network; costs are in the units of the network's `weight` (or CRS length units when the graph was built from geometry length). The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node`: one row per (origin, destination) carrying `x`'s attributes, the destination id, the cost, and (when `geometry = TRUE`) the route geometry. Backed by temporary `.vtr` spills removed when the node is garbage-collected, and carrying the network CRS.

See Also

`spatial_network()` to build the graph, `spatial_service_area()` for reachability, `collect_sf()` to materialize routes as `sf`.

Examples

```
mk <- function(x1, y1, x2, y2)
  sf::st_linestring(rbind(c(x1, y1), c(x2, y2)))
streets <- sf::st_sfc(
  mk(0, 0, 1, 0), mk(1, 0, 2, 0), mk(0, 0, 0, 1),
  mk(0, 1, 1, 1), mk(1, 0, 1, 1), mk(1, 1, 2, 1), mk(2, 0, 2, 1))
net <- spatial_network(streets)
```

```
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = 1:2, x = c(0, 0), y = c(0, 1)), f)
dest <- sf::st_sfc(sf::st_point(c(2, 1)))

tbl(f) |>
  spatial_route(net, to = dest, coords = c("x", "y")) |>
  collect_sf()
unlink(f)
```

`spatial_service_area` *Service areas and isochrones over a network*

Description

Streams a layer of origins `x` past a resident `spatial_network()` and, for each origin, finds every part of the network reachable within a cost budget - the service area, or, with several budgets, nested travel-cost isochrone bands. Each origin is snapped to its nearest graph node and a budget-bounded Dijkstra (native C) collects the reachable nodes; the reachable set is returned as the convex hull (`output = "polygon"`, the generalised service area), the reachable edges (`"lines"`), or the reachable nodes (`"nodes"`).

Usage

```
spatial_service_area(
  x,
  network,
  cost,
  output = c("polygon", "lines", "nodes"),
  band_col = "band",
  geom = "geometry",
  coords = NULL,
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` of origin features (point geometry, or `coords`).
`network`	A `vectra_network` from `spatial_network()`.
`cost`	A cost budget (scalar), or several budgets for nested isochrone bands (e.g. `c(5, 10, 15)`).
`output`	`"polygon"` (default) for the convex hull of the reachable nodes, `"lines"` for the reachable edges, or `"nodes"` for the reachable nodes as a multipoint.
`band_col`	Name of the output column holding each row's budget. Default `"band"`.

geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
coords	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. c("x", "y")), for inputs such as tiff_extract_points() output. The coordinate columns are retained.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

A vector cost returns one row per (origin, band), each band the area reachable within that budget, so the rows nest from the smallest budget out. Costs are in the network's weight units. The convex-hull polygon is a generalisation; use output = "lines" for the exact reachable network. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node: one row per (origin, band) carrying x's attributes, the band value, and the service-area geometry. Backed by temporary .vtr spills removed when the node is garbage-collected, and carrying the network CRS.

See Also

spatial_network() to build the graph, spatial_route() for shortest paths, collect_sf() to materialize as sf.

Examples

```
mk <- function(x1, y1, x2, y2)
  sf::st_linestring(rbind(c(x1, y1), c(x2, y2)))
streets <- sf::st_sfc(
  mk(0, 0, 1, 0), mk(1, 0, 2, 0), mk(0, 0, 0, 1),
  mk(0, 1, 1, 1), mk(1, 0, 1, 1), mk(1, 1, 2, 1), mk(2, 0, 2, 1))
net <- spatial_network(streets)

f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(id = 1L, x = 0, y = 0), f)

tbl(f) |>
  spatial_service_area(net, cost = c(1, 2), output = "lines",
                       coords = c("x", "y")) |>
  collect_sf()
unlink(f)
```

spatial_simplify *Simplify a polygon coverage without tearing shared edges*

Description

Simplifies polygon boundaries while keeping a shared border between two polygons identical on both sides, so adjacent polygons stay edge-matched with no slivers or gaps – the topology-preserving simplification that a per-feature `spatial_map(~ sf::st_simplify(.x))` cannot give, because it simplifies each polygon's copy of a shared border independently. The boundaries of each group are unioned so a shared border is one line, noded into arcs at every junction, each arc simplified once (its junction endpoints pinned), and the arcs re-polygonized; each resulting face inherits the attributes of the source polygon containing it. Like `spatial_dissolve()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per `by` group in a single bounded pass, and each group is simplified as an independent coverage. Peak memory is the routing budget during the pass, then one group's geometry while it is simplified – partition on a key whose groups fit in memory. With no `by`, the whole layer is one coverage.

Usage

```
spatial_simplify(
  x,
  tolerance,
  by = NULL,
  geom = "geometry",
  crs = NA,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`tolerance`	Distance tolerance for the boundary simplification, in CRS units: vertices that deviate less than this from the simplified line are dropped. Larger values simplify more.
`by`	Character vector of attribute columns whose groups are each simplified as an independent coverage. `NULL` (default) treats the whole layer as one coverage.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

The simplification is topology-preserving Douglas-Peucker (`dTolerance = tolerance`) on the noded boundary arcs. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; the noding is **sf**/GEOS and expects projected or unprojected planar data. The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of the simplified polygons, each carrying its source feature's attributes and the input CRS, backed by temporary `.vtr` spills removed when the node is garbage-collected.

See Also

`spatial_map()` with `~ sf::st_simplify(.x)` for independent per-feature simplification, `spatial_smooth()` for Chaikin corner-rounding, `collect_sf()` to materialize as `sf`.

Examples

```
p1 <- sf::st_polygon(list(rbind(
  c(0, 0), c(1, 0), c(1, 0.5), c(1, 1), c(0, 1), c(0, 0))))
p2 <- sf::st_polygon(list(rbind(
  c(1, 0), c(2, 0), c(2, 1), c(1, 1), c(1, 0.5), c(1, 0))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = c("a", "b"),
  geometry = sf::st_as_binary(sf::st_sfc(p1, p2), hex = TRUE)
), f)

# The shared edge is simplified once, so the two polygons stay edge-matched.
tbl(f) |> spatial_simplify(tolerance = 0.6) |> collect_sf()
unlink(f)
```

---

spatial_smooth *Smooth streamed line and polygon geometry*

Description

Rounds the corners of every line and polygon in a streamed layer by Chaikin corner-cutting, one batch at a time. Each iteration replaces every vertex with two points a quarter and three-quarters of the way along its adjacent edges, so sharp angles become a sequence of short chamfers that read as a smooth curve; more `iterations` give a smoother result with more vertices. Open lines keep their endpoints (`keep_ends`); polygon rings are cut cyclically and shrink slightly, as Chaikin smoothing does. Point geometry passes through unchanged.

Usage

```
spatial_smooth(
  x,
  iterations = 2L,
  keep_ends = TRUE,
  geom = "geometry",
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.
iterations	Number of corner-cutting passes (a positive integer). Each pass roughly doubles the vertex count. Default 2.
keep_ends	If TRUE (default), pin the first and last vertex of an open line so it is not shortened at its tips. Ignored for closed rings.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

The smoothing is computed directly on the coordinates (no GEOS call), so it is dependency-light; **sf** is used only to decode and rebuild each batch. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use collect_sf() to materialize.

Value

A vectra_node of the smoothed geometry with x's attributes, backed by temporary .vtr spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

spatial_map() for per-feature transforms such as densifying with ~ sf::st_segmentize(.x, dfMaxLength) or sampling points along a line with ~ sf::st_line_sample(.x, n), collect_sf() to materialize as sf.

Examples

```
zig <- sf::st_linestring(rbind(c(0, 0), c(1, 1), c(2, 0), c(3, 1), c(4, 0)))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = 1L, geometry = sf::st_as_binary(sf::st_sfc(zig), hex = TRUE)
), f)

# Smooth the zig-zag with three corner-cutting passes.
tbl(f) |> spatial_smooth(iterations = 3) |> collect_sf()
unlink(f)
```

---

spatial_snap *Snap a streamed layer toward a resident reference layer*

Description

Streams a large layer `x` through the engine and snaps each batch's vertices toward a small resident reference layer `y` when they lie within `tolerance` (in CRS units), one batch at a time (the QGIS "snap geometries to layer"). Vertices and edges of `x` closer than `tolerance` to `y` are pulled onto `y`, which closes the small gaps and overshoots between two layers that should share a boundary. The reference layer stays resident while the billion-row left stream flows past; the snap itself is **sf**'s `sf::st_snap()`.

Usage

```
spatial_snap(
  x,
  y,
  tolerance,
  geom = "geometry",
  coords = NULL,
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`y`	An `sf` or `sfc` object: the resident reference layer to snap toward.
`tolerance`	Snapping distance in CRS units (a positive number). Vertices and edges of `x` within this distance of `y` are moved onto `y`.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.

coords	Optional length-2 character vector naming the x and y coordinate columns to assemble point geometry from (e.g. c("x", "y")), for inputs such as tiff_extract_points() output. The coordinate columns are retained.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use collect_sf() to materialize. When y carries no CRS it inherits the stream's. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node of the snapped geometry with x's attributes, backed by temporary .vtr spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

spatial_snap_grid() to snap to a grid instead of a layer, spatial_clip() for the resident-mask streaming pattern, collect_sf().

Examples

```
ref <- sf::st_sfc(sf::st_linestring(rbind(c(0, 0), c(10, 0))))
line <- sf::st_linestring(rbind(c(0, 0.2), c(5, 0.1), c(10, 0.2)))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = 1L, geometry = sf::st_as_binary(sf::st_sfc(line), hex = TRUE)
), f)

# Pull the near-zero vertices down onto the reference line.
tbl(f) |> spatial_snap(ref, tolerance = 0.5) |> collect_sf()
unlink(f)
```

---

spatial_snap_grid *Snap a streamed layer's coordinates to a fixed grid*

Description

Rounds every coordinate of a streamed layer to a regular grid of spacing `size` (in CRS units) and repairs the result, one batch at a time. This is the fixed-precision snap-rounding the overlay noder (`spatial_overlay()`) applies internally, exposed as a standalone verb: it merges near-coincident vertices and removes the slivers that floating-point coordinates leave between shared boundaries, so a layer can be cleaned (or pre-noded to a common precision) without running a full overlay. Snapping is done in C straight off the hex-WKB column; one cleaned geometry comes back per input feature, so attributes ride through untouched.

Usage

```
spatial_snap_grid(
  x,
  size,
  geom = "geometry",
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`size`	Grid spacing in CRS units (a positive number). Coordinates are rounded to the nearest multiple; a larger `size` snaps more aggressively.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`out_geom`	Name of the output geometry column. Defaults to `geom` (or `"geometry"` when `coords` is used).
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; use `collect_sf()` to materialize. The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of the snapped geometry with `x`'s attributes, backed by temporary `.vtr` spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

spatial_snap() to snap toward another layer instead of a grid, spatial_overlay() whose noding uses the same snap-rounding, collect_sf().

Examples

```
p <- sf::st_polygon(list(rbind(c(0.04, 0.03), c(1.02, 0.01),
                               c(0.98, 1.03), c(0.01, 0.97), c(0.04, 0.03))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = 1L, geometry = sf::st_as_binary(sf::st_sfc(p), hex = TRUE)
), f)

# Snap the jittered corners back onto a 0.1 grid.
tbl(f) |> spatial_snap_grid(0.1) |> collect_sf()
unlink(f)
```

---

spatial_split *Split a streamed layer by a resident blade, or return its crossing points*

Description

Streams a large layer x through the engine and cuts each batch's geometry against a small resident blade layer (the QGIS "split with lines"), one batch at a time. With extract = "pieces" (the default) every feature is divided where the blade crosses it – a polygon into the faces the blade carves out, a line into the arcs between crossings – and each piece is emitted as its own row with the source attributes copied; a feature the blade does not cross passes through as a single piece. With extract = "points" the verb instead returns, per feature, the points where it meets the blade (the "line intersections" tool), dropping features that do not cross.

Usage

```
spatial_split(
  x,
  blade,
  extract = c("pieces", "points"),
  geom = "geometry",
  crs = NA,
  out_geom = NULL,
  flush_rows = NULL
)
```

Arguments

x	A vectra_node (from tbl(), tbl_tiff(), any verb chain, ...). It is consumed by the stream.

blade	An sf or sfc object whose geometry cuts the stream (typically lines, but any geometry whose boundary can node x).
extract	"pieces" (default) to emit the split pieces, or "points" to emit the intersection points of each feature with the blade.
geom	Name of the input geometry column holding hex-WKB or WKT strings. Default "geometry". Ignored when coords is given.
crs	Coordinate reference system of the input geometry, in any form sf::st_crs() accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
out_geom	Name of the output geometry column. Defaults to geom (or "geometry" when coords is used).
flush_rows	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to getOption("vectra.spatial_flush", 5e5).

Details

The split is built from **sf**/GEOS noding and polygonization, so it expects projected or unprojected planar data; geographic coordinates are best projected first. The blade is dissolved to one geometry once and held resident while the left stream flows past. Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; when blade carries no CRS it inherits the stream's. The **sf** package is an optional dependency (Suggests).

Value

A vectra_node: with extract = "pieces", one row per piece carrying x's attributes; with extract = "points", one row per crossing feature carrying its intersection points. Backed by temporary .vtr spills (removed when the node is garbage-collected) and carrying the input CRS.

See Also

spatial_clip() to cut against a mask without dividing into pieces, spatial_overlay() to node two polygon layers into a partition, collect_sf() to materialize as sf.

Examples

```
sq <- sf::st_polygon(list(rbind(c(0, 0), c(4, 0), c(4, 4), c(0, 4), c(0, 0))))
blade <- sf::st_sfc(sf::st_linestring(rbind(c(2, -1), c(2, 5))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = 1L, geometry = sf::st_as_binary(sf::st_sfc(sq), hex = TRUE)
), f)

# Split the square into two halves along the blade.
tbl(f) |> spatial_split(blade) |> collect_sf()
unlink(f)
```

spatial_topology *Build the shared-edge topology of a polygon coverage*

Description

Decomposes a polygon coverage into the arcs of its planar topology: each border is returned once, tagged with the polygons on either side. Where the raw boundaries of adjacent polygons each carry their own copy of a shared edge, this nodes the unioned boundaries so a shared border is a single arc carrying the identifiers of both neighbours – the "build topology" of a GIS, and the adjacency a dissolve or a coverage edit needs. An internal arc names two faces; an outer arc names one and leaves the other side `NA`. Like `spatial_dissolve()` it rides the **partition tier**: `x` is spilled once and routed into one disjoint shard per `by` group in a single bounded pass, and each group is treated as an independent coverage. Peak memory is the routing budget during the pass, then one group's geometry while its arcs are built.

Usage

```
spatial_topology(
  x,
  id = NULL,
  by = NULL,
  geom = "geometry",
  crs = NA,
  face_cols = c("face1", "face2"),
  flush_rows = NULL
)
```

Arguments

`x`	A `vectra_node` (from `tbl()`, `tbl_tiff()`, any verb chain, ...). It is consumed by the stream.
`id`	Optional name of a column of `x` whose value identifies each polygon in the face columns. `NULL` (default) uses the 1-based feature order within the group.
`by`	Character vector of attribute columns whose groups are each built as an independent coverage. `NULL` (default) treats the whole layer as one coverage.
`geom`	Name of the input geometry column holding hex-WKB or WKT strings. Default `"geometry"`. Ignored when `coords` is given.
`crs`	Coordinate reference system of the input geometry, in any form `sf::st_crs()` accepts (EPSG integer, WKT, proj string). Defaults to the CRS the upstream node carries, or unknown.
`face_cols`	Length-2 character vector naming the two output columns that hold the identifiers of the polygons on either side of each arc. Default `c("face1", "face2")`.
`flush_rows`	Transformed rows buffered before a spill flush. Larger values mean fewer, bigger temporary files. Defaults to `getOption("vectra.spatial_flush", 5e5)`.

Details

Geometry travels through the engine as hex-encoded WKB in a string column and the CRS is carried on the returned node; the noding is **sf**/GEOS and expects projected or unprojected planar data. The **sf** package is an optional dependency (Suggests).

Value

A `vectra_node` of one row per arc – the arc geometry plus the two face-identifier columns (and any by columns) – carrying the input CRS and backed by temporary `.vtr` spills removed when the node is garbage-collected.

See Also

`spatial_polygonize()` to rebuild faces from arcs, `spatial_dissolve()` to merge geometries by group, `collect_sf()` to materialize as `sf`.

Examples

```
p1 <- sf::st_polygon(list(rbind(c(0, 0), c(1, 0), c(1, 1), c(0, 1), c(0, 0))))
p2 <- sf::st_polygon(list(rbind(c(1, 0), c(2, 0), c(2, 1), c(1, 1), c(1, 0))))
f <- tempfile(fileext = ".vtr")
write_vtr(data.frame(
  id = c("a", "b"),
  geometry = sf::st_as_binary(sf::st_sfc(p1, p2), hex = TRUE)
), f)

# The shared edge appears once, tagged with both neighbours.
tbl(f) |> spatial_topology(id = "id") |> collect()
unlink(f)
```

---

st_write.vectra_node *Stream a vectra node's geometry to a vector file*

Description

An `sf::st_write()` method (also reached through `sf::write_sf()`) for a `vectra_node`: writes the result a batch at a time, appending each, so the whole layer is never held in memory. This is the streaming counterpart to `collect_sf(x) |> sf::st_write(...)` – that route materializes every feature as an `sf` object first, which for a multi-million-feature result dominates memory; this route's peak is one batch.

Usage

```
## S3 method for class 'vectra_node'
st_write(
  obj,
  dsn,
```

```
  layer = NULL,
  ...,
  geom = "geometry",
  crs = NULL,
  delete_dsn = FALSE,
  quiet = TRUE
)
```

Arguments

obj	A vectra_node whose rows carry a hex-WKB geometry column (from spatial_overlay(), a grouped slice_min() / slice_max() resolution, a .vtr scan, ...). It is consumed by the stream.
dsn	Destination data source name (file path).
layer	Layer name. NULL lets **sf** derive it from dsn.
...	Unused; for S3 generic compatibility.
geom	Name of the hex-WKB geometry column. Default "geometry".
crs	CRS to tag the output with. NULL takes the CRS carried on the node.
delete_dsn	If TRUE, remove an existing dsn before writing.
quiet	Passed to sf::st_write().

Value

The dsn, invisibly.

See Also

collect_sf() to materialize the whole result as one sf object.

---

summarise *Summarise grouped data*

Description

Summarise grouped data

Usage

```
summarise(.data, ..., .groups = NULL)

summarize(.data, ..., .groups = NULL)
```

Arguments

.data	A grouped vectra_node (from group_by()).
...	Named aggregation expressions using n(), sum(), mean(), min(), max(), sd(), var(), first(), last(), any(), all(), median(), n_distinct().
.groups	How to handle groups in the result. One of "drop_last" (default), "drop", or "keep".

Details

Aggregation is hash-based by default. When the engine detects it is advantageous, it switches to a sort-based path that can spill to disk, keeping memory bounded regardless of group count.

All aggregation functions accept na.rm = TRUE to skip NA values. Without na.rm, any NA in a group poisons the result (returns NA). R-matching edge cases: sum(na.rm = TRUE) on all-NA returns 0, mean(na.rm = TRUE) on all-NA returns NaN, min/max(na.rm = TRUE) on all-NA returns Inf/-Inf with a warning.

This is a materializing operation.

Value

A vectra_node with one row per group.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> group_by(cyl) |> summarise(avg_mpg = mean(mpg)) |> collect()
unlink(f)
```

---

tbl *Create a lazy table reference from a .vtr file*

Description

Opens a vectra1 file and returns a lazy query node. No data is read until collect() is called.

Usage

```
tbl(path)
```

Arguments

path	Path to a .vtr file.

Value

A vectra_node object representing a lazy scan of the file.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
node <- tbl(f)
print(node)
unlink(f)
```

`tbl_csv` *Create a lazy table reference from a CSV file*

Description

Opens a CSV file for lazy, streaming query execution. Column types are inferred from the first 1000 rows. No data is read until `collect()` is called. Gzip-compressed files (`.csv.gz`) are supported transparently.

Usage

```
tbl_csv(path, batch_size = .DEFAULT_BATCH_SIZE)
```

Arguments

`path`	Path to a `.csv` or `.csv.gz` file.
`batch_size`	Number of rows per batch (default 65536).

Value

A `vectra_node` object representing a lazy scan of the CSV file.

Examples

```
f <- tempfile(fileext = ".csv")
write.csv(mtcars, f, row.names = FALSE)
node <- tbl_csv(f)
print(node)
unlink(f)
```

tbl_sqlite *Create a lazy table reference from a SQLite database*

Description

Opens a SQLite database and lazily scans a table. Column types are inferred from declared types in the CREATE TABLE statement. All filtering, grouping, and aggregation is handled by vectra's C engine — no SQL parsing needed. No data is read until `collect()` is called.

Usage

```
tbl_sqlite(path, table, batch_size = .DEFAULT_BATCH_SIZE)
```

Arguments

`path`	Path to a SQLite database file.
`table`	Name of the table to scan.
`batch_size`	Number of rows per batch (default 65536).

Value

A `vectra_node` object representing a lazy scan of the table.

Examples

```
f <- tempfile(fileext = ".sqlite")
write_sqlite(mtcars, f, "cars")
node <- tbl_sqlite(f, "cars")
node |> filter(cyl == 6) |> collect()
unlink(f)
```

tbl_tiff *Create a lazy table reference from a GeoTIFF raster*

Description

Opens a GeoTIFF file and returns a lazy query node. Each pixel becomes a row with columns `x`, `y`, `band1`, `band2`, etc. Coordinates are pixel centers derived from the affine geotransform. NoData values become `NA`.

Usage

```
tbl_tiff(path, batch_size = .TIFF_BATCH_SIZE)
```

Arguments

path	Path to a GeoTIFF file.
batch_size	Number of raster rows per batch (default 256).

Details

Use `filter(x >= ..., y <= ...)` for extent-based cropping and `filter(band1 > ...)` for value-based cropping. Results can be converted back to a raster with `terra::rast(df, type = "xyz")`.

Value

A `vectra_node` object representing a lazy scan of the raster.

Examples

```
f <- tempfile(fileext = ".tif")
df <- data.frame(x = as.double(rep(1:4, 3)),
                 y = as.double(rep(1:3, each = 4)),
                 band1 = as.double(1:12))
write_tiff(df, f)
node <- tbl_tiff(f)
node |> filter(band1 > 6) |> collect()
unlink(f)
```

---

tbl_xlsx *Create a lazy table reference from an Excel (.xlsx) file*

Description

Reads a sheet from an Excel workbook into a vectra node for lazy query execution. The sheet is read into memory via `openxlsx2::read_xlsx()` and then converted to vectra's internal format. Requires the **openxlsx2** package.

Usage

```
tbl_xlsx(path, sheet = 1L, batch_size = .DEFAULT_BATCH_SIZE)
```

Arguments

path	Path to an `.xlsx` file.
sheet	Sheet to read: either a name (character) or 1-based index (integer). Default `1L` (first sheet).
batch_size	Number of rows per batch (default 65536).

Value

A `vectra_node` object representing a lazy scan of the sheet.

Examples

```
if (requireNamespace("openxlsx2", quietly = TRUE)) {
  f <- tempfile(fileext = ".xlsx")
  openxlsx2::write_xlsx(mtcars, f)
  node <- tbl_xlsx(f)
  node |> filter(cyl == 6) |> collect()
  unlink(f)
}
```

---

terrain *Terrain derivatives from a streamed elevation raster*

Description

Computes DEM derivatives from a `.vec` elevation raster with Horn's 3x3 method, on the same haloed tile-row strip pass as `focal()` – the input is read one strip at a time and, when `path` is given, the outputs are streamed straight back to a multi-band `.vec`. Matches **terra**'s `terrain()` / `shade()` conventions.

Usage

```
terrain(
  x,
  v = c("slope", "aspect", "hillshade", "TPI", "roughness", "TRI"),
  unit = c("degrees", "radians"),
  azimuth = 315,
  altitude = 45,
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` elevation raster.
`v`	Derivatives to compute, any of `"slope"`, `"aspect"`, `"hillshade"`, `"TPI"` (topographic position index), `"roughness"`, `"TRI"` (terrain ruggedness index). The return follows the input: one matrix for a single `v`, a named list for several.
`unit`	Angular unit for `slope` and `aspect`: `"degrees"` (default) or `"radians"`.

azimuth, altitude
: Sun position for "hillshade", in degrees. Defaults 315 (NW) and 45.

band
: Band to read (1-based). Default 1.

path
: Optional output .vec path (one band per v, named after v). When given the result is streamed to disk and the opened vec_open_raster() handle is returned invisibly; when NULL the result is returned in memory.

dtype
: Storage dtype for .vec output (see vec_write_raster()). Default "f32".

compression
: Compression effort for .vec output. Default "fast".

Details

Slope and aspect use the Horn (1981) finite-difference gradient over the 3x3 neighbourhood; aspect is degrees clockwise from north (flat cells return 90). hillshade is the cosine of the incidence angle for the given sun position, clamped at 0. TPI is the cell minus the mean of its eight neighbours; roughness is the range over the 3x3; TRI is the mean absolute difference to the eight neighbours. Cells whose 3x3 neighbourhood touches a nodata value or the raster edge return NA.

Value

When path is NULL: a numeric matrix for a single v, or a named list of matrices for several, each carrying gt, extent, and crs attributes (row 1 northmost). When path is given, the written multi-band vectra_raster handle (invisibly).

See Also

focal() for arbitrary moving windows.

Examples

```
# A tilted surface so slope and aspect are well defined.
z <- outer(1:8, 1:8, function(r, c) 10 + 2 * c + r)
f <- tempfile(fileext = ".vec")
vec_write_raster(z, f, dtype = "f64", extent = c(0, 0, 8, 8))

slp <- terrain(f, v = "slope")
deriv <- terrain(f, v = c("slope", "aspect", "hillshade"))
names(deriv)
unlink(f)
```

---

tiff_band_names *Read per-band names from a GeoTIFF*

Description

Returns the band names embedded in the file's GDAL_METADATA XML (TIFF tag 42112). GDAL writes per-band names as <Item name="DESCRIPTION" sample="N" role="description">...</Item> entries, where sample is the 0-based band index. Bands without a name in the XML are reported as NA. Files with no GDAL_METADATA tag at all return a length-nbands vector of NA_character_.

Usage

```
tiff_band_names(path)
```

Arguments

path	Path to a GeoTIFF file.

Details

This is a small, dependency-free scanner intended for the common case (`terra::names(r) <- ...` and similar). For arbitrary XML, parse the raw string from `tiff_metadata()` yourself.

Value

A character vector of length `nbands`. Element `i` is the name of band `i` (or `NA_character_` if the file does not name it).

Examples

```
f <- tempfile(fileext = ".tif")
df <- data.frame(x = rep(1:2, 2), y = rep(1:2, each = 2),
                 band1 = as.double(1:4), band2 = as.double(5:8))
xml <- paste0(
  "<GDALMetadata>",
  "<Item name=\"DESCRIPTION\" sample=\"0\" role=\"description\">temperature</Item>",
  "<Item name=\"DESCRIPTION\" sample=\"1\" role=\"description\">humidity</Item>",
  "</GDALMetadata>")
write_tiff(df, f, metadata = xml)
tiff_band_names(f)
unlink(f)
```

---

tiff_crs *Read CRS metadata from a GeoTIFF*

Description

Returns the spatial reference system embedded in a GeoTIFF, parsed from the GeoKey directory (TIFF tag 34735). The projected CRS EPSG (`PCSTypeGeoKey` 3072) is preferred over the geographic CRS EPSG (`GeographicTypeGeoKey` 2048). Citation strings are read from `GeoAsciiParams` (tag 34737) with priority PCS > GeoTIFF > geographic.

Usage

```
tiff_crs(path)
```

Arguments

path	Path to a GeoTIFF file.

Details

Files written without a GeoKey directory return NA for both fields.

Value

A list with elements epsg (integer or NA_integer_) and citation (character or NA_character_).

Examples

```
f <- tempfile(fileext = ".tif")
df <- data.frame(x = 1:4, y = rep(1:2, each = 2), band1 = as.double(1:4))
write_tiff(df, f)
tiff_crs(f)  # epsg = NA, citation = NA — vectra writer omits GeoKeys
unlink(f)
```

---

tiff_extract_points *Extract raster values at point coordinates*

Description

Samples band values from a GeoTIFF at specific (x, y) locations using the file's affine geotransform. Only the strips containing query points are read, making this efficient for sparse point sets on large rasters.

Usage

```
tiff_extract_points(path, x, y = NULL)
```

Arguments

path	Path to a GeoTIFF file.
x	Numeric vector of x coordinates, or a data.frame / matrix with columns named x and y.
y	Numeric vector of y coordinates (ignored if x is a data.frame).

Details

Points that fall outside the raster extent return NA for all bands. Pixel assignment uses nearest-pixel rounding (i.e., the point is assigned to the pixel whose center is closest).

Value

A data.frame with columns `x`, `y`, `band1`, `band2`, etc. One row per input point, in the same order as the input.

Examples

```
f <- tempfile(fileext = ".tif")
df <- data.frame(x = as.double(rep(1:4, 3)),
                 y = as.double(rep(1:3, each = 4)),
                 band1 = as.double(1:12))
write_tiff(df, f)

# Sample at specific locations via data.frame
pts <- data.frame(x = c(2, 3), y = c(1, 2))
tiff_extract_points(f, pts)

# Or pass x and y separately
tiff_extract_points(f, x = c(2, 3), y = c(1, 2))
unlink(f)
```

---

tiff_metadata *Read GDAL_METADATA from a GeoTIFF*

Description

Returns the GDAL_METADATA XML string (TIFF tag 42112) embedded in a GeoTIFF file. Returns `NA` if the tag is not present.

Usage

```
tiff_metadata(path)
```

Arguments

`path`	Path to a GeoTIFF file.

Value

A single character string containing the XML, or `NA_character_`.

Examples

```
f <- tempfile(fileext = ".tif")
df <- data.frame(x = 1:4, y = rep(1:2, each = 2), band1 = as.double(1:4))
write_tiff(df, f, metadata = "<GDALMetadata></GDALMetadata>")
tiff_metadata(f)
unlink(f)
```

transmute *Keep only columns from mutate expressions*

Description

Like `mutate()` but drops all other columns.

Usage

```
transmute(.data, ...)
```

Arguments

`.data`	A `vectra_node` object.
`...`	Named expressions.

Value

A new `vectra_node` with only the computed columns.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> transmute(kpl = mpg * 0.425) |> collect() |> head()
unlink(f)
```

ungroup *Remove grouping from a vectra query*

Description

Remove grouping from a vectra query

Usage

```
ungroup(x, ...)
```

Arguments

`x`	A `vectra_node` object.
`...`	Ignored.

Value

An ungrouped vectra_node.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)
tbl(f) |> group_by(cyl) |> ungroup()
unlink(f)
```

---

vec_build_overviews *Build overview pyramids for a .vec raster*

Description

Appends n_levels - 1 reduced-resolution copies of the raster to the file. Each level is computed by 2x downsampling the previous level with the chosen kernel. Reading via vec_read_window(level = L) picks tiles at level L; the file's n_levels is updated in place.

Usage

```
vec_build_overviews(
  path,
  levels,
  resampling = c("average", "nearest", "bilinear", "mode", "gauss"),
  compression = c("fast", "balanced", "max")
)
```

Arguments

path	Path to a .vec raster file. The file is modified in place.
levels	Total levels including level 0 (so levels = 5 adds four overviews: levels 1..4). Must be in [2, 16].
resampling	One of "nearest", "average", "bilinear", "mode", "gauss". "average" is the right choice for continuous rasters; "mode" for categorical/land-cover.
compression	Compression effort for the new tiles. Defaults to "fast" because overview tiles are usually one-shot writes.

Value

Invisible NULL.

vec_close_raster *Close a .vec raster handle*

Description

Idempotent. The handle is also auto-released by R's garbage collector.

Usage

```
vec_close_raster(r)
```

Arguments

r	A vectra_raster returned by vec_open_raster().

Value

Invisible NULL.

vec_extract_points *Extract band values at (x, y) points from a .vec raster*

Description

Extract band values at (x, y) points from a .vec raster

Usage

```
vec_extract_points(r, x, y)
```

Arguments

r	A vectra_raster from vec_open_raster().
x	Numeric vector of x coordinates in CRS units.
y	Numeric vector of y coordinates, same length as x.

Value

A data.frame with columns x, y, then one column per band (named after r$band_names if recorded, otherwise band1, band2, ...). NA marks pixels outside the raster or matching nodata.

vec_open_raster *Open a .vec raster*

Description

Lazy open: parses the header and tile index but does not decode any tiles. Returns a list with metadata and an external pointer handle. The pointer is auto-finalized when garbage collected; call `vec_close_raster()` to release earlier.

Usage

```
vec_open_raster(path)
```

Arguments

`path`	Path to a `.vec` raster file.

Value

A `vectra_raster` list with elements: `ptr`, `width`, `height`, `n_bands`, `tile_size`, `dtype`, `gt`, `epsg`, `nodata`, `band_names`.

vec_raster_layout *Tile layout of an open .vec raster*

Description

Returns `"image"` (default Phase 6 layout — one tile per `(band, time, ty, tx)`) or `"pixel"` (Phase 6b transpose layout — one tile per `(band, ty, tx)` holding the full time stack).

Usage

```
vec_raster_layout(r)
```

Arguments

`r`	A `vectra_raster`.

Value

Character(1) `"image"` or `"pixel"`.

vec_raster_times *Distinct time stamps stored in a .vec time cube*

Description

Returns the ascending vector of time stamps recorded for the given (band, level). Pixel-major files store one consolidated table; image- major files derive the list from the per-tile time field.

Usage

```
vec_raster_times(r, band = 1L, level = 0L)
```

Arguments

r	A vectra_raster.
band	1-based band index.
level	Overview level.

Value

Numeric vector of stamps (length 0 when the file has no time information).

vec_read_pixel_series *Read the full time series at a single pixel from a .vec time cube*

Description

Returns a numeric vector of length n_time — one value per time step recorded in the file, in ascending time-stamp order.

Usage

```
vec_read_pixel_series(
  r,
  x = NULL,
  y = NULL,
  col = NULL,
  row = NULL,
  band = 1L,
  level = 0L
)
```

Arguments

r	A vectra_raster from vec_open_raster().
x, y	Pixel coordinates. Either both x and y (CRS units; the geotransform is used to map to col/row) or both col and row (1-based pixel indices).
col, row	1-based pixel coordinates (alternative to x/y).
band	Band index (1-based).
level	Overview level. Default 0.

Details

For pixel-major files (written with vec_write_time_cube(layout = "pixel")) this is the optimal access pattern: a single tile decode yields all time values for the pixel. For image-major files the reader scans the index for distinct time stamps, decodes one spatial tile per stamp, and extracts the pixel from each — correct but n_time slower than the optimal layout.

Value

A numeric vector of length n_time. NA marks pixels outside the raster or matching nodata. The corresponding time stamps can be obtained from vec_raster_times(r, band, level).

---

vec_read_time_slice *Read a single time slice from a .vec time cube*

Description

Performs a linear scan of the index for tiles with time == time and decodes the matching window. The lookup is O(n_tiles) per call — Phase 6's optimized hash-map lookup is a follow-up.

Usage

```
vec_read_time_slice(r, time, band = 1L, level = 0L, cols = NULL, rows = NULL)
```

Arguments

r	A vectra_raster from vec_open_raster().
time	Time value to match (numeric/integer).
band	Band index (1-based).
level	Overview level. Default 0.
cols, rows	1-based ranges, same as vec_read_window.

Value

A numeric matrix.

vec_read_window *Read a window of pixels from a .vec raster*

Description

Decodes only the tiles overlapping the requested window. Pixels outside the raster extent come back as NA.

Usage

```
vec_read_window(r, band = 1L, level = 0L, cols = NULL, rows = NULL)
```

Arguments

r	A vectra_raster from vec_open_raster().
band	Band index (1-based). Default 1.
level	Overview level — 0 = full resolution, 1 = half, 2 = quarter, etc. Must be < r$n_levels (which is 1 unless vec_build_overviews() has been run on the file).
cols	1-based column range c(col_min, col_max). Inclusive. Coordinates are in the chosen level's pixel grid (so at level 1 the raster is half as wide). Default c(1, level_width).
rows	1-based row range c(row_min, row_max). Inclusive. Default c(1, level_height).

Value

A numeric matrix with nrow = row_max - row_min + 1 and ncol = col_max - col_min + 1. Nodata pixels become NA.

vec_to_tiff *Export a .vec raster to GeoTIFF*

Description

Writes the level-0 pixels of a .vec raster to a GeoTIFF file. The TIFF inherits dtype, geotransform, EPSG, and nodata from the source. Strip layout; the writer supports "none", "deflate", and "lzw" compression. LZW also applies horizontal differencing (Predictor 2) for integer pixel types, which dramatically improves compression on smooth raster data and matches the layout most production GIS tools produce by default. Tiled and BigTIFF output land in a follow-up.

Usage

```
vec_to_tiff(r, path, compression = c("deflate", "lzw", "none"))
```

Arguments

r	Either a path to a .vec raster or a vectra_raster returned by vec_open_raster(). If a handle is passed it is left open.
path	Output .tif path.
compression	One of "deflate" (default), "lzw", or "none".

Value

Invisible NULL.

---

vec_write_raster *Write a raster matrix or 3D array to a .vec raster file*

Description

Writes a row-major raster (one band) or a band-major 3D array (multi-band) to the VECR raster format. Each tile is encoded as a self-describing tdc block (PRED_2D + BYTE_SHUFFLE + LZ).

Usage

```
vec_write_raster(
  x,
  path,
  dtype = "f32",
  tile_size = 512L,
  extent = NULL,
  gt = NULL,
  epsg = 0L,
  nodata = NA_real_,
  band_names = NULL,
  compression = c("fast", "balanced", "max")
)
```

Arguments

x	A numeric matrix c(rows, cols) for a single band, or a numeric 3D array c(rows, cols, bands) for multi-band.
path	Output file path.
dtype	Storage dtype, one of "f64", "f32", "i8", "u8", "i16", "u16", "i32", "u32", "i64", "u64". Defaults to "f32" for floating-point input — "f64" doubles file size with no information gain for typical climate rasters.
tile_size	Square tile edge in pixels. Default 512.
extent	Numeric vector c(xmin, ymin, xmax, ymax). Used together with the raster dimensions to derive the geotransform. Either extent or gt must be supplied for georeferenced output.

gt	Numeric(6) GDAL-style geotransform. Overrides extent if both are given.
epsg	EPSG code (integer) or 0L for none.
nodata	Nodata value, or NA_real_ to skip recording one.
band_names	Optional character vector of length equal to the number of bands.
compression	Compression effort, one of "fast" (single spec, fast encode), "balanced" (probe two entropy coders, ~2x encode time), or "max" (probe six candidate specs per tile, slowest encode but smallest file). Decode cost is unchanged across levels because each tile records its own codec spec. Default "fast".

Value

Invisible NULL.

`vec_write_time_cube` *Write a 4D time-cube raster to .vec*

Description

Each (band, time) combination becomes a stack of tiles tagged with the chosen time stamp. Stamps are stored as int64 in the per-tile index entry; a value of 0 is reserved for "untimed" so this writer remaps any caller-supplied 0 to 1 internally.

Usage

```
vec_write_time_cube(
  x,
  times,
  path,
  dtype = "f32",
  tile_size = 512L,
  layout = c("image", "pixel"),
  extent = NULL,
  gt = NULL,
  epsg = 0L,
  nodata = NA_real_,
  band_names = NULL,
  compression = c("fast", "balanced", "max")
)
```

Arguments

x	Numeric 4D array c(rows, cols, bands, time).
times	Numeric/integer vector with length(times) == dim(x)[4], in the unit of your choice (epoch ms, year, step index).
path	Output .vec path.

dtype	Storage dtype (see vec_write_raster).
tile_size	Tile edge in pixels.
layout	Tile layout — one of "image" (default; one tile per (band, time, ty, tx), optimal for "give me one full image at time T" reads) or "pixel" (Phase 6b; one tile per (band, ty, tx) holding the full time stack as [tw*th, n_time], optimal for "give me the time series at pixel (x, y)" reads).
extent, gt, epsg, nodata, band_names, compression	Same semantics as vec_write_raster().

Value

Invisible NULL.

---

vtr_schema *Create a star schema over linked vectra tables*

Description

Registers a fact table with named dimension links. The schema enables lookup() to resolve columns from dimension tables without writing explicit joins.

Usage

```
vtr_schema(fact, ...)
```

Arguments

fact	A vectra_node object (the central fact table). Must be file-backed (created via tbl(), tbl_csv(), or tbl_sqlite()).
...	Named vectra_link objects created by link(). Names become the dimension aliases used in lookup() (e.g., species$name).

Value

A vectra_schema object.

Examples

```
f_obs <- tempfile(fileext = ".vtr")
f_sp  <- tempfile(fileext = ".vtr")
f_ct  <- tempfile(fileext = ".vtr")
write_vtr(data.frame(sp_id = 1:3, ct_code = c("AT", "DE", "FR"),
                     value = 10:12), f_obs)
write_vtr(data.frame(sp_id = 1:3,
                     name = c("Oak", "Beech", "Pine")), f_sp)
write_vtr(data.frame(ct_code = c("AT", "DE", "FR"),
                     gdp = c(400, 3800, 2700)), f_ct)
```

```
s <- vtr_schema(
  fact    = tbl(f_obs),
  species = link("sp_id", tbl(f_sp)),
  country = link("ct_code", tbl(f_ct))
)
print(s)
unlink(c(f_obs, f_sp, f_ct))
```

`warp` *Resample or reproject a streamed raster onto a target grid*

Description

Warps a `.vec` raster onto a target grid, walking the *output* one tile-row strip at a time. For each strip the target pixel-centre coordinates are built, projected into the source coordinate reference system when the two CRSs differ (delegated to PROJ via **sf**), mapped through the source geotransform to fractional source pixels, and sampled from the bounded source window those coordinates fall in. The output is assembled in memory or streamed straight back to a new `.vec`, so the whole output grid is never resident; the source is read in bounded windows rather than held whole.

Usage

```
warp(
  x,
  template,
  method = c("near", "bilinear", "cubic"),
  band = 1L,
  path = NULL,
  dtype = "f32",
  compression = c("fast", "balanced", "max")
)
```

Arguments

`x`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster to warp.
`template`	The target grid: a `vectra_raster` / `.vec` path whose grid and CRS are borrowed, or a list `list(crs =, extent =, res =, dims =)`. With `crs` and `res` but no `extent`, the target extent is the source's corners projected into `crs`.
`method`	Resampling method: `"near"`, `"bilinear"`, or `"cubic"`. Default `"near"`.
`band`	Band to warp (1-based). Default 1.
`path`	Optional output `.vec` path. When given the result is streamed to disk and the opened `vec_open_raster()` handle is returned invisibly; when `NULL` the result is returned as an in-memory matrix.
`dtype`	Storage dtype for `.vec` output (see `vec_write_raster()`). Default `"f32"`.
`compression`	Compression effort for `.vec` output. Default `"fast"`.

Details

This is the *sort / partition* tier of the spatial toolbox: each output strip reads the source window it projects onto. For a mild reprojection or a plain resample that window is a thin band; a strong reprojection can make it large, but the output stays streamed throughout.

Sampling follows the GDAL / **terra** convention (pixel centres at half-integer coordinates). `"near"` takes the nearest source cell; `"bilinear"` the 2x2 weighted mean; `"cubic"` the 4x4 cubic convolution (Catmull-Rom, a = -0.5). A target cell whose sampling kernel reaches outside the source extent, or touches a nodata cell, comes back `NA`.

Reprojection happens only when both rasters carry a known EPSG code and the codes differ; otherwise `warp()` resamples within a shared CRS and needs no **sf**.

Value

When `path` is `NULL`, a numeric matrix on the target grid (row 1 northmost) carrying `gt`, `extent`, and `crs` attributes. When `path` is given, the written `vectra_raster` handle (invisibly).

See Also

`rasterize()` to build a raster from streamed vector features, `focal()` for moving-window statistics.

Examples

```
z <- outer(1:8, 1:8, function(r, c) r + 2 * c)
f <- tempfile(fileext = ".vec")
vec_write_raster(z, f, dtype = "f64", extent = c(0, 0, 8, 8))

# Resample onto a finer grid over the same extent.
fine <- warp(f, list(extent = c(0, 0, 8, 8), res = 0.5), method = "bilinear")
dim(fine)
unlink(f)
```

---

write_csv *Write query results or a data.frame to a CSV file*

Description

For `vectra_node` inputs, data is streamed batch-by-batch to disk without materializing the full result in memory. For `data.frame` inputs, the data is written directly.

Usage

```
write_csv(x, path, ...)
```

Arguments

x	A vectra_node (lazy query) or a data.frame.
path	File path for the output CSV file.
...	Reserved for future use.

Value

Invisible NULL.

Examples

```
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars[1:5, ], f)
csv <- tempfile(fileext = ".csv")
tbl(f) |> write_csv(csv)
unlink(c(f, csv))
```

---

write_sqlite *Write query results or a data.frame to a SQLite table*

Description

For vectra_node inputs, data is streamed batch-by-batch to disk without materializing the full result in memory. For data.frame inputs, the data is written directly.

Usage

```
write_sqlite(x, path, table, ...)
```

Arguments

x	A vectra_node (lazy query) or a data.frame.
path	File path for the SQLite database.
table	Name of the table to create/write into.
...	Reserved for future use.

Value

Invisible NULL.

Examples

```
db <- tempfile(fileext = ".sqlite")
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars[1:5, ], f)
tbl(f) |> write_sqlite(db, "cars")
unlink(c(f, db))
```

write_tiff *Write query results to a GeoTIFF file*

Description

The data must contain x and y columns (pixel center coordinates) and one or more numeric band columns. Grid dimensions and geotransform are inferred from the x/y coordinate arrays. Missing pixels are written as NaN (or the type-appropriate nodata value for integer pixel types).

Usage

```
write_tiff(
  x,
  path,
  compress = FALSE,
  pixel_type = "float64",
  metadata = NULL,
  crs = NULL,
  tiled = FALSE,
  tile_size = 256L,
  bigtiff = "auto",
  ...
)
```

Arguments

x	A vectra_node (lazy query) or a data.frame.
path	File path for the output GeoTIFF file.
compress	Logical; use DEFLATE compression? Default FALSE.
pixel_type	Character string specifying the output pixel type. One of "float64" (default), "float32", "int16", "int32", "uint8", or "uint16".
metadata	Optional character string of GDAL_METADATA XML to embed in the file (tag 42112). Use tiff_metadata() to read it back.
crs	Optional CRS to embed as a GeoKey directory (TIFF tag 34735). Accepts an integer EPSG code, an "EPSG:xxxx" string, or a list with named fields epsg, geographic (TRUE/FALSE), and optionally citation. Codes that are not auto-classified as projected/geographic default to projected; pass geographic = TRUE to override. Use tiff_crs() to read it back.
tiled	Logical; write a tiled GeoTIFF (TIFF tags 322/323/324/325) instead of strips. Default FALSE. Tiled layout enables random-access block reads and is required for Cloud-Optimized GeoTIFF (COG).
tile_size	Integer; tile edge length in pixels. Must be a positive multiple of 16 (TIFF spec). Either a single value (square tiles) or a length-2 vector c(width, height). Default 256. Edge tiles at the right and bottom of the image are padded to full tile size with the NoData / NaN value.

bigtiff	Controls BigTIFF dispatch. "auto" (default) emits BigTIFF when the expected raw payload would exceed the classic-TIFF 4 GB ceiling, otherwise emits classic TIFF. TRUE forces BigTIFF (magic 0x002B, 64-bit offsets), useful for round-trip tests on small data. FALSE forces classic TIFF — beware that classic TIFF will silently corrupt outputs larger than 4 GB. Tiled BigTIFF is not yet supported.
...	Reserved for future use.

Value

Invisible NULL.

Examples

```
# Write as int16 with DEFLATE compression and an EPSG:4326 GeoKey
df <- data.frame(x = 1:4, y = rep(1:2, each = 2), band1 = c(100, 200, 300, 400))
f <- tempfile(fileext = ".tif")
write_tiff(df, f, compress = TRUE, pixel_type = "int16", crs = 4326L)
tiff_crs(f)
unlink(f)
```

---

write_vtr *Write data to a .vtr file*

Description

For vectra_node inputs (lazy queries from any format: CSV, SQLite, TIFF, or another .vtr), data is streamed batch-by-batch to disk without materializing the full result in memory. Each batch becomes one row group. The output file is written atomically (via temp file + rename) so readers never see a partial file.

Usage

```
write_vtr(
  x,
  path,
  compress = c("fast", "small", "none"),
  batch_size = NULL,
  col_types = NULL,
  quantize = NULL,
  spatial = NULL,
  ...
)
```

Arguments

x	A vectra_node (lazy query) or a data.frame.
path	File path for the output .vtr file.
compress	Compression level: "fast" (default, byte-shuffle + greedy LZ), "small" (per-block adaptive — tries greedy LZ, separated-streams LZ, and LZ + Huffman entropy coding, and writes whichever shrank the block the most; never worse than "fast" on any block, typically 10-25 percent smaller files at the cost of slower encode), or "none".
batch_size	Target number of rows per row group in the output file. Defaults to 131072 for data.frames (1 MB per double column, cache-friendly for decompression). For nodes, defaults to NULL (one row group per upstream batch).
col_types	Optional named character vector specifying narrow integer storage types. Names must match column names; values must be "int8", "int16", or "int32". Only applies to integer columns. Example: col_types = c(age = "int8", year = "int16").
quantize	Optional named list for lossy quantization of double columns. Each element is named after a column and is itself a named list with scale (or precision = 1/scale), type ("int8", "int16", "int32"; default "int16"), and optionally offset (default 0). Example: quantize = list(temp = list(precision = 0.001, type = "int16")).
spatial	Optional list for 2D spatial predictor encoding. Either a global spec applied to all numeric columns (list(nx = 2000, ny = 2000)) or per-column specs (list(temp = list(nx = 2000, ny = 2000))). When provided, a spatial predictor removes smooth 2D trends before compression, dramatically improving compression of raster data. Combines with quantize for maximum effect.
...	Additional arguments passed to methods.

Details

For data.frame inputs, the data is written directly from memory.

Value

Invisible NULL.

Examples

```
# From a data.frame
f <- tempfile(fileext = ".vtr")
write_vtr(mtcars, f)

# Streaming format conversion (CSV -> VTR)
csv <- tempfile(fileext = ".csv")
write.csv(mtcars, csv, row.names = FALSE)
f2 <- tempfile(fileext = ".vtr")
tbl_csv(csv) |> write_vtr(f2)
```

```
unlink(c(f, f2, csv))
```

`zonal` *Summarise raster values within zones*

Description

Reduces a raster to one summary row per zone, streaming the raster one tile-row strip at a time so the whole grid never has to be resident. Zones come either from a second raster aligned to the value grid (each pixel's zone is that raster's value, the `terra::zonal` pattern) or from an **sf** polygon layer (each pixel is assigned the polygon its centre falls in). The per-zone running moments (count, sum, sum of squares, min, max) are folded in memory as strips arrive, so peak memory is one strip plus the small per-zone table regardless of raster size. This is the *monoid fold* tier of the spatial toolbox: bounded memory, a single streaming pass, no spill.

Usage

```
zonal(
  raster,
  zones,
  fun = "mean",
  band = 1L,
  zone_band = 1L,
  zone_field = NULL,
  na.rm = TRUE
)
```

Arguments

`raster`	A `vectra_raster` (from `vec_open_raster()`) or a path to a `.vec` raster holding the values to summarise.
`zones`	The zones to summarise within: a `vectra_raster` / `.vec` path aligned to `raster` (zone id per pixel), or an `sf`/`sfc` polygon layer.
`fun`	One or more of `"mean"`, `"sum"`, `"count"`, `"min"`, `"max"`, `"sd"`. Each becomes a column in the result. Default `"mean"`.
`band`	Band of the value `raster` to summarise (1-based). Default 1.
`zone_band`	Band of a raster `zones` holding the zone ids. Default 1.
`zone_field`	For an `sf` `zones` layer, the column giving each polygon's zone id. Default `NULL` uses the polygon row index `1:n`.
`na.rm`	If `TRUE` (default) skip nodata pixels; if `FALSE` let a nodata pixel propagate `NA` to its zone's statistics.

Details

`sd` is derived from the streamed moments (`sqrt((sum2 - sum^2 / n) / (n - 1))`), so it needs no second pass. With `na.rm = TRUE` (the default) nodata pixels are skipped; with `na.rm = FALSE` any nodata pixel in a zone makes that zone's `sum`/`mean`/`min`/`max`/`sd` `NA`, matching the resident behaviour. `count` always reports the number of non-nodata cells in the zone.

Polygon zones assign each pixel centre to a polygon natively on the GEOS C API (the polygons parsed once into a spatial index, every strip's centres located in C), so **sf** is touched only to read the polygons in; geographic polygons with spherical geometry on (`sf::sf_use_s2()`) keep the **sf** point-in-polygon path. Raster zones are fully **sf**-free. The zone raster must share the value raster's dimensions and geotransform.

Value

A data.frame with a `zone` column (sorted) followed by one column per `fun`, one row per zone.

See Also

`rasterize()` to build a value raster from streamed points, `vec_open_raster()` to open the inputs.

Examples

```
# A value raster and an aligned 2x2-block zone raster on a 4x4 grid.
vals <- matrix(1:16, 4, 4, byrow = TRUE)
zone <- matrix(c(1, 1, 2, 2, 1, 1, 2, 2,
                 3, 3, 4, 4, 3, 3, 4, 4), 4, 4, byrow = TRUE)
fv <- tempfile(fileext = ".vec"); fz <- tempfile(fileext = ".vec")
vec_write_raster(vals, fv, dtype = "f64", extent = c(0, 0, 4, 4))
vec_write_raster(zone, fz, dtype = "f64", extent = c(0, 0, 4, 4))

zonal(fv, fz, fun = c("mean", "sum", "count"))
unlink(c(fv, fz))
```

Index