

Package: dplyneage (via r-universe)

October 18, 2026

Type Package

Title Column Lineage Visualization for 'dplyr' Pipelines

Version 0.3.1.9000

Description Extracts column-level lineage from 'dplyr' and 'dbplyr' pipelines and from SQL queries, and renders it as an interactive 'React Flow' diagram. The same lineage is available as plain data for impact analysis, for comparing pipeline versions (including a continuous-integration check that fails on breaking provenance changes), and for export to JSON, 'GraphML', 'Mermaid', and 'OpenLineage' interchange formats.

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 arrow, data.table, DBI, dbplyr, dplyr, dtplyr, duckdb, duckplyr, httr2, igraph, knitr, reticulate, rlang, rmarkdown, RSQLite, shiny, testthat (>= 3.2.0), webfakes, withr, xml2

VignetteBuilder knitr

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/Needs/website ducklake=github::tgerke/ducklake-r,
pharmaversesdtm, targets

Config/pak/sysreqs cmake make libuv1-dev

Repository <https://tgerke.r-universe.dev>

Date/Publication 2026-09-18 16:44:29 UTC

RemoteUrl <https://github.com/tgerke/dplyneage>
RemoteRef HEAD
RemoteSha ef3d30d9f8a055c55065fb804256ad8b2e299da0

Contents

create_column_edge	2
create_table_node	3
extract_lineage	4
has_bundle	8
has_sqlglot	9
lineage_check	9
lineage_diff	11
lineage_edges	12
lineage_emit	13
lineage_example	15
lineage_flow	16
lineage_flow-shiny	18
lineage_graphml	19
lineage_json	20
lineage_mermaid	22
lineage_openlineage	23
lineage_tables	26
lineage_unused	27
lineage_upstream	28
Index	30

create_column_edge	<i>Connect two columns in a lineage diagram</i>
--------------------	---

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	<i>Create a table node for a lineage diagram</i>
-------------------	--

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 lazy table straight into it (dbplyr, dtplyr, duckplyr, or arrow), 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 = NULL,
  schema = NULL,
  labels = NULL,
  show_sql = FALSE,
  engine = c("auto", "sqlglot", "r"),
  include_indirect = FALSE
)
```

Arguments

sql	A lazy table (a dbplyr <code>tbl_lazy</code> , a <code>dtplyr::lazy_dt()</code> pipeline, a duckplyr frame, or an arrow query, Table, or Dataset), a single SQL query string, or a named list of these (one element per pipeline model; see Details). Lazy inputs are analyzed directly from their query tree; the query text recorded in metadata comes from the backend (<code>dbplyr::sql_render()</code> , dtplyr's generated <code>data.table</code> call, or the rewritten duckdb SQL for duckplyr; arrow plans have none). When a dbplyr table 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 frame first: <code>dbplyr::tbl_lazy(df, name = "df")</code> builds a lazy table with no database at all, which is enough for lineage; <code>dtplyr::lazy_dt(df, name = "df")</code> and <code>duckplyr::as_duckdb_tibble(df)</code> do the same on those backends; <code>dbplyr::memdb_frame()</code> (or <code>copy_to(dbplyr::memdb(), df, name = "df")</code>) also makes the pipeline collectable. See vignette("getting-started").
dialect	SQL dialect the query is written in, e.g. "duckdb", "postgres", "mysql", "snowflake", "bigquery". Any dialect sqlglot understands works here. The default NULL infers the dialect from a lazy table's database connection (falling back to "duckdb" for connections it does not recognize); SQL strings are parsed as "duckdb" unless a dialect is given. dtplyr and arrow pipelines have no SQL dialect and record the labels "data.table" and "arrow"; a value given here replaces that label in the metadata but changes nothing about the analysis.
schema	Optional table schema used by the sqlglot engine to attribute unqualified columns to the right table and to expand <code>SELECT *</code> : a named list mapping table names to character vectors of column names, e.g. <code>list(orders = c("order_id", "amount"))</code> . A typed form also works, mapping column names to type strings (<code>list(orders = c(order_id = "INTEGER", amount = "DOUBLE"))</code>); the types then show as column tooltips, the same way a harvested database schema's do. Mainly relevant for SQL strings: the R walkers read exact provenance from the query tree, a dbplyr table that falls back to sqlglot harvests its schema from the database connection automatically, and duckplyr frames harvest file-reader schemas themselves (entries given here win).
labels	Optional human-readable column labels: a named list mapping node ids to named character vectors, e.g. <code>list(orders = c(amount = "Order amount in USD"))</code> . Node ids are base table names, pipeline model names, or "output" for a single query's result. Labels ride on the matching nodes, show in <code>lineage_flow()</code> tooltips, land in <code>lineage_json()</code> , and become each schema-facet field's description in <code>lineage_openlineage()</code> . Entries here win over the two automatic sources: label attributes on a local frame's columns (the haven/labelled convention), and column comments read from the table's live database connection (duckdb and postgres; other backends are skipped quietly).
show_sql	If TRUE, print the query text being analyzed: the SQL a dbplyr pipeline generated, or dtplyr's <code>data.table</code> call. arrow pipelines have no query text to show. Default: FALSE.
engine	Which lineage engine to use. "auto" (the default) picks by input: the native R walker for dbplyr (falling back to sqlglot for unsupported constructs), dtplyr, and arrow input, and sqlglot for SQL strings and duckplyr frames. "r" forces

the native walker and errors on anything it cannot trace; duckplyr frames always refuse it, since their relation is only readable through SQL. "sqlglot" needs SQL, so it accepts SQL strings, dbplyr lazy tables (rendered first), and duckplyr frames.

`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

Which engine runs depends on the input. dbplyr lazy tables, dtplyr `lazy_dt()` steps, and arrow queries are walked directly in R, no Python required; each records a matching dialect in the meta-data (the connection's dialect, "data.table", or "arrow"). SQL strings are analyzed by `sqlglot`'s lineage engine via `reticulate` (a Suggests dependency: install `reticulate` to enable it; `sqlglot` itself is provisioned automatically). duckplyr frames also need `sqlglot`: their lazy tree lives inside duckdb as an opaque relation, so it is rendered to duckdb SQL and parsed. If a dbplyr pipeline uses a construct the R walker cannot trace (e.g. raw SQL injected with `dbplyr::sql()`), it falls back to `sqlglot` automatically; dtplyr and arrow pipelines compile to `data.table` code and Acero plans rather than SQL, so an untraceable construct there is an error instead.

Every engine traces the same verbs. `select()`, `rename()`, and `relocate()` draw identity edges to the new name and position. `mutate()` and `transmute()` draw an edge from every column an expression reads, whether the expression arrives through `across()`, refers to a column defined earlier in the same call, overwrites its own input, or runs per group with `.by`; `summarise()`, joins, set operations, `distinct()`, and window functions resolve the same way, since each walker reads the compiled form its backend produced rather than the verbs. What a backend refuses fails before lineage runs: `rowwise()` has no lazy method anywhere, dbplyr and dtplyr reject `across(where(...))`, arrow rejects anonymous functions inside `across()`, and duckplyr falls back to eager dplyr for grouped `mutate()` and for functions it cannot translate (see `vignette("getting-started")`).

Source nodes take their names from the backend: table paths for dbplyr, the name given to `dtplyr::lazy_dt()` (auto-named `_DT1`, `_DT2`, ... otherwise, so passing name is worthwhile), and file paths for duckplyr file readers and arrow datasets. In-memory duckplyr frames and arrow tables carry no name of their own and get placeholders (`df`, `df_1`, ... and `arrow_table`, `arrow_table_2`, ...).

Every engine traces select-list lineage by default: columns used only in `filter()`, join conditions, or `arrange()` do not create lineage edges. A window function's partition and ordering columns do: they sit inside the expression's `OVER` clause, so `row_number()` under `group_by(g)` and `window_order(d)` draws direct edges from both `g` and `d`. That window rule belongs to the SQL backends; `data.table` and Acero have no `OVER` clause, so on dtplyr and arrow pipelines a windowed column's grouping keys stay indirect ("group_by") instead of becoming direct sources. Set `include_indirect = TRUE` to add the rest 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 (window `ORDER BY` columns included) 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 dialect, the engine used, node/edge counts, and a models map holding each model's analyzed SQL, engine, and dialect (one entry, keyed by the output table, for a single query).

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()

# Column labels ride along: database comments are read automatically,
# propagate to passthrough output columns, and a labels argument
# documents the computed ones. Hover a column in the diagram to see them.
invisible(DBI::dbExecute(
  con, "COMMENT ON COLUMN orders.customer_id IS 'Customer surrogate key'"
))
```

```

))
tbl(con, "orders") |>
  group_by(customer_id) |>
  summarise(total = sum(amount, na.rm = TRUE)) |>
  extract_lineage(labels = list(output = c(total = "Total spent"))) |>
  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 duckplyr frames (or falls back to sqlglot for a dbplyr pipeline it cannot trace in R); dbplyr, dtplyr, and arrow pipelines are analyzed by pure-R engines. 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 needs the sqlglot engine (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_check

*Gate a CI run on lineage changes***Description**

Compares two lineage extractions and fails when the changes cross a severity threshold: the one-call version of "diff the lineage and stop the merge". Findings print one per line, and on GitHub Actions they become `::error/::warning` annotations on the run.

Usage

```
lineage_check(
  old,
  new,
  fail_on = c("breaking", "any", "none"),
  annotate = NULL
)
```

Arguments

old, new	Lineage objects from <code>extract_lineage()</code> (or lists with nodes and edges), in before/after order: typically main's extraction and the branch's.
fail_on	Which changes fail the check: "breaking" (the default) stops on breaking changes only, "any" stops on any change, and "none" always passes, printing findings as a report.
annotate	Print findings as GitHub Actions workflow commands (<code>::error::</code> / <code>::warning::</code>) instead of plain lines. The default NULL turns annotations on exactly when the GITHUB_ACTIONS environment variable is "true", as it is on every Actions runner.

Details

The failure is a classed condition, so a wrapper can catch it and report its own way:

```
tryCatch(
  lineage_check(old, new),
  dplyneage_lineage_check_failure = function(cnd) post_comment(cnd$diff)
)
```

The package site's [lineage checks in CI](#) article walks through the full setup, including a GitHub Actions job that extracts lineage on a pull request branch and on main, then fails the merge on breaking changes.

Value

The `lineage_diff()` of the two lineages, invisibly. When the check fails, the error condition (class "dplyneage_lineage_check_failure") carries the same diff in its `diff` field.

See Also

Other lineage accessors: [lineage_diff\(\)](#), [lineage_edges\(\)](#), [lineage_tables\(\)](#), [lineage_unused\(\)](#), [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"))
)
# an unchanged lineage passes
lineage_check(old, old, annotate = FALSE)

# removing the edge into a target column is breaking
new <- old
new$edges <- list()
```

```
try(lineage_check(old, new, annotate = FALSE))

# report-only mode never fails
lineage_check(old, new, fail_on = "none", annotate = FALSE)
```

lineage_diff	<i>Compare two lineage extractions</i>
--------------	--

Description

Reports the column-level edges and table columns that changed between two lineage objects, typically the same pipeline before and after an edit. Edges are keyed by their endpoints, and an edge present in both objects still counts as changed when its transformation classification or defining expression differs: rewriting `sum(amount)` as `mean(amount)` changes provenance even though the same columns stay connected.

Usage

```
lineage_diff(old, new)

lineage_has_changes(diff)
```

Arguments

old, new	Lineage objects from <code>extract_lineage()</code> (or lists with nodes and edges), in before/after order.
diff	The result of <code>lineage_diff()</code> .

Details

Each change is also classified by blast radius: a `severity` column on every element marks it "breaking" when it could invalidate something built on the old lineage, "non-breaking" otherwise. See Details for the rule.

`lineage_check()` wraps the diff for CI: it prints each change and errors when the changes cross a severity threshold.

Severity is judged against old, the lineage existing consumers were built on. A removed or changed edge (and a removed column) is "breaking" when its target column fed other columns downstream, or belonged to a target node: target columns are the pipeline's consumed surface, and the graph cannot see the dashboards and jobs reading them, so changing one is assumed to break something. Nodes without a declared type get the same cautious treatment. Everything else is "non-breaking": additions cannot invalidate an existing consumer, and neither can removing an intermediate column nothing consumed.

One caveat for hand-built lineages whose edges carry no expressions: there, adding a source to an existing column shows up only in `added_edges`, which is always non-breaking. Engine-extracted lineage also reports the column's surviving edges in `changed_edges`, because its defining expression changed, and those get the breaking check.

Value

A `dplyr::lineage_diff` list with data frame elements `added_edges`, `removed_edges`, `changed_edges`, `added_columns`, and `removed_columns`, each carrying a `severity` column ("breaking" or "non-breaking"). `changed_edges` also records the old and new transformation and expression for each edge whose endpoints matched but whose definition differs. The `print` method summarises the changes; zero-row elements mean no change. `lineage_has_changes()` returns `TRUE` when any element has rows.

See Also

Other lineage accessors: [lineage_check\(\)](#), [lineage_edges\(\)](#), [lineage_tables\(\)](#), [lineage_unused\(\)](#), [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. A column used in several of those ways keeps one edge classified by the first kind; the full set is in the [lineage_json\(\)](#) artifact and the `OpenLineage` export.

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_check\(\)](#), [lineage_diff\(\)](#), [lineage_tables\(\)](#), [lineage_unused\(\)](#), [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_emit

Send OpenLineage events to a backend over HTTP

Description

Builds the same events as [lineage_openlineage\(\)](#) and POSTs each one to an OpenLineage endpoint, so dplyneage-extracted lineage lands in a running catalog (Marquez, DataHub, or anything else that speaks the protocol) next to lineage from dbt, Airflow, or Spark. One request per event, JSON body, matching how the reference OpenLineage clients transport events.

Usage

```
lineage_emit(
  lineage,
  url = NULL,
  events = "run",
  namespace = NULL,
```

```

    job_name = "extract_lineage",
    run_id = NULL,
    event_time = NULL,
    event_type = "COMPLETE",
    output_name = NULL,
    nominal_time = NULL,
    parent = NULL,
    api_key = NULL,
    headers = NULL,
    endpoint = "api/v1/lineage"
)

```

Arguments

lineage	The result of <code>extract_lineage()</code> , or any list with nodes and edges built with <code>create_table_node()</code> and <code>create_column_edge()</code> .
url	Base URL of the OpenLineage backend, e.g. "http://localhost:5000" for a local Marquez. Defaults to the <code>OPENLINEAGE_URL</code> environment variable, the same one the reference clients read.
events	Which event kinds to emit: any combination of "run" (the default; one <code>RunEvent</code> for the whole extraction), "job" (run-less <code>JobEvents</code> , one per model), and "dataset" (run-less <code>DatasetEvents</code> , one per dataset). See the Static lineage events section.
namespace	Dataset and job namespace recorded in the event. The default <code>NULL</code> resolves each dataset's namespace from the connection captured at extraction time (see the Namespaces section); pass a string to override everything, matching 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 <code>NULL</code> (the default); pass a fixed UUID for reproducible output.
event_time	Event timestamp in ISO-8601 format. The current UTC time when <code>NULL</code> (the default); pass a fixed timestamp for reproducible output.
event_type	The run state a "run" event reports: "COMPLETE" (the default), "START", "RUNNING", "ABORT", "FAIL", or "OTHER". Static events carry no run state.
output_name	Name recorded for the output dataset in place of the synthetic "output" node id of a single-query extraction; use it when the query's result lands in a known table. Errors on multi-model lineage, whose models already carry their real names.
nominal_time	One or two ISO-8601 timestamps (the scheduled <code>nominalStartTime</code> and optionally <code>nominalEndTime</code>) emitted as the <code>nominalTime</code> run facet. <code>NULL</code> (the default) omits the facet.
parent	A <code>list(run_id = , job_name =)</code> , optionally with a namespace, identifying the orchestrating run this event belongs under (an Airflow task, a dbt run); emitted as the parent run facet. <code>NULL</code> (the default) omits the facet.
api_key	Bearer token added as an Authorization header. Defaults to the <code>OPENLINEAGE_API_KEY</code> environment variable; unset means no auth header.

headers	A named character vector or list of extra HTTP headers.
endpoint	Path of the lineage endpoint under url.

Details

The default endpoint path `api/v1/lineage` is where Marquez listens; for other backends set `endpoint` (DataHub, for example, exposes the OpenLineage endpoint under its own prefix). Marquez accepts run events and the static events = "job" / events = "dataset" kinds alike.

Value

Invisibly, a data frame with one row per event sent: `event` (index) and `status` (HTTP status code).

Errors

A failed request (connection refused, or an HTTP error status) stops with a condition of class `dplyneage_emit_failure` carrying event (the failing index), status (the HTTP status, NA when the request never got a response), and results (the rows for events already sent). Events after the failing one are not sent.

See Also

[lineage_openlineage\(\)](#) for the event documents themselves

Other lineage exporters: [lineage_graphml\(\)](#), [lineage_json\(\)](#), [lineage_mermaid\(\)](#), [lineage_openlineage\(\)](#)

Examples

```
## Not run:
lineage <- extract_lineage(my_query)

# A local Marquez quickstart listens on port 5000
lineage_emit(lineage, url = "http://localhost:5000")

# Design-time lineage, no fabricated run, nested under a known job
lineage_emit(lineage, url = "http://localhost:5000", events = "job")

## End(Not run)
```

lineage_example	<i>A built-in example lineage diagram</i>
-----------------	---

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	<i>Render an interactive column lineage diagram</i>
--------------	---

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,
  minimap = FALSE,
  legend = TRUE,
  theme = c("light", "dark", "auto"),
  export_button = TRUE
)
```

Arguments

nodes	The output of <code>extract_lineage()</code> , or a list of nodes created with <code>create_table_node()</code> .
edges	A list of edges created with <code>create_column_edge()</code> . Ignored when nodes is an <code>extract_lineage()</code> 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.
minimap	If TRUE, draws a pannable overview map in the corner. Off by default; it earns its space on large multi-model graphs.
legend	If TRUE (the default), overlays a small legend naming the node colors and edge styles present in the graph.
theme	"light" (the default), "dark", or "auto", which follows the viewer's prefers-color-scheme. That is the operating-system setting, not the theme of a surrounding Quarto or R Markdown document, so documents rendered dark should pass theme = "dark" explicitly.
export_button	If TRUE (the default), the zoom controls gain a button that downloads the diagram as a PNG.

Details

Clicking a column isolates its trace cone (the transitive upstream and downstream subgraph) and dims everything else; clicking it again, clicking the background, or pressing Escape releases it. In Shiny, the traced column is reported as `input$<outputId>_selected`, a list with table and column entries (NULL when nothing is traced).

Hovering a column shows a small card with its captured type and label (see `extract_lineage()`'s `schema` and `labels` arguments); columns with neither stay quiet. The static SVG fallback shows the same information through native browser tooltips.

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)

# Dark theme with the overview map
lineage_flow(nodes, edges, theme = "dark", minimap = TRUE)
```

```
# 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. When the viewer clicks a column to trace it, the app receives `input$<outputId>_selected`: a list with table and column entries, or NULL once the trace is released, ready to feed `lineage_upstream()` and `lineage_downstream()` on the server side.

Usage

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

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

Arguments

<code>outputId</code>	output variable to read from
<code>width, height</code>	Must be a valid CSS unit (like '100%', '400px', 'auto') or a number, which will be coerced to a string and have 'px' appended.
<code>expr</code>	An expression that generates a <code>lineage_flow</code>
<code>env</code>	The environment in which to evaluate <code>expr</code> .
<code>quoted</code>	Is <code>expr</code> a quoted expression (with <code>quote()</code>)? 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	<i>Export lineage as GraphML</i>
-----------------	----------------------------------

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 <code>extract_lineage()</code> , or any list with nodes and edges built with <code>create_table_node()</code> and <code>create_column_edge()</code> .
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_emit()`, `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, and read it back

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), feeding to a data catalog, or storing on the lake commit that materialized a table.

Usage

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

lineage_from_json(x)
```

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.
x	The document to read: a JSON string as <code>lineage_json()</code> returns it, or the path to a file it wrote.

Details

`lineage_from_json()` reads that document back into a lineage object, rebuilding the nodes and edges every accessor and exporter works on. A committed artifact can then stand in for a fresh extraction as the old side of [lineage_diff\(\)](#) or [lineage_check\(\)](#), and lineage kept in a catalog, a database row, or the `commit_extra_info` of a ducklake commit comes back for [lineage_upstream\(\)](#) and the exports. The [lineage that travels with the data](#) article stores each lake layer's lineage on the commit that materialized it and diffs it between snapshots.

Value

`lineage_json()` returns a JSON string; see the Document shape section. `lineage_from_json()` returns a lineage object (class `dplyneage_lineage`) with nodes, edges, and, when the document carries it, metadata.

Document shape

The top-level keys, in order:

- `format_version`: integer version of this document shape, currently
 1. It bumps only when a change would break an existing consumer; added fields do not bump it.
- `metadata`: present on `extract_lineage()` results. Carries dialect, engine, models, `node_count`, and `edge_count`. `models` maps each model name to its `sql`, `engine`, and `dialect`; a single-query extraction is a one-model map keyed by the output table's name, so consumers read per-model SQL the same way for both shapes. A model extracted from a table on a live database connection also carries namespace, the OpenLineage namespace URI inferred from that connection. The top-level dialect and engine collapse across models, to "mixed" when models disagree.
- `nodes`: objects with `id`, `type` ("source", "transform", or "target"), and `columns`. A node whose `columns` carry types (from a live connection's schema, or a typed schema argument) adds `types`, a column-to-type map over that subset; one whose `columns` carry labels (label attributes on a local frame, database column comments, or a `labels` argument) adds `labels`, shaped the same way. Both spread along identity edges, so a passthrough column reports its source column's type and label.
- `edges`: objects with `source`, `source_column`, `target`, and `target_column`. Edges from `extract_lineage()` also carry `transformation` and, on direct edges, `expression`. An indirect edge whose column shapes the result in several ways (filtered and sorted on, say) adds `transformations`, the full set of kinds with the first one leading.

`lineage_from_json()` rebuilds a lineage object from these fields. Node positions and edge labels are recomputed by the layout rules rather than restored, and the label and animated settings of a hand-built edge do not come back, since the document never carried them.

See Also

`extract_lineage()` to compute lineage automatically

Other lineage exporters: `lineage_emit()`, `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)

  # Read it back: the rebuilt lineage feeds every accessor and exporter
  lineage_diff(lineage, lineage_from_json(path))

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

```

lineage_mermaid	<i>Export lineage as a Mermaid flowchart</i>
-----------------	--

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 <code>extract_lineage()</code> , or any list with nodes and edges built with <code>create_table_node()</code> and <code>create_column_edge()</code> .
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_emit\(\)](#), [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	<i>Export lineage as OpenLineage events</i>
---------------------	---

Description

Serializes a lineage object to **OpenLineage** JSON, the interchange format that data catalogs and lineage backends (Marquez, DataHub, OpenMetadata, ...) ingest. The default is one RunEvent with a ColumnLineage facet on each output dataset; POST it to an OpenLineage endpoint (see [lineage_emit\(\)](#)) and dplyneage-extracted lineage appears alongside lineage from dbt, Airflow, or Spark.

Usage

```
lineage_openlineage(
  lineage,
  path = NULL,
  namespace = NULL,
  job_name = "extract_lineage",
  run_id = NULL,
  event_time = NULL,
  pretty = TRUE,
  events = "run",
  event_type = "COMPLETE",
  output_name = NULL,
  nominal_time = NULL,
  parent = NULL
)
```

Arguments

lineage	The result of <code>extract_lineage()</code> , or any list with nodes and edges built with <code>create_table_node()</code> and <code>create_column_edge()</code> .
path	Optional file to write the JSON to. When supplied, the string is returned invisibly.
namespace	Dataset and job namespace recorded in the event. The default NULL resolves each dataset's namespace from the connection captured at extraction time (see the Namespaces section); pass a string to override everything, matching 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.
events	Which event kinds to emit: any combination of "run" (the default; one RunEvent for the whole extraction), "job" (run-less JobEvents, one per model), and "dataset" (run-less DatasetEvents, one per dataset). See the Static lineage events section.
event_type	The run state a "run" event reports: "COMPLETE" (the default), "START", "RUNNING", "ABORT", "FAIL", or "OTHER". Static events carry no run state.
output_name	Name recorded for the output dataset in place of the synthetic "output" node id of a single-query extraction; use it when the query's result lands in a known table. Errors on multi-model lineage, whose models already carry their real names.
nominal_time	One or two ISO-8601 timestamps (the scheduled nominalStartTime and optionally nominalEndTime) emitted as the nominalTime run facet. NULL (the default) omits the facet.
parent	A <code>list(run_id = , job_name =)</code> , optionally with a namespace, identifying the orchestrating run this event belongs under (an Airflow task, a dbt run); emitted as the parent run facet. NULL (the default) omits the facet.

Value

A JSON string: one indented event document, or NDJSON with one event per line (see the Static lineage events section for when each applies).

Static lineage events

dplyneage extracts lineage from code without running it, which is the case OpenLineage defines run-less events for. `events = "job"` emits one JobEvent per model: its job named after the model, its inputs the datasets the model reads (upstream models included), its output carrying the columnLineage facet. `events = "dataset"` emits one DatasetEvent per dataset, sources included, as a static schema registration. Both carry no run, so no run has to be fabricated for design-time lineage. Kinds combine: `events = c("job", "dataset")` emits both sets in one document.

When the selection yields a single event and `pretty = TRUE`, the result is one indented JSON document. Anything else (several events, or `pretty = FALSE`) is NDJSON, one compact event per line, the format OpenLineage's FileTransport writes and replay tooling reads: a committed events file can later be replayed into any backend.

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.

Namespaces

OpenLineage groups datasets by namespace, and catalogs join events to known datasets through it, so the spec expects scheme URIs derived from the data store (`postgres://host:port`, `mysql://host:port`). `extract_lineage()` captures that URI from the table's live connection: `duckdb:<path>` and `sqlite:<path>` for file-backed databases (bare `duckdb/sqlite` in memory, since the spec names no convention for either), the spec's `host:port` form for server databases. Each dataset in the event uses the namespace captured for its model. A URI-shaped namespace is also recorded in the dataset's `dataSource` facet. Where nothing was captured (local frames via `dbplyr::tbl_lazy()`, hand-built graphs, unrecognized drivers), datasets fall back to `"dplyneage"`. The job namespace is always `"dplyneage"`: OpenLineage job namespaces identify the producer, not the data store. Passing an explicit namespace overrides all of this, for datasets and job alike.

Facets

Beyond schema, `dataSource`, and `columnLineage` on datasets, the event carries job facets: `jobType` (BATCH/DPLYNEAGE/QUERY) and, for single-model lineage, `sql` with the analyzed query and its dialect. Schema facet fields include a `type` when column types were captured (from a live connection's tables, or a schema argument with named entries like `list(orders = list(amount = "DOUBLE"))`), and a `description` when column labels were: from label attributes on a local frame (the haven/labelled convention), database column comments, or `extract_lineage()`'s `labels` argument. Both propagate along identity edges, so an output dataset's passthrough columns report their source columns' types and descriptions. Indirect edges land in the `columnLineage` facet's dataset-level `dataset` array rather than under individual output columns: filter, join, group and sort columns shape the whole result, which is exactly what that array expresses. Optional run facets (`nominalTime`, `parent`) attach through the matching arguments.

See Also

`extract_lineage()` to compute lineage automatically

Other lineage exporters: `lineage_emit()`, `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	<i>Lineage tables as a data frame</i>
----------------	---------------------------------------

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_check\(\)](#), [lineage_diff\(\)](#), [lineage_edges\(\)](#), [lineage_unused\(\)](#), [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_unused	<i>Report columns with no path to any target</i>
----------------	--

Description

The dead-column report: every column on a source or transform table from which no chain of edges reaches a target table. In a multi-model lineage that surfaces base-table columns nothing reads and intermediate-model outputs no downstream model consumes, both safe to drop as far as the graph can see.

Usage

```
lineage_unused(lineage)
```

Arguments

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

Details

A single-query extraction usually reports nothing, because its source nodes only carry columns the query referenced. Tables whose type is unknown (hand-built nodes without a `table_type`) count as targets, on the same reasoning as [lineage_diff\(\)](#) severity: the graph cannot see who consumes them, so their columns, and columns feeding them, are not called unused.

Value

A data frame with columns `table`, `column`, and `table_type`, one row per unused column, sorted by table then column. Zero rows when every column reaches a target.

See Also

Other lineage accessors: [lineage_check\(\)](#), [lineage_diff\(\)](#), [lineage_edges\(\)](#), [lineage_tables\(\)](#), [lineage_upstream\(\)](#)

Examples

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

lineage_upstream	<i>Trace a column's ancestry or descendants</i>
------------------	---

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. Passing a table name instead of a column traces from every column of that table at once.

Usage

```
lineage_upstream(lineage, column)

lineage_downstream(lineage, column)
```

Arguments

lineage	The result of <code>extract_lineage()</code> , or any list with nodes and edges built with <code>create_table_node()</code> and <code>create_column_edge()</code> .
column	A "table.column" string identifying the column to trace from, e.g. "output.total_spent", or a table name to trace from all of that table's columns. Names are matched exactly, never split on dots; a string that is both a column key and a schema-qualified table name (a node "main" with column "silver" alongside a node "main.silver") is read as the column key.

Value

A character vector of "table.column" identifiers, sorted. Empty when the column has no upstream (or downstream) connections. For a table, the table's own columns are not part of the answer.

See Also

Other lineage accessors: `lineage_check()`, `lineage_diff()`, `lineage_edges()`, `lineage_tables()`, `lineage_unused()`

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")  
lineage_downstream(lineage, "orders")
```

Index

- * **lineage accessors**
 - lineage_check, 9
 - lineage_diff, 11
 - lineage_edges, 12
 - lineage_tables, 26
 - lineage_unused, 27
 - lineage_upstream, 28
- * **lineage exporters**
 - lineage_emit, 13
 - lineage_graphml, 19
 - lineage_json, 20
 - lineage_mermaid, 22
 - lineage_openlineage, 23
- * **manual lineage builders**
 - create_column_edge, 2
 - create_table_node, 3
 - lineage_example, 15
- create_column_edge, 2
- create_column_edge(), 3, 4, 13, 14, 16, 19, 20, 22, 24, 26–28
- create_table_node, 3
- create_table_node(), 2, 3, 13, 14, 16, 19, 20, 22, 24, 26–28
- dbplyr::memdb_frame(), 5
- dbplyr::sql_render(), 5
- dbplyr::tbl_lazy(), 25
- dtplyr::lazy_dt(), 5, 6
- extract_lineage, 4
- extract_lineage(), 4, 9–14, 16, 17, 19–28
- has_bundle, 8
- has_sqlglot, 9
- lineage_check, 9
- lineage_check(), 11–13, 20, 26–28
- lineage_diff, 11
- lineage_diff(), 10, 13, 20, 26–28
- lineage_downstream(lineage_upstream), 28
- lineage_downstream(), 18
- lineage_edges, 12
- lineage_edges(), 10, 12, 26–28
- lineage_emit, 13
- lineage_emit(), 19, 21, 23, 25
- lineage_example, 15
- lineage_example(), 3, 4
- lineage_flow, 16
- lineage_flow(), 2–5, 7, 8, 16, 22
- lineage_flow-shiny, 18
- lineage_flowOutput
 - (lineage_flow-shiny), 18
- lineage_flowOutput(), 17
- lineage_from_json(lineage_json), 20
- lineage_graphml, 19
- lineage_graphml(), 15, 21, 23, 25
- lineage_has_changes(lineage_diff), 11
- lineage_json, 20
- lineage_json(), 5, 12, 15, 19, 23, 25
- lineage_mermaid, 22
- lineage_mermaid(), 15, 19, 21, 25
- lineage_openlineage, 23
- lineage_openlineage(), 5, 13, 15, 19, 21, 23
- lineage_tables, 26
- lineage_tables(), 10, 12, 13, 27, 28
- lineage_unused, 27
- lineage_unused(), 10, 12, 13, 26, 28
- lineage_upstream, 28
- lineage_upstream(), 10, 12, 13, 18, 20, 26, 27
- renderLineageFlow(lineage_flow-shiny), 18
- renderLineageFlow(), 17
- reticulate::py_require(), 9