Package 'vectra'

April 21, 2026

Title Columnar Query Engine for Larger-than-RAM Data

Version 0.5.1

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').

License MIT + file LICENSE

Depends R (>= 4.1.0)

SystemRequirements GNU make

Encoding UTF-8

Imports tidyselect, rlang

RoxygenNote 7.3.3

Suggests bit64, knitr, openxlsx2, rmarkdown, 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

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 2026-04-21 20:00:02 UTC

Contents

across . . . 3
append_vtr . . . 4
arrange . . . 5
bind_rows . . . 5
block_fuzzy_lookup . . . 6
block_lookup . . . 7
collect . . . 8
count . . . 9
create_index . . . 10
cross_join . . . 11
delete_vtr . . . 11
desc . . . 12
diff_vtr . . . 13
distinct . . . 14
explain . . . 15
filter . . . 15
fuzzy_join . . . 16
glimpse . . . 17
group_by . . . 18
has_index . . . 18
head.vectra_node . . . 19
left_join . . . 19
link . . . 21
lookup . . . 21
materialize . . . 23
mutate . . . 23
print.vectra_node . . . 24
pull . . . 25
reframe . . . 25
relocate . . . 26
rename . . . 27
select . . . 27
slice . . . 28
slice_head . . . 28
summarise . . . 29
tbl . . . 30
tbl_csv . . . 31
tbl_sqlite . . . 32
tbl_tiff . . . 32
tbl_xlsx . . . 33
tiff_extract_points . . . 34
tiff_metadata . . . 35
transmute . . . 36
ungroup . . . 36
vtr_schema . . . 37
write_csv . . . 38

write_sqlite 38
write_tiff 39
write_vtr 40

Index **42**

`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)
```

---

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)
```

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)
```

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)
```

---

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_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)
```

`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,
  ...
)
```

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.
...	Reserved for future use.

Value

Invisible NULL.

Examples

```
# Write as int16 with DEFLATE compression
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")
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

across, 3
anti_join *(*left_join*)*, 19
append_vtr, 4
arrange, 5
arrange(), *12*

bind_cols *(*bind_rows*)*, 5
bind_rows, 5
block_fuzzy_lookup, 6
block_lookup, 7
block_lookup(), *23*

collect, 8
collect(), *13*, *30–32*
count, 9
create_index, 10
cross_join, 11

delete_vtr, 11
desc, 12
desc(), *5*
diff_vtr, 13
distinct, 14

explain, 15

filter, 15
full_join *(*left_join*)*, 19
fuzzy_join, 16

glimpse, 17
group_by, 18
group_by(), *30*

has_index, 18
head.vectra_node, 19

inner_join *(*left_join*)*, 19

left_join, 19
link, 21
link(), *37*
lookup, 21
lookup(), *37*

materialize, 23
materialize(), *7*
mutate, 23
mutate(), *3*, *36*

openxlsx2::read_xlsx(), *33*

print.vectra_node, 24
pull, 25

reframe, 25
relocate, 26
rename, 27
right_join *(*left_join*)*, 19

select, 27
semi_join *(*left_join*)*, 19
slice, 28
slice_head, 28
slice_max *(*slice_head*)*, 28
slice_min *(*slice_head*)*, 28
slice_tail *(*slice_head*)*, 28
summarise, 29
summarise(), *3*, *25*
summarize *(*summarise*)*, 29

tally *(*count*)*, 9
tbl, 30
tbl(), *10*, *11*, *13*, *21*, *37*
tbl_csv, 31
tbl_csv(), *21*, *37*
tbl_sqlite, 32
tbl_sqlite(), *21*, *37*
tbl_tiff, 32
tbl_xlsx, 33
tiff_extract_points, 34
tiff_metadata, 35

tiff_metadata(), *40*
transmute, 36

ungroup, 36

vtr_schema, 37
vtr_schema(), *21*

write_csv, 38
write_sqlite, 38
write_tiff, 39
write_vtr, 40