Package 'AstraeaDB'

August 21, 2026

Type Package

Title Client for the 'AstraeaDB' Graph Database

Version 0.2.1

Description Provides a client for 'AstraeaDB', a graph database with vector search capabilities. Supports node and edge create, read, update, and delete operations, label and edge-type lookups, graph traversals (breadth-first search, depth-first search, shortest path), temporal (time-travel) queries, graph algorithms (PageRank, Louvain community detection, connected components, and degree and betweenness centrality), vector similarity search, hybrid graph-vector search, Graph Query Language (GQL) execution, and graph-based retrieval-augmented generation (subgraph extraction with large language model integration). Communicates with the 'AstraeaDB' server over a JSON-over-TCP protocol. An optional 'Apache Arrow Flight' transport is available for high-performance bulk operations when the 'arrow' package is installed.

License MIT + file LICENSE

URL https://github.com/AstraeaDB/R-AstraeaDB

BugReports https://github.com/AstraeaDB/R-AstraeaDB/issues

Encoding UTF-8

RoxygenNote 7.3.2

Depends R (>= 3.5.0)

Imports jsonlite, R6

Suggests arrow, knitr, rmarkdown, testthat (>= 3.0.0), withr

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author James Harris [aut, cre]

Maintainer James Harris <jimeharrisjr@gmail.com>

Repository CRAN

Date/Publication 2026-08-21 13:50:28 UTC

Contents

`ArrowClient` *ArrowClient*

Description

An R6 class providing a high-performance client for AstraeaDB using the Apache Arrow Flight protocol. Arrow Flight enables efficient columnar data transfer, making it well-suited for bulk queries and analytical workloads.

The **arrow** package is optional and listed in `Suggests`. This client checks for its availability at initialization and provides an informative error message if **arrow** is not installed.

Details

Arrow Flight is a gRPC-based protocol for high-performance data transport. When the AstraeaDB server exposes a Flight endpoint, this client can execute GQL queries and receive results as Apache Arrow Tables or record batches, which are more efficient than JSON for large result sets.

Typical usage:

1. Create an `ArrowClient` instance with the Flight URI.
2. Call `$connect()` to establish the connection.
3. Execute queries with `$query()` (returns Arrow Table) or `$query_df()` (returns data.frame).
4. Call `$disconnect()` when finished.

For most users, the `UnifiedClient` is recommended as it automatically selects the best transport available.

Public fields

`uri` Character. The Arrow Flight server URI. Defaults to `"grpc://localhost:7689"`.

`client` The Arrow Flight client connection object, or `NULL` when disconnected.

Methods

Public methods:

Method `new()`**:** Create a new ArrowClient.

Checks that the **arrow** package is installed and available. If not, an informative error is raised directing the user to install it.

Usage:

```
ArrowClient$new(uri = "grpc://localhost:7689")
```

Arguments:

`uri` Character. The Arrow Flight gRPC URI. Defaults to `"grpc://localhost:7689"`.

Returns: A new `ArrowClient` object.

Method `connect()`**:** Connect to the Arrow Flight server.

Establishes a gRPC connection to the AstraeaDB Flight endpoint specified by `uri`.

Usage:

```
ArrowClient$connect()
```

Returns: Invisibly returns `self` for method chaining.

Method `disconnect()`**:** Disconnect from the Arrow Flight server.

Closes the connection and sets the client to `NULL`.

Usage:

```
ArrowClient$disconnect()
```

Returns: Invisibly returns `self` for method chaining.

Method `is_connected()`**:** Check whether the client is currently connected.

Usage:

```
ArrowClient$is_connected()
```

Returns: Logical. `TRUE` if connected, `FALSE` otherwise.

Method `query()`**:** Execute a GQL query and return an Arrow Table.

Sends the query as a Flight command descriptor, retrieves flight info, and fetches the data from the first endpoint.

Usage:

```
ArrowClient$query(gql)
```

Arguments:

`gql` Character. A GQL query string.

Returns: An Arrow Table containing the query results.

Examples:

```
\donttest{
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  table <- client$query("MATCH (n:Person) RETURN n.name")
  client$disconnect()
}
}
```

Method `query_df()`**:** Execute a GQL query and return the result as a data.frame.

This is a convenience wrapper around `$query()` that converts the Arrow Table to a base R data.frame.

Usage:

```
ArrowClient$query_df(gql)
```

Arguments:

`gql` Character. A GQL query string.

Returns: A data.frame containing the query results.

Examples:

```
\donttest{
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")
  client$disconnect()
}
}
```

Method `query_batches()`**:** Execute a GQL query and process results in record batches.

Instead of reading all results into memory at once, this method streams results as individual Arrow RecordBatch objects and passes each to the provided callback function. This is useful for processing large result sets with bounded memory usage.

Usage:

```
ArrowClient$query_batches(gql, callback)
```

Arguments:

`gql` Character. A GQL query string.

`callback` Function. A function that accepts a single argument (an Arrow RecordBatch). Called once for each batch received.

Returns: Invisibly returns `NULL`.

Examples:

```
\donttest{
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  client$query_batches(
    "MATCH (n) RETURN n",
    callback = function(batch) {
      cat("Batch with", nrow(batch), "rows\n")
    }
  )
  client$disconnect()
}
}
```

Method `list_flights()`**:** List available flights on the server.

Queries the Flight server for its list of available flight descriptors.

Usage:

```
ArrowClient$list_flights()
```

Returns: A list of available flights as reported by the server.

Examples:

```
\donttest{
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  flights <- client$list_flights()
  client$disconnect()
}
}
```

Method `print()`**:** Print a summary of the ArrowClient.

Usage:

```
ArrowClient$print(...)
```

Arguments:

`...` Ignored. Present for compatibility with the generic.

Returns: Invisibly returns `self`.

Examples

```
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  # Requires the arrow package: install.packages("arrow")
  client <- ArrowClient$new("grpc://localhost:7689")
```

```
  client$connect()

  # Execute a GQL query and get an Arrow Table
  table <- client$query("MATCH (n:Person) RETURN n.name, n.age")

  # Or get a data.frame directly
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")

  # Process large results in batches
  client$query_batches(
    "MATCH (n) RETURN n",
    callback = function(batch) {
      cat("Received batch with", nrow(batch), "rows\n")
    }
  )

  # List available flights
  flights <- client$list_flights()

  client$disconnect()
}



## ------------------------------------------------
## Method `ArrowClient$query`
## ------------------------------------------------


if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  table <- client$query("MATCH (n:Person) RETURN n.name")
  client$disconnect()
}


## ------------------------------------------------
## Method `ArrowClient$query_df`
## ------------------------------------------------


if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")
  client$disconnect()
}


## ------------------------------------------------
```

```
## Method `ArrowClient$query_batches`
## ------------------------------------------------


if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  client$query_batches(
    "MATCH (n) RETURN n",
    callback = function(batch) {
      cat("Batch with", nrow(batch), "rows\n")
    }
  )
  client$disconnect()
}


## ------------------------------------------------
## Method `ArrowClient$list_flights`
## ------------------------------------------------


if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- ArrowClient$new()
  client$connect()
  flights <- client$list_flights()
  client$disconnect()
}
```

`AstraeaClient` *AstraeaDB Client*

Description

R6 client for the AstraeaDB graph database using the JSON-over-TCP protocol.

The client communicates with an AstraeaDB server by sending JSON-encoded request lines over a TCP socket and reading JSON-encoded response lines back. Each request contains a `"type"` field identifying the operation, and each response contains a `"status"` field (`"ok"` or `"error"`) along with a `"data"` payload on success.

The client supports a comprehensive set of operations including:

- Node and edge CRUD (create, read, update, delete)
- Node and edge lookup by label or edge type
- Graph traversals (BFS, DFS, shortest path)
- Temporal queries (time-travel over edges with validity windows)

- Graph algorithms (PageRank, Louvain community detection, connected components, degree and betweenness centrality)
- GQL query execution
- Vector similarity search (k-NN)
- Hybrid graph-vector search
- Semantic neighbor ranking and semantic walks
- GraphRAG (subgraph extraction for LLM integration)
- Graph statistics and raw subgraph export
- Batch and data frame import/export operations

Connection

Create a client with `AstraeaClient$new()`, then call `$connect()` to open the TCP socket. Always call `$disconnect()` when finished, or use `on.exit` to ensure cleanup.

Authentication

If the server requires authentication, pass an `auth_token` to the constructor. The token is automatically attached to every request.

Public fields

`host` Character scalar. Server hostname. Default `"127.0.0.1"`.

`port` Integer scalar. Server port. Default `7687L`.

`con` Socket connection object, or `NULL` when disconnected.

`auth_token` Character scalar or `NULL`. Optional authentication token sent with every request.

Methods

Public methods:

- `AstraeaClient$new()`
- `AstraeaClient$connect()`
- `AstraeaClient$disconnect()`
- `AstraeaClient$is_connected()`
- `AstraeaClient$print()`
- `AstraeaClient$ping()`
- `AstraeaClient$create_node()`
- `AstraeaClient$get_node()`
- `AstraeaClient$update_node()`
- `AstraeaClient$delete_node()`
- `AstraeaClient$create_edge()`
- `AstraeaClient$get_edge()`
- `AstraeaClient$update_edge()`
- `AstraeaClient$delete_edge()`

- AstraeaClient$neighbors()
- AstraeaClient$bfs()
- AstraeaClient$shortest_path()
- AstraeaClient$neighbors_at()
- AstraeaClient$bfs_at()
- AstraeaClient$shortest_path_at()
- AstraeaClient$query()
- AstraeaClient$vector_search()
- AstraeaClient$hybrid_search()
- AstraeaClient$semantic_neighbors()
- AstraeaClient$semantic_walk()
- AstraeaClient$extract_subgraph()
- AstraeaClient$graph_rag()
- AstraeaClient$dfs()
- AstraeaClient$dfs_at()
- AstraeaClient$find_by_label()
- AstraeaClient$find_edge_by_type()
- AstraeaClient$delete_by_label()
- AstraeaClient$get_subgraph()
- AstraeaClient$graph_stats()
- AstraeaClient$run_pagerank()
- AstraeaClient$run_louvain()
- AstraeaClient$run_connected_components()
- AstraeaClient$run_degree_centrality()
- AstraeaClient$run_betweenness_centrality()
- AstraeaClient$create_nodes()
- AstraeaClient$create_edges()
- AstraeaClient$delete_nodes()
- AstraeaClient$delete_edges()
- AstraeaClient$import_nodes_df()
- AstraeaClient$import_edges_df()
- AstraeaClient$export_nodes_df()
- AstraeaClient$export_bfs_df()
- AstraeaClient$results_to_dataframe()
- AstraeaClient$nodes_to_dataframe()
- AstraeaClient$clone()

Method `new()`**:** Create a new AstraeaDB client.

Usage:

```
AstraeaClient$new(host = "127.0.0.1", port = 7687L, auth_token = NULL)
```

Arguments:

`host` Character scalar. Server hostname. Default `"127.0.0.1"`.

`port` Integer scalar. Server port. Default `7687L`.

`auth_token` Character scalar or `NULL`. Optional authentication token.

Returns: An `AstraeaClient` object (invisibly).

Examples:

```
client <- AstraeaClient$new()
client <- AstraeaClient$new(host = "db.example.com", port = 7688L)
client <- AstraeaClient$new(auth_token = "my-secret-token")
```

Method `connect()`**:** Open a TCP socket connection to the AstraeaDB server.

Usage:

```
AstraeaClient$connect()
```

Returns: The client object (invisibly), for method chaining.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- AstraeaClient$new()
  client$connect()
}
}
```

Method `disconnect()`**:** Close the TCP socket connection.

Usage:

```
AstraeaClient$disconnect()
```

Returns: The client object (invisibly).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  client$disconnect()
  client$disconnect()
}
}
```

Method `is_connected()`**:** Check whether the client is currently connected.

Usage:

```
AstraeaClient$is_connected()
```

Returns: Logical scalar. `TRUE` if connected, `FALSE` otherwise.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  client$is_connected()
  client$disconnect()
}
}
```

Method `print()`**:** Print method showing connection status.

Usage:

```
AstraeaClient$print(...)
```

Arguments:

`...` Ignored. Present for compatibility with the generic.

Returns: The client object (invisibly).

Method `ping()`**:** Health-check ping. Returns server information.

Usage:

```
AstraeaClient$ping()
```

Returns: A list with server information (e.g., `version`, `pong`).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  info <- client$ping()
  message(info$version)
  client$disconnect()
}
}
```

Method `create_node()`**:** Create a node.

Usage:

```
AstraeaClient$create_node(labels, properties, embedding = NULL)
```

Arguments:

`labels` Character vector of labels for the node.

`properties` Named list of node properties.

`embedding` Optional numeric vector. An embedding associated with the node for vector search.

Returns: Integer scalar: the ID of the newly created node.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  nid <- client$create_node(
    labels = c("Person"),
    properties = list(name = "Alice", age = 30),
    embedding = rep(0.1, dim)
  )
  client$disconnect()
}
}
```

Method `get_node()`**:** Retrieve a node by its ID.

Usage:

```
AstraeaClient$get_node(node_id)
```

Arguments:

`node_id` Integer scalar. The node ID to look up.

Returns: A list with `labels` (character vector) and `properties` (named list).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  node <- client$get_node(a)
  node$labels
  node$properties$name
  client$disconnect()
}
}
```

Method `update_node()`**:** Update a node's properties using merge semantics. Existing properties not present in the update are preserved.

Usage:

```
AstraeaClient$update_node(node_id, properties)
```

Arguments:

`node_id` Integer scalar. The node ID to update.

`properties` Named list of properties to merge.

Returns: The server response data (invisibly).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  client$update_node(a, list(city = "San Francisco"))
  client$disconnect()
}
}
```

Method `delete_node()`**:** Delete a node and all edges connected to it.

Usage:

```
AstraeaClient$delete_node(node_id)
```

Arguments:

`node_id` Integer scalar. The node ID to delete.

Returns: The server response data (invisibly).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  client$delete_node(a)
  client$disconnect()
}
}
```

Method `create_edge()`**:** Create an edge between two nodes, with optional temporal validity.

Usage:

```
AstraeaClient$create_edge(
  source,
  target,
  edge_type,
  properties = list(),
  weight = 1,
  valid_from = NULL,
  valid_to = NULL
)
```

Arguments:

`source` Integer scalar. Source node ID.

`target` Integer scalar. Target node ID.

`edge_type` Character scalar. The relationship type (e.g., `"KNOWS"`).

`properties` Named list of edge properties. Default `list()`.

`weight` Numeric scalar. Edge weight. Default `1.0`.

`valid_from` Numeric scalar or `NULL`. Start of temporal validity window (milliseconds since epoch).

`valid_to` Numeric scalar or `NULL`. End of temporal validity window (milliseconds since epoch).

Returns: Integer scalar: the ID of the newly created edge.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  eid <- client$create_edge(
    source    = a,
    target    = b,
    edge_type = "KNOWS",
    weight    = 0.9
  )
  client$disconnect()
}
}
```

Method `get_edge()`**:** Retrieve an edge by its ID.

Usage:

```
AstraeaClient$get_edge(edge_id)
```

Arguments:

`edge_id` Integer scalar. The edge ID to look up.

Returns: A list with `source`, `target`, `edge_type`, `properties`, and optional temporal fields.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  edge <- client$get_edge(e)
  edge$edge_type
  client$disconnect()
}
}
```

Method `update_edge()`**:** Update an edge's properties using merge semantics.

Usage:

```
AstraeaClient$update_edge(edge_id, properties)
```

Arguments:

`edge_id` Integer scalar. The edge ID to update.

`properties` Named list of properties to merge.

Returns: The server response data (invisibly).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  client$update_edge(e, list(strength = "strong"))
  client$disconnect()
}
}
```

Method `delete_edge()`**:** Delete an edge.

Usage:

```
AstraeaClient$delete_edge(edge_id)
```

Arguments:

`edge_id` Integer scalar. The edge ID to delete.

Returns: The server response data (invisibly).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  client$delete_edge(e)
  client$disconnect()
}
}
```

Method `neighbors()`**:** Get neighbors of a node, optionally filtered by direction and edge type.

Usage:

```
AstraeaClient$neighbors(node_id, direction = "outgoing", edge_type = NULL)
```

Arguments:

`node_id` Integer scalar. The node whose neighbors to retrieve.

`direction` Character scalar. One of `"outgoing"`, `"incoming"`, or `"both"`. Default `"outgoing"`.

`edge_type` Character scalar or `NULL`. If non-`NULL`, only return neighbors connected by this edge type.

Returns: A list of neighbor entries. Each entry is a list with at least `node_id` and `edge_id`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  nbrs <- client$neighbors(a, direction = "outgoing")
  nbrs_knows <- client$neighbors(a, edge_type = "KNOWS")
  client$disconnect()
}
}
```

Method `bfs()`**:** Breadth-first search starting from a node.

Usage:

```
AstraeaClient$bfs(start, max_depth = 3L)
```

Arguments:

`start` Integer scalar. The starting node ID.

`max_depth` Integer scalar. Maximum traversal depth. Default `3L`.

Returns: A list of entries, each a list with `node_id` (integer) and `depth` (integer).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  bfs_result <- client$bfs(a, max_depth = b)
  client$disconnect()
}
}
```

Method `shortest_path()`**:** Find the shortest path between two nodes.

Usage:

```
AstraeaClient$shortest_path(from_node, to_node, weighted = FALSE)
```

Arguments:

`from_node` Integer scalar. Source node ID.

`to_node` Integer scalar. Target node ID.

`weighted` Logical scalar. If `TRUE`, use edge weights (Dijkstra). If `FALSE`, use hop count. Default `FALSE`.

Returns: A list with `path` (integer vector of node IDs), `length` (hop count), and optionally `cost` (total weight when `weighted = TRUE`).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sp <- client$shortest_path(a, b, weighted = TRUE)
  sp$path
  sp$cost
  client$disconnect()
}
}
```

Method `neighbors_at()`**:** Get neighbors of a node at a specific point in time. Only edges whose temporal validity window includes `timestamp` are traversed.

Usage:

```
AstraeaClient$neighbors_at(
  node_id,
  direction = "outgoing",
  timestamp,
  edge_type = NULL
)
```

Arguments:

`node_id` Integer scalar. The node whose neighbors to retrieve.

`direction` Character scalar. One of `"outgoing"`, `"incoming"`, or `"both"`. Default `"outgoing"`.

`timestamp` Numeric scalar. Point in time as milliseconds since the Unix epoch.

`edge_type` Character scalar or `NULL`. Optional edge type filter.

Returns: A list of neighbor entries valid at the given timestamp.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  # Neighbors as of January 1 2023 (ms since epoch)
  nbrs <- client$neighbors_at(a, "outgoing", 1672531200000)
  client$disconnect()
}
}
```

Method `bfs_at()`**:** Breadth-first search at a specific point in time.

Usage:

```
AstraeaClient$bfs_at(start, max_depth = 3L, timestamp)
```

Arguments:

`start` Integer scalar. The starting node ID.

`max_depth` Integer scalar. Maximum traversal depth. Default `3L`.

`timestamp` Numeric scalar. Point in time (ms since epoch).

Returns: A list of entries with `node_id` and `depth`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  result <- client$bfs_at(a, max_depth = b, timestamp = 1672531200000)
  client$disconnect()
}
}
```

Method `shortest_path_at()`**:** Find the shortest path at a specific point in time.

Usage:

```
AstraeaClient$shortest_path_at(from_node, to_node, timestamp, weighted = FALSE)
```

Arguments:

`from_node` Integer scalar. Source node ID.

`to_node` Integer scalar. Target node ID.

`timestamp` Numeric scalar. Point in time (ms since epoch).

`weighted` Logical scalar. Use edge weights? Default `FALSE`.

Returns: A list with `path`, `length`, and optionally `cost`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sp <- client$shortest_path_at(a, b, timestamp = 1672531200000)
  client$disconnect()
}
}
```

Method `query()`**:** Execute a GQL (Graph Query Language) query string.

Usage:

```
AstraeaClient$query(gql)
```

Arguments:

`gql` Character scalar. The GQL query to execute.

Returns: The query result data as returned by the server.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  result <- client$query("MATCH (p:Person) RETURN p.name, p.city")
  client$disconnect()
}
}
```

Method `vector_search()`**:** Perform k-nearest neighbor vector similarity search.

Usage:

```
AstraeaClient$vector_search(query_vector, k = 10L)
```

Arguments:

`query_vector` Numeric vector. The query embedding.

`k` Integer scalar. Number of nearest neighbors to return. Default `10L`.

Returns: A list of result entries, each containing at least `node_id` and `distance` (smaller is closer). A legacy `score` alias equal to `distance` is also present for backward compatibility with older clients.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  results <- client$vector_search(rep(0.1, dim), k = 5L)
  client$disconnect()
}
}
```

Method `hybrid_search()`**:** Combined graph proximity and vector similarity search.

The `alpha` parameter controls the blend between graph proximity and vector similarity. `alpha = 0.0` uses pure graph distance; `alpha = 1.0` uses pure vector similarity.

Usage:

```
AstraeaClient$hybrid_search(
  anchor,
  query_vector,
  max_hops = 3L,
  k = 10L,
  alpha = 0.5
)
```

Arguments:

`anchor` Integer scalar. Anchor node ID for graph proximity.

`query_vector` Numeric vector. Query embedding.

`max_hops` Integer scalar. Maximum graph hops from anchor. Default `3L`.

`k` Integer scalar. Number of results. Default `10L`.

`alpha` Numeric scalar in `[0, 1]`. Blend factor. Default `0.5`.

Returns: A list of result entries with `node_id` and combined scores.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  results <- client$hybrid_search(
    anchor = a,
    query_vector = rep(0.1, dim),
    k = b,
    alpha = 0.7
  )
  client$disconnect()
}
}
```

Method `semantic_neighbors()`**:** Get neighbors ranked by semantic similarity to a concept vector.

Usage:

```
AstraeaClient$semantic_neighbors(
  node_id,
  concept,
  direction = "outgoing",
  k = 10L
)
```

Arguments:

`node_id` Integer scalar. The node whose neighbors to rank.

`concept` Numeric vector. The concept embedding to rank against.

`direction` Character scalar. One of `"outgoing"`, `"incoming"`, or `"both"`. Default `"outgoing"`.

`k` Integer scalar. Maximum number of ranked neighbors. Default `10L`.

Returns: A list of neighbor entries with `node_id` and `distance` (smaller is closer to the concept).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  nbrs <- client$semantic_neighbors(a, rep(0.1, dim), k = b)
  client$disconnect()
}
}
```

Method `semantic_walk()`**:** Greedy walk following edges whose targets are most similar to a concept vector.

Usage:

```
AstraeaClient$semantic_walk(start, concept, max_hops = 3L)
```

Arguments:

`start` Integer scalar. Starting node ID.

`concept` Numeric vector. Concept embedding guiding the walk.

`max_hops` Integer scalar. Maximum walk length. Default `3L`.

Returns: A list representing the walk path.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  path <- client$semantic_walk(a, rep(0.1, dim), max_hops = 4L)
  client$disconnect()
}
}
```

Method `extract_subgraph()`**:** Extract a subgraph centered on a node and linearize it to text.

Usage:

```
AstraeaClient$extract_subgraph(
  center,
  hops = 2L,
  max_nodes = 50L,
  format = "structured"
)
```

Arguments:

`center` Integer scalar. Center node ID.

`hops` Integer scalar. Radius in hops. Default `2L`.

`max_nodes` Integer scalar. Maximum number of nodes to include. Default `50L`.

`format` Character scalar. Output format: one of `"structured"`, `"prose"`, `"triples"`, or `"json"`. Default `"structured"`.

Returns: A list with extracted subgraph data including `text`, `node_count`, and `edge_count`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sg <- client$extract_subgraph(a, hops = b, max_nodes = 20L)
  sg$text
  client$disconnect()
}
}
```

Method `graph_rag()`**:** Execute a full GraphRAG pipeline: extract a subgraph and send it to a language model.

Provide either `anchor` (a node ID to center the subgraph on) or `question_embedding` (a vector to locate the closest node via vector search), or both.

Usage:

```
AstraeaClient$graph_rag(
  question,
  anchor = NULL,
  question_embedding = NULL,
  hops = 2L,
  max_nodes = 50L,
  format = "structured"
)
```

Arguments:

`question` Character scalar. The natural-language question.

`anchor` Integer scalar or `NULL`. Anchor node ID.

`question_embedding` Numeric vector or `NULL`. Embedding of the question for vector-based anchor selection.

`hops` Integer scalar. Subgraph radius. Default `2L`.

`max_nodes` Integer scalar. Maximum subgraph nodes. Default `50L`.

`format` Character scalar. Linearization format. Default `"structured"`.

Returns: A list with the RAG result, typically including an `answer` field.

Examples:

```
\donttest{
if (astraea_server_available()) {
```

```
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  answer <- client$graph_rag(
    question = "What does Alice work on?",
    anchor = a
  )
  client$disconnect()
}
}
```

Method `dfs()`**:** Depth-first search starting from a node.

Usage:

```
AstraeaClient$dfs(start, max_depth = 3L)
```

Arguments:

`start` Integer scalar. The starting node ID.

`max_depth` Integer scalar. Maximum traversal depth. Default `3L`.

Returns: A list of node IDs (integers) in depth-first visitation order.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  visited <- client$dfs(a, max_depth = b)
  client$disconnect()
}
}
```

Method `dfs_at()`**:** Depth-first search as of a specific point in time.

Usage:

```
AstraeaClient$dfs_at(start, max_depth = 3L, timestamp)
```

Arguments:

`start` Integer scalar. The starting node ID.

`max_depth` Integer scalar. Maximum traversal depth. Default `3L`.

`timestamp` Numeric scalar. Point in time (ms since epoch).

Returns: A list of node IDs (integers) in depth-first visitation order as of `timestamp`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  visited <- client$dfs_at(a, max_depth = b, timestamp = 1672531200000)
```

```
  client$disconnect()
}
}
```

Method `find_by_label()`**:** Find all nodes carrying a given label.

Usage:

```
AstraeaClient$find_by_label(label)
```

Arguments:

`label` Character scalar. The node label to match.

Returns: A list of matching node IDs (integers).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  ids <- client$find_by_label("Person")
  client$disconnect()
}
}
```

Method `find_edge_by_type()`**:** Find all edges of a given edge type.

Usage:

```
AstraeaClient$find_edge_by_type(edge_type)
```

Arguments:

`edge_type` Character scalar. The edge type to match.

Returns: A list of entries, each a list with `edge_id`, `source`, and `target` node IDs.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  edges <- client$find_edge_by_type("KNOWS")
  client$disconnect()
}
}
```

Method `delete_by_label()`**:** Delete every node carrying a given label, along with all its edges.

Usage:

```
AstraeaClient$delete_by_label(label)
```

Arguments:

`label` Character scalar. The node label to match.

Returns: Integer scalar: the number of nodes deleted.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  n_removed <- client$delete_by_label("Temporary")
  client$disconnect()
}
}
```

Method `get_subgraph()`**:** Retrieve the raw subgraph (nodes and edges) around a center node, suitable for visualization or client-side processing.

Usage:

```
AstraeaClient$get_subgraph(center, hops = 3L, max_nodes = 50L)
```

Arguments:

`center` Integer scalar. The center node ID.

`hops` Integer scalar. Neighborhood radius in hops. Default `3L`.

`max_nodes` Integer scalar. Maximum number of nodes to return. Default `50L`.

Returns: A list with `nodes` (each a list with `id`, `labels`, `properties`, `has_embedding`) and `edges` (each a list with `id`, `source`, `target`, `edge_type`, `properties`, `weight`, `valid_from`, `valid_to`).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sg <- client$get_subgraph(a, hops = b, max_nodes = 100L)
  client$disconnect()
}
}
```

Method `graph_stats()`**:** Retrieve graph-wide statistics: node and edge counts, per-label node counts, and vector-index information when available.

Usage:

```
AstraeaClient$graph_stats()
```

Returns: A named list of statistics, including `total_nodes`, `total_edges`, and `labels` (a per-label node count).

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  stats <- client$graph_stats()
  stats$total_nodes
  client$disconnect()
}
}
```

Method `run_pagerank()`**:** Run the PageRank algorithm over the whole graph or a node subset.

Usage:

```
AstraeaClient$run_pagerank(
  nodes = NULL,
  damping = 0.85,
  max_iterations = 100L,
  tolerance = 1e-06
)
```

Arguments:

`nodes` Optional integer vector. Restrict the computation to these node IDs. `NULL` (default) uses the whole graph.

`damping` Numeric scalar. Damping factor. Default `0.85`.

`max_iterations` Integer scalar. Maximum iterations. Default `100L`.

`tolerance` Numeric scalar. Convergence tolerance. Default `1e-6`.

Returns: A named list mapping node ID (a character key) to its PageRank score.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  scores <- client$run_pagerank()
  client$disconnect()
}
}
```

Method `run_louvain()`**:** Run Louvain community detection over the whole graph or a subset.

Usage:

```
AstraeaClient$run_louvain(nodes = NULL)
```

Arguments:

`nodes` Optional integer vector. Restrict the computation to these node IDs. `NULL` (default) uses the whole graph.

Returns: A list with `communities` (a named list mapping node ID to community index) and `num_communities`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  res <- client$run_louvain()
  res$num_communities
  client$disconnect()
}
}
```

Method `run_connected_components()`**:** Find connected components of the whole graph or a node subset.

Usage:

```
AstraeaClient$run_connected_components(nodes = NULL, strong = FALSE)
```

Arguments:

`nodes` Optional integer vector. Restrict the computation to these node IDs. `NULL` (default) uses the whole graph.

`strong` Logical scalar. If `TRUE`, compute strongly connected components; otherwise weakly connected. Default `FALSE`.

Returns: A list with `components` (a list of integer vectors of node IDs) and `count`.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  cc <- client$run_connected_components()
  cc$count
  client$disconnect()
}
}
```

Method `run_degree_centrality()`**:** Compute degree centrality for the whole graph or a node subset.

Usage:

```
AstraeaClient$run_degree_centrality(nodes = NULL, direction = "outgoing")
```

Arguments:

`nodes` Optional integer vector. Restrict the computation to these node IDs. `NULL` (default) uses the whole graph.

`direction` Character scalar. One of `"outgoing"` (default), `"incoming"`, or `"both"`.

Returns: A named list mapping node ID (a character key) to its degree-centrality score.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  scores <- client$run_degree_centrality(direction = "both")
  client$disconnect()
}
}
```

Method `run_betweenness_centrality()`**:** Compute betweenness centrality for the whole graph or a node subset.

Usage:

```
AstraeaClient$run_betweenness_centrality(nodes = NULL)
```

Arguments:

`nodes` Optional integer vector. Restrict the computation to these node IDs. `NULL` (default) uses the whole graph.

Returns: A named list mapping node ID (a character key) to its betweenness-centrality score.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  scores <- client$run_betweenness_centrality()
  client$disconnect()
}
}
```

Method `create_nodes()`**:** Create multiple nodes in a single batch.

Usage:

```
AstraeaClient$create_nodes(nodes_list)
```

Arguments:

`nodes_list` A list of node specifications. Each element must be a list with `labels` (character vector) and `properties` (named list). An optional `embedding` (numeric vector) may be included.

Returns: An integer vector of created node IDs, in the same order as the input.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  ids <- client$create_nodes(list(
    list(labels = "Person", properties = list(name = "Alice")),
    list(labels = "Person", properties = list(name = "Bob"))
  ))
  client$disconnect()
}
}
```

Method `create_edges()`**:** Create multiple edges in a single batch.

Usage:

```
AstraeaClient$create_edges(edges_list)
```

Arguments:

`edges_list` A list of edge specifications. Each element must be a list with `source`, `target`, and `edge_type`. Optional fields: `properties`, `weight`, `valid_from`, `valid_to`.

Returns: An integer vector of created edge IDs, in the same order as the input.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  eids <- client$create_edges(list(
```

```
    list(source = a, target = b, edge_type = "KNOWS"),
    list(source = b, target = b, edge_type = "FOLLOWS", weight = 0.5)
  ))
  client$disconnect()
}
}
```

Method `delete_nodes()`**:** Delete multiple nodes. Errors for individual nodes are silently skipped.

Usage:

```
AstraeaClient$delete_nodes(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs to delete.

Returns: Integer scalar: the count of successfully deleted nodes.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  deleted <- client$delete_nodes(c(a, b))
  client$disconnect()
}
}
```

Method `delete_edges()`**:** Delete multiple edges. Errors for individual edges are silently skipped.

Usage:

```
AstraeaClient$delete_edges(edge_ids)
```

Arguments:

`edge_ids` Integer vector of edge IDs to delete.

Returns: Integer scalar: the count of successfully deleted edges.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  deleted <- client$delete_edges(e)
  client$disconnect()
}
}
```

Method `import_nodes_df()`**:** Import nodes from a `data.frame`.

Each row becomes a node. One column supplies the label(s), and the remaining columns (excluding any embedding columns) become node properties.

Usage:

```
AstraeaClient$import_nodes_df(
  df,
  label_col = "label",
  id_col = NULL,
  embedding_cols = NULL
)
```

Arguments:

`df` A `data.frame` with one row per node.

`label_col` Character scalar. Name of the column containing node labels. Default `"label"`.

`id_col` Character scalar or `NULL`. If non-`NULL`, this column is stored in the node properties as an external identifier.

`embedding_cols` Character vector or `NULL`. Column names whose values form the embedding vector.

Returns: An integer vector of created node IDs.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  df <- data.frame(
    label = c("Person", "Person"),
    name  = c("Alice", "Bob"),
    age   = c(30, 25),
    stringsAsFactors = FALSE
  )
  ids <- client$import_nodes_df(df)
  client$disconnect()
}
}
```

Method `import_edges_df()`**:** Import edges from a `data.frame`.

Each row becomes an edge. Columns supply the source/target node IDs, edge type, and optionally weight and temporal bounds. Remaining columns become edge properties.

Usage:

```
AstraeaClient$import_edges_df(
  df,
  source_col = "source",
  target_col = "target",
  type_col = "type",
  weight_col = NULL,
  valid_from_col = NULL,
  valid_to_col = NULL
)
```

Arguments:

`df` A `data.frame` with one row per edge.

`source_col` Character scalar. Column with source node IDs. Default `"source"`.

`target_col` Character scalar. Column with target node IDs. Default `"target"`.

`type_col` Character scalar. Column with edge type strings. Default `"type"`.

`weight_col` Character scalar or `NULL`. Column with edge weights.

`valid_from_col` Character scalar or `NULL`. Column with temporal start (ms since epoch).

`valid_to_col` Character scalar or `NULL`. Column with temporal end (ms since epoch).

Returns: An integer vector of created edge IDs.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  edf <- data.frame(
    source = c(a, b),
    target = c(b, a),
    type   = c("KNOWS", "FOLLOWS"),
    stringsAsFactors = FALSE
  )
  eids <- client$import_edges_df(edf)
  client$disconnect()
}
}
```

Method `export_nodes_df()`**:** Export nodes to a `data.frame` with `node_id`, a comma-separated `labels` column, and flattened property columns.

Usage:

```
AstraeaClient$export_nodes_df(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs to export.

Returns: A `data.frame` with one row per node.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  df <- client$export_nodes_df(c(a, b))
  client$disconnect()
}
}
```

Method `export_bfs_df()`**:** Run BFS from a starting node and return the results as a `data.frame` that includes node details.

Usage:

```
AstraeaClient$export_bfs_df(start, max_depth = 3L)
```

Arguments:

`start` Integer scalar. Starting node ID.

`max_depth` Integer scalar. Maximum BFS depth. Default `3L`.

Returns: A `data.frame` with `node_id`, `depth`, `labels`, and flattened property columns.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  bfs_df <- client$export_bfs_df(a, max_depth = b)
  client$disconnect()
}
}
```

Method `results_to_dataframe()`**:** Convert a list of search results to a `data.frame`.

This is a convenience method for converting results from `vector_search()`, `hybrid_search()`, or similar methods into a tabular format.

Usage:

```
AstraeaClient$results_to_dataframe(results)
```

Arguments:

`results` A list of result entries (e.g., from `vector_search()`). Each entry should be a list with named elements.

Returns: A `data.frame` with one row per result entry. Returns an empty `data.frame` if `results` is empty.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  results <- client$vector_search(rep(0.1, dim), k = 5L)
  df <- client$results_to_dataframe(results)
  client$disconnect()
}
}
```

Method `nodes_to_dataframe()`**:** Fetch multiple nodes by ID and return as a `data.frame`.

Similar to `export_nodes_df()` but preserves the `labels` column as a list column (using `I`) rather than collapsing to a comma-separated string.

Usage:

```
AstraeaClient$nodes_to_dataframe(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs to fetch.

Returns: A `data.frame` with columns `id`, `labels` (list column), and flattened property columns.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  df <- client$nodes_to_dataframe(c(a, b))
  df$labels[[1]]
  client$disconnect()
}
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AstraeaClient$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
if (astraea_server_available()) {
  # Connect to a local AstraeaDB server
  client <- AstraeaClient$new()
  client$connect()

  # Health check
  client$ping()

  # Create nodes
  alice_id <- client$create_node(
    labels = c("Person"),
    properties = list(name = "Alice", age = 30)
  )
  bob_id <- client$create_node(
    labels = c("Person"),
    properties = list(name = "Bob", age = 25)
  )

  # Create an edge
  edge_id <- client$create_edge(
    source = alice_id,
    target = bob_id,
    edge_type = "KNOWS",
    properties = list(since = 2020)
  )

  # Traverse the graph
  client$neighbors(alice_id, direction = "outgoing")
  client$bfs(alice_id, max_depth = 2L)
```

```
  # Clean up
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$new`
## ------------------------------------------------

client <- AstraeaClient$new()
client <- AstraeaClient$new(host = "db.example.com", port = 7688L)
client <- AstraeaClient$new(auth_token = "my-secret-token")

## ------------------------------------------------
## Method `AstraeaClient$connect`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- AstraeaClient$new()
  client$connect()
}


## ------------------------------------------------
## Method `AstraeaClient$disconnect`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  client$disconnect()
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$is_connected`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  client$is_connected()
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$ping`
## ------------------------------------------------
```

```
if (astraea_server_available()) {
  client <- astraea_connect()
  info <- client$ping()
  message(info$version)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$create_node`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  nid <- client$create_node(
    labels = c("Person"),
    properties = list(name = "Alice", age = 30),
    embedding = rep(0.1, dim)
  )
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$get_node`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  node <- client$get_node(a)
  node$labels
  node$properties$name
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$update_node`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  client$update_node(a, list(city = "San Francisco"))
  client$disconnect()
}
```

```
## ------------------------------------------------
## Method `AstraeaClient$delete_node`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  client$delete_node(a)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$create_edge`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  eid <- client$create_edge(
    source    = a,
    target    = b,
    edge_type = "KNOWS",
    weight    = 0.9
  )
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$get_edge`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  edge <- client$get_edge(e)
  edge$edge_type
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$update_edge`
```

```
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  client$update_edge(e, list(strength = "strong"))
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$delete_edge`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  e <- client$create_edge(a, b, "KNOWS")
  client$delete_edge(e)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$neighbors`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  nbrs <- client$neighbors(a, direction = "outgoing")
  nbrs_knows <- client$neighbors(a, edge_type = "KNOWS")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$bfs`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  bfs_result <- client$bfs(a, max_depth = b)
```

```
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$shortest_path`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sp <- client$shortest_path(a, b, weighted = TRUE)
  sp$path
  sp$cost
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$neighbors_at`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  # Neighbors as of January 1 2023 (ms since epoch)
  nbrs <- client$neighbors_at(a, "outgoing", 1672531200000)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$bfs_at`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  result <- client$bfs_at(a, max_depth = b, timestamp = 1672531200000)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$shortest_path_at`
## ------------------------------------------------
```

```
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sp <- client$shortest_path_at(a, b, timestamp = 1672531200000)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$query`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  result <- client$query("MATCH (p:Person) RETURN p.name, p.city")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$vector_search`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  results <- client$vector_search(rep(0.1, dim), k = 5L)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$hybrid_search`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  results <- client$hybrid_search(
    anchor = a,
    query_vector = rep(0.1, dim),
    k = b,
    alpha = 0.7
  )
  client$disconnect()
}
```

```
## ------------------------------------------------
## Method `AstraeaClient$semantic_neighbors`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  nbrs <- client$semantic_neighbors(a, rep(0.1, dim), k = b)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$semantic_walk`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  dim <- client$ping()$vector_dim
  path <- client$semantic_walk(a, rep(0.1, dim), max_hops = 4L)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$extract_subgraph`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sg <- client$extract_subgraph(a, hops = b, max_nodes = 20L)
  sg$text
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$graph_rag`
## ------------------------------------------------


if (astraea_server_available()) {
```

```
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  answer <- client$graph_rag(
    question = "What does Alice work on?",
    anchor = a
  )
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$dfs`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  visited <- client$dfs(a, max_depth = b)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$dfs_at`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  visited <- client$dfs_at(a, max_depth = b, timestamp = 1672531200000)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$find_by_label`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  ids <- client$find_by_label("Person")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$find_edge_by_type`
```

```
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  edges <- client$find_edge_by_type("KNOWS")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$delete_by_label`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  n_removed <- client$delete_by_label("Temporary")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$get_subgraph`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  sg <- client$get_subgraph(a, hops = b, max_nodes = 100L)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$graph_stats`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  stats <- client$graph_stats()
  stats$total_nodes
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$run_pagerank`
## ------------------------------------------------
```

```
if (astraea_server_available()) {
  client <- astraea_connect()
  scores <- client$run_pagerank()
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$run_louvain`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  res <- client$run_louvain()
  res$num_communities
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$run_connected_components`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  cc <- client$run_connected_components()
  cc$count
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$run_degree_centrality`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  scores <- client$run_degree_centrality(direction = "both")
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$run_betweenness_centrality`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
```

```
  scores <- client$run_betweenness_centrality()
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$create_nodes`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  ids <- client$create_nodes(list(
    list(labels = "Person", properties = list(name = "Alice")),
    list(labels = "Person", properties = list(name = "Bob"))
  ))
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$create_edges`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  eids <- client$create_edges(list(
    list(source = a, target = b, edge_type = "KNOWS"),
    list(source = b, target = b, edge_type = "FOLLOWS", weight = 0.5)
  ))
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$delete_nodes`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  deleted <- client$delete_nodes(c(a, b))
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$delete_edges`
```

```
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  deleted <- client$delete_edges(e)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$import_nodes_df`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  df <- data.frame(
    label = c("Person", "Person"),
    name  = c("Alice", "Bob"),
    age   = c(30, 25),
    stringsAsFactors = FALSE
  )
  ids <- client$import_nodes_df(df)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$import_edges_df`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  edf <- data.frame(
    source = c(a, b),
    target = c(b, a),
    type   = c("KNOWS", "FOLLOWS"),
    stringsAsFactors = FALSE
  )
  eids <- client$import_edges_df(edf)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$export_nodes_df`
## ------------------------------------------------
```

```
if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  df <- client$export_nodes_df(c(a, b))
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$export_bfs_df`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  bfs_df <- client$export_bfs_df(a, max_depth = b)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$results_to_dataframe`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  dim <- client$ping()$vector_dim
  results <- client$vector_search(rep(0.1, dim), k = 5L)
  df <- client$results_to_dataframe(results)
  client$disconnect()
}


## ------------------------------------------------
## Method `AstraeaClient$nodes_to_dataframe`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- astraea_connect()
  a <- client$create_node(c("Person"), list(name = "Alice"))
  b <- client$create_node(c("Person"), list(name = "Bob"))
  df <- client$nodes_to_dataframe(c(a, b))
  df$labels[[1]]
  client$disconnect()
}
```

astraea_arrow_connect *Connect to AstraeaDB via Arrow Flight*

Description

Creates an ArrowClient and connects to the Arrow Flight server. This is a convenience wrapper that instantiates the client, calls connect(), and returns the connected client object.

Usage

```
astraea_arrow_connect(uri = "grpc://localhost:7689")
```

Arguments

uri	Character. Flight server URI. Default: "grpc://localhost:7689".

Details

The **arrow** package must be installed to use this function.

Value

A connected ArrowClient object.

Examples

```
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  client <- astraea_arrow_connect()
  result <- client$query_df("MATCH (n) RETURN n")
  client$disconnect()
}
```

astraea_connect *Connect to AstraeaDB Server*

Description

Creates an AstraeaClient and connects to the server in one step. This is a convenience wrapper that instantiates the client, calls connect(), and returns the connected client object.

Usage

```
astraea_connect(host = "127.0.0.1", port = 7687L, auth_token = NULL)
```

Arguments

host	Character. Server hostname. Default: "127.0.0.1".
port	Integer. Server port. Default: 7687L.
auth_token	Character or NULL. Optional authentication token.

Value

A connected AstraeaClient object.

Examples

```
if (astraea_server_available()) {
  client <- astraea_connect()
  client$ping()
  client$disconnect()
}
```

---

astraea_server_available

Check if AstraeaDB Server is Available

Description

Tests whether an AstraeaDB server is reachable at the given host and port. Useful for conditionally running examples and tests that require a live server.

Usage

```
astraea_server_available(host = "127.0.0.1", port = 7687L, timeout = 2)
```

Arguments

host	Character. Server hostname. Default: "127.0.0.1".
port	Integer. Server port. Default: 7687L.
timeout	Numeric. Connection timeout in seconds. Default: 2.

Value

Logical. TRUE if server is reachable, FALSE otherwise.

Examples

```
if (astraea_server_available()) {
  client <- astraea_connect()
}
```

classify_service_port *Classify the service port from a source/destination port pair*

Description

Given vectors of source and destination ports, determines which port represents the "service" (server) side of the connection using a three-tier heuristic:

1. If exactly one port is in the privileged range (< 1024), it is the service port.
2. If exactly one port appears in the well-known ports lookup table, it wins.
3. Otherwise the lower port number is chosen (statistically more likely to be the service).

Usage

```
classify_service_port(src_port, dst_port)
```

Arguments

src_port	Integer vector of source port numbers.
dst_port	Integer vector of destination port numbers.

Value

Integer vector of the same length containing the service port for each pair. Returns NA_integer_ where either input is NA.

Examples

```
classify_service_port(52000, 443)
classify_service_port(c(52000, 3306), c(80, 52000))
```

ports *Port Classification Utilities*

Description

Functions for classifying network service ports and looking up well-known port names. Used by the PCAP ingestion script to aggregate packets into flows keyed by service port.

port_service_name *Look up the service name for a port number*

Description

Returns the human-readable service name for a given port number based on the package's built-in well-known ports table, or `NA_character_` if the port is not recognized.

Usage

```
port_service_name(port)
```

Arguments

`port`	Integer vector of port numbers.

Value

Character vector of service names, with `NA_character_` for unrecognized ports.

Examples

```
port_service_name(443)
port_service_name(c(80, 3306, 99999))
```

UnifiedClient *UnifiedClient*

Description

An R6 class that provides a unified interface to AstraeaDB by automatically selecting the best available transport. It always creates a JSON/TCP client (`AstraeaClient`) and optionally creates an Arrow Flight client (`ArrowClient`) when the **arrow** package is installed.

Details

The UnifiedClient delegates all CRUD operations (node, edge, traversal, vector search, etc.) to the JSON/TCP client. For GQL query execution, it prefers the Arrow Flight transport when available, as Arrow provides more efficient columnar data transfer for analytical workloads. If Arrow is not available or its connection fails, queries transparently fall back to the JSON/TCP transport.

Typical usage:

1. Create a `UnifiedClient` instance.
2. Call `$connect()` to establish connections.
3. Use CRUD methods (`$create_node()`, `$get_node()`, etc.) and query methods (`$query()`, `$query_df()`).
4. Call `$disconnect()` when finished.

Public fields

`json_client` An `AstraeaClient` instance for JSON/TCP communication. Always available.

`arrow_client` An `ArrowClient` instance for Arrow Flight communication, or `NULL` if the **arrow** package is not installed or initialization failed.

`use_arrow` Logical. Whether Arrow Flight transport is active and should be used for queries.

Methods

Public methods:

- `UnifiedClient$new()`
- `UnifiedClient$connect()`
- `UnifiedClient$disconnect()`
- `UnifiedClient$create_node()`
- `UnifiedClient$get_node()`
- `UnifiedClient$update_node()`
- `UnifiedClient$delete_node()`
- `UnifiedClient$create_edge()`
- `UnifiedClient$get_edge()`
- `UnifiedClient$update_edge()`
- `UnifiedClient$delete_edge()`
- `UnifiedClient$neighbors()`
- `UnifiedClient$bfs()`
- `UnifiedClient$shortest_path()`
- `UnifiedClient$neighbors_at()`
- `UnifiedClient$bfs_at()`
- `UnifiedClient$shortest_path_at()`
- `UnifiedClient$vector_search()`
- `UnifiedClient$hybrid_search()`
- `UnifiedClient$semantic_neighbors()`
- `UnifiedClient$semantic_walk()`
- `UnifiedClient$extract_subgraph()`
- `UnifiedClient$graph_rag()`
- `UnifiedClient$dfs()`
- `UnifiedClient$dfs_at()`
- `UnifiedClient$find_by_label()`
- `UnifiedClient$find_edge_by_type()`
- `UnifiedClient$delete_by_label()`
- `UnifiedClient$get_subgraph()`
- `UnifiedClient$graph_stats()`
- `UnifiedClient$run_pagerank()`
- `UnifiedClient$run_louvain()`
- `UnifiedClient$run_connected_components()`

- UnifiedClient$run_degree_centrality()
- UnifiedClient$run_betweenness_centrality()
- UnifiedClient$create_nodes()
- UnifiedClient$create_edges()
- UnifiedClient$delete_nodes()
- UnifiedClient$delete_edges()
- UnifiedClient$import_nodes_df()
- UnifiedClient$import_edges_df()
- UnifiedClient$export_nodes_df()
- UnifiedClient$export_bfs_df()
- UnifiedClient$results_to_dataframe()
- UnifiedClient$nodes_to_dataframe()
- UnifiedClient$ping()
- UnifiedClient$query()
- UnifiedClient$query_df()
- UnifiedClient$is_arrow_enabled()
- UnifiedClient$print()

Method `new()`**:** Create a new UnifiedClient.

Always creates a JSON/TCP client. If the **arrow** package is available, also creates an Arrow Flight client. Arrow initialization failure is non-fatal and results in a message.

Usage:

```
UnifiedClient$new(
  host = "127.0.0.1",
  port = 7687L,
  flight_uri = NULL,
  auth_token = NULL
)
```

Arguments:

`host` Character. Server hostname for JSON/TCP transport. Defaults to `"127.0.0.1"`.

`port` Integer. Server port for JSON/TCP transport. Defaults to `7687L`.

`flight_uri` Character or `NULL`. Arrow Flight gRPC URI. If `NULL`, constructed automatically as `sprintf("grpc://%s:7689", host)`.

`auth_token` Character or `NULL`. Authentication token for the JSON/TCP client.

Returns: A new `UnifiedClient` object.

Method `connect()`**:** Connect to AstraeaDB server(s).

Connects the JSON/TCP client. If Arrow Flight is enabled, also attempts to connect the Arrow client. Arrow connection failure is non-fatal and results in a fallback to JSON/TCP with a message.

Usage:

```
UnifiedClient$connect()
```

Returns: Invisibly returns `self` for method chaining.

Method `disconnect()`**:** Disconnect all client connections.
Closes the JSON/TCP connection and, if active, the Arrow Flight connection.

Usage:

```
UnifiedClient$disconnect()
```

Returns: Invisibly returns `self` for method chaining.

Method `create_node()`**:** Create a node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$create_node(labels, properties, embedding = NULL)
```

Arguments:

`labels` List of character labels for the node.

`properties` Named list of node properties.

`embedding` Optional numeric vector for vector similarity search.

Returns: Integer. The ID of the newly created node.

Method `get_node()`**:** Get a node by ID. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$get_node(node_id)
```

Arguments:

`node_id` Integer. The node ID.

Returns: A list containing node data (labels, properties, etc.).

Method `update_node()`**:** Update a node's properties. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$update_node(node_id, properties)
```

Arguments:

`node_id` Integer. The node ID.

`properties` Named list of properties to merge.

Returns: The server response data.

Method `delete_node()`**:** Delete a node and its connected edges. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$delete_node(node_id)
```

Arguments:

`node_id` Integer. The node ID.

Returns: The server response data.

Method `create_edge()`**:** Create an edge. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$create_edge(
  source,
  target,
  edge_type,
  properties = list(),
  weight = 1,
  valid_from = NULL,
  valid_to = NULL
)
```

Arguments:

`source` Integer. Source node ID.

`target` Integer. Target node ID.

`edge_type` Character. The edge type label.

`properties` Named list of edge properties.

`weight` Numeric. Edge weight. Defaults to `1.0`.

`valid_from` Numeric or `NULL`. Temporal start (ms epoch).

`valid_to` Numeric or `NULL`. Temporal end (ms epoch).

Returns: Integer. The ID of the newly created edge.

Method `get_edge()`**:** Get an edge by ID. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$get_edge(edge_id)
```

Arguments:

`edge_id` Integer. The edge ID.

Returns: A list containing edge data.

Method `update_edge()`**:** Update an edge's properties. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$update_edge(edge_id, properties)
```

Arguments:

`edge_id` Integer. The edge ID.

`properties` Named list of properties to merge.

Returns: The server response data.

Method `delete_edge()`**:** Delete an edge. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$delete_edge(edge_id)
```

Arguments:

`edge_id` Integer. The edge ID.

Returns: The server response data.

Method `neighbors()`**:** Get neighbors of a node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$neighbors(node_id, direction = "outgoing", edge_type = NULL)
```

Arguments:

`node_id` Integer. The node ID.

`direction` Character. One of `"outgoing"`, `"incoming"`, or `"both"`. Defaults to `"outgoing"`.

`edge_type` Character or `NULL`. Filter by edge type.

Returns: A list of neighbor entries.

Method `bfs()`**:** Breadth-first search from a start node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$bfs(start, max_depth = 3L)
```

Arguments:

`start` Integer. Starting node ID.

`max_depth` Integer. Maximum traversal depth. Defaults to `3L`.

Returns: A list of entries with `node_id` and `depth`.

Method `shortest_path()`**:** Find the shortest path between two nodes. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$shortest_path(from_node, to_node, weighted = FALSE)
```

Arguments:

`from_node` Integer. Source node ID.

`to_node` Integer. Target node ID.

`weighted` Logical. Use edge weights? Defaults to `FALSE`.

Returns: A list with path information.

Method `neighbors_at()`**:** Get neighbors at a specific point in time. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$neighbors_at(
  node_id,
  direction = "outgoing",
  timestamp,
  edge_type = NULL
)
```

Arguments:

`node_id` Integer. The node ID.

`direction` Character. Direction filter. Defaults to `"outgoing"`.

`timestamp` Numeric. Point-in-time timestamp (ms epoch).

`edge_type` Character or `NULL`. Filter by edge type.

Returns: A list of neighbor entries valid at the given timestamp.

Method `bfs_at()`**:** BFS traversal at a specific point in time. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$bfs_at(start, max_depth = 3L, timestamp)
```

Arguments:

`start` Integer. Starting node ID.

`max_depth` Integer. Maximum depth. Defaults to `3L`.

`timestamp` Numeric. Point-in-time timestamp (ms epoch).

Returns: A list of entries with `node_id` and `depth`.

Method `shortest_path_at()`**:** Find shortest path at a specific point in time. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$shortest_path_at(from_node, to_node, timestamp, weighted = FALSE)
```

Arguments:

`from_node` Integer. Source node ID.

`to_node` Integer. Target node ID.

`timestamp` Numeric. Point-in-time timestamp (ms epoch).

`weighted` Logical. Use edge weights? Defaults to `FALSE`.

Returns: A list with path information.

Method `vector_search()`**:** k-nearest neighbor vector search. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$vector_search(query_vector, k = 10L)
```

Arguments:

`query_vector` Numeric vector. The query embedding.

`k` Integer. Number of results. Defaults to `10L`.

Returns: A list of search results with `node_id` and `similarity`.

Method `hybrid_search()`**:** Hybrid graph-vector search. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$hybrid_search(
  anchor,
  query_vector,
  max_hops = 3L,
  k = 10L,
  alpha = 0.5
)
```

Arguments:

`anchor` Integer. Anchor node ID for graph proximity.

`query_vector` Numeric vector. Query embedding for similarity.

`max_hops` Integer. Maximum graph hops. Defaults to `3L`.

`k` Integer. Number of results. Defaults to `10L`.

`alpha` Numeric. Balance between graph (0) and vector (1). Defaults to `0.5`.

Returns: A list of search results.

Method `semantic_neighbors()`**:** Get neighbors ranked by semantic similarity. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$semantic_neighbors(
  node_id,
  concept,
  direction = "outgoing",
  k = 10L
)
```

Arguments:

`node_id` Integer. The node ID.

`concept` Numeric vector. Concept embedding for ranking.

`direction` Character. Direction filter. Defaults to `"outgoing"`.

`k` Integer. Number of results. Defaults to `10L`.

Returns: A list of neighbor entries with similarity scores.

Method `semantic_walk()`**:** Greedy semantic walk following edges most similar to a concept. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$semantic_walk(start, concept, max_hops = 3L)
```

Arguments:

`start` Integer. Starting node ID.

`concept` Numeric vector. Concept embedding for guidance.

`max_hops` Integer. Maximum walk length. Defaults to `3L`.

Returns: A list representing the walk path.

Method `extract_subgraph()`**:** Extract a subgraph centered on a node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$extract_subgraph(
  center,
  hops = 2L,
  max_nodes = 50L,
  format = "structured"
)
```

Arguments:

`center` Integer. Center node ID.

`hops` Integer. Radius in hops. Defaults to `2L`.

`max_nodes` Integer. Maximum nodes to return. Defaults to `50L`.

`format` Character. Output format: `"structured"`, `"prose"`, `"triples"`, or `"json"`. Defaults to `"structured"`.

Returns: A list with subgraph data and linearized text.

Method `graph_rag()`**:** Execute a GraphRAG query. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$graph_rag(
  question,
  anchor = NULL,
  question_embedding = NULL,
  hops = 2L,
  max_nodes = 50L,
  format = "structured"
)
```

Arguments:

`question` Character. The question to answer.

`anchor` Integer or `NULL`. Anchor node ID.

`question_embedding` Numeric vector or `NULL`. Embedding of the question.

`hops` Integer. Subgraph radius. Defaults to `2L`.

`max_nodes` Integer. Maximum subgraph nodes. Defaults to `50L`.

`format` Character. Linearization format. Defaults to `"structured"`.

Returns: A list with the LLM-generated answer and context.

Method `dfs()`**:** Depth-first search from a node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$dfs(start, max_depth = 3L)
```

Arguments:

`start` Integer. The starting node ID.

`max_depth` Integer. Maximum traversal depth. Default `3L`.

Returns: A list of node IDs in depth-first visitation order.

Method `dfs_at()`**:** Depth-first search as of a point in time. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$dfs_at(start, max_depth = 3L, timestamp)
```

Arguments:

`start` Integer. The starting node ID.

`max_depth` Integer. Maximum traversal depth. Default `3L`.

`timestamp` Numeric. Point in time (ms since epoch).

Returns: A list of node IDs in depth-first visitation order.

Method `find_by_label()`**:** Find nodes by label. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$find_by_label(label)
```

Arguments:

`label` Character. The node label to match.

Returns: A list of matching node IDs.

Method `find_edge_by_type()`**:** Find edges by edge type. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$find_edge_by_type(edge_type)
```

Arguments:

`edge_type` Character. The edge type to match.

Returns: A list of entries with `edge_id`, `source`, `target`.

Method `delete_by_label()`**:** Delete all nodes with a label. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$delete_by_label(label)
```

Arguments:

`label` Character. The node label to match.

Returns: Integer scalar: the number of nodes deleted.

Method `get_subgraph()`**:** Retrieve the raw subgraph around a node. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$get_subgraph(center, hops = 3L, max_nodes = 50L)
```

Arguments:

`center` Integer. The center node ID.

`hops` Integer. Neighborhood radius. Default `3L`.

`max_nodes` Integer. Maximum nodes to return. Default `50L`.

Returns: A list with `nodes` and `edges`.

Method `graph_stats()`**:** Retrieve graph-wide statistics. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$graph_stats()
```

Returns: A named list of statistics.

Method `run_pagerank()`**:** Run PageRank. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$run_pagerank(
  nodes = NULL,
  damping = 0.85,
  max_iterations = 100L,
  tolerance = 1e-06
)
```

Arguments:

`nodes` Optional integer vector restricting the computation.

`damping` Numeric. Damping factor. Default `0.85`.

`max_iterations` Integer. Maximum iterations. Default `100L`.

`tolerance` Numeric. Convergence tolerance. Default `1e-6`.

Returns: A named list mapping node ID to PageRank score.

Method `run_louvain()`**:** Run Louvain community detection. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$run_louvain(nodes = NULL)
```

Arguments:

`nodes` Optional integer vector restricting the computation.

Returns: A list with `communities` and `num_communities`.

Method `run_connected_components()`**:** Find connected components. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$run_connected_components(nodes = NULL, strong = FALSE)
```

Arguments:

`nodes` Optional integer vector restricting the computation.

`strong` Logical. Strongly (`TRUE`) or weakly (`FALSE`, default) connected.

Returns: A list with `components` and `count`.

Method `run_degree_centrality()`**:** Compute degree centrality. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$run_degree_centrality(nodes = NULL, direction = "outgoing")
```

Arguments:

`nodes` Optional integer vector restricting the computation.

`direction` Character. `"outgoing"` (default), `"incoming"`, or `"both"`.

Returns: A named list mapping node ID to degree-centrality score.

Method `run_betweenness_centrality()`**:** Compute betweenness centrality. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$run_betweenness_centrality(nodes = NULL)
```

Arguments:

`nodes` Optional integer vector restricting the computation.

Returns: A named list mapping node ID to betweenness-centrality score.

Method `create_nodes()`**:** Create multiple nodes. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$create_nodes(nodes_list)
```

Arguments:

`nodes_list` A list of lists, each with `labels`, `properties`, and optionally `embedding`.

Returns: An integer vector of created node IDs.

Method `create_edges()`**:** Create multiple edges. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$create_edges(edges_list)
```

Arguments:

`edges_list` A list of lists, each with `source`, `target`, `edge_type`, and optionally `properties`, `weight`, `valid_from`, `valid_to`.

Returns: An integer vector of created edge IDs.

Method `delete_nodes()`**:** Delete multiple nodes. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$delete_nodes(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs to delete.

Returns: Integer. Number of successfully deleted nodes.

Method `delete_edges()`**:** Delete multiple edges. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$delete_edges(edge_ids)
```

Arguments:

`edge_ids` Integer vector of edge IDs to delete.

Returns: Integer. Number of successfully deleted edges.

Method `import_nodes_df()`**:** Import nodes from a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$import_nodes_df(
  df,
  label_col = "label",
  id_col = NULL,
  embedding_cols = NULL
)
```

Arguments:

`df` A data.frame containing node data.

`label_col` Character. Column name for labels. Defaults to `"label"`.

`id_col` Character or `NULL`. Column for external IDs.

`embedding_cols` Character vector or `NULL`. Columns for embedding values.

Returns: An integer vector of created node IDs.

Method `import_edges_df()`**:** Import edges from a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$import_edges_df(
  df,
  source_col = "source",
  target_col = "target",
  type_col = "type",
  weight_col = NULL,
  valid_from_col = NULL,
  valid_to_col = NULL
)
```

Arguments:

`df` A data.frame containing edge data.

`source_col` Character. Column for source node IDs. Defaults to `"source"`.

`target_col` Character. Column for target node IDs. Defaults to `"target"`.

`type_col` Character. Column for edge types. Defaults to `"type"`.

`weight_col` Character or `NULL`. Column for weights.

`valid_from_col` Character or `NULL`. Column for temporal start.

`valid_to_col` Character or `NULL`. Column for temporal end.

Returns: An integer vector of created edge IDs.

Method `export_nodes_df()`**:** Export nodes to a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$export_nodes_df(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs.

Returns: A data.frame with columns for node ID, labels, and properties.

Method `export_bfs_df()`**:** Run BFS and return results as a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$export_bfs_df(start, max_depth = 3L)
```

Arguments:

`start` Integer. Starting node ID.

`max_depth` Integer. Maximum depth. Defaults to `3L`.

Returns: A data.frame with BFS results including node details.

Method `results_to_dataframe()`**:** Convert search results to a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$results_to_dataframe(results)
```

Arguments:

`results` A list of result entries.

Returns: A data.frame with one row per result.

Method `nodes_to_dataframe()`**:** Fetch nodes by ID and return as a data.frame. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$nodes_to_dataframe(node_ids)
```

Arguments:

`node_ids` Integer vector of node IDs.

Returns: A data.frame with columns for ID, labels, and properties.

Method `ping()`**:** Health check. Delegates to the JSON/TCP client.

Usage:

```
UnifiedClient$ping()
```

Returns: A list with server status information.

Method `query()`**:** Execute a GQL query.

Uses Arrow Flight transport if available for higher performance. Falls back to JSON/TCP transport otherwise.

Usage:

```
UnifiedClient$query(gql)
```

Arguments:

`gql` Character. A GQL query string.

Returns: Query results. An Arrow Table when using Arrow transport, or a list when using JSON/TCP.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- UnifiedClient$new()
  client$connect()
  result <- client$query("MATCH (n:Person) RETURN n.name")
  client$disconnect()
}
}
```

Method `query_df()`**:** Execute a GQL query and return a data.frame.

Uses Arrow Flight transport if available. Falls back to JSON/TCP, converting the result to a data.frame.

Usage:

```
UnifiedClient$query_df(gql)
```

Arguments:

`gql` Character. A GQL query string.

Returns: A data.frame containing the query results.

Examples:

```
\donttest{
if (astraea_server_available()) {
  client <- UnifiedClient$new()
  client$connect()
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")
  client$disconnect()
}
}
```

Method `is_arrow_enabled()`**:** Check whether Arrow Flight transport is currently active.

Usage:

```
UnifiedClient$is_arrow_enabled()
```

Returns: Logical. `TRUE` if Arrow Flight is enabled and connected, `FALSE` otherwise.

Method `print()`**:** Print a summary of the UnifiedClient.

Usage:

```
UnifiedClient$print(...)
```

Arguments:

`...` Ignored. Present for compatibility with the generic.

Returns: Invisibly returns `self`.

Examples

```
if (requireNamespace("arrow", quietly = TRUE) &&
    astraea_server_available(port = 7689L)) {
  # Basic usage
  client <- UnifiedClient$new()
  client$connect()

  # CRUD operations use JSON/TCP
  node_id <- client$create_node(list("Person"), list(name = "Alice", age = 30))
  node <- client$get_node(node_id)

  # Queries use Arrow Flight if available, otherwise JSON/TCP
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")

  # Check transport status
  client$is_arrow_enabled()

  client$disconnect()

  # With authentication
  client <- UnifiedClient$new(auth_token = "my-secret-token")
  client$connect()

  # Custom Flight URI
  client <- UnifiedClient$new(
    host = "db.example.com",
    port = 7687L,
```

```
    flight_uri = "grpc://db.example.com:7689"
  )
  client$connect()
}



## ------------------------------------------------
## Method `UnifiedClient$query`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- UnifiedClient$new()
  client$connect()
  result <- client$query("MATCH (n:Person) RETURN n.name")
  client$disconnect()
}


## ------------------------------------------------
## Method `UnifiedClient$query_df`
## ------------------------------------------------


if (astraea_server_available()) {
  client <- UnifiedClient$new()
  client$connect()
  df <- client$query_df("MATCH (n:Person) RETURN n.name, n.age")
  client$disconnect()
}
```

Index