Package 'dplyneage'

September 16, 2026

Type Package

Title Column Lineage Visualization for 'dplyr' Pipelines

Version 0.3.1

Description Implements column lineage visualizations using 'React Flow' for 'dplyr' and 'dbplyr' pipelines. Provides a tidyverse-style interface for tracking data transformations through pipeline operations.

License MIT + file LICENSE

URL https://tgerke.github.io/dplyneage/, https://github.com/tgerke/dplyneage

BugReports https://github.com/tgerke/dplyneage/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports htmlwidgets, jsonlite

Suggests DBI, dbplyr, dplyr, duckdb, igraph, knitr, reticulate, rlang, rmarkdown, RSQLite, shiny, testthat (>= 3.2.0), withr, xml2

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/Needs/website tgerke/ducklake-r

NeedsCompilation no

Author Travis Gerke [aut, cre, cph], Meta Platforms, Inc. and affiliates [cph] (React and scheduler libraries bundled in inst/htmlwidgets/lib/reactflow), xyflow GmbH [cph] (React Flow library bundled in inst/htmlwidgets/lib/reactflow)

Maintainer Travis Gerke <travisgerke@gmail.com>

Repository CRAN

Date/Publication 2026-09-16 12:10:02 UTC

Contents

create_column_edge . . . 2
create_table_node . . . 3
extract_lineage . . . 4
has_bundle . . . 7
has_sqlglot . . . 7
lineage_diff . . . 8
lineage_edges . . . 9
lineage_example . . . 10
lineage_flow . . . 10
lineage_flow-shiny . . . 11
lineage_graphml . . . 12
lineage_json . . . 13
lineage_mermaid . . . 15
lineage_openlineage . . . 16
lineage_tables . . . 17
lineage_upstream . . . 18

Index **20**

`create_column_edge` *Connect two columns in a lineage diagram*

Description

Creates an edge from one table's column to another's, for diagrams built with `create_table_node()` and rendered by `lineage_flow()`. Table and column names must match the `table_name` and `columns` used when creating the nodes.

Usage

```
create_column_edge(
  from_table,
  from_column,
  to_table,
  to_column,
  label = NULL,
  animated = FALSE
)
```

Arguments

`from_table`, `from_column`
: Table and column the data comes from.

`to_table`, `to_column`
: Table and column the data flows into.

label	Optional label drawn on the edge, typically the transformation applied (e.g. "SUM()").
animated	If TRUE, the edge is drawn with a moving dash pattern. Useful for drawing attention to aggregations.

Value

An edge list ready to pass to lineage_flow()

See Also

Other manual lineage builders: create_table_node(), lineage_example()

Examples

```
# A direct column mapping
create_column_edge("customers", "id", "customer_summary", "customer_id")

# An aggregation, labeled and animated
create_column_edge("orders", "amount", "customer_summary", "total_spent",
  label = "SUM()", animated = TRUE
)
```

---

create_table_node *Create a table node for a lineage diagram*

Description

Builds one table, with its columns, for lineage diagrams rendered by lineage_flow(). Use this together with create_column_edge() when you want full control over the diagram instead of extracting lineage automatically.

Usage

```
create_table_node(table_name, columns, x = 0, y = 0, table_type = "source")
```

Arguments

table_name	Name shown in the node header. Also used as the node id, so it must be unique within a diagram.
columns	Character vector of column names listed in the node. Each column gets connection handles that edges can attach to.
x, y	Position of the node on the canvas, in pixels. Nodes remain draggable, so these only set the starting layout.
table_type	One of "source" (blue), "transform" (orange), or "target" (green). The colors follow the conventions used by tools like dbt and SQLMesh.

Value

A node list ready to pass to `lineage_flow()`

See Also

`extract_lineage()` to build nodes and edges automatically

Other manual lineage builders: `create_column_edge()`, `lineage_example()`

Examples

```
create_table_node(
  table_name = "customers",
  columns = c("id", "name", "email"),
  table_type = "source"
)
```

`extract_lineage` *Extract column lineage from a dplyr pipeline or SQL query*

Description

`extract_lineage()` traces every output column of a query back to the source table columns it was computed from. Pipe a dbplyr lazy table straight into it, or pass a SQL string. Aliases, CTEs, subqueries, set operations like `UNION`, and multi-source expressions such as `COALESCE(a.x, b.x)` all resolve to their true source columns.

Usage

```
extract_lineage(
  sql,
  dialect = "duckdb",
  schema = NULL,
  show_sql = FALSE,
  engine = c("auto", "sqlglot", "r"),
  include_indirect = FALSE
)
```

Arguments

`sql`	A dbplyr lazy table (`tbl_lazy`), a single SQL query string, or a named list of these (one element per pipeline model; see Details). Lazy tables are analyzed directly from their lazy query tree (the SQL recorded in `metadata` still comes from `dbplyr::sql_render()`); when one is handled by the sqlglot engine instead, its database connection is used to harvest table schemas automatically. Plain data frames are not accepted — dplyr executes each verb on them immediately, leaving no query tree to read. Wrap the data with `dbplyr::memdb_frame()` (or copy an existing frame with `copy_to(dbplyr::memdb(), df, name = "df")`) and the same pipeline becomes traceable; see `vignette("getting-started")`.

`dialect` SQL dialect the query is written in, e.g. `"duckdb"` (the default), `"postgres"`, `"mysql"`, `"snowflake"`, `"bigquery"`. Any dialect sqlglot understands works here.

`schema` Optional table schema used by the sqlglot engine to attribute unqualified columns to the right table and to expand `SELECT *`: a named list mapping table names to character vectors of column names, e.g. `list(orders = c("order_id", "amount"))`. Only relevant for SQL strings — the R engine reads exact provenance from the lazy query tree, and a lazy table that falls back to sqlglot harvests its schema from the database connection automatically.

`show_sql` If `TRUE`, print the SQL being analyzed. Useful for seeing what dbplyr generated from your pipeline. Default: `FALSE`.

`engine` Which lineage engine to use. `"auto"` (the default) uses the pure-R engine for lazy tables when dbplyr (>= 2.5.0) is installed, falling back to sqlglot for SQL strings or unsupported constructs. `"r"` forces the pure-R engine and errors on anything it cannot trace. `"sqlglot"` always renders to SQL and analyzes with sqlglot.

`include_indirect` If `TRUE`, columns used in `filter()`/`WHERE`, join conditions, `group_by()`, and `arrange()`/`ORDER BY` also appear in the diagram, connected by dashed edges (see Details). Default: `FALSE`, matching most lineage tools.

Details

Two engines are available. dbplyr lazy tables are analyzed by a pure-R fast path that walks the pipeline's lazy query tree directly — no Python required. SQL strings are analyzed by sqlglot's lineage engine via reticulate (a Suggests dependency: install reticulate to enable this engine; sqlglot itself is provisioned automatically). If a pipeline uses a construct the R engine cannot trace (e.g. raw SQL injected with `dbplyr::sql()`), it falls back to sqlglot automatically.

Both engines trace select-list lineage by default: columns used only in `filter()`, join conditions, or `arrange()` do not create lineage edges. Set `include_indirect = TRUE` to add them as dashed edges — a column that only filters the result still breaks the pipeline if it is dropped, so impact analysis usually wants them. Indirect edges connect each filter/join/group/sort column to every output column, since these conditions shape the whole result, and are classified by how the column is used (`"filter"`, `"join"`, `"group_by"`, `"sort"`).

A named list stitches a multi-model pipeline into one graph. Each element (lazy table or SQL string) is analyzed on its own, and any source table whose name matches another element's name connects to that model's node — so a bronze/silver/gold flow where each layer is materialized under its model's name renders as a single multi-hop DAG, with intermediate models drawn as orange transform nodes and terminal models as green targets.

Value

A list with `nodes` and `edges` ready to pass to `lineage_flow()`, plus `metadata` recording the analyzed SQL, the dialect, the engine used, and node/edge counts.

See Also

lineage_flow() to render the result; vignette("getting-started") for a tour from simple pipelines to CTEs and multi-source columns.

Examples

```
# Raw SQL: qualified columns resolve on their own
extract_lineage("SELECT c.id, c.name FROM customers c") |>
  lineage_flow()

# Supply a schema so unqualified columns attribute to the right table
# and SELECT * expands
extract_lineage(
  "SELECT c.name, order_date FROM customers c
   JOIN orders o ON c.id = o.customer_id",
  schema = list(
    customers = c("id", "name"),
    orders = c("customer_id", "order_date")
  )
)


# dbplyr pipelines: pipe straight in; the pure-R engine reads exact
# provenance from the pipeline itself, no Python needed
library(dplyr)

con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbWriteTable(con, "customers", data.frame(id = 1, name = "a"))
DBI::dbWriteTable(con, "orders", data.frame(customer_id = 1, amount = 10))

tbl(con, "customers") |>
  left_join(tbl(con, "orders"), by = c("id" = "customer_id")) |>
  group_by(id, name) |>
  summarise(total_spent = sum(amount, na.rm = TRUE), .groups = "drop") |>
  extract_lineage() |>
  lineage_flow()

# Multi-model pipelines: name each step and pass a named list; source
# tables matching a model name stitch the layers into one DAG
silver <- tbl(con, "orders") |>
  group_by(customer_id) |>
  summarise(total_spent = sum(amount, na.rm = TRUE), .groups = "drop")
invisible(compute(silver, name = "silver", temporary = TRUE))
gold <- tbl(con, "silver") |>
  mutate(big_spender = total_spent > 100)

extract_lineage(list(silver = silver, gold = gold)) |>
  lineage_flow()

DBI::dbDisconnect(con)
```

has_bundle *Is the React Flow bundle available?*

Description

The JavaScript bundle that powers `lineage_flow()` ships pre-built with the package, so this normally returns `TRUE`. If it returns `FALSE`, diagrams fall back to a static SVG rendering; see `vignette("building-reactflow")` for how to rebuild the bundle from source.

Usage

```
has_bundle()
```

Value

`TRUE` if the pre-built React Flow bundle is present

Examples

```
has_bundle()
```

has_sqlglot *Is the Python sqlglot dependency available?*

Description

Python is only involved when `extract_lineage()` analyzes raw SQL strings (or falls back to sqlglot for a pipeline it cannot trace in R); dbplyr pipelines are analyzed by a pure-R engine. The sqlglot engine needs the reticulate package (a Suggests dependency — install it with `install.packages("reticulate")`); dplyneage then declares its sqlglot dependency via `reticulate::py_require()`, so sqlglot itself is provisioned automatically the first time it is needed. Use this to check availability, or to gate code that extracts lineage from raw SQL (examples, vignette chunks, Shiny apps). Returns `FALSE` when reticulate is not installed. Note that calling it may initialize Python.

Usage

```
has_sqlglot()
```

Value

`TRUE` if sqlglot can be loaded, `FALSE` otherwise

See Also

`vignette("python-integration")` for using your own Python environment

Examples

```
has_sqlglot()
```

`lineage_diff` *Compare two lineage extractions*

Description

Reports the column-level edges and table columns that were added or removed between two lineage objects — typically the same pipeline before and after an edit. This makes the CI story concrete: extract lineage on both branches and fail (or comment) when provenance changed.

Usage

```
lineage_diff(old, new)
```

Arguments

`old`, `new`	Lineage objects from `extract_lineage()` (or lists with `nodes` and `edges`), in before/after order.

Value

A `dplyneage_lineage_diff` list with data frame elements `added_edges`, `removed_edges`, `added_columns`, and `removed_columns`. Its print method summarises the changes; zero-row elements mean no change.

See Also

Other lineage accessors: `lineage_edges()`, `lineage_tables()`, `lineage_upstream()`

Examples

```
old <- list(
  nodes = list(
    create_table_node("orders", "amount"),
    create_table_node("out", "total", table_type = "target")
  ),
  edges = list(create_column_edge("orders", "amount", "out", "total"))
)
new <- list(
  nodes = list(
    create_table_node("orders", c("amount", "tax")),
    create_table_node("out", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "out", "total"),
    create_column_edge("orders", "tax", "out", "total")
```

```
  )
)
lineage_diff(old, new)
```

`lineage_edges` *Lineage edges as a data frame*

Description

Flattens a lineage object's column-level edges into one row per edge, for filtering, joining, and summarising with ordinary data frame tools. For edges produced by `extract_lineage()`, the `transformation` column classifies each edge (`"identity"` for plain column passthrough, `"aggregation"`, or `"transformation"`) and `expression` records the output column's defining expression; both are `NA` for hand-built edges. With `include_indirect = TRUE`, indirect edges are classified by how the source column is used — `"filter"`, `"join"`, `"group_by"`, or `"sort"` — with `NA` for `expression`.

Usage

```
lineage_edges(lineage)
```

Arguments

`lineage`	The result of `extract_lineage()`, or any list with `nodes` and `edges` built with `create_table_node()` and `create_column_edge()`.

Value

A data frame with columns `source_table`, `source_column`, `target_table`, `target_column`, `transformation`, and `expression`.

See Also

Other lineage accessors: `lineage_diff()`, `lineage_tables()`, `lineage_upstream()`

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
lineage_edges(lineage)
```

lineage_example *A built-in example lineage diagram*

Description

Renders a small customers/orders lineage diagram built with the manual helpers. Handy for checking that the visualization works in your environment, and as a template for building diagrams by hand.

Usage

```
lineage_example()
```

Value

A `lineage_flow()` htmlwidget

See Also

Other manual lineage builders: `create_column_edge()`, `create_table_node()`

Examples

```
lineage_example()
```

lineage_flow *Render an interactive column lineage diagram*

Description

Draws a lineage graph with React Flow: tables as draggable nodes, column-to-column edges, and zoom/pan controls. Pass the result of `extract_lineage()` directly (it is detected automatically, so piping works), or build `nodes` and `edges` yourself with `create_table_node()` and `create_column_edge()`.

Usage

```
lineage_flow(
  nodes = list(),
  edges = list(),
  width = NULL,
  height = NULL,
  elementId = NULL
)
```

Arguments

nodes	The output of extract_lineage(), or a list of nodes created with create_table_node().
edges	A list of edges created with create_column_edge(). Ignored when nodes is an extract_lineage() result, which carries its own edges.
width, height	CSS dimensions of the widget, e.g. "100%" or "600px". Default to full width and 600px tall.
elementId	Explicit HTML element id for the widget. Usually left NULL so one is generated.

Value

An htmlwidget that prints in the RStudio viewer, R Markdown / Quarto documents, and Shiny apps.

See Also

extract_lineage() to compute lineage automatically; lineage_flowOutput() and renderLineageFlow() for Shiny.

Examples

```
# Build a small diagram by hand
nodes <- list(
  create_table_node("orders", c("order_id", "amount"), x = 0, y = 0),
  create_table_node("daily_totals", c("total"),
    x = 400, y = 0, table_type = "target"
  )
)
edges <- list(
  create_column_edge("orders", "amount", "daily_totals", "total",
    label = "SUM()", animated = TRUE
  )
)
lineage_flow(nodes, edges)

# Or pipe from extract_lineage()
extract_lineage("SELECT id, name FROM customers") |>
  lineage_flow()
```

---

lineage_flow-shiny *Shiny bindings for lineage_flow*

Description

Output and render functions for using lineage_flow within Shiny applications and interactive Rmd documents.

Usage

```
lineage_flowOutput(outputId, width = "100%", height = "400px")

renderLineageFlow(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

outputId	output variable to read from
width, height	Must be a valid CSS unit (like '100%', '400px', 'auto') or a number, which will be coerced to a string and have 'px' appended.
expr	An expression that generates a lineage_flow
env	The environment in which to evaluate expr.
quoted	Is expr a quoted expression (with quote())? This is useful if you want to save an expression in a variable.

Value

lineage_flowOutput() returns a shiny.tag.list holding the HTML element and dependencies that place the lineage widget in a Shiny UI. renderLineageFlow() returns a shiny.render.function to assign to an output slot; Shiny calls it to render the widget expr produces.

Examples

```
if (interactive() && requireNamespace("shiny", quietly = TRUE)) {
  library(shiny)

  ui <- fluidPage(
    lineage_flowOutput("lineage", height = "600px")
  )
  server <- function(input, output, session) {
    output$lineage <- renderLineageFlow(lineage_example())
  }
  shinyApp(ui, server)
}
```

---

lineage_graphml *Export lineage as GraphML*

Description

Serializes a lineage object to GraphML, the XML graph format read by igraph, Gephi, yEd, and most other graph tools. Each table column becomes a node (id "table.column", duplicated in a name attribute so igraph picks it up as the vertex name) with table, column, and node_type attributes; each column-level edge becomes a directed edge. That granularity is what makes the export useful downstream: igraph::subcomponent(g, "output.total", mode = "in") lists every source column feeding an output, and Gephi can color the graph by table.

Usage

```
lineage_graphml(lineage, path = NULL)
```

Arguments

lineage	The result of extract_lineage(), or any list with nodes and edges built with create_table_node() and create_column_edge().
path	Optional file to write the GraphML to. When supplied, the string is returned invisibly.

Value

A string containing the GraphML document.

See Also

extract_lineage() to compute lineage automatically

Other lineage exporters: lineage_json(), lineage_mermaid(), lineage_openlineage()

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
cat(lineage_graphml(lineage))

# Round-trip through igraph for ancestry queries

path <- tempfile(fileext = ".graphml")
lineage_graphml(lineage, path = path)
g <- igraph::read_graph(path, format = "graphml")
igraph::subcomponent(g, "daily_totals.total", mode = "in")
```

---

lineage_json *Export lineage as JSON*

Description

Serializes a lineage object to a small, stable JSON document: node ids with their columns and table type, plus one record per column-level edge. React Flow presentation details (positions, colors) are deliberately dropped, so the output is suitable for scripting with jq, committing to version control (a CI diff catches accidental provenance changes when a pipeline is edited), or feeding to a data catalog.

Usage

```
lineage_json(lineage, path = NULL, pretty = TRUE)
```

Arguments

lineage	The result of extract_lineage(), or any list with nodes and edges built with create_table_node() and create_column_edge().
path	Optional file to write the JSON to. When supplied, the string is returned invisibly.
pretty	If TRUE (the default), indent the output for readability. Use FALSE for a single-line document.

Value

A JSON string. With metadata (present on extract_lineage() results), nodes (objects with id, type, and columns), and edges (objects with source, source_column, target, and target_column; edges produced by extract_lineage() also carry transformation and expression).

See Also

extract_lineage() to compute lineage automatically

Other lineage exporters: lineage_graphml(), lineage_mermaid(), lineage_openlineage()

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
lineage_json(lineage)

# Write to a file instead
path <- tempfile(fileext = ".json")
lineage_json(lineage, path = path)

extract_lineage("SELECT customer_id, SUM(amount) AS total
                 FROM orders GROUP BY customer_id") |>
  lineage_json()
```

`lineage_mermaid` *Export lineage as a Mermaid flowchart*

Description

Serializes a lineage object to Mermaid flowchart text. Mermaid renders natively in GitHub markdown, Quarto, and most documentation tools, so this is the exporter to reach for when lineage should live *in the docs*: paste the output into a ```` ```mermaid ```` code fence and the diagram renders with no R, no htmlwidget, and no JavaScript bundle.

Usage

```
lineage_mermaid(lineage, path = NULL)
```

Arguments

`lineage`	The result of `extract_lineage()`, or any list with `nodes` and `edges` built with `create_table_node()` and `create_column_edge()`.
`path`	Optional file to write the Mermaid text to. When supplied, the string is returned invisibly.

Details

Each table becomes a subgraph containing its columns, colored by table type with the same palette as `lineage_flow()`. Non-identity edges are labeled with the column's defining expression, and indirect edges (from `extract_lineage(include_indirect = TRUE)`) draw dashed.

Value

A string containing the Mermaid flowchart definition.

See Also

`extract_lineage()` to compute lineage automatically

Other lineage exporters: `lineage_graphml()`, `lineage_json()`, `lineage_openlineage()`

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
cat(lineage_mermaid(lineage))
```

```
# Ready to paste into a GitHub README or Quarto document:
cat("```mermaid\n", lineage_mermaid(lineage), "```\n", sep = "")
```

`lineage_openlineage` *Export lineage as an OpenLineage run event*

Description

Serializes a lineage object to an OpenLineage `RunEvent` JSON document with a `ColumnLineage` facet on each output dataset — the interchange format that data catalogs and lineage backends (Marquez, DataHub, OpenMetadata, ...) ingest. POST the document to an OpenLineage endpoint and dplyneage-extracted lineage appears alongside lineage from dbt, Airflow, or Spark.

Usage

```
lineage_openlineage(
  lineage,
  path = NULL,
  namespace = "dplyneage",
  job_name = "extract_lineage",
  run_id = NULL,
  event_time = NULL,
  pretty = TRUE
)
```

Arguments

`lineage`	The result of `extract_lineage()`, or any list with `nodes` and `edges` built with `create_table_node()` and `create_column_edge()`.
`path`	Optional file to write the JSON to. When supplied, the string is returned invisibly.
`namespace`	Dataset and job namespace recorded in the event. OpenLineage uses namespaces to group datasets by system; the default `"dplyneage"` is fine for standalone use, but match your catalog's namespace when integrating.
`job_name`	Name recorded for the job that produced this lineage.
`run_id`	UUID identifying the run. Generated when `NULL` (the default); pass a fixed UUID for reproducible output.
`event_time`	Event timestamp in ISO-8601 format. The current UTC time when `NULL` (the default); pass a fixed timestamp for reproducible output.
`pretty`	If `TRUE` (the default), indent the output for readability. Use `FALSE` for a single-line document.

Details

Source tables become the event's `inputs` (with a schema facet listing their referenced columns); transform and target tables become `outputs`, each carrying a `columnLineage` facet that maps every output column to its input fields. Edge classifications translate to OpenLineage transformation types: `identity`/`transformation`/ `aggregation` edges become `DIRECT` transformations with the matching subtype, and indirect edges (from `extract_lineage(include_indirect = TRUE)`) become `INDIRECT` with subtype `FILTER`, `JOIN`, `GROUP_BY`, or `SORT`. A direct edge's defining expression is carried in the transformation's `description`.

Value

A JSON string containing one OpenLineage `RunEvent` of type `COMPLETE`.

See Also

`extract_lineage()` to compute lineage automatically

Other lineage exporters: `lineage_graphml()`, `lineage_json()`, `lineage_mermaid()`

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
lineage_openlineage(
  lineage,
  run_id = "00000000-0000-4000-8000-000000000000",
  event_time = "2026-01-01T00:00:00.000Z"
)
```

---

lineage_tables *Lineage tables as a data frame*

Description

Summarises a lineage object's nodes: one row per table with its diagram role and column count.

Usage

```
lineage_tables(lineage)
```

Arguments

lineage	The result of extract_lineage(), or any list with nodes and edges built with create_table_node() and create_column_edge().

Value

A data frame with columns table, type ("source", "transform", or "target"), and n_columns.

See Also

Other lineage accessors: lineage_diff(), lineage_edges(), lineage_upstream()

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", c("order_id", "amount")),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list()
)
lineage_tables(lineage)
```

---

lineage_upstream *Trace a column's ancestry or descendants*

Description

lineage_upstream() lists every column that feeds into column, following edges transitively; lineage_downstream() lists every column column feeds into. This is the core impact-analysis question — "what breaks if this column changes?" — answered directly on the lineage object, without exporting to a graph tool.

Usage

```
lineage_upstream(lineage, column)

lineage_downstream(lineage, column)
```

Arguments

lineage	The result of extract_lineage(), or any list with nodes and edges built with create_table_node() and create_column_edge().
column	A "table.column" string identifying the column to trace from, e.g. "output.total_spent".

Value

A character vector of "table.column" identifiers, sorted. Empty when the column has no upstream (or downstream) connections.

See Also

Other lineage accessors: lineage_diff(), lineage_edges(), lineage_tables()

Examples

```
lineage <- list(
  nodes = list(
    create_table_node("orders", "amount"),
    create_table_node("daily_totals", "total", table_type = "target")
  ),
  edges = list(
    create_column_edge("orders", "amount", "daily_totals", "total")
  )
)
lineage_upstream(lineage, "daily_totals.total")
lineage_downstream(lineage, "orders.amount")
```

Index

∗ **lineage accessors**
  lineage_diff, 8
  lineage_edges, 9
  lineage_tables, 17
  lineage_upstream, 18
∗ **lineage exporters**
  lineage_graphml, 12
  lineage_json, 13
  lineage_mermaid, 15
  lineage_openlineage, 16
∗ **manual lineage builders**
  create_column_edge, 2
  create_table_node, 3
  lineage_example, 10

create_column_edge, 2
create_column_edge(), *3, 4*, *9–11*, *13–16*, *18*
create_table_node, 3
create_table_node(), *2, 3*, *9–11*, *13–16*, *18*

dbplyr::memdb_frame(), *4*
dbplyr::sql_render(), *4*

extract_lineage, 4
extract_lineage(), *4*, *7–11*, *13–18*

has_bundle, 7
has_sqlglot, 7

lineage_diff, 8
lineage_diff(), *9*, *18, 19*
lineage_downstream *(*lineage_upstream*)*, 18
lineage_edges, 9
lineage_edges(), *8*, *18, 19*
lineage_example, 10
lineage_example(), *3, 4*
lineage_flow, 10
lineage_flow(), *2–7*, *10*, *15*
lineage_flow-shiny, 11
lineage_flowOutput *(*lineage_flow-shiny*)*, 11
lineage_flowOutput(), *11*
lineage_graphml, 12
lineage_graphml(), *14, 15*, *17*
lineage_json, 13
lineage_json(), *13*, *15*, *17*
lineage_mermaid, 15
lineage_mermaid(), *13, 14*, *17*
lineage_openlineage, 16
lineage_openlineage(), *13–15*
lineage_tables, 17
lineage_tables(), *8, 9*, *19*
lineage_upstream, 18
lineage_upstream(), *8, 9*, *18*

renderLineageFlow *(*lineage_flow-shiny*)*, 11
renderLineageFlow(), *11*
reticulate::py_require(), *7*