Package: vectra (via r-universe)

June 12, 2026

Title Columnar Query Engine for Larger-than-RAM Data

Version 0.7.1

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.

License MIT + file LICENSE

Depends R (>= 4.1.0)

SystemRequirements GNU make

Encoding UTF-8

Imports tidyselect, rlang

Suggests biglm, bit64, knitr, openxlsx2, rmarkdown, 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>

Config/pak/sysreqs make

Repository https://cran.r-universe.dev

Date/Publication 2026-06-12 00:00:02 UTC

RemoteUrl https://github.com/cran/vectra

RemoteRef HEAD

RemoteSha 8c680df70de5475494115c7fe0bf8fcd7cf7a4fa

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)
```

---

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)
```

`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).

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)
```

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))
```

`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)
```

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)
```

---

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`.

`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)
```

`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.

Value

A vectra_node for slice_head() and slice_min/max(..., with_ties = FALSE). A data.frame for slice_tail() and 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()
unlink(f)
```

---

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)
}
```

---

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))
```

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))
```

Index