

# GraphqlGraphs

`class GraphqlGraphs`

Bases: [object](#)

A class for accessing graphs hosted in a Raphtory GraphQL server and running global search for graph documents

Methods:

`get(name)`

Return the VectorisedGraph with name name or None if it doesn't exist

---

`search_graph_documents(query, limit, window)`

Return the top documents with the smallest cosine distance to query

---

`search_graph_documents_with_scores(query, ...)`

Same as `search_graph_documents` but it also returns the scores alongside the documents

---

`get(name)`

Return the VectorisedGraph with name name or None if it doesn't exist

**Parameters:**

name ([str](#)) – the name of the graph

**Returns:**

the graph if it exists

**Return type:**

[Optional\[VectorisedGraph\]](#)

`search_graph_documents(query, limit, window)`

Return the top documents with the smallest cosine distance to query

**Parameters:**

- query ([str](#)) – the text or the embedding to score against
- limit ([int](#)) – the maximum number of documents to return

- `window` (`Tuple[TimeInput, TimeInput], optional`) – the window where documents need to belong to in order to be considered

**Returns:**

A list of documents

**Return type:**

`list[Document]`

`search_graph_documents_with_scores(query, limit, window)`

Same as `search_graph_documents` but it also returns the scores alongside the documents

**Parameters:**

- `query` (`str`) – the text or the embedding to score against
- `limit` (`int`) – the maximum number of documents to return
- `window` (`Tuple[TimeInput, TimeInput], optional`) – the window where documents need to belong to in order to be considered

**Returns:**

A list of documents and their scores

**Return type:**

`list[Tuple[Document, float]]`

# GraphServer

```
class GraphServer(work_dir, cache_capacity=None,
cache_tti_seconds=None, log_level=None, tracing=None,
otlp_agent_host=None, otlp_agent_port=None,
otlp_tracing_service_name=None, auth_public_key=None,
auth_enabled_for_reads=None, config_path=None)
```

Bases: `object`

A class for defining and running a Raphtory GraphQL server

**Parameters:**

- `work_dir` (`str` | `PathLike`) – the working directory for the server
- `cache_capacity` (`int, optional`) – the maximum number of graphs to keep in memory at once
- `cache_tti_seconds` (`int, optional`) – the inactive time in seconds after which a graph is evicted from the cache
- `log_level` (`str, optional`) – the log level for the server
- `tracing` (`bool, optional`) – whether tracing should be enabled
- `otlp_agent_host` (`str, optional`) – OTLP agent host for tracing

- `otlp_agent_port` (`str`, *optional*) – OTLP agent port for tracing
- `otlp_tracing_service_name` (`str`, *optional*) – The OTLP tracing service name
- `config_path` (`str` | `PathLike`, *optional*) – Path to the config file

Methods:

`run`([port, timeout\_ms])

Run the server until completion.

`set_embeddings`(cache[, embedding, graphs, ...])

Setup the server to vectorise graphs with a default template.

`start`([port, timeout\_ms])

Start the server and return a handle to it.

`turn_off_index`()

Turn off index for all graphs

`with_global_search_function`(name, input, ...)

Register a function in the GraphQL schema for document search among all the graphs.

`with_vectorised_graphs`(graph\_names[, ...])

Vectorise a subset of the graphs of the server.

`run (port=1736, timeout_ms=180000)`

Run the server until completion.

Parameters:

- `port` (`int`) – The port to use. Defaults to 1736.
- `timeout_ms` (`int`) – Timeout for waiting for the server to start. Defaults to 180000.

Return type:

`None`

`set_embeddings(cache, embedding=None, graphs=..., nodes=..., edges=...)`

Setup the server to vectorise graphs with a default template.

**Parameters:**

- cache ([str](#)) – the directory to use as cache for the embeddings.
- embedding ([Callable](#), *optional*) – the embedding function to translate documents to embeddings.
- graphs ([bool](#) | [str](#)) – if graphs have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- nodes ([bool](#) | [str](#)) – if nodes have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- edges ([bool](#) | [str](#)) – if edges have to be embedded or not or the custom template to use if a str is provided. Defaults to True.

**Returns:**

A new server object with embeddings setup.

**Return type:**

[GraphServer](#)

`start(port=1736, timeout_ms=5000)`

Start the server and return a handle to it.

**Parameters:**

- port ([int](#)) – the port to use. Defaults to 1736.
- timeout\_ms ([int](#)) – wait for server to be online. Defaults to 5000. The server is stopped if not online within timeout\_ms but manages to come online as soon as timeout\_ms finishes!

**Returns:**

The running server

**Return type:**

[RunningGraphServer](#)

`turn_off_index()`

Turn off index for all graphs

**Returns:**

The server with indexing disabled

**Return type:**

[GraphServer](#)

`with_global_search_function(name, input, function)`

Register a function in the GraphQL schema for document search among all the graphs.

The function needs to take a `GraphQLGraphs` object as the first argument followed by a pre-defined set of keyword arguments. Supported types are `str`, `int`, and `float`. They

have to be specified using the input parameter as a dict where the keys are the names of the parameters and the values are the types, expressed as strings.

#### Parameters:

- name ([str](#)) – the name of the function in the GraphQL schema.
- input ([dict\[str, str\]](#)) – the keyword arguments expected by the function.
- function ([Callable](#)) – the function to run.

#### Returns:

A new server object with the function registered

#### Return type:

[GraphServer](#)

`with_vectorised_graphs(graph_names, graphs=..., nodes=..., edges=...)`

Vectorise a subset of the graphs of the server.

#### Parameters:

- graph\_names ([list\[str\]](#)) – the names of the graphs to vectorise. All by default.
- graphs ([bool](#) | [str](#)) – if graphs have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- nodes ([bool](#) | [str](#)) – if nodes have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- edges ([bool](#) | [str](#)) – if edges have to be embedded or not or the custom template to use if a str is provided. Defaults to True.

#### Returns:

A new server object containing the vectorised graphs.

#### Return type:

[GraphServer](#)

# RunningGraphServer

`class RunningGraphServer`

Bases: [object](#)

A Raphtory server handler that also enables querying the server

#### Methods:

[get\\_client\(\)](#)

Get the client for the server

---

[\*\*stop\(\)\*\*](#) Stop the server and wait for it to finish

[\*\*get\\_client\(\)\*\*](#)

Get the client for the server

**Returns:**

the client

**Return type:**

[RaphtoryClient](#)

[\*\*stop\(\)\*\*](#)

Stop the server and wait for it to finish

**Return type:**

[None](#)

# RaphtoryClient

`class RaphtoryClient(url, token=None)`

Bases: [object](#)

A client for handling GraphQL operations in the context of Raphtory.

**Parameters:**

`url (str)` – the URL of the Raphtory GraphQL server

**Methods:**

[\*\*copy\\_graph\(path, new\\_path\)\*\*](#)

Copy graph from a path path on the server to a new\_path on the server

[\*\*delete\\_graph\(path\)\*\*](#)

Delete graph from a path path on the server

[\*\*is\\_server\\_online\(\)\*\*](#)

Check if the server is online.

[\*\*move\\_graph\(path, new\\_path\)\*\*](#)

Move graph from a path path on the server to a new\_path on the server

---

<code><a href="#">new_graph</a>(path, graph_type)</code>	Create a new empty Graph on the server at path
--	--

---

<code><a href="#">query</a>(query[, variables])</code>	Make a GraphQL query against the server.
--	--

---

<code><a href="#">receive_graph</a>(path)</code>	Receive graph from a path path on the server
--	--

---

<code><a href="#">remote_graph</a>(path)</code>	Get a RemoteGraph reference to a graph on the server at path
---	--

---

<code><a href="#">send_graph</a>(path, graph[, overwrite])</code>	Send a graph to the server
---	----------------------------

---

<code><a href="#">upload_graph</a>(path, file_path[, overwrite])</code>	Upload graph file from a path file_path on the client
---	---

---

### `copy\_graph\(path, new\_path\)`

Copy graph from a path path on the server to a new\_path on the server

**Parameters:**

- path ([str](#)) – the path of the graph to be copied
- new\_path ([str](#)) – the new path of the copied graph

**Return type:**

[None](#)

### `delete\_graph\(path\)`

Delete graph from a path path on the server

**Parameters:**

path ([str](#)) – the path of the graph to be deleted

**Return type:**

[None](#)

### `is\_server\_online\(\)`

Check if the server is online.

**Returns:**

Returns true if server is online otherwise false.

**Return type:**

`bool`

`move_graph(path, new_path)`

Move graph from a path path on the server to a new\_path on the server

**Parameters:**

- `path (str)` – the path of the graph to be moved
- `new_path (str)` – the new path of the moved graph

**Return type:**

`None`

`new_graph(path, graph_type)`

Create a new empty Graph on the server at path

**Parameters:**

- `path (str)` – the path of the graph to be created
- `graph_type (Literal["EVENT", "PERSISTENT"])` – the type of graph that should be created - this can be EVENT or PERSISTENT

**Return type:**

`None`

`query(query, variables=None)`

Make a GraphQL query against the server.

**Parameters:**

- `query (str)` – the query to make.
- `variables (dict[str, Any], optional)` – a dict of variables present on the query and their values.

**Returns:**

The data field from the GraphQL response.

**Return type:**

`dict[str, Any]`

`receive_graph(path)`

Receive graph from a path path on the server

**Note**

This downloads a copy of the graph. Modifications are not persistet to the server.

**Parameters:**

path ([str](#)) – the path of the graph to be received

**Returns:**

A copy of the graph

**Return type:**

[Union\[Graph, PersistentGraph\]](#)

`remote_graph(path)`

Get a RemoteGraph reference to a graph on the server at path

**Parameters:**

path ([str](#)) – the path of the graph to be created

**Returns:**

the remote graph reference

**Return type:**

[RemoteGraph](#)

`send_graph(path, graph, overwrite=False)`

Send a graph to the server

**Parameters:**

- path ([str](#)) – the path of the graph
- graph ([Graph](#) | [PersistentGraph](#)) – the graph to send
- overwrite ([bool](#)) – overwrite existing graph. Defaults to False.

**Returns:**

The data field from the graphQL response after executing the mutation.

**Return type:**

[dict\[str, Any\]](#)

`upload_graph(path, file_path, overwrite=False)`

Upload graph file from a path file\_path on the client

**Parameters:**

- path ([str](#)) – the name of the graph
- file\_path ([str](#)) – the path of the graph on the client
- overwrite ([bool](#)) – overwrite existing graph. Defaults to False.

**Returns:**

The data field from the graphQL response after executing the mutation.

**Return type:**

[dict\[str, Any\]](#)

# RemoteGraph

`class RemoteGraph`

Bases: [object](#)

Methods:

[`add\_constant\_properties\(properties\)`](#)

Adds constant properties to the remote graph.

[`add\_edge\(timestamp, src, dst\[, properties, ...\]\)`](#)

Adds a new edge with the given source and destination nodes and properties to the remote graph.

[`add\_edges\(updates\)`](#)

Batch add edge updates to the remote graph

[`add\_node\(timestamp, id\[, properties, node\_type\]\)`](#)

Adds a new node with the given id and properties to the remote graph.

[`add\_nodes\(updates\)`](#)

Batch add node updates to the remote graph

[`add\_property\(timestamp, properties\)`](#)

Adds properties to the remote graph.

[`create\_node\(timestamp, id\[, properties, ...\]\)`](#)

Create a new node with the given id and properties to the remote graph and fail if the node already exists.

---

<code><a href="#">delete_edge</a>(timestamp, src, dst[, layer])</code>	Deletes an edge in the remote graph, given the timestamp, src and dst nodes and layer (optional)
--	--

---

<code><a href="#">edge</a>(src, dst)</code>	Gets a remote edge with the specified source and destination nodes
---	--

---

<code><a href="#">node</a>(id)</code>	Gets a remote node with the specified id
---------------------------------------	--

---

<code><a href="#">update_constant_properties</a>(properties)</code>	Updates constant properties on the remote graph.
---	--

### `add\_constant\_properties(properties)`

Adds constant properties to the remote graph.

#### Parameters:

`properties` ([dict](#)) – The constant properties of the graph.

#### Return type:

[None](#)

### `add\_edge(timestamp, src, dst, properties=None, layer=None)`

Adds a new edge with the given source and destination nodes and properties to the remote graph.

#### Parameters:

- `timestamp` ([int](#) | [str](#) | [datetime](#)) – The timestamp of the edge.
- `src` ([str](#) | [int](#)) – The id of the source node.
- `dst` ([str](#) | [int](#)) – The id of the destination node.
- `properties` ([dict](#), *optional*) – The properties of the edge, as a dict of string and properties.
- `layer` ([str](#), *optional*) – The layer of the edge.

#### Returns:

the remote edge

#### Return type:

[RemoteEdge](#)

## `add_edges(updates)`

Batch add edge updates to the remote graph

### Parameters:

`updates (List[RemoteEdgeAddition])` – The list of updates you want to apply to the remote graph

### Return type:

`None`

## `add_node(timestamp, id, properties=None, node_type=None)`

Adds a new node with the given id and properties to the remote graph.

### Parameters:

- `timestamp (int | str | datetime)` – The timestamp of the node.
- `id (str | int)` – The id of the node.
- `properties (dict, optional)` – The properties of the node.
- `node_type (str, optional)` – The optional string which will be used as a node type

### Returns:

the new remote node

### Return type:

`RemoteNode`

## `add_nodes(updates)`

Batch add node updates to the remote graph

### Parameters:

`updates (List[RemoteNodeAddition])` – The list of updates you want to apply to the remote graph

### Return type:

`None`

## `add_property(timestamp, properties)`

Adds properties to the remote graph.

### Parameters:

- `timestamp (int | str | datetime)` – The timestamp of the temporal property.
- `properties (dict)` – The temporal properties of the graph.

### Return type:

`None`

## `create_node(timestamp, id, properties=None, node_type=None)`

Create a new node with the given id and properties to the remote graph and fail if the node already exists.

**Parameters:**

- timestamp ([int](#) | [str](#) | [datetime](#)) – The timestamp of the node.
- id ([str](#) | [int](#)) – The id of the node.
- properties ([dict](#), [optional](#)) – The properties of the node.
- node\_type ([str](#), [optional](#)) – The optional string which will be used as a node type

**Returns:**

the new remote node

**Return type:**

[RemoteNode](#)

**delete\_edge(timestamp, src, dst, layer=None)**

Deletes an edge in the remote graph, given the timestamp, src and dst nodes and layer (optional)

**Parameters:**

- timestamp ([int](#)) – The timestamp of the edge.
- src ([str](#) | [int](#)) – The id of the source node.
- dst ([str](#) | [int](#)) – The id of the destination node.
- layer ([str](#), [optional](#)) – The layer of the edge.

**Returns:**

the remote edge

**Return type:**

[RemoteEdge](#)

**edge(src, dst)**

Gets a remote edge with the specified source and destination nodes

**Parameters:**

- src ([str](#) | [int](#)) – the source node id
- dst ([str](#) | [int](#)) – the destination node id

**Returns:**

the remote edge reference

**Return type:**

[RemoteEdge](#)

**node(id)**

Gets a remote node with the specified id

**Parameters:**

`id (str | int)` – the node id

**Returns:**

the remote node reference

**Return type:**

[RemoteNode](#)

`update_constant_properties(properties)`

Updates constant properties on the remote graph.

**Parameters:**

`properties (dict)` – The constant properties of the graph.

**Return type:**

[None](#)

# RemoteEdge

`class RemoteEdge`

Bases: [object](#)

A remote edge reference

Returned by [RemoteGraph.edge\(\)](#), [RemoteGraph.add\\_edge\(\)](#), and  
[RemoteGraph.delete\\_edge\(\)](#).

**Methods:**

`add_constant_properties(properties[, layer])`

Add constant properties to the edge within the remote graph.

---

`add_updates(t[, properties, layer])`

Add updates to an edge in the remote graph at a specified time.

---

`delete(t[, layer])`

Mark the edge as deleted at the specified time.

---

<code>update_constant_properties(properties[, layer])</code>	Update constant properties of an edge in the remote graph overwriting existing values.
--	--

---

### `add_constant_properties(properties, layer=None)`

Add constant properties to the edge within the remote graph. This function is used to add properties to an edge that remain constant and do not change over time. These properties are fundamental attributes of the edge.

#### Parameters:

- `properties (Dict[str, Prop])` – A dictionary of properties to be added to the edge.
- `layer (str, optional)` – The layer you want these properties to be added on to.

#### Return type:

`None`

### `add_updates(t, properties=None, layer=None)`

Add updates to an edge in the remote graph at a specified time.

This function allows for the addition of property updates to an edge within the graph. The updates are time-stamped, meaning they are applied at the specified time.

#### Parameters:

- `t (int | str | datetime)` – The timestamp at which the updates should be applied.
- `properties (Optional[Dict[str, Prop]])` – A dictionary of properties to update.
- `layer (str, optional)` – The layer you want the updates to be applied.

#### Return type:

`None`

### `delete(t, layer=None)`

Mark the edge as deleted at the specified time.

#### Parameters:

- `t (int | str | datetime)` – The timestamp at which the deletion should be applied.
- `layer (str, optional)` – The layer you want the deletion applied to.

#### Return type:

`None`

### `update_constant_properties(properties, layer=None)`

Update constant properties of an edge in the remote graph overwriting existing values. This function is used to add properties to an edge that remains constant and does not change over time. These properties are fundamental attributes of the edge.

**Parameters:**

- `properties` (`Dict[str, Prop]`) – A dictionary of properties to be added to the edge.
- `layer` (`str, optional`) – The layer you want these properties to be added on to.

**Return type:**

`None`

# RemoteNode

`class RemoteNode`

Bases: `object`

**Methods:**

`add_constant_properties(properties)`

Add constant properties to a node in the remote graph.

`add_updates(t[, properties])`

Add updates to a node in the remote graph at a specified time.

`set_node_type(new_type)`

Set the type on the node.

`update_constant_properties(properties)`

Update constant properties of a node in the remote graph overwriting existing values.

**`add_constant_properties(properties)`**

Add constant properties to a node in the remote graph. This function is used to add properties to a node that remain constant and does not change over time. These properties are fundamental attributes of the node.

**Parameters:**

`properties` (`Dict[str, Prop]`) – A dictionary of properties to be added to the node.

**Return type:**

`None`

### `add_updates(t, properties=None)`

Add updates to a node in the remote graph at a specified time. This function allows for the addition of property updates to a node within the graph. The updates are time-stamped, meaning they are applied at the specified time.

#### **Parameters:**

- `t` (`int` | `str` | `datetime`) – The timestamp at which the updates should be applied.
- `properties` (`Dict[str, Prop]`, *optional*) – A dictionary of properties to update.

#### **Return type:**

`None`

### `set_node_type(new_type)`

Set the type on the node. This only works if the type has not been previously set, otherwise will throw an error

#### **Parameters:**

`new_type` (`str`) – The new type to be set

#### **Return type:**

`None`

### `update_constant_properties(properties)`

Update constant properties of a node in the remote graph overwriting existing values. This function is used to add properties to a node that remain constant and do not change over time. These properties are fundamental attributes of the node.

#### **Parameters:**

`properties` (`Dict[str, Prop]`) – A dictionary of properties to be added to the node.

#### **Return type:**

`None`

## RemoteNodeAddition

### `class RemoteNodeAddition(name, node_type=None, constant_properties=None, updates=None)`

Bases: `object`

Node addition update

#### **Parameters:**

- `name` (`GID`) – the id of the node
- `node_type` (`str`, *optional*) – the node type
- `constant_properties` (`PropInput`, *optional*) – the constant properties

- `updates` – (`list[RemoteUpdate]`, optional): the temporal updates

# RemoteUpdate

```
class RemoteUpdate(time, properties=None)
```

Bases: `object`

A temporal update

**Parameters:**

- `time` (`TimeInput`) – the timestamp for the update
- `properties` (`PropInput`, optional) – the properties for the update

# RemoteEdgeAddition

```
class RemoteEdgeAddition(src, dst, layer=None,
constant_properties=None, updates=None)
```

Bases: `object`

An edge update

**Parameters:**

- `src` (`GID`) – the id of the source node
- `dst` (`GID`) – the id of the destination node
- `layer` (`str`, optional) – the layer for the update
- `constant_properties` (`PropInput`, optional) – the constant properties for the edge
- `updates` (`list[RemoteUpdate]`, optional) – the temporal updates for the edge

# encode\_graph

```
encode_graph(graph)
```

Encode a graph using Base64 encoding

**Parameters:**

`graph` (`Graph` | `PersistentGraph`) – the graph

**Returns:**

the encoded graph

**Return type:**

`str`

# decode\_graph

[decode\\_graph\(graph\)](#)

Decode a Base64-encoded graph

**Parameters:**

`graph (str)` – the encoded graph

**Returns:**

the decoded graph

**Return type:**

`Union[Graph, PersistentGraph]`

# Matching

`class Matching`

Bases: `object`

A Matching (i.e., a set of edges that do not share any nodes)

**Methods:**

---

`dst(src)` Get the matched destination node for a source node

---

`edge_for_dst(dst)` Get the matched edge for a destination node

---

`edge_for_src(src)` Get the matched edge for a source node

---

`edges()` Get a view of the matched edges

---

`src(dst)` Get the matched source node for a destination node

---

`dst(src)`

Get the matched destination node for a source node

**Parameters:**

`src (NodeInput)` – The source node

**Returns:**

The matched destination node if it exists

**Return type:**

[Optional\[Node\]](#)

`edge_for_dst(dst)`

Get the matched edge for a destination node

**Parameters:**

`dst (NodeInput)` – The source node

**Returns:**

The matched edge if it exists

**Return type:**

[Optional\[Edge\]](#)

`edge_for_src(src)`

Get the matched edge for a source node

**Parameters:**

`src (NodeInput)` – The source node

**Returns:**

The matched edge if it exists

**Return type:**

[Optional\[Edge\]](#)

`edges()`

Get a view of the matched edges

**Returns:**

The edges in the matching

**Return type:**

[Edges](#)

`src(dst)`

Get the matched source node for a destination node

**Parameters:**

`dst (NodeInput)` – The destination node

**Returns:**

The matched source node if it exists

**Return type:**

[Optional\[Node\]](#)

# Infected

**class Infected**

Bases: [object](#)

Attributes:

[\*\*active\*\*](#)

The timestamp at which the infected node started spreading the infection

[\*\*infected\*\*](#)

The timestamp at which the node was infected

[\*\*recovered\*\*](#)

The timestamp at which the infected node stopped spreading the infection

[\*\*active\*\*](#)

The timestamp at which the infected node started spreading the infection

**Return type:**

[int](#)

[\*\*infected\*\*](#)

The timestamp at which the node was infected

**Return type:**

[int](#)

[\*\*recovered\*\*](#)

The timestamp at which the infected node stopped spreading the infection

**Return type:**

[int](#)

## dijkstra\_single\_source\_shortest\_paths

**dijkstra\_single\_source\_shortest\_paths(graph, source, targets, direction=..., weight='weight')**

Finds the shortest paths from a single source to multiple targets in a graph.

**Parameters:**

- graph ([GraphView](#)) – The graph to search in.
- source ([NodeInput](#)) – The source node.
- targets ([list\[NodeInput\]](#)) – A list of target nodes.
- direction ([Direction](#)) – The direction of the edges to be considered for the shortest path. Defaults to “both”.
- weight ([str](#)) – The name of the weight property for the edges. Defaults to “weight”.

**Returns:**

Mapping from nodes to a tuple containing the total cost and the nodes representing the shortest path.

**Return type:**

[NodeStateWeightedSP](#)

## global\_reciprocity

**global\_reciprocity(graph)**

Reciprocity - measure of the symmetry of relationships in a graph, the global reciprocity of the entire graph. This calculates the number of reciprocal connections (edges that go in both directions) in a graph and normalizes it by the total number of directed edges.

**Parameters:**

graph ([GraphView](#)) – a directed Raphtory graph

**Returns:**

reciprocity of the graph between 0 and 1.

**Return type:**

[float](#)

## betweenness\_centrality

**betweenness\_centrality(graph, k=None, normalized=True)**

Computes the betweenness centrality for nodes in a given graph.

**Parameters:**

- graph ([GraphView](#)) – A reference to the graph.
- k ([int, optional](#)) – Specifies the number of nodes to consider for the centrality computation. All nodes are considered by default.
- normalized ([bool](#)) – Indicates whether to normalize the centrality values. Defaults to True.

**Returns:**

Mapping from nodes to their betweenness centrality.

**Return type:**

[NodeStateF64](#)

## all\_local\_reciprocity

**all\_local\_reciprocity(graph)**

Local reciprocity - measure of the symmetry of relationships associated with a node

This measures the proportion of a node's outgoing edges which are reciprocated with an incoming edge.

**Parameters:**

graph ([GraphView](#)) – a directed Raphtory graph

**Returns:**

Mapping of nodes to their reciprocity value.

**Return type:**

[NodeStateF64](#)

## triplet\_count

**triplet\_count(graph)**

Computes the number of connected triplets within a graph

A connected triplet (also known as a wedge, 2-hop path) is a pair of edges with one node in common. For example, the triangle made up of edges A-B, B-C, C-A is formed of three connected triplets.

**Parameters:**

graph ([GraphView](#)) – a Raphtory graph, treated as undirected

**Returns:**

the number of triplets in the graph

**Return type:**

[int](#)

## local\_triangle\_count

**local\_triangle\_count(graph, v)**

Implementations of various graph algorithms that can be run on a graph.

To run an algorithm simply import the module and call the function with the graph as the argument

Local triangle count - calculates the number of triangles (a cycle of length 3) a node participates in.

This function returns the number of pairs of neighbours of a given node which are themselves connected.

**Parameters:**

- graph ([GraphView](#)) – Raphtory graph, this can be directed or undirected but will be treated as undirected
- v ([NodeInput](#)) – node id or name

**Returns:**

number of triangles associated with node v

**Return type:**

[int](#)

## average\_degree

**average\_degree (graph)**

The average (undirected) degree of all nodes in the graph.

Note that this treats the graph as simple and undirected and is equal to twice the number of undirected edges divided by the number of nodes.

**Parameters:**

graph ([GraphView](#)) – a Raphtory graph

**Returns:**

the average degree of the nodes in the graph

**Return type:**

[float](#)

## directed\_graph\_density

**directed\_graph\_density (graph)**

Graph density - measures how dense or sparse a graph is.

The ratio of the number of directed edges in the graph to the total number of possible directed edges (given by  $N * (N-1)$  where N is the number of nodes).

**Parameters:**graph (*GraphView*) – a directed Raphtory graph**Returns:**

Directed graph density of graph.

**Return type:**`float`

## degree\_centrality

**degree\_centrality(graph)**

Computes the degree centrality of all nodes in the graph. The values are normalized by dividing each result with the maximum possible degree. Graphs with self-loops can have values of centrality greater than 1.

**Parameters:**graph (*GraphView*) – The graph view on which the operation is to be performed.**Returns:**

Mapping of nodes to their associated degree centrality.

**Return type:**`NodeStateF64`

## max\_degree

**max\_degree(graph)**

Returns the largest degree found in the graph

**Parameters:**graph (*GraphView*) – The graph view on which the operation is to be performed.**Returns:**

The largest degree

**Return type:**`int`

## min\_degree

**min\_degree(graph)**

Returns the smallest degree found in the graph

**Parameters:**

`graph` ([GraphView](#)) – The graph view on which the operation is to be performed.

**Returns:**

The smallest degree found

**Return type:**

[int](#)

## max\_out\_degree

**max\_out\_degree** (`graph`)

The maximum out degree of any node in the graph.

**Parameters:**

`graph` ([GraphView](#)) – a directed Raphtory graph

**Returns:**

value of the largest outdegree

**Return type:**

[int](#)

## max\_in\_degree

**max\_in\_degree** (`graph`)

The maximum in degree of any node in the graph.

**Parameters:**

`graph` ([GraphView](#)) – a directed Raphtory graph

**Returns:**

value of the largest indegree

**Return type:**

[int](#)

## min\_out\_degree

**min\_out\_degree** (`graph`)

The minimum out degree of any node in the graph.

**Parameters:**

`graph` ([GraphView](#)) – a directed Raphtory graph

**Returns:**

value of the smallest outdegree

**Return type:**

[int](#)

## min\_in\_degree

[\*\*min\\_in\\_degree\(graph\)\*\*](#)

The minimum in degree of any node in the graph.

**Parameters:**

graph ([GraphView](#)) – a directed Raphtory graph

**Returns:**

value of the smallest indegree

**Return type:**

[int](#)

## pagerank

[\*\*pagerank\(graph, iter\\_count=20, max\\_diff=None, use\\_l2\\_norm=True, damping\\_factor=0.85\)\*\*](#)

Pagerank – pagerank centrality value of the nodes in a graph

This function calculates the Pagerank value of each node in a graph. See

<https://en.wikipedia.org/wiki/PageRank> for more information on PageRank centrality. A

default damping factor of 0.85 is used. This is an iterative algorithm which terminates if the sum of the absolute difference in pagerank values between iterations is less than the max diff value given.

**Parameters:**

- graph ([GraphView](#)) – Raphtory graph
- iter\_count ([int](#)) – Maximum number of iterations to run. Note that this will terminate early if convergence is reached. Defaults to 20.
- max\_diff ([Optional/float](#)) – Optional parameter providing an alternative stopping condition. The algorithm will terminate if the sum of the absolute difference in pagerank values between iterations is less than the max diff value given.
- use\_l2\_norm ([bool](#)) – Flag for choosing the norm to use for convergence checks, True for l2 norm, False for l1 norm. Defaults to True.
- damping\_factor ([float](#)) – The damping factor for the PageRank calculation. Defaults to 0.85.

**Returns:**

Mapping of nodes to their pagerank value.

**Return type:**

[NodeStateF64](#)

# single\_source\_shortest\_path

**single\_source\_shortest\_path**(*graph*, *source*, *cutoff*=None)

Calculates the single source shortest paths from a given source node.

## Parameters:

- *graph* ([GraphView](#)) – A reference to the graph. Must implement GraphViewOps.
- *source* ([NodeInput](#)) – The source node.
- *cutoff* ([int](#), optional) – An optional cutoff level. The algorithm will stop if this level is reached.

## Returns:

Mapping from end node to shortest path from the source node.

## Return type:

[NodeStateNodes](#)

# global\_clustering\_coefficient

**global\_clustering\_coefficient**(*graph*)

Computes the global clustering coefficient of a graph. The global clustering coefficient is defined as the number of triangles in the graph divided by the number of triplets in the graph.

Note that this is also known as transitivity and is different to the average clustering coefficient.

## Parameters:

*graph* ([GraphView](#)) – a Raphtry graph, treated as undirected

## Returns:

the global clustering coefficient of the graph

## Return type:

[float](#)

## See also

[Triplet Count](triplet\_count)

# temporally\_reachable\_nodes

```
temporally_reachable_nodes(graph, max_hops, start_time,  
seed_nodes, stop_nodes=None)
```

Temporally reachable nodes – the nodes that are reachable by a time respecting path followed out from a set of seed nodes at a starting time.

This function starts at a set of seed nodes and follows all time respecting paths until either a) a maximum number of hops is reached, b) one of a set of stop nodes is reached, or c) no further time respecting edges exist. A time respecting path is a sequence of nodes  $v_1, v_2, \dots, v_k$  such that there exists a sequence of edges  $(v_i, v_{i+1}, t_i)$  with  $t_i < t_{i+1}$  for  $i = 1, \dots, k - 1$ .

#### Parameters:

- graph ([GraphView](#)) – directed Raphtory graph
- max\_hops ([int](#)) – maximum number of hops to propagate out
- start\_time ([int](#)) – time at which to start the path (such that  $t_1 > start\_time$  for any path starting from these seed nodes)
- seed\_nodes ([list\[NodeInput\]](#)) – list of node names or ids which should be the starting nodes
- stop\_nodes ([Optional\[list\[NodeInput\]\]](#)) – nodes at which a path shouldn't go any further

#### Returns:

Mapping of nodes to their reachability history.

#### Return type:

[NodeStateReachability](#)

## temporal\_bipartite\_graph\_projection

```
temporal_bipartite_graph_projection(graph, delta, pivot_type)
```

Projects a temporal bipartite graph into an undirected temporal graph over the pivot node type. Let G be a bipartite graph with node types A and B. Given  $\delta > 0$ , the projection graph  $G'$  pivoting over type B nodes, will make a connection between nodes  $n_1$  and  $n_2$  (of type A) at time  $(t_1 + t_2)/2$  if they respectively have an edge at time  $t_1, t_2$  with the same node of type B in G, and  $|t_2 - t_1| < \delta$ .

#### Parameters:

- graph ([GraphView](#)) – A directed raphtory graph
- delta ([int](#)) – Time period
- pivot\_type ([str](#)) – node type to pivot over. If a bipartite graph has types A and B, and B is the pivot type, the new graph will consist of type A nodes.

#### Returns:

Projected (unipartite) temporal graph.

**Return type:**

[Graph](#)

## local\_clustering\_coefficient

```
local_clustering_coefficient(graph, v)
```

Local clustering coefficient - measures the degree to which nodes in a graph tend to cluster together.

The proportion of pairs of neighbours of a node who are themselves connected.

**Parameters:**

- graph ([GraphView](#)) – Raphtry graph, can be directed or undirected but will be treated as undirected.
- v ([NodeInput](#)) – node id or name

**Returns:**

the local clustering coefficient of node v in graph.

**Return type:**

[float](#)

## local\_clustering\_coefficient\_batch

```
local_clustering_coefficient_batch(graph, v)
```

## weakly\_connected\_components

```
weakly_connected_components(graph, iter_count=None)
```

Weakly connected components – partitions the graph into node sets which are mutually reachable by an undirected path.

This function assigns a component id to each node such that nodes with the same component id are mutually reachable by an undirected path.

**Parameters:**

- graph ([GraphView](#)) – Raphtry graph

- `iter_count (int, optional)` – Maximum number of iterations to run. Note that this will terminate early if the labels converge prior to the number of iterations being reached.

**Returns:**

Mapping of nodes to their component ids.

**Return type:**

[NodeStateUsize](#)

## strongly\_connected\_components

**strongly\_connected\_components (graph)**

Strongly connected components

Partitions the graph into node sets which are mutually reachable by an directed path

**Parameters:**

graph ([GraphView](#)) – Raphtory graph

**Returns:**

Mapping of nodes to their component ids

**Return type:**

[NodeStateUsize](#)

## in\_components

**in\_components (graph)**

In components – Finding the “in-component” of a node in a directed graph involves identifying all nodes that can be reached following only incoming edges.

**Parameters:**

graph ([GraphView](#)) – Raphtory graph

**Returns:**

Mapping of nodes to the nodes in their ‘in-component’

**Return type:**

[NodeStateNodes](#)

## in\_component

**in\_component (node)**

**In component** – Finding the “in-component” of a node in a directed graph involves identifying all nodes that can be reached following only incoming edges.

**Parameters:**

node ([Node](#)) – The node whose in-component we wish to calculate

**Returns:**

Mapping of nodes in the in-component to the distance from the starting node.

**Return type:**

[NodeStateUsize](#)

## out\_components

**out\_components (graph)**

**Out components** – Finding the “out-component” of a node in a directed graph involves identifying all nodes that can be reached following only outgoing edges.

**Parameters:**

graph ([GraphView](#)) – Raphtry graph

**Returns:**

Mapping of nodes to the nodes within their ‘out-component’

**Return type:**

[NodeStateNodes](#)

## out\_component

**out\_component (node)**

**Out component** – Finding the “out-component” of a node in a directed graph involves identifying all nodes that can be reached following only outgoing edges.

**Parameters:**

node ([Node](#)) – The node whose out-component we wish to calculate

**Returns:**

A NodeState mapping the nodes in the out-component to their distance from the starting node.

**Return type:**

[NodeStateUsize](#)

## fast\_rp

```
fast_rp(graph, embedding_dim, normalization_strength,  
iter_weights, seed=None, threads=None)
```

Computes embedding vectors for each vertex of an undirected/bidirectional graph according to the Fast RP algorithm. Original Paper:

<https://doi.org/10.48550/arXiv.1908.11512> :param graph: The graph view on which embeddings are generated. :type graph: GraphView :param embedding\_dim: The size (dimension) of the generated embeddings. :type embedding\_dim: int :param normalization\_strength: The extent to which high-degree vertices should be discounted (range: 1-0) :type normalization\_strength: float :param iter\_weights: The scalar weights to apply to the results of each iteration :type iter\_weights: list[float] :param seed: The seed for initialisation of random vectors :type seed: int, optional :param threads: The number of threads to be used for parallel execution. :type threads: int, optional

**Returns:**

Mapping from nodes to embedding vectors.

**Return type:**

[NodeStateListF64](#)

## global\_temporal\_three\_node\_motif

```
global_temporal_three_node_motif(graph, delta,  
threads=None)
```

Computes the number of three edge, up-to-three node delta-temporal motifs in the graph, using the algorithm of Paranjape et al, Motifs in Temporal Networks (2017). We point the reader to this reference for more information on the algorithm and background, but provide a short summary below.

Motifs included:

### Stars

There are three classes (in the order they are outputted) of star motif on three nodes based on the switching behaviour of the edges between the two leaf nodes.

- PRE: Stars of the form  $i \leftrightarrow j, i \leftrightarrow j, i \leftrightarrow k$  (ie two interactions with leaf  $j$  followed by one with leaf  $k$ )
- MID: Stars of the form  $i \leftrightarrow j, i \leftrightarrow k, i \leftrightarrow j$  (ie switching interactions from leaf  $j$  to leaf  $k$ , back to  $j$  again)

- POST: Stars of the form  $i \leftrightarrow j$ ,  $i \leftrightarrow k$ ,  $i \leftrightarrow k$  (ie one interaction with leaf  $j$  followed by two with leaf  $k$ )

Within each of these classes is 8 motifs depending on the direction of the first to the last edge – incoming “I” or outgoing “O”. These are enumerated in the order III, IIO, IOI, IOO, OII, OIO, OOI, OOO (like binary with “I”-0 and “O”-1).

Two node motifs:

Also included are two node motifs, of which there are 8 when counted from the perspective of each node. These are characterised by the direction of each edge, enumerated in the above order. Note that for the global graph counts, each motif is counted in both directions (a single III motif for one node is an OOO motif for the other node).

Triangles:

There are 8 triangle motifs:

1.  $i \rightarrow j, k \rightarrow j, i \rightarrow k$
2.  $i \rightarrow j, k \rightarrow j, k \rightarrow i$
3.  $i \rightarrow j, j \rightarrow k, i \rightarrow k$
4.  $i \rightarrow j, j \rightarrow k, k \rightarrow i$
5.  $i \rightarrow j, k \rightarrow i, j \rightarrow k$
6.  $i \rightarrow j, k \rightarrow i, k \rightarrow j$
7.  $i \rightarrow j, i \rightarrow k, j \rightarrow k$
8.  $i \rightarrow j, i \rightarrow k, k \rightarrow j$

Parameters:

- graph ([GraphView](#)) – A directedraphory graph
- delta ([int](#)) – Maximum time difference between the first and last edge of the motif. NB if time for edges was given as a UNIX epoch, this should be given in seconds, otherwise milliseconds should be used (if edge times were given as string)
- threads ([int, optional](#)) – The number of threads to use when running the algorithm.

Returns:

A 40 dimensional array with the counts of each motif, given in the same order as described above. Note that the two-node motif counts are symmetrical so it may be more useful just to consider the first four elements.

Return type:

[list\[int\]](#)

Notes

---

This is achieved by calling the local motif counting algorithm, summing the resulting arrays and dealing with overcounted motifs: the triangles (by dividing each motif count by three) and two-node motifs (dividing by two).

## global\_temporal\_three\_node\_motif\_multi

```
global_temporal_three_node_motif_multi(graph, deltas,  
threads=None)
```

Computes the global counts of three-edge up-to-three node temporal motifs for a range of timescales. See `global_temporal_three_node_motif` for an interpretation of each row returned.

### Parameters:

- `graph` ([GraphView](#)) – A directedraphory graph
- `deltas` ([list\[int\]](#)) – A list of delta values to use.
- `threads` ([int](#), *optional*) – The number of threads to use.

### Returns:

A list of 40d arrays, each array is the motif count for a particular value of delta, returned in the order that the deltas were given as input.

### Return type:

[list\[list\[int\]\]](#)

## local\_temporal\_three\_node\_motifs

```
local_temporal_three_node_motifs(graph, delta,  
threads=None)
```

Computes the number of each type of motif that each node participates in. See `global_temporal_three_node_motifs` for a summary of the motifs involved.

### Parameters:

- `graph` ([GraphView](#)) – A directedraphory graph
- `delta` ([int](#)) – Maximum time difference between the first and last edge of the motif. NB if time for edges was given as a UNIX epoch, this should be given in seconds, otherwise milliseconds should be used (if edge times were given as string)

**Returns:**

A mapping from nodes to lists of motif counts (40 counts in the same order as the global motif counts) with the number of each motif that node participates in.

**Return type:**

[NodeStateMotifs](#)

# hits

**hits**(graph, iter\_count=20, threads=None)

HITS (Hubs and Authority) Algorithm:

AuthScore of a node (A) = Sum of HubScore of all nodes pointing at node (A) from previous iteration / Sum of HubScore of all nodes in the current iteration

HubScore of a node (A) = Sum of AuthScore of all nodes pointing away from node (A) from previous iteration / Sum of AuthScore of all nodes in the current iteration

**Parameters:**

- graph ([GraphView](#)) – Graph to run the algorithm on
- iter\_count ([int](#)) – How many iterations to run the algorithm. Defaults to 20.
- threads ([int, optional](#)) – Number of threads to use

**Returns:**

A mapping from nodes their hub and authority scores

**Return type:**

[NodeStateHits](#)

# balance

**balance**(graph, name=..., direction=...)

Sums the weights of edges in the graph based on the specified direction.

This function computes the sum of edge weights based on the direction provided, and can be executed in parallel using a given number of threads.

**Parameters:**

- graph ([GraphView](#)) – The graph view on which the operation is to be performed.
- name ([str](#)) – The name of the edge property used as the weight. Defaults to “weight”.
- direction ([Direction](#)) – Specifies the direction of the edges to be considered for summation. Defaults to “both”. \* “out”: Only consider outgoing edges. \* “in”: Only consider incoming edges. \* “both”: Consider both outgoing and incoming edges. This is the default.

**Returns:**

Mapping of nodes to the computed sum of their associated edge weights.

**Return type:**

[NodeStateF64](#)

# label\_propagation

```
label_propagation(graph, seed=None)
```

Computes components using a label propagation algorithm

**Parameters:**

- graph ([GraphView](#)) – A reference to the graph
- seed ([bytes](#), *optional*) – Array of 32 bytes of u8 which is set as the rng seed

**Returns:**

A list of sets each containing nodes that have been grouped

**Return type:**

[list\[set\[Node\]\]](#)

# temporal\_SEIR

```
temporal_SEIR(graph, seeds, infection_prob,
initial_infection, recovery_rate=None, incubation_rate=None,
rng_seed=None)
```

Simulate an SEIR dynamic on the network

The algorithm uses the event-based sampling strategy from

<https://doi.org/10.1371/journal.pone.0246961>

**Parameters:**

- graph ([GraphView](#)) – the graph view
- seeds ([int](#) | [float](#) | [list\[NodeInput\]](#)) – the seeding strategy to use for the initial infection (if int, choose fixed number of nodes at random, if float infect each node with this probability, if list initially infect the specified nodes)
- infection\_prob ([float](#)) – the probability for a contact between infected and susceptible nodes to lead to a transmission
- initial\_infection ([int](#) | [str](#) | [datetime](#)) – the time of the initial infection
- recovery\_rate ([float](#) | [None](#)) – optional recovery rate (if None, simulates SEI dynamic where nodes never recover) the actual recovery time is sampled from an exponential distribution with this rate
- incubation\_rate ([float](#) | [None](#)) – optional incubation rate (if None, simulates SI or SIR dynamics where infected nodes are infectious at the next time step)

the actual incubation time is sampled from an exponential distribution with this rate

- `rng_seed` (`int` | `None`) – optional seed for the random number generator

**Returns:**

**Mapping from nodes toInfectedobjects for each infected node with attributes**

infected: the time stamp of the infection event active: the time stamp at which the node actively starts spreading the infection (i.e., the end of the incubation period)

recovered: the time stamp at which the node recovered (i.e., stopped spreading the infection)

**Return type:**

[NodeStateSEIR](#)

# louvain

**`louvain(graph, resolution=1.0, weight_prop=None, tol=None)`**

Louvain algorithm for community detection

**Parameters:**

- `graph` ([GraphView](#)) – the graph view
- `resolution` (`float`) – the resolution parameter for modularity. Defaults to 1.0.
- `weight_prop` (`str` | `None`) – the edge property to use for weights (has to be float)
- `tol` (`None` | `float`) – the floating point tolerance for deciding if improvements are significant (default: 1e-8)

**Returns:**

Mapping of nodes to their community assignment

**Return type:**

[NodeStateUsize](#)

# fruchterman\_reingold

**`fruchterman_reingold(graph, iterations=100, scale=1.0, node_start_size=1.0, cooloff_factor=0.95, dt=0.1)`**

Fruchterman Reingold layout algorithm

**Parameters:**

- `graph` ([GraphView](#)) – the graph view
- `iterations` (`int` | `None`) – the number of iterations to run. Defaults to 100.
- `scale` (`float` | `None`) – the scale to apply. Defaults to 1.0.
- `node_start_size` (`float` | `None`) – the start node size to assign random positions. Defaults to 1.0.

- `cooloff_factor` (`float` | `None`) – the cool off factor for the algorithm. Defaults to 0.95.
- `dt` (`float` | `None`) – the time increment between iterations. Defaults to 0.1.

**Returns:**

A mapping from nodes to their [x, y] positions

**Return type:**

[NodeLayout](#)

## cohesive\_fruchterman\_reingold

```
cohesive_fruchterman_reingold(graph, iter_count=100,
scale=1.0, node_start_size=1.0, cooloff_factor=0.95, dt=0.1)
```

Cohesive version of fruchterman\_reingold that adds virtual edges between isolated nodes  
:param graph: A reference to the graph :type graph: GraphView :param iter\_count: The number of iterations to run. Defaults to 100. :type iter\_count: int :param scale: Global scaling factor to control the overall spread of the graph. Defaults to 1.0. :type scale: float :param node\_start\_size: Initial size or movement range for nodes. Defaults to 1.0. :type node\_start\_size: float :param cooloff\_factor: Factor to reduce node movement in later iterations, helping stabilize the layout. Defaults to 0.95. :type cooloff\_factor: float :param dt: Time step or movement factor in each iteration. Defaults to 0.1. :type dt: float

**Returns:**

A mapping from nodes to their [x, y] positions

**Return type:**

[NodeLayout](#)

## max\_weight\_matching

```
max_weight_matching(graph, weight_prop=None,
max_cardinality=True, verify_optimum_flag=False)
```

Compute a maximum-weighted matching in the general undirected weighted graph given by “edges”. If `max_cardinality` is true, only maximum-cardinality matchings are considered as solutions.

The algorithm is based on “Efficient Algorithms for Finding Maximum Matching in Graphs” by Zvi Galil, ACM Computing Surveys, 1986.

Based on networkx implementation <[networkx/networkx](#)>

With reference to the standalone prototype implementation from:

<<http://jorisvr.nl/article/maximum-matching>>

<<http://jorisvr.nl/files/graphmatching/20130407/mwmatching.py>>

The function takes time  $O(n^{**3})$

### Parameters:

- graph ([GraphView](#)) – The graph to compute the maximum weight matching for
- weight\_prop ([str, optional](#)) – The property on the edge to use for the weight. If not provided,
- max\_cardinality ([bool](#)) – If set to True, consider only maximum-cardinality matchings. Defaults to True. If True, finds the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings, otherwise, finds the maximum weight matching irrespective of cardinality.
- verify\_optimum\_flag ([bool](#)) – Whether the optimum should be verified. Defaults to False. If true prior to returning, an additional routine to verify the optimal solution was found will be run after computing the maximum weight matching. If it's true and the found matching is not an optimal solution this function will panic. This option should normally be only set true during testing.

### Returns:

The matching

### Return type:

[Matching](#)

# VectorisedGraph

`class VectorisedGraph`

Bases: [object](#)

### Methods:

[`documents\_by\_similarity\(query, limit\[, window\]\)`](#)

Search the top scoring documents according to query with no more than limit documents

[`edges\_by\_similarity\(query, limit\[, window\]\)`](#)

Search the top scoring edges according to query with no more than limit edges

---

<code><a href="#">empty_selection()</a></code>	Return an empty selection of documents
--	--

---

<code><a href="#">entities_by_similarity(query, limit[, window])</a></code>	Search the top scoring entities according to query with no more than limit entities
---	---

---

<code><a href="#">get_graph_documents()</a></code>	Return all the graph level documents
--	--------------------------------------

---

<code><a href="#">nodes_by_similarity(query, limit[, window])</a></code>	Search the top scoring nodes according to query with no more than limit nodes
--	---

---

<code><a href="#">save_embeddings(file)</a></code>	Save the embeddings present in this graph to file so they can be further used in a call to vectorise
--	--

---

### `documents\_by\_similarity\(query, limit, window=None\)`

Search the top scoring documents according to query with no more than limit documents

#### Parameters:

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the maximum number of documents to search
- `window (Tuple\[int | str, int | str\], optional)` – the window where documents need to belong to in order to be considered

#### Returns:

The vector selection resulting from the search

#### Return type:

`VectorSelection`

### `edges\_by\_similarity\(query, limit, window=None\)`

Search the top scoring edges according to query with no more than limit edges

#### Parameters:

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the maximum number of new edges to search
- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Returns:**

The vector selection resulting from the search

**Return type:**

`VectorSelection`

`empty_selection()`

Return an empty selection of documents

`entities_by_similarity(query, limit, window=None)`

Search the top scoring entities according to query with no more than limit entities

**Parameters:**

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the maximum number of new entities to search
- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Returns:**

The vector selection resulting from the search

**Return type:**

`VectorSelection`

`get_graph_documents()`

Return all the graph level documents

**Returns:**

list of graph level documents

**Return type:**

`list[Document]`

`nodes_by_similarity(query, limit, window=None)`

Search the top scoring nodes according to query with no more than limit nodes

**Parameters:**

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the maximum number of new nodes to search
- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Returns:**

The vector selection resulting from the search

**Return type:**

[VectorSelection](#)

`save_embeddings(file)`

Save the embeddings present in this graph to file so they can be further used in a call to vectorise

# Document

`class Document`

Bases: [object](#)

A Document

**Parameters:**

- `content (str)` – the document content
- `life (int | Tuple[int, int], optional)` – the optional lifespan for the document  
(single value corresponds to an event, a tuple corresponds to a window).

Attributes:

[content](#) the document content

---

[embedding](#) the embedding

---

[entity](#) the entity corresponding to the document

---

[life](#) the life span

---

[content](#)

the document content

**Return type:**

[str](#)

[embedding](#)

the embedding

**Returns:**

the embedding for the document if it was computed

**Return type:**

[Optional\[Embedding\]](#)

**entity**

the entity corresponding to the document

**Return type:**

[Optional\[Any\]](#)

**life**

the life span

**Return type:**

[Optional\[Union\[int | Tuple\[int, int\]\]\]](#)

# Embedding

`class Embedding`

Bases: [object](#)

# VectorSelection

`class VectorSelection`

Bases: [object](#)

**Methods:**

[add\\_edges\(edges\)](#)

Add all the documents  
associated with the edges to  
the current selection

[add\\_nodes\(nodes\)](#)

Add all the documents  
associated with the nodes to  
the current selection

[append\(selection\)](#)

Add all the documents in  
selection to the current  
selection

[edges\(\)](#)

Return the edges present in  
the current selection

---

<code><u>expand</u>(hops[, window])</code>	Add all the documents hops hops away to the selection
--	---

---

<code><u>expand_documents_by_similarity</u>(query, limit)</code>	Add the top limit adjacent documents with higher score for query to the selection
--	---

---

<code><u>expand_edges_by_similarity</u>(query, limit[, ...])</code>	Add the top limit adjacent edges with higher score for query to the selection
---	---

---

<code><u>expand_entities_by_similarity</u>(query, limit)</code>	Add the top limit adjacent entities with higher score for query to the selection
---	--

---

<code><u>expand_nodes_by_similarity</u>(query, limit[, ...])</code>	Add the top limit adjacent nodes with higher score for query to the selection
---	---

---

<code><u>get_documents</u>()</code>	Return the documents present in the current selection
-------------------------------------	---

---

<code><u>get_documents_with_scores</u>()</code>	Return the documents alongside their scores present in the current selection
---	--

---

<code><u>nodes</u>()</code>	Return the nodes present in the current selection
-----------------------------	---

---

### **add\_edges(edges)**

Add all the documents associated with the edges to the current selection

Documents added by this call are assumed to have a score of 0.

**Parameters:**

`edges (list)` – a list of the edge ids or edges to add

**Return type:**

[None](#)

`add_nodes (nodes)`

Add all the documents associated with the nodes to the current selection

Documents added by this call are assumed to have a score of 0.

**Parameters:**

`nodes (list)` – a list of the node ids or nodes to add

**Return type:**

[None](#)

`append (selection)`

Add all the documents in selection to the current selection

**Parameters:**

`selection (VectorSelection)` – a selection to be added

**Returns:**

The selection with the new documents

**Return type:**

[VectorSelection](#)

`edges ()`

Return the edges present in the current selection

**Returns:**

list of edges in the current selection

**Return type:**

[list\[Edge\]](#)

`expand (hops, window=None)`

Add all the documents hops hops away to the selection

Two documents A and B are considered to be 1 hop away of each other if they are on the same entity or if they are on the same node/edge pair. Provided that, two nodes A and C are n hops away of each other if there is a document B such that A is n - 1 hops away of B and B is 1 hop away of C.

**Parameters:**

- `hops (int)` – the number of hops to carry out the expansion

- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Return type:**

`None`

`expand_documents_by_similarity(query, limit, window=None)`

Add the top limit adjacent documents with higher score for query to the selection

**The expansion algorithm is a loop with two steps on each iteration:**

1. All the documents 1 hop away of some of the documents included on the selection (and not already selected) are marked as candidates.
2. Those candidates are added to the selection in descending order according to the similarity score obtained against the query.

This loops goes on until the current selection reaches a total of limit documents or until no more documents are available

**Parameters:**

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the number of documents to add
- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Return type:**

`None`

`expand_edges_by_similarity(query, limit, window=None)`

Add the top limit adjacent edges with higher score for query to the selection

This function has the same behavior as `expand_entities_by_similarity` but it only considers edges.

**Parameters:**

- `query (str | list)` – the text or the embedding to score against
- `limit (int)` – the maximum number of new edges to add
- `window (Tuple[int | str, int | str], optional)` – the window where documents need to belong to in order to be considered

**Return type:**

`None`

`expand_entities_by_similarity(query, limit, window=None)`

Add the top limit adjacent entities with higher score for query to the selection

**The expansion algorithm is a loop with two steps on each iteration:**

1. All the entities 1 hop away of some of the entities included on the selection (and not already selected) are marked as candidates.
2. Those candidates are added to the selection in descending order according to the similarity score obtained against the query.

This loops goes on until the number of new entities reaches a total of limit entities or until no more documents are available

#### Parameters:

- query ([str](#) | [list](#)) – the text or the embedding to score against
- limit ([int](#)) – the number of documents to add
- window ([Tuple\[int | str, int | str\]](#), optional) – the window where documents need to belong to in order to be considered

#### Return type:

[None](#)

**`expand_nodes_by_similarity(query, limit, window=None)`**

Add the top limit adjacent nodes with higher score for query to the selection

This function has the same behavior as `expand_entities_by_similarity` but it only considers nodes.

#### Parameters:

- query ([str](#) | [list](#)) – the text or the embedding to score against
- limit ([int](#)) – the maximum number of new nodes to add
- window ([Tuple\[int | str, int | str\]](#), optional) – the window where documents need to belong to in order to be considered

#### Return type:

[None](#)

**`get_documents()`**

Return the documents present in the current selection

#### Returns:

list of documents in the current selection

#### Return type:

[list\[Document\]](#)

**`get_documents_with_scores()`**

Return the documents alongside their scores present in the current selection

#### Returns:

list of documents and scores

#### Return type:

`list[Tuple[Document, float]]`

**nodes ()**

Return the nodes present in the current selection

**Returns:**

list of nodes in the current selection

**Return type:**

`list[Node]`

# NodeGroups

`class NodeGroups`

Bases: `object`

**Methods:**

`group(index)`

Get group nodes and value

---

`group_subgraph(index)`

Get group as subgraph

---

`iter_subgraphs()`

Iterate over group subgraphs

**group (index)**

Get group nodes and value

**Parameters:**

index (`int`) – the group index

**Returns:**

Nodes and corresponding value

**Return type:**

`Tuple[Any, Nodes]`

`group_subgraph(index)`

Get group as subgraph

**Parameters:**

index (`int`) – the group index

**Returns:**

The group as a subgraph and corresponding value

**Return type:**

[Tuple\[Any, GraphView\]](#)

**iter\_subgraphs()**

Iterate over group subgraphs

**Returns:**

Iterator over subgraphs with corresponding value

**Return type:**

[Iterator\[Tuple\[Any, GraphView\]\]](#)

# DegreeView

**class DegreeView**

Bases: [object](#)

A lazy view over node values

**Methods:**

[after](#)(start)

Create a view of the DegreeView including all events after start (exclusive).

---

[at](#)(time)

Create a view of the DegreeView including all events at time.

---

[before](#)(end)

Create a view of the DegreeView including all events before end (exclusive).

---

[bottom\\_k](#)(k)

Compute the k smallest values

---

[collect](#)()

Compute all values and return the result as a list

---

[compute](#)()

Compute all values and return the result as a node view

<code><u>default_layer</u>()</code>	Return a view of DegreeView containing only the default edge layer :returns: The layered view :rtype: DegreeView
<code><u>exclude_layer</u>(name)</code>	Return a view of DegreeView containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of DegreeView containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of DegreeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of DegreeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>get</u>(node[, default])</code>	Get value for node
<code><u>groups</u>()</code>	Group by value
<code><u>has_layer</u>(name)</code>	Check if DegreeView has the layer "name"
<code><u>items</u>()</code>	Iterate over items
<code><u>latest</u>()</code>	Create a view of the DegreeView including all events at the latest time.

---

<code><u>layer</u>(name)</code>	Return a view of DegreeView containing the layer "name" Errors if the layer does not exist
<code><u>layers</u>(names)</code>	Return a view of DegreeView containing all layers names Errors if any of the layers do not exist.
<code><u>max</u>()</code>	Return the maximum value
<code><u>max_item</u>()</code>	Return largest value and corresponding node
<code><u>mean</u>()</code>	mean of values over all nodes
<code><u>median</u>()</code>	Return the median value
<code><u>median_item</u>()</code>	Return median value and corresponding node
<code><u>min</u>()</code>	Return the minimum value
<code><u>min_item</u>()</code>	Return smallest value and corresponding node
<code><u>nodes</u>()</code>	Iterate over nodes
<code><u>rolling</u>(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><u>shrink_end</u>(end)</code>	Set the end of the window to the smaller of end and self.end()
<code><u>shrink_start</u>(start)</code>	Set the start of the window to the larger of start and self.start()

---

<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling <code>shrink_start</code> followed by <code>shrink_end</code> but more efficient)
<code>snapshot_at(time)</code>	Create a view of the DegreeView including all events that have not been explicitly deleted at time.
<code>snapshot_latest()</code>	Create a view of the DegreeView including all events that have not been explicitly deleted at the latest time.
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>sum()</code>	sum of values over all nodes
<code>to_df()</code>	Convert results to pandas DataFrame
<code>top_k(k)</code>	Compute the k largest values
<code>valid_layers(names)</code>	Return a view of DegreeView containing all layers names Any layers that do not exist are ignored
<code>values()</code>	Iterate over values
<code>window(start, end)</code>	Create a view of the DegreeView including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

<code>end</code>	Gets the latest time that this DegreeView is valid.
------------------	---

---

---

<code>end_date_time</code>	Gets the latest datetime that this DegreeView is valid
<code>start</code>	Gets the start time for rolling and expanding windows for this DegreeView
<code>start_date_time</code>	Gets the earliest datetime that this DegreeView is valid
<code>window_size</code>	Get the window size (difference between start and end) for this DegreeView

---

### `after(start)`

Create a view of the DegreeView including all events after start (exclusive).

#### **Parameters:**

`start` ([TimeInput](#)) – The start time of the window.

#### **Return type:**

[DegreeView](#)

### `at(time)`

Create a view of the DegreeView including all events at time.

#### **Parameters:**

`time` ([TimeInput](#)) – The time of the window.

#### **Return type:**

[DegreeView](#)

### `before(end)`

Create a view of the DegreeView including all events before end (exclusive).

#### **Parameters:**

`end` ([TimeInput](#)) – The end time of the window.

#### **Return type:**

[DegreeView](#)

### `bottom_k(k)`

Compute the k smallest values

#### **Parameters:**

`k` ([int](#)) – The number of values to return

#### **Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateUsize](#)

`collect()`

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[List\[int\]](#)

`compute()`

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateUsize](#)

`default_layer()`

Return a view of DegreeView containing only the default edge layer :returns: The layered view :rtype: DegreeView

`end`

Gets the latest time that this DegreeView is valid.

**Returns:**

The latest time that this DegreeView is valid or None if the DegreeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`end_date_time`

Gets the latest datetime that this DegreeView is valid

**Returns:**

The latest datetime that this DegreeView is valid or None if the DegreeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`exclude_layer(name)`

**Return a view of DegreeView containing all layers except the excluded name Errors**  
if any of the layers do not exist.

**Parameters:**

`name (str) – layer name that is excluded for the new view`

**Returns:**

The layered view

**Return type:**

[DegreeView](#)

**exclude\_layers(names)**

**Return a view of DegreeView containing all layers except the excluded names Errors**  
if any of the layers do not exist.

**Parameters:**

`names (list[str]) – list of layer names that are excluded for the new view`

**Returns:**

The layered view

**Return type:**

[DegreeView](#)

**exclude\_valid\_layer(name)**

**Return a view of DegreeView containing all layers except the excluded name** :param  
name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[DegreeView](#)

**exclude\_valid\_layers(names)**

**Return a view of DegreeView containing all layers except the excluded names**  
:param names: list of layer names that are excluded for the new view :type names:  
list[str]

**Returns:**

The layered view

**Return type:**

[DegreeView](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[int\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[int\]](#)

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

`has_layer(name)`

Check if DegreeView has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator[Tuple[Node, int]]`

`latest()`

Create a view of the DegreeView including all events at the latest time.

**Return type:**

`DegreeView`

`layer (name)`

Return a view of DegreeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

`name (str)` – then name of the layer.

**Returns:**

The layered view

**Return type:**

`DegreeView`

`layers (names)`

Return a view of DegreeView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

`DegreeView`

`max ()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

`Optional[int]`

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

`Optional[Tuple[Node, int]]`

`mean()`

mean of values over all nodes

**Returns:**

mean value

**Return type:**

`float`

`median()`

Return the median value

**Return type:**

`Optional[int]`

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

`Optional[Tuple[Node, int]]`

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

`Optional[int]`

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

`Optional[Tuple[Node, int]]`

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`shrink_end(end)`

Set the end of the window to the smaller of end and self.end()

**Parameters:**

`end (TimeInput)` – the new end time of the window

**Return type:**

[DegreeView](#)

`shrink_start(start)`

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start (TimeInput)` – the new start time of the window

**Return type:**

[DegreeView](#)

`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start (TimeInput)` – the new start time for the window
- `end (TimeInput)` – the new end time for the window

**Return type:**

[DegreeView](#)

`snapshot_at(time)`

Create a view of the DegreeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[DegreeView](#)

`snapshot_latest()`

Create a view of the DegreeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[DegreeView](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

reverse ([bool](#)) – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateUsize](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateUsize](#)

`start`

Gets the start time for rolling and expanding windows for this DegreeView

**Returns:**

The earliest time that this DegreeView is valid or None if the DegreeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this DegreeView is valid

**Returns:**

The earliest datetime that this DegreeView is valid or None if the DegreeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**sum()**

sum of values over all nodes

**Returns:**

the sum

**Return type:**

[int](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**top\_k(k)**

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

## NodeStateUsize

### **valid\_layers(names)**

Return a view of DegreeView containing all layers names Any layers that do not exist are ignored

#### **Parameters:**

names (*list[str]*) – list of layer names for the new view

#### **Returns:**

The layered view

#### **Return type:**

[DegreeView](#)

### **values()**

Iterate over values

#### **Returns:**

Iterator over values

#### **Return type:**

[Iterator\[int\]](#)

### **window(start, end)**

Create a view of the DegreeView including all events between start (inclusive) and end (exclusive)

#### **Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

#### **Return type:**

[DegreeView](#)

### **window\_size**

Get the window size (difference between start and end) for this DegreeView

#### **Return type:**

[Optional\[int\]](#)

# NodeStateUsize

**class NodeStateUsize**

Bases: [object](#)

Methods:

<code>bottom_k(k)</code>	Compute the k smallest values
<code>get(node[, default])</code>	Get value for node
<code>groups()</code>	Group by value
<code>items()</code>	Iterate over items
<code>max()</code>	Return the maximum value
<code>max_item()</code>	Return largest value and corresponding node
<code>mean()</code>	mean of values over all nodes
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>sum()</code>	sum of values over all nodes
<code>to_df()</code>	Convert results to pandas DataFrame
<code>top_k(k)</code>	Compute the k largest values
<code>values()</code>	Iterate over values
<b><code>bottom_k(k)</code></b>	Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateUsize](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[int\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[int\]](#)

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, int\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[int\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

`mean()`

mean of values over all nodes

**Returns:**

mean value

**Return type:**

[float](#)

`median()`

Return the median value

**Return type:**

[Optional\[int\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[int\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateUsize](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateUsize](#)

`sum()`

sum of values over all nodes

**Returns:**

the sum

**Return type:**

[int](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateUsize](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[int\]](#)

# NodeStateU64

`class NodeStateU64`

Bases: [object](#)

**Methods:**

[`bottom\_k\(k\)`](#)

Compute the k smallest values

[`get\(node\[, default\]\)`](#)

Get value for node

[`items\(\)`](#)

Iterate over items

[`max\(\)`](#)

Return the maximum value

[`max\_item\(\)`](#)

Return largest value and corresponding node

[`mean\(\)`](#)

mean of values over all nodes

[`median\(\)`](#)

Return the median value

[`median\_item\(\)`](#)

Return median value and corresponding node

[`min\(\)`](#)

Return the minimum value

---

`min_item()` Return smallest value and corresponding node

`nodes()` Iterate over nodes

`sorted([reverse])` Sort by value

`sorted_by_id()` Sort results by node id

`sum()` sum of values over all nodes

`to_df()` Convert results to pandas DataFrame

`top_k(k)` Compute the k largest values

`values()` Iterate over values

`bottom_k(k)`

Compute the k smallest values

**Parameters:**

k (`int`) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateU64`

`get(node, default=...)`

Get value for node

**Parameters:**

- node (`NodeInput`) – the node
- default (`Optional[int]`) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional[int]`

`items()`

**Iterate over items**

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, int\]\]](#)

**max()**

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[int\]](#)

**max\_item()**

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

**mean()**

mean of values over all nodes

**Returns:**

mean value

**Return type:**

[float](#)

**median()**

Return the median value

**Return type:**

[Optional\[int\]](#)

**median\_item()**

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

**min()**

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[int\]](#)

**min\_item()**

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, int\]\]](#)

**nodes()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted(reverse=False)**

Sort by value

**Parameters:**

reverse ([bool](#)) – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateU64](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

## NodeStateU64

**sum()**

sum of values over all nodes

**Returns:**

the sum

**Return type:**

int

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

DataFrame

**top\_k(k)**

Compute the k largest values

**Parameters:**

k (int) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

NodeStateU64

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

Iterator[int]

## NodeStateOptionI64

**class NodeStateOptionI64**

Bases: `object`

Methods:

---

`bottom_k(k)` Compute the k smallest values

---

`get(node[, default])` Get value for node

---

`groups()` Group by value

---

`items()` Iterate over items

---

`max()` Return the maximum value

---

`max_item()` Return largest value and corresponding node

---

`median()` Return the median value

---

`median_item()` Return median value and corresponding node

---

`min()` Return the minimum value

---

`min_item()` Return smallest value and corresponding node

---

`nodes()` Iterate over nodes

---

`sorted([reverse])` Sort by value

---

`sorted_by_id()` Sort results by node id

---

`to_df()` Convert results to pandas DataFrame

---

`top_k(k)` Compute the k largest values

---

`values()` Iterate over values

## **bottom\_k(k)**

Compute the k smallest values

### **Parameters:**

k ([int](#)) – The number of values to return

### **Returns:**

The k smallest values as a node state

### **Return type:**

[NodeStateOptionI64](#)

## **get(node, default=...)**

Get value for node

### **Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[int\]\]](#)) – the default value. Defaults to None.

### **Returns:**

the value for the node or the default value

### **Return type:**

[Optional\[Optional\[int\]\]](#)

## **groups()**

Group by value

### **Returns:**

The grouped nodes

### **Return type:**

[NodeGroups](#)

## **items()**

Iterate over items

### **Returns:**

Iterator over items

### **Return type:**

[Iterator\[Tuple\[Node, Optional\[int\]\]\]](#)

## **max()**

Return the maximum value

### **Returns:**

The maximum value or None if empty

**Return type:**

`Optional[Optional[int]]`

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

`Optional[Tuple[Node, Optional[int]]]`

`median()`

Return the median value

**Return type:**

`Optional[Optional[int]]`

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

`Optional[Tuple[Node, Optional[int]]]`

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

`Optional[Optional[int]]`

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

`Optional[Tuple[Node, Optional[int]]]`

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionI64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionI64](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionI64](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[int\]\]](#)

# IdView

**class IdView**

Bases: [object](#)

A lazy view over node values

Methods:

---

[\*\*bottom\\_k\(k\)\*\*](#) Compute the k smallest values

---

[\*\*collect\(\)\*\*](#) Compute all values and return the result as a list

---

[\*\*compute\(\)\*\*](#) Compute all values and return the result as a node view

---

[\*\*get\(node\[, default\]\)\*\*](#) Get value for node

---

[\*\*items\(\)\*\*](#) Iterate over items

---

[\*\*max\(\)\*\*](#) Return the maximum value

---

[\*\*max\\_item\(\)\*\*](#) Return largest value and corresponding node

---

[\*\*median\(\)\*\*](#) Return the median value

---

`median_item()`      Return median value and corresponding node

---

`min()`      Return the minimum value

---

`min_item()`      Return smallest value and corresponding node

---

`nodes()`      Iterate over nodes

---

`sorted([reverse])`      Sort by value

---

`sorted_by_id()`      Sort results by node id

---

`to_df()`      Convert results to pandas DataFrame

---

`top_k(k)`      Compute the k largest values

---

`values()`      Iterate over values

---

`bottom_k(k)`

Compute the k smallest values

**Parameters:**

`k` (`int`) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateGID`

`collect()`

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

`list[GID]`

`compute()`

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeState\[GID\]](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[GID\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[GID\]](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, GID\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[GID\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[GID\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[GID\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateGID](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateGID](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateGID](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[GID\]](#)

# NodeStateGID

**class** `NodeStateGID`

Bases: `object`

Methods:

`bottom_k(k)` Compute the k smallest values

`get(node[, default])` Get value for node

`items()` Iterate over items

`max()` Return the maximum value

`max_item()` Return largest value and corresponding node

`median()` Return the median value

`median_item()` Return median value and corresponding node

`min()` Return the minimum value

`min_item()` Return smallest value and corresponding node

`nodes()` Iterate over nodes

`sorted([reverse])` Sort by value

`sorted_by_id()` Sort results by node id

`to_df()` Convert results to pandas DataFrame

`top_k(k)` Compute the k largest values

`values()` Iterate over values

`bottom_k(k)`

Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateGID](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[GID\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[GID\]](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, GID\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[GID\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[GID\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[GID\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, GID\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateGID](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateGID](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateGID](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[GID\]](#)

# EarliestTimeView

`class EarliestTimeView`

Bases: [object](#)

A lazy view over node values

**Methods:**

---

`after(start)`

Create a view of the EarliestTimeView including all events after start (exclusive).

---

`at(time)`

Create a view of the EarliestTimeView including all events at time.

---

`before(end)`

Create a view of the EarliestTimeView including all events before end (exclusive).

---

`bottom_k(k)`

Compute the k smallest values

---

`collect()`

Compute all values and return the result as a list

---

`compute()`

Compute all values and return the result as a node view

---

`default_layer()`

Return a view of EarliestTimeView containing only the default edge layer :returns: The layered view :rtype: EarliestTimeView

---

`exclude_layer(name)`

Return a view of EarliestTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

---

`exclude_layers(names)`

Return a view of EarliestTimeView containing all layers except the excluded names Errors if any of the layers do not exist.

---

`exclude_valid_layer(name)`

Return a view of EarliestTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

---

<code><u>exclude_valid_layers</u>(names)</code>	Return a view of EarliestTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>get</u>(node[, default])</code>	Get value for node
<code><u>groups</u>()</code>	Group by value
<code><u>has_layer</u>(name)</code>	Check if EarliestTimeView has the layer "name"
<code><u>items</u>()</code>	Iterate over items
<code><u>latest</u>()</code>	Create a view of the EarliestTimeView including all events at the latest time.
<code><u>layer</u>(name)</code>	Return a view of EarliestTimeView containing the layer "name" Errors if the layer does not exist
<code><u>layers</u>(names)</code>	Return a view of EarliestTimeView containing all layers names Errors if any of the layers do not exist.
<code><u>max</u>()</code>	Return the maximum value
<code><u>max_item</u>()</code>	Return largest value and corresponding node
<code><u>median</u>()</code>	Return the median value

---

<code><a href="#">median_item()</a></code>	Return median value and corresponding node
<code><a href="#">min()</a></code>	Return the minimum value
<code><a href="#">min_item()</a></code>	Return smallest value and corresponding node
<code><a href="#">nodes()</a></code>	Iterate over nodes
<code><a href="#">rolling(window[, step])</a></code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><a href="#">shrink_end(end)</a></code>	Set the end of the window to the smaller of end and self.end()
<code><a href="#">shrink_start(start)</a></code>	Set the start of the window to the larger of start and self.start()
<code><a href="#">shrink_window(start, end)</a></code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code><a href="#">snapshot_at(time)</a></code>	Create a view of the EarliestTimeView including all events that have not been explicitly deleted at time.
<code><a href="#">snapshot_latest()</a></code>	Create a view of the EarliestTimeView including all events that have not been explicitly deleted at the latest time.
<code><a href="#">sorted([reverse])</a></code>	Sort by value
<code><a href="#">sorted_by_id()</a></code>	Sort results by node id

---

<a href="#"><code>to_df()</code></a>	Convert results to pandas DataFrame
<a href="#"><code>top_k(k)</code></a>	Compute the k largest values
<a href="#"><code>valid_layers(names)</code></a>	Return a view of EarliestTimeView containing all layers names Any layers that do not exist are ignored
<a href="#"><code>values()</code></a>	Iterate over values
<a href="#"><code>window(start, end)</code></a>	Create a view of the EarliestTimeView including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

---

<a href="#"><code>end</code></a>	Gets the latest time that this EarliestTimeView is valid.
<a href="#"><code>end_date_time</code></a>	Gets the latest datetime that this EarliestTimeView is valid
<a href="#"><code>start</code></a>	Gets the start time for rolling and expanding windows for this EarliestTimeView
<a href="#"><code>start_date_time</code></a>	Gets the earliest datetime that this EarliestTimeView is valid
<a href="#"><code>window_size</code></a>	Get the window size (difference between start and end) for this EarliestTimeView
<a href="#"><code>after(start)</code></a>	Create a view of the EarliestTimeView including all events after start (exclusive).

---

#### Parameters:

`start` ([`TimeInput`](#)) – The start time of the window.

#### Return type:

[`EarliestTimeView`](#)

[`at\(time\)`](#)

Create a view of the EarliestTimeView including all events at time.

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[EarliestTimeView](#)

**before (end)**

Create a view of the EarliestTimeView including all events before end (exclusive).

**Parameters:**

end ([TimeInput](#)) – The end time of the window.

**Return type:**

[EarliestTimeView](#)

**bottom\_k (k)**

Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateOptionI64](#)

**collect ()**

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[list\[Optional\[int\]\]](#)

**compute ()**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateOptionI64](#)

**default\_layer ()**

**Returns:** Return a view of EarliestTimeView containing only the default edge layer :returns:

The layered view :rtype: EarliestTimeView

**end**

Gets the latest time that this EarliestTimeView is valid.

**Returns:**

The latest time that this EarliestTimeView is valid or None if the EarliestTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this EarliestTimeView is valid

**Returns:**

The latest datetime that this EarliestTimeView is valid or None if the EarliestTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of EarliestTimeView containing all layers except the excluded name

Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**exclude\_layers(names)**

Return a view of EarliestTimeView containing all layers except the excluded names

Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**`exclude_valid_layer(name)`**

Return a view of EarliestTimeView containing all layers except the excluded name  
:param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**`exclude_valid_layers(names)`**

Return a view of EarliestTimeView containing all layers except the excluded names  
:param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**`expanding(step)`**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**`get(node, default=...)`**

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[int\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[int\]\]](#)

**groups()**

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

**has\_layer(name)**

Check if EarliestTimeView has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**items()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[int\]\]\]](#)

**latest()**

Create a view of the EarliestTimeView including all events at the latest time.

**Return type:**

[EarliestTimeView](#)

**layer(name)**

Return a view of EarliestTimeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

name ([str](#)) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**layers(names)**

**Return a view of EarliestTimeView containing all layers names Errors if any of the layers do not exist.**

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**max ()**

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[int\]\]](#)

**max\_item ()**

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

**median ()**

Return the median value

**Return type:**

[Optional\[Optional\[int\]\]](#)

**median\_item ()**

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

**min ()**

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[int\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`shrink_end(end)`

Set the end of the window to the smaller of end and self.end()

**Parameters:**

`end (TimeInput)` – the new end time of the window

**Return type:**

[EarliestTimeView](#)

**`shrink_start(start)`**

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start` ([TimeInput](#)) – the new start time of the window

**Return type:**

[EarliestTimeView](#)

**`shrink_window(start, end)`**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start` ([TimeInput](#)) – the new start time for the window
- `end` ([TimeInput](#)) – the new end time for the window

**Return type:**

[EarliestTimeView](#)

**`snapshot_at(time)`**

Create a view of the EarliestTimeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[EarliestTimeView](#)

**`snapshot_latest()`**

Create a view of the EarliestTimeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[EarliestTimeView](#)

**`sorted(reverse=False)`**

Sort by value

**Parameters:**

`reverse` ([bool](#)) – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionI64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionI64](#)

`start`

Gets the start time for rolling and expanding windows for this EarliestTimeView

**Returns:**

The earliest time that this EarliestTimeView is valid or None if the EarliestTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this EarliestTimeView is valid

**Returns:**

The earliest datetime that this EarliestTimeView is valid or None if the EarliestTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOption64](#)

**valid\_layers(names)**

Return a view of EarliestTimeView containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[EarliestTimeView](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[int\]\]](#)

**window(start, end)**

Create a view of the EarliestTimeView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

[EarliestTimeView](#)

**window\_size**

Get the window size (difference between start and end) for this EarliestTimeView

**Return type:**

`Optional[int]`

# LatestTimeView

`class LatestTimeView`

Bases: `object`

A lazy view over node values

Methods:

`after(start)`

Create a view of the LatestTimeView including all events after start (exclusive).

`at(time)`

Create a view of the LatestTimeView including all events at time.

`before(end)`

Create a view of the LatestTimeView including all events before end (exclusive).

`bottom_k(k)`

Compute the k smallest values

`collect()`

Compute all values and return the result as a list

`compute()`

Compute all values and return the result as a node view

`default_layer()`

Return a view of LatestTimeView containing only the default edge layer :returns: The layered view :rtype: LatestTimeView

`exclude_layer(name)`

Return a view of LatestTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

---

<b><code>exclude_layers(names)</code></b>	Return a view of LatestTimeView containing all layers except the excluded names Errors if any of the layers do not exist.
<b><code>exclude_valid_layer(name)</code></b>	Return a view of LatestTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<b><code>exclude_valid_layers(names)</code></b>	Return a view of LatestTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<b><code>expanding(step)</code></b>	Creates a WindowSet with the given step size using an expanding window.
<b><code>get(node[, default])</code></b>	Get value for node
<b><code>groups()</code></b>	Group by value
<b><code>has_layer(name)</code></b>	Check if LatestTimeView has the layer "name"
<b><code>items()</code></b>	Iterate over items
<b><code>latest()</code></b>	Create a view of the LatestTimeView including all events at the latest time.
<b><code>layer(name)</code></b>	Return a view of LatestTimeView containing the layer "name" Errors if the layer does not exist
<b><code>layers(names)</code></b>	Return a view of LatestTimeView containing all layers names Errors if any of the layers do not exist.

---

---

<code>max()</code>	Return the maximum value
<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>rolling(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code>shrink_end(end)</code>	Set the end of the window to the smaller of end and self.end()
<code>shrink_start(start)</code>	Set the start of the window to the larger of start and self.start()
<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code>snapshot_at(time)</code>	Create a view of the LatestTimeView including all events that have not been explicitly deleted at time.

---

<code><a href="#">snapshot_latest()</a></code>	Create a view of the LatestTimeView including all events that have not been explicitly deleted at the latest time.
<code><a href="#">sorted([reverse])</a></code>	Sort by value
<code><a href="#">sorted_by_id()</a></code>	Sort results by node id
<code><a href="#">to_df()</a></code>	Convert results to pandas DataFrame
<code><a href="#">top_k(k)</a></code>	Compute the k largest values
<code><a href="#">valid_layers(names)</a></code>	Return a view of LatestTimeView containing all layers names Any layers that do not exist are ignored
<code><a href="#">values()</a></code>	Iterate over values
<code><a href="#">window(start, end)</a></code>	Create a view of the LatestTimeView including all events between start (inclusive) and end (exclusive)
<hr/>	
Attributes:	
<code><a href="#">end</a></code>	Gets the latest time that this LatestTimeView is valid.
<code><a href="#">end_date_time</a></code>	Gets the latest datetime that this LatestTimeView is valid
<code><a href="#">start</a></code>	Gets the start time for rolling and expanding windows for this LatestTimeView
<code><a href="#">start_date_time</a></code>	Gets the earliest datetime that this LatestTimeView is valid
<code><a href="#">window_size</a></code>	Get the window size (difference between start and end) for this LatestTimeView

---

**`after(start)`**

Create a view of the LatestTimeView including all events after start (exclusive).

**Parameters:**

`start (TimeInput)` – The start time of the window.

**Return type:**

[LatestTimeView](#)

**`at(time)`**

Create a view of the LatestTimeView including all events at time.

**Parameters:**

`time (TimeInput)` – The time of the window.

**Return type:**

[LatestTimeView](#)

**`before(end)`**

Create a view of the LatestTimeView including all events before end (exclusive).

**Parameters:**

`end (TimeInput)` – The end time of the window.

**Return type:**

[LatestTimeView](#)

**`bottom_k(k)`**

Compute the k smallest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateOptionI64](#)

**`collect()`**

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

`list[Optional\[int\]]`

**`compute()`**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateOptionI64](#)

`default_layer()`

Return a view of LatestTimeView containing only the default edge layer :returns: The layered view :rtype: LatestTimeView

`end`

Gets the latest time that this LatestTimeView is valid.

**Returns:**

The latest time that this LatestTimeView is valid or None if the LatestTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`end_date_time`

Gets the latest datetime that this LatestTimeView is valid

**Returns:**

The latest datetime that this LatestTimeView is valid or None if the LatestTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`exclude_layer(name)`

Return a view of LatestTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

`exclude_layers(names)`

Return a view of LatestTimeView containing all layers except the excluded names

Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

**exclude\_valid\_layer(name)**

Return a view of LatestTimeView containing all layers except the excluded name

:param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

**exclude\_valid\_layers(names)**

Return a view of LatestTimeView containing all layers except the excluded names

:param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**get(node, default=...)**

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[int\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[int\]\]](#)

**groups ()**

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

**has\_layer (name)**

Check if LatestTimeView has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**items ()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[int\]\]\]](#)

**latest ()**

Create a view of the LatestTimeView including all events at the latest time.

**Return type:**

[LatestTimeView](#)

**layer (name)**

Return a view of LatestTimeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

name ([str](#)) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

`layers (names)`

Return a view of LatestTimeView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

`max ()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[int\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

`median ()`

Return the median value

**Return type:**

[Optional\[Optional\[int\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[int\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[int\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`shrink_end(end)`

Set the end of the window to the smaller of end and self.end()

**Parameters:**end (*TimeInput*) – the new end time of the window**Return type:**[LatestTimeView](#)`shrink_start(start)`

Set the start of the window to the larger of start and self.start()

**Parameters:**start (*TimeInput*) – the new start time of the window**Return type:**[LatestTimeView](#)`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start (*TimeInput*) – the new start time for the window
- end (*TimeInput*) – the new end time for the window

**Return type:**[LatestTimeView](#)`snapshot_at(time)`

Create a view of the LatestTimeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**time (*TimeInput*) – The time of the window.**Return type:**[LatestTimeView](#)`snapshot_latest()`

Create a view of the LatestTimeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**[LatestTimeView](#)`sorted(reverse=False)`

**Sort by value**

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionI64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionI64](#)

`start`

Gets the start time for rolling and expanding windows for this LatestTimeView

**Returns:**

The earliest time that this LatestTimeView is valid or None if the LatestTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this LatestTimeView is valid

**Returns:**

The earliest datetime that this LatestTimeView is valid or None if the LatestTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionI64](#)

`valid_layers(names)`

Return a view of LatestTimeView containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[LatestTimeView](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[int\]\]](#)

`window(start, end)`

Create a view of the LatestTimeView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

[LatestTimeView](#)

## **window\_size**

Get the window size (difference between start and end) for this LatestTimeView

**Return type:**

[Optional\[int\]](#)

# NameView

**class NameView**

Bases: [object](#)

A lazy view over node values

**Methods:**

---

[\*\*bottom\\_k\(k\)\*\*](#) Compute the k smallest values

---

[\*\*collect\(\)\*\*](#) Compute all values and return the result as a list

---

[\*\*compute\(\)\*\*](#) Compute all values and return the result as a node view

---

[\*\*get\(node\[, default\]\)\*\*](#) Get value for node

---

[\*\*groups\(\)\*\*](#) Group by value

---

[\*\*items\(\)\*\*](#) Iterate over items

---

[\*\*max\(\)\*\*](#) Return the maximum value

---

[\*\*max\\_item\(\)\*\*](#) Return largest value and corresponding node

---

[\*\*median\(\)\*\*](#) Return the median value

---

[\*\*median\\_item\(\)\*\*](#) Return median value and corresponding node

---

<code><u>min</u>()</code>	Return the minimum value
<code><u>min_item</u>()</code>	Return smallest value and corresponding node
<code><u>nodes</u>()</code>	Iterate over nodes
<code><u>sorted</u>([reverse])</code>	Sort by value
<code><u>sorted_by_id</u>()</code>	Sort results by node id
<code><u>to_df</u>()</code>	Convert results to pandas DataFrame
<code><u>top_k</u>(k)</code>	Compute the k largest values
<code><u>values</u>()</code>	Iterate over values
<code><u>bottom_k</u>(k)</code>	Compute the k smallest values

**Parameters:**

k (`int`) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateString`

`collect()`

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

`list[str]`

`compute()`

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateString](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[str\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[str\]](#)

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, str\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[str\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[str\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[str\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateString](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateString](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateString](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[str\]](#)

# NodeStateString

**class NodeStateString**

Bases: [object](#)

Methods:

<a href="#"><b>bottom_k</b>(k)</a>	Compute the k smallest values
<a href="#"><b>get</b>(node[, default])</a>	Get value for node
<a href="#"><b>groups</b>()</a>	Group by value
<a href="#"><b>items</b>()</a>	Iterate over items
<a href="#"><b>max</b>()</a>	Return the maximum value
<a href="#"><b>max_item</b>()</a>	Return largest value and corresponding node
<a href="#"><b>median</b>()</a>	Return the median value
<a href="#"><b>median_item</b>()</a>	Return median value and corresponding node
<a href="#"><b>min</b>()</a>	Return the minimum value
<a href="#"><b>min_item</b>()</a>	Return smallest value and corresponding node
<a href="#"><b>nodes</b>()</a>	Iterate over nodes
<a href="#"><b>sorted</b>([reverse])</a>	Sort by value
<a href="#"><b>sorted_by_id</b>()</a>	Sort results by node id
<a href="#"><b>to_df</b>()</a>	Convert results to pandas DataFrame

---

**`top_k(k)`** Compute the k largest values

---

**`values()`** Iterate over values

---

**`bottom_k(k)`**

Compute the k smallest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateString](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional\[str\])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[str\]](#)

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, str\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[str\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[str\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[str\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, str\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateString](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateString](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateString](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[str\]](#)

# EarliestDateTimeView

**class** `EarliestDateTimeView`

Bases: [object](#)

A lazy view over node values

Methods:

<a href="#"><code>after</code>(start)</a>	Create a view of the EarliestDateTimeView including all events after start (exclusive).
<a href="#"><code>at</code>(time)</a>	Create a view of the EarliestDateTimeView including all events at time.
<a href="#"><code>before</code>(end)</a>	Create a view of the EarliestDateTimeView including all events before end (exclusive).
<a href="#"><code>bottom_k</code>(k)</a>	Compute the k smallest values
<a href="#"><code>collect</code>()</a>	Compute all values and return the result as a list
<a href="#"><code>compute</code>()</a>	Compute all values and return the result as a node view
<a href="#"><code>default_layer</code>()</a>	Return a view of EarliestDateTimeView containing only the default edge layer :returns: The layered view :rtype: EarliestDateTimeView
<a href="#"><code>exclude_layer</code>(name)</a>	Return a view of EarliestDateTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

---

<b>exclude_layers</b> (names)	Return a view of EarliestDateTimeView containing all layers except the excluded names Errors if any of the layers do not exist.
<b>exclude_valid_layer</b> (name)	Return a view of EarliestDateTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<b>exclude_valid_layers</b> (names)	Return a view of EarliestDateTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<b>expanding</b> (step)	Creates a WindowSet with the given step size using an expanding window.
<b>get</b> (node[, default])	Get value for node
<b>groups</b> ()	Group by value
<b>has_layer</b> (name)	Check if EarliestDateTimeView has the layer "name"
<b>items</b> ()	Iterate over items
<b>latest</b> ()	Create a view of the EarliestDateTimeView including all events at the latest time.
<b>layer</b> (name)	Return a view of EarliestDateTimeView containing the layer "name" Errors if the layer does not exist
<b>layers</b> (names)	Return a view of EarliestDateTimeView containing all layers names Errors if any of the layers do not exist.
<b>max</b> ()	Return the maximum value

---

---

<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>rolling(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code>shrink_end(end)</code>	Set the end of the window to the smaller of end and self.end()
<code>shrink_start(start)</code>	Set the start of the window to the larger of start and self.start()
<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code>snapshot_at(time)</code>	Create a view of the EarliestDateTimeView including all events that have not been explicitly deleted at time.
<code>snapshot_latest()</code>	Create a view of the EarliestDateTimeView including all events that have not been explicitly deleted at the latest time.
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame

---

---

<code>top_k(k)</code>	Compute the k largest values
<code>valid_layers(names)</code>	Return a view of EarliestDateTimeView containing all layers names Any layers that do not exist are ignored
<code>values()</code>	Iterate over values
<code>window(start, end)</code>	Create a view of the EarliestDateTimeView including all events between start (inclusive) and end (exclusive)

---

Attributes:

---

<code>end</code>	Gets the latest time that this EarliestDateTimeView is valid.
<code>end_date_time</code>	Gets the latest datetime that this EarliestDateTimeView is valid
<code>start</code>	Gets the start time for rolling and expanding windows for this EarliestDateTimeView
<code>start_date_time</code>	Gets the earliest datetime that this EarliestDateTimeView is valid
<code>window_size</code>	Get the window size (difference between start and end) for this EarliestDateTimeView

---

### `after(start)`

Create a view of the EarliestDateTimeView including all events after start (exclusive).

#### **Parameters:**

`start (TimeInput)` – The start time of the window.

#### **Return type:**

[EarliestDateTimeView](#)

### `at(time)`

Create a view of the EarliestDateTimeView including all events at time.

#### **Parameters:**

`time (TimeInput)` – The time of the window.

#### **Return type:**

## [EarliestDateTimeView](#)

### **before (end)**

Create a view of the EarliestDateTimeView including all events before end (exclusive).

#### **Parameters:**

end ([TimeInput](#)) – The end time of the window.

#### **Return type:**

## [EarliestDateTimeView](#)

### **bottom\_k (k)**

Compute the k smallest values

#### **Parameters:**

k ([int](#)) – The number of values to return

#### **Returns:**

The k smallest values as a node state

#### **Return type:**

## [NodeStateOptionDateTime](#)

### **collect()**

Compute all values and return the result as a list

#### **Returns:**

all values as a list

#### **Return type:**

## [list\[Optional\[datetime\]\]](#)

### **compute()**

Compute all values and return the result as a node view

#### **Returns:**

the computed NodeState

#### **Return type:**

## [NodeStateOptionDateTime](#)

### **default\_layer()**

Return a view of EarliestDateTimeView containing only the default edge layer :returns: The layered view :rtype: EarliestDateTimeView

### **end**

Gets the latest time that this EarliestDateTimeView is valid.

#### **Returns:**

The latest time that this EarliestDateTimeView is valid or None if the EarliestDateTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this EarliestDateTimeView is valid

**Returns:**

The latest datetime that this EarliestDateTimeView is valid or None if the EarliestDateTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of EarliestDateTimeView containing all layers except the excluded name  
Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

**exclude\_layers(names)**

Return a view of EarliestDateTimeView containing all layers except the excluded names  
Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

**exclude\_valid\_layer(name)**

Return a view of EarliestDateTimeView containing all layers except the excluded name  
:param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

**exclude\_valid\_layers(names)**

Return a view of EarliestDateTimeView containing all layers except the excluded names  
:param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

**expanding (step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**get (node, default=...)**

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[datetime\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

**groups ()**

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

**has\_layer (name)**

Check if EarliestDateTimeView has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**items ()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`latest()`

Create a view of the EarliestDateTimeView including all events at the latest time.

**Return type:**

[EarliestDateTimeView](#)

`layer(name)`

Return a view of EarliestDateTimeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

name ([str](#)) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

`layers(names)`

Return a view of EarliestDateTimeView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- window ([int](#) | [str](#)) – The size of the window.
- step ([int](#) | [str](#) | [None](#)) – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[EarliestDateTimeView](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

start ([TimeInput](#)) – the new start time of the window

**Return type:**

[EarliestDateTimeView](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start ([TimeInput](#)) – the new start time for the window
- end ([TimeInput](#)) – the new end time for the window

**Return type:**

[EarliestDateTimeView](#)

**snapshot\_at(time)**

Create a view of the EarliestDateTimeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[EarliestDateTimeView](#)

**snapshot\_latest()**

Create a view of the EarliestDateTimeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[EarliestDateTimeView](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionDateTime](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionDateTime](#)

`start`

Gets the start time for rolling and expanding windows for this EarliestDateTimeView

**Returns:**

The earliest time that this EarliestDateTimeView is valid or None if the EarliestDateTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this EarliestDateTimeView is valid

**Returns:**

The earliest datetime that this EarliestDateTimeView is valid or None if the EarliestDateTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionDateTime](#)

`valid_layers(names)`

Return a view of EarliestDateTimeView containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[EarliestDateTimeView](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[datetime\]\]](#)

`window(start, end)`

Create a view of the EarliestDateTimeView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

## [EarliestDateTimeView](#)

`window_size`

Get the window size (difference between start and end) for this EarliestDateTimeView

**Return type:**

[Optional\[int\]](#)

# LatestDateTimeView

**class LatestDateTimeView**

Bases: [object](#)

A lazy view over node values

Methods:

<a href="#"><code>after</code>(start)</a>	Create a view of the LatestDateTimeView including all events after start (exclusive).
<a href="#"><code>at</code>(time)</a>	Create a view of the LatestDateTimeView including all events at time.
<a href="#"><code>before</code>(end)</a>	Create a view of the LatestDateTimeView including all events before end (exclusive).
<a href="#"><code>bottom_k</code>(k)</a>	Compute the k smallest values
<a href="#"><code>collect</code>()</a>	Compute all values and return the result as a list
<a href="#"><code>compute</code>()</a>	Compute all values and return the result as a node view
<a href="#"><code>default_layer</code>()</a>	Return a view of LatestDateTimeView containing only the default edge layer :returns: The layered view :rtype: LatestDateTimeView
<a href="#"><code>exclude_layer</code>(name)</a>	Return a view of LatestDateTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

---

<b><code>exclude_layers</code></b> (names)	Return a view of LatestDateTimeView containing all layers except the excluded names Errors if any of the layers do not exist.
<b><code>exclude_valid_layer</code></b> (name)	Return a view of LatestDateTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<b><code>exclude_valid_layers</code></b> (names)	Return a view of LatestDateTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<b><code>expanding</code></b> (step)	Creates a WindowSet with the given step size using an expanding window.
<b><code>get</code></b> (node[, default])	Get value for node
<b><code>groups</code></b> ()	Group by value
<b><code>has_layer</code></b> (name)	Check if LatestDateTimeView has the layer "name"
<b><code>items</code></b> ()	Iterate over items
<b><code>latest</code></b> ()	Create a view of the LatestDateTimeView including all events at the latest time.
<b><code>layer</code></b> (name)	Return a view of LatestDateTimeView containing the layer "name" Errors if the layer does not exist
<b><code>layers</code></b> (names)	Return a view of LatestDateTimeView containing all layers names Errors if any of the layers do not exist.
<b><code>max</code></b> ()	Return the maximum value

---

---

<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>rolling(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code>shrink_end(end)</code>	Set the end of the window to the smaller of end and self.end()
<code>shrink_start(start)</code>	Set the start of the window to the larger of start and self.start()
<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code>snapshot_at(time)</code>	Create a view of the LatestDateTimeView including all events that have not been explicitly deleted at time.
<code>snapshot_latest()</code>	Create a view of the LatestDateTimeView including all events that have not been explicitly deleted at the latest time.
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame

---

---

<code>top_k(k)</code>	Compute the k largest values
<code>valid_layers(names)</code>	Return a view of LatestDateTimeView containing all layers names Any layers that do not exist are ignored
<code>values()</code>	Iterate over values
<code>window(start, end)</code>	Create a view of the LatestDateTimeView including all events between start (inclusive) and end (exclusive)

---

Attributes:

---

<code>end</code>	Gets the latest time that this LatestDateTimeView is valid.
<code>end_date_time</code>	Gets the latest datetime that this LatestDateTimeView is valid
<code>start</code>	Gets the start time for rolling and expanding windows for this LatestDateTimeView
<code>start_date_time</code>	Gets the earliest datetime that this LatestDateTimeView is valid
<code>window_size</code>	Get the window size (difference between start and end) for this LatestDateTimeView

---

### `after(start)`

Create a view of the LatestDateTimeView including all events after start (exclusive).

#### **Parameters:**

`start (TimeInput)` – The start time of the window.

#### **Return type:**

[LatestDateTimeView](#)

### `at(time)`

Create a view of the LatestDateTimeView including all events at time.

#### **Parameters:**

`time (TimeInput)` – The time of the window.

#### **Return type:**

[LatestDateTimeView](#)

**before (end)**

Create a view of the LatestDateTimeView including all events before end (exclusive).

**Parameters:**

end ([TimeInput](#)) – The end time of the window.

**Return type:**

[LatestDateTimeView](#)

**bottom\_k (k)**

Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateOptionDateTime](#)

**collect ()**

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[list\[Optional\[datetime\]\]](#)

**compute ()**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateOptionDateTime](#)

**default\_layer ()**

Return a view of LatestDateTimeView containing only the default edge layer :returns: The layered view :rtype: LatestDateTimeView

**end**

Gets the latest time that this LatestDateTimeView is valid.

**Returns:**

The latest time that this LatestDateTimeView is valid or None if the LatestDateTimeView is valid for all times.

**Return type:**

Optional[int]

**end\_date\_time**

Gets the latest datetime that this LatestDateTimeView is valid

**Returns:**

The latest datetime that this LatestDateTimeView is valid or None if the LatestDateTimeView is valid for all times.

**Return type:**

Optional[datetime]

**exclude\_layer(name)**

Return a view of LatestDateTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name (str) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

LatestDateTimeView

**exclude\_layers(names)**

Return a view of LatestDateTimeView containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names (list[str]) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

LatestDateTimeView

**exclude\_valid\_layer(name)**

Return a view of LatestDateTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

LatestDateTimeView

**exclude\_valid\_layers(names)**

Return a view of LatestDateTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[LatestDateTimeView](#)

**expanding (step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**get (node, default=...)**

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[datetime\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

**groups ()**

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

**has\_layer (name)**

Check if LatestDateTimeView has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**items ()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`latest()`

Create a view of the LatestDateTimeView including all events at the latest time.

**Return type:**

[LatestDateTimeView](#)

`layer(name)`

Return a view of LatestDateTimeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

`name (str)` – then name of the layer.

**Returns:**

The layered view

**Return type:**

[LatestDateTimeView](#)

`layers(names)`

Return a view of LatestDateTimeView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[LatestDateTimeView](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- window ([int](#) | [str](#)) – The size of the window.
- step ([int](#) | [str](#) | [None](#)) – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[LatestDateTimeView](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

start ([TimeInput](#)) – the new start time of the window

**Return type:**

[LatestDateTimeView](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start ([TimeInput](#)) – the new start time for the window
- end ([TimeInput](#)) – the new end time for the window

**Return type:**

[LatestDateTimeView](#)

**snapshot\_at(time)**

Create a view of the LatestDateTimeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[LatestDateTimeView](#)

**snapshot\_latest()**

Create a view of the LatestDateTimeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[LatestDateTimeView](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionDateTime](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionDateTime](#)

`start`

Gets the start time for rolling and expanding windows for this LatestDateTimeView

**Returns:**

The earliest time that this LatestDateTimeView is valid or None if the LatestDateTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this LatestDateTimeView is valid

**Returns:**

The earliest datetime that this LatestDateTimeView is valid or None if the LatestDateTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionDateTime](#)

`valid_layers(names)`

Return a view of LatestDateTimeView containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[LatestDateTimeView](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[datetime\]\]](#)

`window(start, end)`

Create a view of the LatestDateTimeView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

## [LatestDateTimeView](#)

`window_size`

Get the window size (difference between start and end) for this LatestDateTimeView

**Return type:**

[Optional\[int\]](#)

# NodeStateOptionDateTime

`class NodeStateOptionDateTime`

Bases: [object](#)

Methods:

---

`bottom_k(k)` Compute the k smallest values

---

`get(node[, default])` Get value for node

---

`groups()` Group by value

---

`items()` Iterate over items

---

`max()` Return the maximum value

---

`max_item()` Return largest value and corresponding node

---

`median()` Return the median value

---

`median_item()` Return median value and corresponding node

---

`min()` Return the minimum value

---

`min_item()` Return smallest value and corresponding node

---

`nodes()` Iterate over nodes

---

<code><a href="#">sorted</a>([reverse])</code>	Sort by value
<code><a href="#">sorted_by_id</a>()</code>	Sort results by node id
<code><a href="#">to_df</a>()</code>	Convert results to pandas DataFrame
<code><a href="#">top_k</a>(k)</code>	Compute the k largest values
<code><a href="#">values</a>()</code>	Iterate over values
<b>bottom_k(k)</b>	Compute the k smallest values

---

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateOptionDateTime`

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional\[Optional\[datetime\]\])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional\[Optional\[datetime\]\]`

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

`NodeGroups`

`items()`

**Iterate over items**

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[datetime\]\]\]](#)

**max()**

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

**max\_item()**

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

**median()**

Return the median value

**Return type:**

[Optional\[Optional\[datetime\]\]](#)

**median\_item()**

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[datetime\]\]\]](#)

**min()**

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

`Optional[Optional[datetime]]`

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

`Optional[Tuple[Node, Optional[datetime]]]`

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

`Nodes`

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

`NodeStateOptionDateTime`

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

`NodeStateOptionDateTime`

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**top\_k(k)**

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionDate](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[datetime\]\]](#)

## HistoryView

`class HistoryView`

Bases: [object](#)

A lazy view over node values

Methods:

[after\(start\)](#)

Create a view of the HistoryView including all events after start (exclusive).

[at\(time\)](#)

Create a view of the HistoryView including all events at time.

[before\(end\)](#)

Create a view of the HistoryView including all events before end (exclusive).

---

<code><u>bottom_k</u>(k)</code>	Compute the k smallest values
<code><u>collect</u>()</code>	Compute all values and return the result as a list
<code><u>compute</u>()</code>	Compute all values and return the result as a node view
<code><u>default_layer</u>()</code>	Return a view of HistoryView containing only the default edge layer :returns: The layered view :rtype: HistoryView
<code><u>exclude_layer</u>(name)</code>	Return a view of HistoryView containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of HistoryView containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of HistoryView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of HistoryView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>get</u>(node[, default])</code>	Get value for node
<code><u>has_layer</u>(name)</code>	Check if HistoryView has the layer "name"

---

---

<code><a href="#">items()</a></code>	Iterate over items
<code><a href="#">latest()</a></code>	Create a view of the HistoryView including all events at the latest time.
<code><a href="#">layer(name)</a></code>	Return a view of HistoryView containing the layer "name" Errors if the layer does not exist
<code><a href="#">layers(names)</a></code>	Return a view of HistoryView containing all layers names Errors if any of the layers do not exist.
<code><a href="#">max()</a></code>	Return the maximum value
<code><a href="#">max_item()</a></code>	Return largest value and corresponding node
<code><a href="#">median()</a></code>	Return the median value
<code><a href="#">median_item()</a></code>	Return median value and corresponding node
<code><a href="#">min()</a></code>	Return the minimum value
<code><a href="#">min_item()</a></code>	Return smallest value and corresponding node
<code><a href="#">nodes()</a></code>	Iterate over nodes
<code><a href="#">rolling(window[, step])</a></code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><a href="#">shrink_end(end)</a></code>	Set the end of the window to the smaller of end and self.end()

---

<code>shrink_start</code> (start)	Set the start of the window to the larger of start and self.start()
<code>shrink_window</code> (start, end)	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code>snapshot_at</code> (time)	Create a view of the HistoryView including all events that have not been explicitly deleted at time.
<code>snapshot_latest</code> ()	Create a view of the HistoryView including all events that have not been explicitly deleted at the latest time.
<code>sorted</code> ([reverse])	Sort by value
<code>sorted_by_id</code> ()	Sort results by node id
<code>to_df</code> ()	Convert results to pandas DataFrame
<code>top_k</code> (k)	Compute the k largest values
<code>valid_layers</code> (names)	Return a view of HistoryView containing all layers names Any layers that do not exist are ignored
<code>values</code> ()	Iterate over values
<code>window</code> (start, end)	Create a view of the HistoryView including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

<code>end</code>	Gets the latest time that this HistoryView is valid.
------------------	--

---

<code>end_date_time</code>	Gets the latest datetime that this HistoryView is valid
<code>start</code>	Gets the start time for rolling and expanding windows for this HistoryView
<code>start_date_time</code>	Gets the earliest datetime that this HistoryView is valid
<code>window_size</code>	Get the window size (difference between start and end) for this HistoryView

---

**`after(start)`**  
Create a view of the HistoryView including all events after start (exclusive).

**Parameters:**

`start` ([TimeInput](#)) – The start time of the window.

**Return type:**

[HistoryView](#)

**`at(time)`**

Create a view of the HistoryView including all events at time.

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[HistoryView](#)

**`before(end)`**

Create a view of the HistoryView including all events before end (exclusive).

**Parameters:**

`end` ([TimeInput](#)) – The end time of the window.

**Return type:**

[HistoryView](#)

**`bottom_k(k)`**

Compute the k smallest values

**Parameters:**

`k` ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateListI64](#)

**collect()**

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[list\[list\[int\]\]](#)

**compute()**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateListI64](#)

**default\_layer()**

Return a view of HistoryView containing only the default edge layer :returns: The layered view :rtype: HistoryView

**end**

Gets the latest time that this HistoryView is valid.

**Returns:**

The latest time that this HistoryView is valid or None if the HistoryView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this HistoryView is valid

**Returns:**

The latest datetime that this HistoryView is valid or None if the HistoryView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of HistoryView containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name (*str*) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[HistoryView](#)

**exclude\_layers(names)**

Return a view of HistoryView containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names (*list[str]*) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[HistoryView](#)

**exclude\_valid\_layer(name)**

Return a view of HistoryView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[HistoryView](#)

**exclude\_valid\_layers(names)**

Return a view of HistoryView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[HistoryView](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step (*int* | *str*) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[list\[int\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[int\]\]](#)

`has_layer(name)`

Check if HistoryView has the layer “name”

**Parameters:**

name (*str*) – the name of the layer to check

**Return type:**

`bool`

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[int\]\]\]](#)

`latest()`

Create a view of the HistoryView including all events at the latest time.

**Return type:**

[HistoryView](#)

`layer(name)`

**Return a view of HistoryView containing the layer “name” Errors if the layer does not exist**

**Parameters:**

`name (str) – then name of the layer.`

**Returns:**

`The layered view`

**Return type:**

[HistoryView](#)

`layers (names)`

**Return a view of HistoryView containing all layers names Errors if any of the layers do not exist.**

**Parameters:**

`names (list[str]) – list of layer names for the new view`

**Returns:**

`The layered view`

**Return type:**

[HistoryView](#)

`max ()`

**Return the maximum value**

**Returns:**

`The maximum value or None if empty`

**Return type:**

[Optional\[list\[int\]\]](#)

`max_item ()`

**Return largest value and corresponding node**

**Returns:**

`The Node and maximum value or None if empty`

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`median ()`

**Return the median value**

**Return type:**

[Optional\[list\[int\]\]](#)

## `median_item()`

Return median value and corresponding node

### Returns:

The median value or None if empty

### Return type:

`Optional[Tuple[Node, list[int]]]`

## `min()`

Return the minimum value

### Returns:

The minimum value or None if empty

### Return type:

`Optional[list[int]]`

## `min_item()`

Return smallest value and corresponding node

### Returns:

The Node and minimum value or None if empty

### Return type:

`Optional[Tuple[Node, list[int]]]`

## `nodes()`

Iterate over nodes

### Returns:

The nodes

### Return type:

`Nodes`

## `rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

### Parameters:

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

### Returns:

A WindowSet object.

**Return type:**

[WindowSet](#)

`shrink_end(end)`

Set the end of the window to the smaller of end and self.end()

**Parameters:**

`end (TimeInput)` – the new end time of the window

**Return type:**

[HistoryView](#)

`shrink_start(start)`

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start (TimeInput)` – the new start time of the window

**Return type:**

[HistoryView](#)

`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start (TimeInput)` – the new start time for the window
- `end (TimeInput)` – the new end time for the window

**Return type:**

[HistoryView](#)

`snapshot_at(time)`

Create a view of the HistoryView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

`time (TimeInput)` – The time of the window.

**Return type:**

[HistoryView](#)

`snapshot_latest()`

Create a view of the HistoryView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[HistoryView](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateListI64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateListI64](#)

`start`

Gets the start time for rolling and expanding windows for this HistoryView

**Returns:**

The earliest time that this HistoryView is valid or None if the HistoryView is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this HistoryView is valid

**Returns:**

The earliest datetime that this HistoryView is valid or None if the HistoryView is valid for all times.

**Return type:**

`Optional[datetime]`

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

`DataFrame`

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

`NodeStateListI64`

`valid_layers(names)`

Return a view of HistoryView containing all layers names Any layers that do not exist are ignored

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

`HistoryView`

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

`Iterator[list[int]]`

`window(start, end)`

Create a view of the HistoryView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start` (`TimeInput` | `None`) – The start time of the window (unbounded if `None`).
- `end` (`TimeInput` | `None`) – The end time of the window (unbounded if `None`).

**Return type:**

`HistoryView`

`window_size`

Get the window size (difference between start and end) for this HistoryView

**Return type:**

`Optional[int]`

# NodeStateListI64

`class NodeStateListI64`

Bases: `object`

Methods:

<code>bottom_k(k)</code>	Compute the k smallest values
<code>get(node[, default])</code>	Get value for node
<code>items()</code>	Iterate over items
<code>max()</code>	Return the maximum value
<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node

---

<code><a href="#">nodes()</a></code>	Iterate over nodes
<code><a href="#">sorted([reverse])</a></code>	Sort by value
<code><a href="#">sorted_by_id()</a></code>	Sort results by node id
<code><a href="#">to_df()</a></code>	Convert results to pandas DataFrame
<code><a href="#">top_k(k)</a></code>	Compute the k largest values
<code><a href="#">values()</a></code>	Iterate over values
<b>bottom_k (k)</b>	Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateListI64](#)

`get\(node, default=...\)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[list\[int\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[int\]\]](#)

`items\(\)`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[int\]\]\]](#)

`max\(\)`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[list\[int\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[list\[int\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[list\[int\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateListI64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateListI64](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateListI64](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[list\[int\]\]](#)

# HistoryDateTimeView

`class HistoryDateTimeView`

Bases: [object](#)

A lazy view over node values

**Methods:**

[`after\(start\)`](#)

Create a view of the HistoryDateTimeView including all events after start (exclusive).

[`at\(time\)`](#)

Create a view of the HistoryDateTimeView including all events at time.

[`before\(end\)`](#)

Create a view of the HistoryDateTimeView including all events before end (exclusive).

[`bottom\_k\(k\)`](#)

Compute the k smallest values

[`collect\(\)`](#)

Compute all values and return the result as a list

[`compute\(\)`](#)

Compute all values and return the result as a node view

[`default\_layer\(\)`](#)

Return a view of HistoryDateTimeView containing only the default edge layer  
:returns: The layered view :rtype:  
HistoryDateTimeView

<code><u>exclude_layer</u>(name)</code>	Return a view of HistoryDateTimeView containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of HistoryDateTimeView containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of HistoryDateTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of HistoryDateTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>get</u>(node[, default])</code>	Get value for node
<code><u>has_layer</u>(name)</code>	Check if HistoryDateTimeView has the layer "name"
<code><u>items</u>()</code>	Iterate over items
<code><u>latest</u>()</code>	Create a view of the HistoryDateTimeView including all events at the latest time.
<code><u>layer</u>(name)</code>	Return a view of HistoryDateTimeView containing the layer "name" Errors if the layer does not exist

---

<code><a href="#">layers</a>(names)</code>	Return a view of HistoryDateTimeView containing all layers names Errors if any of the layers do not exist.
<code><a href="#">max()</a></code>	Return the maximum value
<code><a href="#">max_item()</a></code>	Return largest value and corresponding node
<code><a href="#">median()</a></code>	Return the median value
<code><a href="#">median_item()</a></code>	Return median value and corresponding node
<code><a href="#">min()</a></code>	Return the minimum value
<code><a href="#">min_item()</a></code>	Return smallest value and corresponding node
<code><a href="#">nodes()</a></code>	Iterate over nodes
<code><a href="#">rolling</a>(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><a href="#">shrink_end</a>(end)</code>	Set the end of the window to the smaller of end and self.end()
<code><a href="#">shrink_start</a>(start)</code>	Set the start of the window to the larger of start and self.start()
<code><a href="#">shrink_window</a>(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)

---

---

**`snapshot_at`(time)**

Create a view of the HistoryDateTimeView including all events that have not been explicitly deleted at time.

---

**`snapshot_latest()`**

Create a view of the HistoryDateTimeView including all events that have not been explicitly deleted at the latest time.

---

**`sorted`([reverse])**

Sort by value

---

**`sorted_by_id()`**

Sort results by node id

---

**`to_df()`**

Convert results to pandas DataFrame

---

**`top_k`(k)**

Compute the k largest values

---

**`valid_layers`(names)**

Return a view of HistoryDateTimeView containing all layers names Any layers that do not exist are ignored

---

**`values()`**

Iterate over values

---

**`window`(start, end)**

Create a view of the HistoryDateTimeView including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

---

**`end`**

Gets the latest time that this HistoryDateTimeView is valid.

---

**`end_date_time`**

Gets the latest datetime that this HistoryDateTimeView is valid

---

**`start`**

Gets the start time for rolling and expanding windows for this HistoryDateTimeView

---

**`start_date_time`** Gets the earliest datetime that this HistoryDateTimeView is valid

---

**`window_size`** Get the window size (difference between start and end) for this HistoryDateTimeView

**`after(start)`**

Create a view of the HistoryDateTimeView including all events after start (exclusive).

**Parameters:**

`start` ([TimeInput](#)) – The start time of the window.

**Return type:**

[HistoryDateTimeView](#)

**`at(time)`**

Create a view of the HistoryDateTimeView including all events at time.

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[HistoryDateTimeView](#)

**`before(end)`**

Create a view of the HistoryDateTimeView including all events before end (exclusive).

**Parameters:**

`end` ([TimeInput](#)) – The end time of the window.

**Return type:**

[HistoryDateTimeView](#)

**`bottom_k(k)`**

Compute the k smallest values

**Parameters:**

`k` ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateOptionListDateTime](#)

**`collect()`**

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[list\[Optional\[list\[datetime\]\]\]](#)

**compute()**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateOptionListDate Time](#)

**default\_layer()**

Return a view of HistoryDateTimeView containing only the default edge layer

:returns: The layered view :rtype: HistoryDateTimeView

**end**

Gets the latest time that this HistoryDateTimeView is valid.

**Returns:**

The latest time that this HistoryDateTimeView is valid or None if the HistoryDateTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this HistoryDateTimeView is valid

**Returns:**

The latest datetime that this HistoryDateTimeView is valid or None if the HistoryDateTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of HistoryDateTimeView containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

**exclude\_layers(names)**

Return a view of HistoryDateTimeView containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

**exclude\_valid\_layer(name)**

Return a view of HistoryDateTimeView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

**exclude\_valid\_layers(names)**

Return a view of HistoryDateTimeView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[Optional\[list\[datetime\]\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

`has_layer(name)`

Check if HistoryDateTimeView has the layer “name”

**Parameters:**

`name` ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

`latest()`

Create a view of the HistoryDateTimeView including all events at the latest time.

**Return type:**

[HistoryDateTimeView](#)

`layer(name)`

Return a view of HistoryDateTimeView containing the layer “name” Errors if the layer does not exist

**Parameters:**

name ([str](#)) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

`layers (names)`

Return a view of HistoryDateTimeView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

`max ()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

`max_item ()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

`median ()`

Return the median value

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

`median_item ()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`rolling(window, step=None)`

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[HistoryDateTimeView](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

start ([TimeInput](#)) – the new start time of the window

**Return type:**

[HistoryDateTimeView](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start ([TimeInput](#)) – the new start time for the window
- end ([TimeInput](#)) – the new end time for the window

**Return type:**

[HistoryDateTimeView](#)

**snapshot\_at(time)**

Create a view of the HistoryDateTimeView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[HistoryDateTimeView](#)

**snapshot\_latest()**

Create a view of the HistoryDateTimeView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[HistoryDateTimeView](#)

**sorted(*reverse=False*)**

Sort by value

**Parameters:**

**reverse (*bool*)** – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionListDateTime](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionListDateTime](#)

**start**

Gets the start time for rolling and expanding windows for this HistoryDateTimeView

**Returns:**

The earliest time that this HistoryDateTimeView is valid or None if the HistoryDateTimeView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this HistoryDateTimeView is valid

**Returns:**

The earliest datetime that this HistoryDateTimeView is valid or None if the HistoryDateTimeView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionListDateTime](#)

`valid_layers(names)`

Return a view of HistoryDateTimeView containing all layers names Any layers that do not exist are ignored

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[HistoryDateTimeView](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[list\[datetime\]\]\]](#)

`window(start, end)`

Create a view of the HistoryDateTimeView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start` (`TimeInput` | `None`) – The start time of the window (unbounded if `None`).
- `end` (`TimeInput` | `None`) – The end time of the window (unbounded if `None`).

**Return type:**

[HistoryDateTimeView](#)

**window\_size**

Get the window size (difference between start and end) for this `HistoryDateTimeView`

**Return type:**

[Optional\[int\]](#)

# NodeStateOptionListDateTime

`class NodeStateOptionListDateTime`

Bases: [object](#)

**Methods:**

---

[`bottom\_k`\(k\)](#) Compute the k smallest values

---

[`get`\(node\[, default\]\)](#) Get value for node

---

[`items`\(\)](#) Iterate over items

---

[`max`\(\)](#) Return the maximum value

---

[`max\_item`\(\)](#) Return largest value and corresponding node

---

[`median`\(\)](#) Return the median value

---

[`median\_item`\(\)](#) Return median value and corresponding node

---

[`min`\(\)](#) Return the minimum value

---

[`min\_item`\(\)](#) Return smallest value and corresponding node

---

[`nodes`\(\)](#) Iterate over nodes

---

<code><a href="#">sorted</a>([reverse])</code>	Sort by value
<code><a href="#">sorted_by_id</a>()</code>	Sort results by node id
<code><a href="#">to_df</a>()</code>	Convert results to pandas DataFrame
<code><a href="#">top_k</a>(k)</code>	Compute the k largest values
<code><a href="#">values</a>()</code>	Iterate over values
<b>bottom_k(k)</b>	Compute the k smallest values

---

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateOptionListDateTime`

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional\[Optional\[list\[datetime\]\]\])` – the default value. Defaults to `None`.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional\[Optional\[list\[datetime\]\]\]`

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]`

**max ()**

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

**max\_item()**

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

**median()**

Return the median value

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

**median\_item()**

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

**min()**

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[list\[datetime\]\]\]](#)

**min\_item()**

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[list\[datetime\]\]\]\]](#)

**nodes ()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted(reverse=False)**

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionListDateTime](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionListDateTime](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**top\_k(k)**

Compute the k largest values

**Parameters:**

`k` ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionListDateTime](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[list\[datetime\]\]\]](#)

# NodeTypeView

`class NodeTypeView`

Bases: [object](#)

A lazy view over node values

**Methods:**

---

[`bottom\_k\(k\)`](#) Compute the k smallest values

---

[`collect\(\)`](#) Compute all values and return the result as a list

---

[`compute\(\)`](#) Compute all values and return the result as a node view

---

[`get\(node\[, default\]\)`](#) Get value for node

---

[`groups\(\)`](#) Group by value

---

[`items\(\)`](#) Iterate over items

---

<code>max()</code>	Return the maximum value
<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame
<code>top_k(k)</code>	Compute the k largest values
<code>values()</code>	Iterate over values
<b>bottom_k(k)</b> Compute the k smallest values	

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateOptionStr`

`collect()`

Compute all values and return the result as a list

**Returns:**

all values as a list

**Return type:**

[list\[Optional\[str\]\]](#)

**compute()**

Compute all values and return the result as a node view

**Returns:**

the computed NodeState

**Return type:**

[NodeStateOptionStr](#)

**get(node, default=...)**

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Optional\[str\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Optional\[str\]\]](#)

**groups()**

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

**items()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[str\]\]\]](#)

**max()**

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[str\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[Optional\[str\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[str\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`nodes()`

**Iterate over nodes**

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionStr](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionStr](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionStr](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[str\]\]](#)

# NodeStateOptionStr

`class NodeStateOptionStr`

Bases: [object](#)

**Methods:**

[`bottom\_k\(k\)`](#) Compute the k smallest values

---

[`get\(node\[, default\]\)`](#) Get value for node

---

[`groups\(\)`](#) Group by value

---

[`items\(\)`](#) Iterate over items

---

[`max\(\)`](#) Return the maximum value

---

[`max\_item\(\)`](#) Return largest value and corresponding node

---

[`median\(\)`](#) Return the median value

---

[`median\_item\(\)`](#) Return median value and corresponding node

---

[`min\(\)`](#) Return the minimum value

---

`min_item()`      Return smallest value and corresponding node

---

`nodes()`      Iterate over nodes

---

`sorted([reverse])`      Sort by value

---

`sorted_by_id()`      Sort results by node id

---

`to_df()`      Convert results to pandas DataFrame

---

`top_k(k)`      Compute the k largest values

---

`values()`      Iterate over values

---

### `bottom_k(k)`

Compute the k smallest values

**Parameters:**

k (`int`) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateOptionStr`

`get(node, default=...)`

Get value for node

**Parameters:**

- node (`NodeInput`) – the node
- default (`Optional[Optional[str]]`) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional[Optional[str]]`

`groups()`

Group by value

**Returns:**

The grouped nodes

**Return type:**

[NodeGroups](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Optional\[str\]\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Optional\[str\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[Optional\[str\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Optional\[str\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Optional\[str\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateOptionStr](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateOptionStr](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateOptionStr](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Optional\[str\]\]](#)

## NodeStateListDateTime

`class NodeStateListDateTime`

Bases: [object](#)

Methods:

---

`bottom_k(k)` Compute the k smallest values

---

`get(node[, default])` Get value for node

---

`items()` Iterate over items

---

`max()` Return the maximum value

---

<code>max_item()</code>	Return largest value and corresponding node
<code>median()</code>	Return the median value
<code>median_item()</code>	Return median value and corresponding node
<code>min()</code>	Return the minimum value
<code>min_item()</code>	Return smallest value and corresponding node
<code>nodes()</code>	Iterate over nodes
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame
<code>top_k(k)</code>	Compute the k largest values
<code>values()</code>	Iterate over values
<code>bottom_k(k)</code>	Compute the k smallest values

---

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateListDateTime`

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional\[list/datetime\])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[datetime\]\]](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[datetime\]\]\]](#)

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[list\[datetime\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[datetime\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[list\[datetime\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[datetime\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[list\[datetime\]\]](#)

**min\_item()**

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[datetime\]\]\]](#)

**nodes()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted(*reverse=False*)**

Sort by value

**Parameters:**

*reverse (bool)* – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateListDateTime](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateListDateTime](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

## `top_k(k)`

Compute the k largest values

### **Parameters:**

`k (int)` – The number of values to return

### **Returns:**

The k largest values as a node state

### **Return type:**

[NodeStateListDateTime](#)

## `values()`

Iterate over values

### **Returns:**

Iterator over values

### **Return type:**

[Iterator\[list\[datetime\]\]](#)

# NodeStateWeightedSP

## `class NodeStateWeightedSP`

Bases: [object](#)

### **Methods:**

<code>get(node[, default])</code>	Get value for node
-----------------------------------	--------------------

---

<code>items()</code>	Iterate over items
----------------------	--------------------

---

<code>nodes()</code>	Iterate over nodes
----------------------	--------------------

---

<code>sorted_by_id()</code>	Sort results by node id
-----------------------------	-------------------------

---

<code>to_df()</code>	Convert results to pandas DataFrame
----------------------	-------------------------------------

---

<code>values()</code>	Iterate over values
-----------------------	---------------------

## `get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[Tuple\[float, Nodes\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[Tuple\[float, Nodes\]\]](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, Tuple\[float, Nodes\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateWeightedSP](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Tuple\[float, Nodes\]\]](#)

# NodeStateF64

**class NodeStateF64**

Bases: [object](#)

**Methods:**

[bottom\\_k\(k\)](#)

Compute the k smallest values

[get\(node\[, default\]\)](#)

Get value for node

[items\(\)](#)

Iterate over items

[max\(\)](#)

Return the maximum value

[max\\_item\(\)](#)

Return largest value and corresponding node

[mean\(\)](#)

mean of values over all nodes

[median\(\)](#)

Return the median value

[median\\_item\(\)](#)

Return median value and corresponding node

[min\(\)](#)

Return the minimum value

[min\\_item\(\)](#)

Return smallest value and corresponding node

[nodes\(\)](#)

Iterate over nodes

---

<code><u>sorted</u>([reverse])</code>	Sort by value
<code><u>sorted_by_id</u>()</code>	Sort results by node id
<code><u>sum</u>()</code>	sum of values over all nodes
<code><u>to_df</u>()</code>	Convert results to pandas DataFrame
<code><u>top_k</u>(k)</code>	Compute the k largest values
<code><u>values</u>()</code>	Iterate over values
<code><u>bottom_k</u>(k)</code>	Compute the k smallest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

[NodeStateF64](#)

`get(node, default=...)`

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[float\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[float\]](#)

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator[Tuple[Node, float]]`

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

`Optional[float]`

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

`Optional[Tuple[Node, float]]`

`mean()`

mean of values over all nodes

**Returns:**

mean value

**Return type:**

`float`

`median()`

Return the median value

**Return type:**

`Optional[float]`

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

`Optional[Tuple[Node, float]]`

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[float\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, float\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

`sorted(reverse=False)`

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateF64](#)

`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateF64](#)

`sum()`

sum of values over all nodes

**Returns:**

the sum

**Return type:**

[float](#)

`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

`top_k(k)`

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateF64](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[float\]](#)

# NodeStateNodes

`class NodeStateNodes`

Bases: [object](#)

Methods:

<code>get(node[, default])</code>	Get value for node
<code>items()</code>	Iterate over items
<code>nodes()</code>	Iterate over nodes
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame
<code>values()</code>	Iterate over values
<code>get(node, default=...)</code>	Get value for node

#### Parameters:

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[Nodes\]](#)) – the default value. Defaults to None.

#### Returns:

the value for the node or the default value

#### Return type:

[Optional\[Nodes\]](#)

#### items()

Iterate over items

#### Returns:

Iterator over items

#### Return type:

[Iterator\[Tuple\[Node, Nodes\]\]](#)

#### nodes()

Iterate over nodes

#### Returns:

The nodes

#### Return type:

[Nodes](#)

#### sorted\_by\_id()

Sort results by node id

**Returns:**

The sorted node state

**Return type:**[NodeStateNodes](#)`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**[DataFrame](#)`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**[Iterator\[Nodes\]](#)

# NodeStateReachability

`class NodeStateReachability`Bases: [object](#)**Methods:**

---

<a href="#">get</a> (node[, default])	Get value for node
---------------------------------------	--------------------

---

<a href="#">items</a> ()	Iterate over items
--------------------------	--------------------

---

<a href="#">nodes</a> ()	Iterate over nodes
--------------------------	--------------------

---

<a href="#">sorted_by_id</a> ()	Sort results by node id
---------------------------------	-------------------------

---

<a href="#">to_df</a> ()	Convert results to pandas DataFrame
--------------------------	-------------------------------------

---

**`values()`** Iterate over values

**`get(node, default=...)`**

Get value for node

**Parameters:**

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[list\[Tuple\[int, str\]\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[Tuple\[int, str\]\]\]](#)

**`items()`**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[Tuple\[int, str\]\]\]\]](#)

**`nodes()`**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**`sorted_by_id()`**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateReachability](#)

**`to_df()`**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**[DataFrame](#)[values\(\)](#)

Iterate over values

**Returns:**

Iterator over values

**Return type:**[Iterator\[list\[Tuple\[int, str\]\]\]](#)

# NodeStateListF64

`class NodeStateListF64`Bases: [object](#)**Methods:**

---

[`get\(node\[, default\]\)`](#) Get value for node

---

[`items\(\)`](#) Iterate over items

---

[`nodes\(\)`](#) Iterate over nodes

---

[`sorted\_by\_id\(\)`](#) Sort results by node id

---

[`to\_df\(\)`](#) Convert results to pandas DataFrame

---

[`values\(\)`](#) Iterate over values

---

[`get\(node, default=...\)`](#)

Get value for node

**Parameters:**

- node ([NodeInput](#)) – the node
- default ([Optional\[list\[float\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[float\]\]](#)

**items ()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[float\]\]\]](#)

**nodes ()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted\_by\_id ()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateListF64](#)

**to\_df ()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**values ()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[list\[float\]\]](#)

# NodeStateMotifs

`class NodeStateMotifs`

Bases: [object](#)

**Methods:**

---

[`bottom\_k\(k\)`](#) Compute the k smallest values

---

[`get\(node\[, default\]\)`](#) Get value for node

---

[`items\(\)`](#) Iterate over items

---

[`max\(\)`](#) Return the maximum value

---

[`max\_item\(\)`](#) Return largest value and corresponding node

---

[`median\(\)`](#) Return the median value

---

[`median\_item\(\)`](#) Return median value and corresponding node

---

[`min\(\)`](#) Return the minimum value

---

[`min\_item\(\)`](#) Return smallest value and corresponding node

---

[`nodes\(\)`](#) Iterate over nodes

---

[`sorted\(\[reverse\]\)`](#) Sort by value

---

[`sorted\_by\_id\(\)`](#) Sort results by node id

---

`to\_df\(\)` Convert results to pandas DataFrame

---

`top\_k\(k\)` Compute the k largest values

---

`values\(\)` Iterate over values

---

`bottom\_k\(k\)`

Compute the k smallest values

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateMotifs`

`get\(node, default=...\)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional\[list\[int\]\])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional\[list\[int\]\]`

`items\(\)`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator\[Tuple\[Node, list\[int\]\]\]`

`max\(\)`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[list\[int\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[list\[int\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[list\[int\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, list\[int\]\]\]](#)

`nodes()`

Iterate over nodes

**Returns:**

The nodes

**Return type:**[Nodes](#)`sorted(reverse=False)`

Sort by value

**Parameters:**`reverse (bool) – If True, sort in descending order, otherwise ascending. Defaults to False.`**Returns:**

Sorted node state

**Return type:**[NodeStateMotifs](#)`sorted_by_id()`

Sort results by node id

**Returns:**

The sorted node state

**Return type:**[NodeStateMotifs](#)`to_df()`

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**[DataFrame](#)`top_k(k)`

Compute the k largest values

**Parameters:**`k (int) – The number of values to return`**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateMotifs](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[list\[int\]\]](#)

# NodeStateHits

**class NodeStateHits**

Bases: [object](#)

Methods:

---

[bottom\\_k\(k\)](#) Compute the k smallest values

---

[get\(node\[, default\]\)](#) Get value for node

---

[items\(\)](#) Iterate over items

---

[max\(\)](#) Return the maximum value

---

[max\\_item\(\)](#) Return largest value and corresponding node

---

[median\(\)](#) Return the median value

---

[median\\_item\(\)](#) Return median value and corresponding node

---

[min\(\)](#) Return the minimum value

---

[min\\_item\(\)](#) Return smallest value and corresponding node

---

[nodes\(\)](#) Iterate over nodes

---

<code><u>sorted</u>([reverse])</code>	Sort by value
<code><u>sorted_by_id</u>()</code>	Sort results by node id
<code><u>to_df</u>()</code>	Convert results to pandas DataFrame
<code><u>top_k</u>(k)</code>	Compute the k largest values
<code><u>values</u>()</code>	Iterate over values
<b>bottom_k(k)</b>	Compute the k smallest values

---

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateHits`

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional[Tuple[float, float]])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional[Tuple[float, float]]`

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator[Tuple[Node, Tuple[float, float]]]`

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

[Optional\[Tuple\[float, float\]\]](#)

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Tuple\[float, float\]\]\]](#)

`median()`

Return the median value

**Return type:**

[Optional\[Tuple\[float, float\]\]](#)

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

[Optional\[Tuple\[Node, Tuple\[float, float\]\]\]](#)

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

[Optional\[Tuple\[float, float\]\]](#)

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

Optional[Tuple[Node, Tuple[float, float]]]

**nodes ()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

Nodes

**sorted (reverse=False)**

Sort by value

**Parameters:**

`reverse (bool)` – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

NodeStateHits

**sorted\_by\_id ()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

NodeStateHits

**to\_df ()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

DataFrame

**top\_k (k)**

Compute the k largest values

**Parameters:**

`k` ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateHits](#)

`values()`

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Tuple\[float, float\]\]](#)

# NodeStateSEIR

`class NodeStateSEIR`

Bases: [object](#)

**Methods:**

---

[`bottom\_k\(k\)`](#) Compute the k smallest values

---

[`get\(node\[, default\]\)`](#) Get value for node

---

[`items\(\)`](#) Iterate over items

---

[`max\(\)`](#) Return the maximum value

---

[`max\_item\(\)`](#) Return largest value and corresponding node

---

[`median\(\)`](#) Return the median value

---

[`median\_item\(\)`](#) Return median value and corresponding node

---

[`min\(\)`](#) Return the minimum value

---

[`min\_item\(\)`](#) Return smallest value and corresponding node

---

<code>nodes()</code>	Iterate over nodes
<code>sorted([reverse])</code>	Sort by value
<code>sorted_by_id()</code>	Sort results by node id
<code>to_df()</code>	Convert results to pandas DataFrame
<code>top_k(k)</code>	Compute the k largest values
<code>values()</code>	Iterate over values
<code>bottom_k(k)</code>	Compute the k smallest values

---

**Parameters:**

`k (int)` – The number of values to return

**Returns:**

The k smallest values as a node state

**Return type:**

`NodeStateSEIR`

`get(node, default=...)`

Get value for node

**Parameters:**

- `node (NodeInput)` – the node
- `default (Optional[Infected])` – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

`Optional[Infected]`

`items()`

Iterate over items

**Returns:**

Iterator over items

**Return type:**

`Iterator[Tuple[Node, Infected]]`

`max()`

Return the maximum value

**Returns:**

The maximum value or None if empty

**Return type:**

`Optional[Infected]`

`max_item()`

Return largest value and corresponding node

**Returns:**

The Node and maximum value or None if empty

**Return type:**

`Optional[Tuple[Node, Infected]]`

`median()`

Return the median value

**Return type:**

`Optional[Infected]`

`median_item()`

Return median value and corresponding node

**Returns:**

The median value or None if empty

**Return type:**

`Optional[Tuple[Node, Infected]]`

`min()`

Return the minimum value

**Returns:**

The minimum value or None if empty

**Return type:**

`Optional[Infected]`

`min_item()`

Return smallest value and corresponding node

**Returns:**

The Node and minimum value or None if empty

**Return type:**

`Optional[Tuple[Node, Infected]]`

**nodes()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted(reverse=False)**

Sort by value

**Parameters:**

reverse ([bool](#)) – If True, sort in descending order, otherwise ascending. Defaults to False.

**Returns:**

Sorted node state

**Return type:**

[NodeStateSEIR](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeStateSEIR](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**top\_k(k)**

Compute the k largest values

**Parameters:**

k ([int](#)) – The number of values to return

**Returns:**

The k largest values as a node state

**Return type:**

[NodeStateSEIR](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[Infected\]](#)

# NodeLayout

`class NodeLayout`

Bases: [object](#)

Methods:

[`get\(node\[, default\]\)`](#)

Get value for node

---

[`items\(\)`](#)

Iterate over items

---

[`nodes\(\)`](#)

Iterate over nodes

---

[`sorted\_by\_id\(\)`](#)

Sort results by node id

---

[`to\_df\(\)`](#)

Convert results to pandas DataFrame

---

[`values\(\)`](#)

Iterate over values

[`get\(node, default=...\)`](#)

Get value for node

**Parameters:**

- `node` ([NodeInput](#)) – the node
- `default` ([Optional\[list\[float\]\]](#)) – the default value. Defaults to None.

**Returns:**

the value for the node or the default value

**Return type:**

[Optional\[list\[float\]\]](#)

**items()**

Iterate over items

**Returns:**

Iterator over items

**Return type:**

[Iterator\[Tuple\[Node, list\[float\]\]\]](#)

**nodes()**

Iterate over nodes

**Returns:**

The nodes

**Return type:**

[Nodes](#)

**sorted\_by\_id()**

Sort results by node id

**Returns:**

The sorted node state

**Return type:**

[NodeLayout](#)

**to\_df()**

Convert results to pandas DataFrame

The DataFrame has two columns, “node” with the node ids and “value” with the corresponding values.

**Returns:**

the pandas DataFrame

**Return type:**

[DataFrame](#)

**values()**

Iterate over values

**Returns:**

Iterator over values

**Return type:**

[Iterator\[list\[float\]\]](#)

# nullmodels

## Description

Generate randomised reference models for a temporal graph edgelist

## Functions

<code>permuted_timestamps_model</code> (graph_df[...])	Returns a DataFrame with the time column shuffled.
<code>shuffle_column</code> (graph_df[, col_number, ...])	Returns an edgelist with a given column shuffled.
<code>shuffle_multiple_columns</code> (graph_df[, ...])	Returns an edgelist with given columns shuffled.

## permuted\_timestamps\_model

`permuted_timestamps_model(graph_df, time_col=None, time_name=None, inplace=False, sorted=False)`

[\[source\]](#)

Returns a DataFrame with the time column shuffled.

### Parameters:

- `graph_df` ([DataFrame](#)) – The input DataFrame representing the graph.
- `time_col` ([int, optional](#)) – The column number of the time column to shuffle. Default is None.
- `time_name` ([str, optional](#)) – The column name of the time column to shuffle. Default is None.
- `inplace` ([bool, optional](#)) – If True, shuffles the time column in-place. Otherwise, creates a copy of the DataFrame. Default is False.
- `sorted` ([bool, optional](#)) – If True, sorts the DataFrame by the shuffled time column. Default is False.

### Returns:

The shuffled DataFrame with the time column, or None if `inplace=True`.

### Return type:

[DataFrame](#) | [None](#)

# shuffle\_column

```
shuffle_column(graph_df, col_number=None, col_name=None,  
inplace=False)  
[source]
```

Returns an edgelist with a given column shuffled. Exactly one of col\_number or col\_name should be specified.

## Parameters:

- graph\_df (DataFrame) – The input DataFrame representing the timestamped edgelist.
- col\_number (int, optional) – The column number to shuffle. Default is None.
- col\_name (str, optional) – The column name to shuffle. Default is None.
- inplace (bool, optional) – If True, shuffles the column in-place. Otherwise, creates a copy of the DataFrame. Default is False.

## Returns:

The shuffled DataFrame with the specified column.

## Return type:

DataFrame

## Raises:

- AssertionError – If neither col\_number nor col\_name is provided.
- AssertionError – If both col\_number and col\_name are provided.

# shuffle\_multiple\_columns

```
shuffle_multiple_columns(graph_df, col_numbers=None,  
col_names=None, inplace=False)  
[source]
```

Returns an edgelist with given columns shuffled. Exactly one of col\_numbers or col\_names should be specified.

## Parameters:

- graph\_df (DataFrame) – The input DataFrame representing the graph.
- col\_numbers (list, optional) – The list of column numbers to shuffle. Default is None.
- col\_names (list, optional) – The list of column names to shuffle. Default is None.
- inplace (bool, optional) – If True, shuffles the columns in-place. Otherwise, creates a copy of the DataFrame. Default is False.

## Returns:

The shuffled DataFrame with the specified columns.

**Return type:**

[DataFrame](#)

**Raises:**

- [AssertionError](#) – If neither col\_numbers nor col\_names are provided.
- [AssertionError](#) – If both col\_numbers and col\_names are provided.

# plottingutils

## Description

Useful code snippets for making commonly used plots in Raphtry.

## Functions

[`ccdf`](#)(observations[, normalised])

Returns x coordinates and y coordinates for a ccdf (complementary cumulative density function) from a list of observations.

[`cdf`](#)(observations[, normalised])

Returns x coordinates and y coordinates for a cdf (cumulative density function) from a list of observations.

[`global\_motif\_heatplot`](#)(motifs[, cmap])

Out-of-the-box plotting of global motif counts corresponding to the layout in Motifs in Temporal Networks (Paranjape et al)

[`human\_format`](#)(num)

Converts a number over 1000 to a string with 1 d.p and the corresponding letter.

[`lorenz`](#)(observations)

Returns x coordinates and y coordinates for a Lorenz Curve from a list of observations.

[ordinal\\_number](#)(number) Returns ordinal number of integer input.

[to\\_motif\\_matrix](#)(motifs[, data\_type]) Converts a 40d vector of global motifs to a 2d grid of motifs corresponding to the layout in Motifs in Temporal Networks (Paranjape et al)

## ccdf

**ccdf**(*observations*, *normalised*=True)

[\[source\]](#)

Returns x coordinates and y coordinates for a ccdf (complementary cumulative density function) from a list of observations.

**Parameters:**

- *observations* ([\*list\*](#)) – list of observations, should be numeric
- *normalised* ([\*bool, optional\*](#)) – Defaults to True. If true, y coordinates normalised such that y is the probability of finding a value greater than than or equal to x, if false y is the number of observations greater than or equal to x.

**Returns:**

x and y coordinates for the cdf

**Return type:**

[\*Tuple\[np.ndarray, np.ndarray\]\*](#)

## cdf

**cdf**(*observations*, *normalised*=True)

[\[source\]](#)

Returns x coordinates and y coordinates for a cdf (cumulative density function) from a list of observations.

**Parameters:**

- *observations* ([\*list\*](#)) – list of observations, should be numeric
- *normalised* ([\*bool\*](#)) – if true, y coordinates normalised such that y is the probability of finding a value less than or equal to x, if false y is the number of observations less than or equal to x. Defaults to True.

**Returns:**

the x and y coordinates for the cdf

**Return type:**

[Tuple\[np.ndarray, np.ndarray\]](#)

## global\_motif\_heatplot

**global\_motif\_heatplot(motifs, cmap='YlGnBu', \*\*kwargs)**  
[\[source\]](#)

Out-of-the-box plotting of global motif counts corresponding to the layout in Motifs in Temporal Networks (Paranjape et al)

**Parameters:**

- motifs ([list](#) | [np.ndarray](#)) – 1 dimensional length-40 array of motifs, which should be the list of motifs returned from the `global_temporal_three_node_motifs` function in Raphtry.
- \*\*kwargs – arguments to

**Returns:**

ax item containing the heatmap with motif labels on the axes.

**Return type:**

[matplotlib.axes.Axes](#)

## human\_format

**human\_format(num)**  
[\[source\]](#)

Converts a number over 1000 to a string with 1 d.p and the corresponding letter. e.g. with input 24134, 24.1k as a string would be returned. This is used in the motif plots to make annotated heatmap cells more concise.

**Parameters:**

num ([int](#)) – number to be abbreviated

**Returns:**

number in abbreviated string format.

**Return type:**

[str](#)

## lorenz

**lorenz(observations)**  
[\[source\]](#)

Returns x coordinates and y coordinates for a Lorenz Curve from a list of observations.

**Parameters:**

observations ([list](#)) – list of observations, should be numeric

**Returns:**

x and y coordinates for the cdf

**Return type:**

[Tuple\[np.ndarray, np.ndarray\]](#)

# ordinal\_number

`ordinal_number(number)`

[\[source\]](#)

Returns ordinal number of integer input.

**Parameters:**

number ([int](#)) – input number

**Returns:**

ordinal for that number as string

**Return type:**

[str](#)

# to\_motif\_matrix

```
to_motif_matrix(motifs, data_type=<class 'int'>)
\[source\]
```

Converts a 40d vector of global motifs to a 2d grid of motifs corresponding to the layout in Motifs in Temporal Networks (Paranjape et al)

## Parameters:

motifs ([list](#) | [np.ndarray](#)) – 1 dimensional length-40 array of motifs.

## Returns:

6x6 array of motifs whose ijth element is M\_ij in Motifs in Temporal Networks (Paranjape et al).

## Return type:

[np.ndarray](#)

# GraphView

```
class GraphView
```

Bases: [object](#)

Graph view is a read-only version of a graph at a certain point in time.

## Methods:

[after](#)(start)

Create a view of the GraphView including all events after start (exclusive).

[at](#)(time)

Create a view of the GraphView including all events at time.

[before](#)(end)

Create a view of the GraphView including all events before end (exclusive).

---

<code>cache_view()</code>	Applies the filters to the graph and retains the node ids and the edge ids in the graph that satisfy the filters creates bitsets per layer for nodes and edges
<code>count_edges()</code>	Number of edges in the graph
<code>count_nodes()</code>	Number of nodes in the graph
<code>count_temporal_edges()</code>	Number of edges in the graph
<code>create_index()</code>	Create graph index
<code>default_layer()</code>	Return a view of GraphView containing only the default edge layer :returns: The layered view :rtype: GraphView
<code>edge(src, dst)</code>	Gets the edge with the specified source and destination nodes
<code>exclude_layer(name)</code>	Return a view of GraphView containing all layers except the excluded name Errors if any of the layers do not exist.

---

<code><u>exclude_layers</u>(names)</code>	Return a view of GraphView containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_nodes</u>(nodes)</code>	Returns a subgraph given a set of nodes that are excluded from the subgraph
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of GraphView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of GraphView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>filter_edges</u>(filter)</code>	Return a filtered view that only includes edges that satisfy the filter
<code><u>filter_exploded_edges</u>(filter)</code>	Return a filtered view that only includes exploded edges that satisfy the filter

---

<code><a href="#">filter_nodes</a>(filter)</code>	Return a filtered view that only includes nodes that satisfy the filter
<code><a href="#">find_edges</a>(properties_dict)</code>	Get the edges that match the properties name and value :param properties_dict: the properties name and value :type properties_dict: dict[str, Prop]
<code><a href="#">find_nodes</a>(properties_dict)</code>	Get the nodes that match the properties name and value :param properties_dict: the properties name and value :type properties_dict: dict[str, Prop]
<code><a href="#">has_edge</a>(src, dst)</code>	Returns true if the graph contains the specified edge
<code><a href="#">has_layer</a>(name)</code>	Check if GraphView has the layer "name"
<code><a href="#">has_node</a>(id)</code>	Returns true if the graph contains the specified node
<code><a href="#">latest()</a></code>	Create a view of the GraphView including all events at the latest time.
<code><a href="#">layer</a>(name)</code>	Return a view of GraphView containing the layer "name" Errors if the layer does not exist

---

---

<code><u>layers</u>(names)</code>	Return a view of GraphView containing all layers names Errors if any of the layers do not exist.
<code><u>materialize</u>()</code>	Returns a 'materialized' clone of the graph view - i.e. a new graph with a copy of the data seen within the view instead of just a mask over the original graph.
<code><u>node</u>(id)</code>	Gets the node with the specified id
<code><u>rolling</u>(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><u>search_edges</u>(filter[, limit, offset])</code>	Searches for edges which match the given filter expression.
<code><u>search_nodes</u>(filter[, limit, offset])</code>	Searches for nodes which match the given filter expression.
<code><u>shrink_end</u>(end)</code>	Set the end of the window to the smaller of end and self.end()
<code><u>shrink_start</u>(start)</code>	Set the start of the window to the larger of start and self.start()

---

<code><u>shrink_window</u>(start, end)</code>	Shrink both the start and end of the window (same as calling <code>shrink_start</code> followed by <code>shrink_end</code> but more efficient)
<code><u>snapshot_at</u>(time)</code>	Create a view of the GraphView including all events that have not been explicitly deleted at time.
<code><u>snapshot_latest</u>()</code>	Create a view of the GraphView including all events that have not been explicitly deleted at the latest time.
<code><u>subgraph</u>(nodes)</code>	Returns a subgraph given a set of nodes
<code><u>subgraph_node_types</u>(node_types)</code>	Returns a subgraph filtered by node types given a set of node types
<code><u>to_networkx</u>([explode_edges, ...])</code>	Returns a graph with NetworkX.
<code><u>to_pyvis</u>([explode_edges, edge_color, shape, ...])</code>	Draw a graph with PyVis.
<code><u>valid_layers</u>(names)</code>	Return a view of GraphView containing all layers names Any layers that do not exist are ignored
<code><u>vectorise</u>(embedding[, cache, ...])</code>	Create a VectorisedGraph from the current graph

---

<b><code>window(start, end)</code></b>	Create a view of the GraphView including all events between start (inclusive) and end (exclusive)
<hr/>	
<b>Attributes:</b>	
<b><code>earliest_date_time</code></b>	DateTime of earliest activity in the graph
<b><code>earliest_time</code></b>	Timestamp of earliest activity in the graph
<b><code>edges</code></b>	Gets all edges in the graph
<b><code>end</code></b>	Gets the latest time that this GraphView is valid.
<b><code>end_date_time</code></b>	Gets the latest datetime that this GraphView is valid
<b><code>latest_date_time</code></b>	DateTime of latest activity in the graph
<b><code>latest_time</code></b>	Timestamp of latest activity in the graph
<b><code>nodes</code></b>	Gets the nodes in the graph
<b><code>properties</code></b>	Get all graph properties
<b><code>start</code></b>	Gets the start time for rolling and expanding windows for this GraphView
<b><code>start_date_time</code></b>	Gets the earliest datetime that this GraphView is valid
<b><code>unique_layers</code></b>	Return all the layer ids in the graph
<b><code>window_size</code></b>	Get the window size (difference between start and end) for this GraphView
<b><code>after(start)</code></b>	

Create a view of the GraphView including all events after start (exclusive).

**Parameters:**

start ([TimeInput](#)) – The start time of the window.

**Return type:**

[GraphView](#)

`at(time)`

Create a view of the GraphView including all events at time.

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[GraphView](#)

`before(end)`

Create a view of the GraphView including all events before end (exclusive).

**Parameters:**

end ([TimeInput](#)) – The end time of the window.

**Return type:**

[GraphView](#)

`cache_view()`

Applies the filters to the graph and retains the node ids and the edge ids in the graph that satisfy the filters creates bitsets per layer for nodes and edges

**Returns:**

Returns the masked graph

**Return type:**

[GraphView](#)

`count_edges()`

Number of edges in the graph

**Returns:**

the number of edges in the graph

**Return type:**

[int](#)

`count_nodes()`

Number of nodes in the graph

**Returns:**

the number of nodes in the graph

**Return type:**

`int`

`count_temporal_edges()`

Number of edges in the graph

**Returns:**

the number of temporal edges in the graph

**Return type:**

`int`

`create_index()`

Create graph index

`default_layer()`

Return a view of GraphView containing only the default edge layer :returns: The layered view :rtype: GraphView

`earliest_date_time`

DateTime of earliest activity in the graph

**Returns:**

the datetime of the earliest activity in the graph

**Return type:**

`Optional[datetime]`

`earliest_time`

Timestamp of earliest activity in the graph

**Returns:**

the timestamp of the earliest activity in the graph

**Return type:**

`Optional[int]`

`edge(src, dst)`

Gets the edge with the specified source and destination nodes

**Parameters:**

- `src` ([NodeInput](#)) – the source node id
- `dst` ([NodeInput](#)) – the destination node id

**Returns:**

the edge with the specified source and destination nodes, or None if the edge does not exist

**Return type:**

[Optional\[Edge\]](#)

**edges**

Gets all edges in the graph

**Returns:**

the edges in the graph

**Return type:**

[Edges](#)

**end**

Gets the latest time that this GraphView is valid.

**Returns:**

The latest time that this GraphView is valid or None if the GraphView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this GraphView is valid

**Returns:**

The latest datetime that this GraphView is valid or None if the GraphView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of GraphView containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[GraphView](#)

**exclude\_layers(names)**

Return a view of GraphView containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names (*list[str]*) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[GraphView](#)

**exclude\_nodes(nodes)**

Returns a subgraph given a set of nodes that are excluded from the subgraph

**Parameters:**

nodes (*list[NodeInput]*) – set of nodes

**Returns:**

Returns the subgraph

**Return type:**

[GraphView](#)

**exclude\_valid\_layer(name)**

Return a view of GraphView containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[GraphView](#)

**exclude\_valid\_layers(names)**

Return a view of GraphView containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[GraphView](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

`filter_edges(filter)`

Return a filtered view that only includes edges that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[GraphView](#)

`filter_exploded_edges(filter)`

Return a filtered view that only includes exploded edges that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the exploded edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[GraphView](#)

`filter_nodes(filter)`

Return a filtered view that only includes nodes that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the node properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[GraphView](#)

`find_edges(properties_dict)`

Get the edges that match the properties name and value :param properties\_dict: the properties name and value :type properties\_dict: dict[str, Prop]

**Returns:**

the edges that match the properties name and value

**Return type:**

`list[Edge]`

`find_nodes(properties_dict)`

Get the nodes that match the properties name and value :param properties\_dict: the properties name and value :type properties\_dict: dict[str, Prop]

**Returns:**

the nodes that match the properties name and value

**Return type:**

`list[Node]`

`has_edge(src, dst)`

Returns true if the graph contains the specified edge

**Parameters:**

- `src` (`NodeInput`) – the source node id
- `dst` (`NodeInput`) – the destination node id

**Returns:**

true if the graph contains the specified edge, false otherwise

**Return type:**

`bool`

`has_layer(name)`

Check if GraphView has the layer “name”

**Parameters:**

`name` (`str`) – the name of the layer to check

**Return type:**

`bool`

`has_node(id)`

Returns true if the graph contains the specified node

**Parameters:**

`id` (`NodeInput`) – the node id

**Returns:**

true if the graph contains the specified node, false otherwise

**Return type:**

`bool`

`latest()`

Create a view of the GraphView including all events at the latest time.

**Return type:**

`GraphView`

`latest_date_time`

`DateTime` of latest activity in the graph

**Returns:**

the datetime of the latest activity in the graph

**Return type:**

`Optional[datetime]`

`latest_time`

`Timestamp` of latest activity in the graph

**Returns:**

the timestamp of the latest activity in the graph

**Return type:**

`Optional[int]`

`layer(name)`

Return a view of GraphView containing the layer “name” Errors if the layer does not exist

**Parameters:**

`name (str)` – then name of the layer.

**Returns:**

The layered view

**Return type:**

`GraphView`

`layers(names)`

Return a view of GraphView containing all layers names Errors if any of the layers do not exist.

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[GraphView](#)

**materialize()**

Returns a ‘materialized’ clone of the graph view - i.e. a new graph with a copy of the data seen within the view instead of just a mask over the original graph

**Returns:**

Returns a graph clone

**Return type:**

[GraphView](#)

**node (id)**

Gets the node with the specified id

**Parameters:**

**id ([NodeInput](#))** – the node id

**Returns:**

the node with the specified id, or None if the node does not exist

**Return type:**

[Optional\[Node\]](#)

**nodes**

Gets the nodes in the graph

**Returns:**

the nodes in the graph

**Return type:**

[Nodes](#)

**properties**

Get all graph properties

**Returns:**

Properties paired with their names

**Return type:**

[Properties](#)

**rolling (window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window` (`int` | `str`) – The size of the window.
- `step` (`int` | `str` | `None`) – The step size of the window. `step` defaults to `window`.

**Returns:**

A `WindowSet` object.

**Return type:**

`WindowSet`

`search_edges(filter, limit=25, offset=0)`

Searches for edges which match the given filter expression. This uses Tantivy's exact search.

**Parameters:**

- `filter` – The filter expression to search for.
- `limit` (`int`) – The maximum number of results to return. Defaults to 25.
- `offset` (`int`) – The number of results to skip. This is useful for pagination. Defaults to 0.

**Returns:**

A list of edges which match the filter expression. The list will be empty if no edges match the query.

**Return type:**

`list[Edge]`

`search_nodes(filter, limit=25, offset=0)`

Searches for nodes which match the given filter expression. This uses Tantivy's exact search.

**Parameters:**

- `filter` – The filter expression to search for.
- `limit` (`int`) – The maximum number of results to return. Defaults to 25.
- `offset` (`int`) – The number of results to skip. This is useful for pagination. Defaults to 0.

**Returns:**

A list of nodes which match the filter expression. The list will be empty if no nodes match.

**Return type:**

`list[Node]`

`shrink_end(end)`

Set the end of the window to the smaller of `end` and `self.end()`

**Parameters:**end (*TimeInput*) – the new end time of the window**Return type:**[GraphView](#)`shrink_start(start)`

Set the start of the window to the larger of start and self.start()

**Parameters:**start (*TimeInput*) – the new start time of the window**Return type:**[GraphView](#)`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start (*TimeInput*) – the new start time for the window
- end (*TimeInput*) – the new end time for the window

**Return type:**[GraphView](#)`snapshot_at(time)`

Create a view of the GraphView including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**time (*TimeInput*) – The time of the window.**Return type:**[GraphView](#)`snapshot_latest()`

Create a view of the GraphView including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**[GraphView](#)`start`

Gets the start time for rolling and expanding windows for this GraphView

**Returns:**

The earliest time that this GraphView is valid or None if the GraphView is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this GraphView is valid

**Returns:**

The earliest datetime that this GraphView is valid or None if the GraphView is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**subgraph(nodes)**

Returns a subgraph given a set of nodes

**Parameters:**

nodes ([list\[NodeInput\]](#)) – set of nodes

**Returns:**

Returns the subgraph

**Return type:**

[GraphView](#)

**subgraph\_node\_types(node\_types)**

Returns a subgraph filtered by node types given a set of node types

**Parameters:**

node\_types ([list\[str\]](#)) – set of node types

**Returns:**

Returns the subgraph

**Return type:**

[GraphView](#)

**to\_networkx(explode\_edges=False,  
include\_node\_properties=True, include\_edge\_properties=True,  
include\_update\_history=True, include\_property\_history=True)**

Returns a graph with NetworkX.

Network X is a required dependency. If you intend to use this function make sure that you install Network X with `pip install networkx`

#### Parameters:

- `explode_edges (bool)` – A boolean that is set to True if you want to explode the edges in the graph. Defaults to False.
- `include_node_properties (bool)` – A boolean that is set to True if you want to include the node properties in the graph. Defaults to True.
- `include_edge_properties (bool)` – A boolean that is set to True if you want to include the edge properties in the graph. Defaults to True.
- `include_update_history (bool)` – A boolean that is set to True if you want to include the update histories in the graph. Defaults to True.
- `include_property_history (bool)` – A boolean that is set to True if you want to include the histories in the graph. Defaults to True.

#### Returns:

A Networkx MultiDiGraph.

#### Return type:

`nx.MultiDiGraph`

```
to_pyvis(explode_edges=False, edge_color="#000000",
shape='dot', node_image=None, edge_weight=None,
edge_label=None, colour_nodes_by_type=False, directed=True,
notebook=False, **kwargs)
```

Draw a graph with PyVis. Pyvis is a required dependency. If you intend to use this function make sure that you install Pyvis with `pip install pyvis`

#### Parameters:

- `explode_edges (bool)` – A boolean that is set to True if you want to explode the edges in the graph. Defaults to False.
- `edge_color (str)` – A string defining the colour of the edges in the graph. Defaults to "#000000".
- `shape (str)` – A string defining what the node looks like. Defaults to "dot".  
There are two types of nodes. One type has the label inside of it and the other type has the label underneath it. The types with the label inside of it are: ellipse, circle, database, box, text. The ones with the label outside of it are: image, circularImage, diamond, dot, star, triangle, triangleDown, square and icon.
- `node_image (str, optional)` – An optional node property used as the url of a custom node image. Use together with `shape="image"`.
- `edge_weight (str, optional)` – An optional string defining the name of the property where edge weight is set on your Raphtry graph. If provided, the default weight for edges that are missing the property is 1.0.

- `edge_label` (`str, optional`) – An optional string defining the name of the property where edge label is set on your Raphtory graph. By default, the edge layer is used as the label.
- `colour_nodes_by_type` (`bool`) – If True, nodes with different types have different colours. Defaults to False.
- `directed` (`bool`) – Visualise the graph as directed. Defaults to True.
- `notebook` (`bool`) – A boolean that is set to True if using jupyter notebook. Defaults to False.
- `kwargs` – Additional keyword arguments that are passed to the pyvis Network class.

**Returns:**

A pyvis network

**Return type:**

`pyvis.network.Network`

**unique\_layers**

Return all the layer ids in the graph

**Returns:**

the names of all layers in the graph

**Return type:**

`list[str]`

**valid\_layers(names)**

Return a view of GraphView containing all layers names Any layers that do not exist are ignored

**Parameters:**

`names` (`list[str]`) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

`GraphView`

```
vectorise(embedding, cache=None, overwrite_cache=False,
graph=..., nodes=..., edges=..., graph_name=None,
verbose=False)
```

Create a VectorisedGraph from the current graph

**Parameters:**

- `embedding` (`Callable[[list], list]`) – the embedding function to translate documents to embeddings

- `cache (str, optional)` – the file to be used as a cache to avoid calling the embedding function
- `overwrite_cache (bool)` – whether or not to overwrite the cache if there are new embeddings. Defaults to False.
- `graph (bool | str)` – if the graph has to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- `nodes (bool | str)` – if nodes have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- `edges (bool | str)` – if edges have to be embedded or not or the custom template to use if a str is provided. Defaults to True.
- `graph_name (str, optional)` – the name of the graph
- `verbose (bool)` – whether or not to print logs reporting the progress. Defaults to False.

**Returns:**

A VectorisedGraph with all the documents/embeddings computed and with an initial empty selection

**Return type:**

[VectorisedGraph](#)

`window (start, end)`

Create a view of the GraphView including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start (TimeInput | None)` – The start time of the window (unbounded if None).
- `end (TimeInput | None)` – The end time of the window (unbounded if None).

**Return type:**

[GraphView](#)

`window_size`

Get the window size (difference between start and end) for this GraphView

**Return type:**

[Optional\[int\]](#)

# Graph

`class Graph(num_shards=None)`

Bases: [GraphView](#)

A temporal graph with event semantics.

**Parameters:**

`num_shards` (*int, optional*) – The number of locks to use in the storage to allow for multithreaded updates.

**Methods:**

---

`add_constant_properties(properties)`

Adds static properties to the graph.

---

`add_edge(timestamp, src, dst[, properties, ...])`

Adds a new edge with the given source and destination nodes and properties to the graph.

---

`add_node(timestamp, id[, properties, ...])`

Adds a new node with the given id and properties to the graph.

---

`add_properties(timestamp, properties[, ...])`

Adds properties to the graph.

---

`cache(path)`

Write Graph to cache file and initialise the cache.

---

`create_node(timestamp, id[, properties, ...])`

Creates a new node with the given id and properties to the graph.

---

`deserialise(bytes)`

Load Graph from serialised bytes.

---

`edge(src, dst)`

Gets the edge with the specified source and destination nodes

---

`event_graph()`

View graph with event semantics

---

<code>from_parquet(graph_dir)</code>	Read graph from parquet files
<code>get_all_node_types()</code>	Returns all the node types in the graph.
<code>import_edge(edge[, merge])</code>	Import a single edge into the graph.
<code>import_edge_as(edge, new_id[, merge])</code>	Import a single edge into the graph with new id.
<code>import_edges(edges[, merge])</code>	Import multiple edges into the graph.
<code>import_edges_as(edges, new_ids[, merge])</code>	Import multiple edges into the graph with new ids.
<code>import_node(node[, merge])</code>	Import a single node into the graph.
<code>import_node_as(node, new_id[, merge])</code>	Import a single node into the graph with new id.
<code>import_nodes(nodes[, merge])</code>	Import multiple nodes into the graph.
<code>import_nodes_as(nodes, new_ids[, merge])</code>	Import multiple nodes into the graph with new ids.
<code>largest_connected_component()</code>	Gives the large connected component of a graph.

---

<code><a href="#">load_cached</a>(path)</code>	Load Graph from a file and initialise it as a cache file.
<code><a href="#">load_edge_props_from_pandas</a>(df, src, dst[, ...])</code>	Load edge properties from a Pandas DataFrame.
<code><a href="#">load_edge_props_from_parquet</a>(parquet_path, ...)</code>	Load edge properties from parquet file
<code><a href="#">load_edges_from_pandas</a>(df, time, src, dst[, ...])</code>	Load edges from a Pandas DataFrame into the graph.
<code><a href="#">load_edges_from_parquet</a>(parquet_path, time, ...)</code>	Load edges from a Parquet file into the graph.
<code><a href="#">load_from_file</a>(path)</code>	Load Graph from a file.
<code><a href="#">load_node_props_from_pandas</a>(df, id[, ...])</code>	Load node properties from a Pandas DataFrame.
<code><a href="#">load_node_props_from_parquet</a>(parquet_path, id)</code>	Load node properties from a parquet file.
<code><a href="#">load_nodes_from_pandas</a>(df, time, id[, ...])</code>	Load nodes from a Pandas DataFrame into the graph.
<code><a href="#">load_nodes_from_parquet</a>(parquet_path, time, id)</code>	Load nodes from a Parquet file into the graph.

---

<code><a href="#">node</a>(id)</code>	Gets the node with the specified id
<code><a href="#">persistent_graph</a>()</code>	View graph with persistent semantics
<code><a href="#">save_to_file</a>(path)</code>	Saves the Graph to the given path.
<code><a href="#">save_to_zip</a>(path)</code>	Saves the Graph to the given path.
<code><a href="#">serialise</a>()</code>	Serialise Graph to bytes.
<code><a href="#">to_parquet</a>(graph_dir)</code>	Persist graph to parquet files
<code><a href="#">update_constant_properties</a>(properties)</code>	Updates static properties to the graph.
<code><a href="#">write_updates</a>()</code>	Persist the new updates by appending them to the cache file.
<b><code><a href="#">add_constant_properties</a>(properties)</code></b>	
Adds static properties to the graph.	
<b>Parameters:</b>	
<code>properties (<a href="#">PropInput</a>)</code> – The static properties of the graph.	
<b>Returns:</b>	
This function does not return a value, if the operation is successful.	
<b>Return type:</b>	
<code><a href="#">None</a></code>	
<b>Raises:</b>	
<code>GraphError</code> – If the operation fails.	
<code><a href="#">add_edge</a>(timestamp, src, dst, properties=None, layer=None, secondary_index=None)</code>	

Adds a new edge with the given source and destination nodes and properties to the graph.

**Parameters:**

- timestamp ([TimeInput](#)) – The timestamp of the edge.
- src ([str|int](#)) – The id of the source node.
- dst ([str|int](#)) – The id of the destination node.
- properties ([PropInput](#), optional) – The properties of the edge, as a dict of string and properties.
- layer ([str](#), optional) – The layer of the edge.
- secondary\_index ([int](#), optional) – The optional integer which will be used as a secondary index

**Returns:**

The added edge.

**Return type:**

[MutableEdge](#)

**Raises:**

GraphError – If the operation fails.

```
add_node(timestamp, id, properties=None, node_type=None,  
secondary_index=None)
```

Adds a new node with the given id and properties to the graph.

**Parameters:**

- timestamp ([TimeInput](#)) – The timestamp of the node.
- id ([str|int](#)) – The id of the node.
- properties ([PropInput](#), optional) – The properties of the node.
- node\_type ([str](#), optional) – The optional string which will be used as a node type
- secondary\_index ([int](#), optional) – The optional integer which will be used as a secondary index

**Returns:**

The added node.

**Return type:**

[MutableNode](#)

**Raises:**

GraphError – If the operation fails.

```
add_properties(timestamp, properties, secondary_index=None)
```

Adds properties to the graph.

**Parameters:**

- `timestamp` (`TimeInput`) – The timestamp of the temporal property.
- `properties` (`PropInput`) – The temporal properties of the graph.
- `secondary_index` (`int, optional`) – The optional integer which will be used as a secondary index

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

`cache(path)`

Write Graph to cache file and initialise the cache.

Future updates are tracked. Use `write_updates` to persist them to the cache file. If the file already exists its contents are overwritten.

**Parameters:**

`path` (`str`) – The path to the cache file

**Return type:**

`None`

`create_node(timestamp, id, properties=None, node_type=None, secondary_index=None)`

Creates a new node with the given id and properties to the graph. It fails if the node already exists.

**Parameters:**

- `timestamp` (`TimeInput`) – The timestamp of the node.
- `id` (`str|int`) – The id of the node.
- `properties` (`PropInput, optional`) – The properties of the node.
- `node_type` (`str, optional`) – The optional string which will be used as a node type
- `secondary_index` (`int, optional`) – The optional integer which will be used as a secondary index

**Returns:**

The created node.

**Return type:**

`MutableNode`

**Raises:**

`GraphError` – If the operation fails.

**static deserialise(bytes)**

Load Graph from serialised bytes.

**Parameters:**

bytes ([bytes](#)) – The serialised bytes to decode

**Return type:**

[Graph](#)

**edge(src, dst)**

Gets the edge with the specified source and destination nodes

**Parameters:**

- src ([str](#)|[int](#)) – the source node id
- dst ([str](#)|[int](#)) – the destination node id

**Returns:**

the edge with the specified source and destination nodes, or None if the edge does not exist

**Return type:**

[MutableEdge](#)

**event\_graph()**

View graph with event semantics

**Returns:**

the graph with event semantics applied

**Return type:**

[Graph](#)

**static from\_parquet(graph\_dir)**

Read graph from parquet files

**Parameters:**

graph\_dir ([str](#) | [PathLike](#)) – the folder where the graph is stored as parquet

**Returns:**

a view of the graph

**Return type:**

[Graph](#)

**get\_all\_node\_types()**

Returns all the node types in the graph.

**Returns:**

the node types

**Return type:**

[List\[str\]](#)

`import_edge(edge, merge=False)`

Import a single edge into the graph.

**Parameters:**

- `edge` ([Edge](#)) – A Edge object representing the edge to be imported.
- `merge` ([bool](#)) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if the imported edge already exists in the graph. If merge is True, the function merges the histories of the imported edge and the existing edge (in the graph).

**Returns:**

An Edge object if the edge was successfully imported.

**Return type:**

[MutableEdge](#)

**Raises:**

`GraphError` – If the operation fails.

`import_edge_as(edge, new_id, merge=False)`

Import a single edge into the graph with new id.

**Parameters:**

- `edge` ([Edge](#)) – A Edge object representing the edge to be imported.
- `new_id` ([tuple](#)) – The ID of the new edge. It's a tuple of the source and destination node ids.
- `merge` ([bool](#)) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if the imported edge already exists in the graph. If merge is True, the function merges the histories of the imported edge and the existing edge (in the graph).

**Returns:**

An Edge object if the edge was successfully imported.

**Return type:**

[Edge](#)

**Raises:**

`GraphError` – If the operation fails.

`import_edges(edges, merge=False)`

Import multiple edges into the graph.

**Parameters:**

- `edges` (`List[Edge]`) – A list of Edge objects representing the edges to be imported.
- `merge` (`bool`) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if any of the imported edges already exists in the graph. If merge is True, the function merges the histories of the imported edges and the existing edges (in the graph).

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

```
import_edges_as(edges, new_ids, merge=False)
```

Import multiple edges into the graph with new ids.

**Parameters:**

- `edges` (`List[Edge]`) – A list of Edge objects representing the edges to be imported.
- `new_ids` (`List[Tuple[int, int]]`) – The IDs of the new edges. It's a vector of tuples of the source and destination node ids.
- `merge` (`bool`) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if any of the imported edges already exists in the graph. If merge is True, the function merges the histories of the imported edges and the existing edges (in the graph).

**Returns:**

This function does not return a value if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

```
import_node(node, merge=False)
```

Import a single node into the graph.

**Parameters:**

- `node` (`Node`) – A Node object representing the node to be imported.
- `merge` (`bool`) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if the imported node already exists in the graph. If merge is True, the function merges the histories of the imported node and the existing node (in the graph).

**Returns:**

A node object if the node was successfully imported.

**Return type:**

[Node](#)

**Raises:**

GraphError – If the operation fails.

```
import_node_as(node, new_id, merge=False)
```

Import a single node into the graph with new id.

**Parameters:**

- node ([Node](#)) – A Node object representing the node to be imported.
- new\_id ([str|int](#)) – The new node id.
- merge ([bool](#)) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if the imported node already exists in the graph. If merge is True, the function merges the histories of the imported node and the existing node (in the graph).

**Returns:**

A node object if the node was successfully imported.

**Return type:**

[MutableNode](#)

**Raises:**

GraphError – If the operation fails.

```
import_nodes(nodes, merge=False)
```

Import multiple nodes into the graph.

**Parameters:**

- nodes ([List\[Node\]](#)) – A vector of Node objects representing the nodes to be imported.
- merge ([bool](#)) – An optional boolean flag. Defaults to False. If merge is False, the function will return an error if any of the imported nodes already exists in the graph. If merge is True, the function merges the histories of the imported nodes and the existing nodes (in the graph).

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
import_nodes_as(nodes, new_ids, merge=False)
```

Import multiple nodes into the graph with new ids.

**Parameters:**

- `nodes` ([List\[Node\]](#)) – A vector of Node objects representing the nodes to be imported.
- `new_ids` ([List\[str|int\]](#)) – A list of node IDs to use for the imported nodes.
- `merge` ([bool](#)) – An optional boolean flag. Defaults to False. If merge is True, the function will return an error if any of the imported nodes already exists in the graph. If merge is False, the function merges the histories of the imported nodes and the existing nodes (in the graph).

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

```
largest_connected_component()
```

Gives the large connected component of a graph.

```
# Example Usage: g.largest_connected_component()
```

**Returns:**

sub-graph of the graph g containing the largest connected component

**Return type:**

[GraphView](#)

```
static load_cached(path)
```

Load Graph from a file and initialise it as a cache file.

Future updates are tracked. Use `write_updates` to persist them to the cache file.

**Parameters:**

`path` ([str](#)) – The path to the cache file

**Returns:**

the loaded graph with initialised cache

**Return type:**

[Graph](#)

```
load_edge_props_from_pandas(df, src, dst,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edge properties from a Pandas DataFrame.

#### Parameters:

- df ([DataFrame](#)) – The Pandas DataFrame containing edge information.
- src ([str](#)) – The column name for the source node.
- dst ([str](#)) – The column name for the destination node.
- constant\_properties ([List\[str\]](#), *optional*) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput](#), *optional*) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str](#), *optional*) – The edge layer name. Defaults to None.
- layer\_col ([str](#), *optional*) – The edge layer col name in dataframe. Defaults to None.

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

[None](#)

#### Raises:

[GraphError](#) – If the operation fails.

```
load_edge_props_from_parquet(parquet_path, src, dst,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edge properties from parquet file

#### Parameters:

- parquet\_path ([str](#)) – Parquet file or directory of Parquet files path containing edge information.
- src ([str](#)) – The column name for the source node.
- dst ([str](#)) – The column name for the destination node.
- constant\_properties ([List\[str\]](#), *optional*) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput](#), *optional*) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str](#), *optional*) – The edge layer name. Defaults to None.
- layer\_col ([str](#), *optional*) – The edge layer col name in dataframe. Defaults to None.

#### Returns:

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_edges_from_pandas(df, time, src, dst, properties=None,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edges from a Pandas DataFrame into the graph.

**Parameters:**

- df ([DataFrame](#)) – The Pandas DataFrame containing the edges.
- time ([str](#)) – The column name for the update timestamps.
- src ([str](#)) – The column name for the source node ids.
- dst ([str](#)) – The column name for the destination node ids.
- properties ([List\[str\], optional](#)) – List of edge property column names. Defaults to None.
- constant\_properties ([List\[str\], optional](#)) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput, optional](#)) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str, optional](#)) – A constant value to use as the layer for all edges. Defaults to None. (cannot be used in combination with layer\_col)
- layer\_col ([str, optional](#)) – The edge layer col name in dataframe. Defaults to None. (cannot be used in combination with layer)

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_edges_from_parquet(parquet_path, time, src, dst,
properties=None, constant_properties=None,
shared_constant_properties=None, layer=None, layer_col=None)
```

Load edges from a Parquet file into the graph.

**Parameters:**

- parquet\_path ([str](#)) – Parquet file or directory of Parquet files path containing edges
- time ([str](#)) – The column name for the update timestamps.
- src ([str](#)) – The column name for the source node ids.

- `dst (str)` – The column name for the destination node ids.
- `properties (List[str], optional)` – List of edge property column names. Defaults to None.
- `constant_properties (List[str], optional)` – List of constant edge property column names. Defaults to None.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every edge. Defaults to None.
- `layer (str, optional)` – A constant value to use as the layer for all edges. Defaults to None. (cannot be used in combination with `layer_col`)
- `layer_col (str, optional)` – The edge layer col name in dataframe. Defaults to None. (cannot be used in combination with `layer`)

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

**static load\_from\_file(path)**

Load Graph from a file.

**Parameters:**

`path (str)` – The path to the file.

**Return type:**

[Graph](#)

**load\_node\_props\_from\_pandas(df, id, node\_type=None, node\_type\_col=None, constant\_properties=None, shared\_constant\_properties=None)**

Load node properties from a Pandas DataFrame.

**Parameters:**

- `df (DataFrame)` – The Pandas DataFrame containing node information.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to None. (cannot be used in combination with `node_type_col`)
- `node_type_col (str, optional)` – The node type col name in dataframe. Defaults to None. (cannot be used in combination with `node_type`)
- `constant_properties (List[str], optional)` – List of constant node property column names. Defaults to None.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every node. Defaults to None.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_node_props_from_parquet(parquet_path, id,
node_type=None, node_type_col=None, constant_properties=None,
shared_constant_properties=None)
```

Load node properties from a parquet file.

**Parameters:**

- `parquet_path (str)` – Parquet file or directory of Parquet files path containing node information.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to None. (cannot be used in combination with `node_type_col`)
- `node_type_col (str, optional)` – The node type col name in dataframe. Defaults to None. (cannot be used in combination with `node_type`)
- `constant_properties (List\[str, optional\])` – List of constant node property column names. Defaults to None.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every node. Defaults to None.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_nodes_from_pandas(df, time, id, node_type=None,
node_type_col=None, properties=None, constant_properties=None,
shared_constant_properties=None)
```

Load nodes from a Pandas DataFrame into the graph.

**Parameters:**

- `df (DataFrame)` – The Pandas DataFrame containing the nodes.
- `time (str)` – The column name for the timestamps.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to None. (cannot be used in combination with `node_type_col`)

- `node_type_col` (`str, optional`) – The node type col name in dataframe. Defaults to `None`. (cannot be used in combination with `node_type`)
- `properties` (`List[str, optional]`) – List of node property column names. Defaults to `None`.
- `constant_properties` (`List[str, optional]`) – List of constant node property column names. Defaults to `None`.
- `shared_constant_properties` (`PropInput, optional`) – A dictionary of constant properties that will be added to every node. Defaults to `None`.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

```
load_nodes_from_parquet(parquet_path, time, id,
node_type=None, node_type_col=None, properties=None,
constant_properties=None, shared_constant_properties=None)
```

Load nodes from a Parquet file into the graph.

**Parameters:**

- `parquet_path` (`str`) – Parquet file or directory of Parquet files containing the nodes
- `time` (`str`) – The column name for the timestamps.
- `id` (`str`) – The column name for the node IDs.
- `node_type` (`str, optional`) – A constant value to use as the node type for all nodes. Defaults to `None`. (cannot be used in combination with `node_type_col`)
- `node_type_col` (`str, optional`) – The node type col name in dataframe. Defaults to `None`. (cannot be used in combination with `node_type`)
- `properties` (`List[str, optional]`) – List of node property column names. Defaults to `None`.
- `constant_properties` (`List[str, optional]`) – List of constant node property column names. Defaults to `None`.
- `shared_constant_properties` (`PropInput, optional`) – A dictionary of constant properties that will be added to every node. Defaults to `None`.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

**`node(id)`**

Gets the node with the specified id

**Parameters:**

`id (str|int)` – the node id

**Returns:**

The node object with the specified id, or None if the node does not exist

**Return type:**

[MutableNode](#)

**`persistent_graph()`**

View graph with persistent semantics

**Returns:**

the graph with persistent semantics applied

**Return type:**

[PersistentGraph](#)

**`save_to_file(path)`**

Saves the Graph to the given path.

**Parameters:**

`path (str)` – The path to the file.

**Return type:**

[None](#)

**`save_to_zip(path)`**

Saves the Graph to the given path.

**Parameters:**

`path (str)` – The path to the file.

**Return type:**

[None](#)

**`serialise()`**

Serialise Graph to bytes.

**Return type:**

[bytes](#)

**`to_parquet(graph_dir)`**

Persist graph to parquet files

**Parameters:**

`graph_dir (str | PathLike)` – the folder where the graph will be persisted as parquet

`update_constant_properties(properties)`

Updates static properties to the graph.

**Parameters:**

`properties (PropInput)` – The static properties of the graph.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

`write_updates()`

Persist the new updates by appending them to the cache file.

**Return type:**

`None`

# PersistentGraph

`class PersistentGraph`

Bases: `GraphView`

A temporal graph that allows edges and nodes to be deleted.

**Methods:**

`add_constant_properties(properties)`

Adds static properties to the graph.

`add_edge(timestamp, src, dst[, properties, ...])`

Adds a new edge with the given source and destination nodes and properties to the graph.

---

<code><u>add_node</u>(timestamp, id[, properties, ...])</code>	Adds a new node with the given id and properties to the graph.
<code><u>add_properties</u>(timestamp, properties[, ...])</code>	Adds properties to the graph.
<code><u>cache</u>(path)</code>	Write PersistentGraph to cache file and initialise the cache.
<code><u>create_node</u>(timestamp, id[, properties, ...])</code>	Creates a new node with the given id and properties to the graph.
<code><u>delete_edge</u>(timestamp, src, dst[, layer, ...])</code>	Deletes an edge given the timestamp, src and dst nodes and layer (optional)
<code><u>deserialise</u>(bytes)</code>	Load PersistentGraph from serialised bytes.
<code><u>edge</u>(src, dst)</code>	Gets the edge with the specified source and destination nodes
<code><u>event_graph</u>()</code>	Get event graph
<code><u>get_all_node_types</u>()</code>	Returns all the node types in the graph.
<code><u>import_edge</u>(edge[, merge])</code>	Import a single edge into the graph.
<code><u>import_edge_as</u>(edge, new_id[, merge])</code>	Import a single edge into the graph with new id.

---

`import\_edges(edges[, merge])`

Import multiple edges into the graph.

---

`import\_edges\_as(edges, new_ids[, merge])`

Import multiple edges into the graph with new ids.

---

`import\_node(node[, merge])`

Import a single node into the graph.

---

`import\_node\_as(node, new_id[, merge])`

Import a single node into the graph with new id.

---

`import\_nodes(nodes[, merge])`

Import multiple nodes into the graph.

---

`import\_nodes\_as(nodes, new_ids[, merge])`

Import multiple nodes into the graph with new ids.

---

`load\_cached(path)`

Load PersistentGraph from a file and initialise it as a cache file.

---

`load\_edge\_deletions\_from\_pandas(df, time, ...)`

Load edges deletions from a Pandas DataFrame into the graph.

---

`load\_edge\_deletions\_from\_parquet(...[, ...])`

Load edges deletions from a Parquet file into the graph.

---

`load\_edge\_props\_from\_pandas(df, src, dst[, ...])`

Load edge properties from a Pandas DataFrame.

---

<code>load_edge_props_from_parquet(parquet_path, ...)</code>	Load edge properties from parquet file
<code>load_edges_from_pandas(df, time, src, dst[, ...])</code>	Load edges from a Pandas DataFrame into the graph.
<code>load_edges_from_parquet(parquet_path, time, ...)</code>	Load edges from a Parquet file into the graph.
<code>load_from_file(path)</code>	Load PersistentGraph from a file.
<code>load_node_props_from_pandas(df, id[, ...])</code>	Load node properties from a Pandas DataFrame.
<code>load_node_props_from_parquet(parquet_path, id)</code>	Load node properties from a parquet file.
<code>load_nodes_from_pandas(df, time, id[, ...])</code>	Load nodes from a Pandas DataFrame into the graph.
<code>load_nodes_from_parquet(parquet_path, time, id)</code>	Load nodes from a Parquet file into the graph.
<code>node(id)</code>	Gets the node with the specified id
<code>persistent_graph()</code>	Get persistent graph
<code>save_to_file(path)</code>	Saves the PersistentGraph to the given path.

---

<code><a href="#">save_to_zip</a>(path)</code>	Saves the PersistentGraph to the given path.
--	--

---

<code><a href="#">serialise</a>()</code>	Serialise PersistentGraph to bytes.
--	-------------------------------------

---

<code><a href="#">update_constant_properties</a>(properties)</code>	Updates static properties to the graph.
---	---

---

<code><a href="#">write_updates</a>()</code>	Persist the new updates by appending them to the cache file.
--	--

### `add\_constant\_properties(properties)`

Adds static properties to the graph.

#### Parameters:

properties ([dict](#)) – The static properties of the graph.

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

[None](#)

#### Raises:

GraphError – If the operation fails.

### `add\_edge(timestamp, src, dst, properties=None, layer=None, secondary_index=None)`

Adds a new edge with the given source and destination nodes and properties to the graph.

#### Parameters:

- timestamp ([int](#)) – The timestamp of the edge.
- src ([str](#) | [int](#)) – The id of the source node.
- dst ([str](#) | [int](#)) – The id of the destination node.
- properties ([PropInput](#), [optional](#)) – The properties of the edge, as a dict of string and properties
- layer ([str](#), [optional](#)) – The layer of the edge.
- secondary\_index ([int](#), [optional](#)) – The optional integer which will be used as a secondary index

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
add_node(timestamp, id, properties=None, node_type=None,
secondary_index=None)
```

Adds a new node with the given id and properties to the graph.

**Parameters:**

- timestamp ([TimeInput](#)) – The timestamp of the node.
- id ([str](#) | [int](#)) – The id of the node.
- properties ([PropInpt](#), *optional*) – The properties of the node.
- node\_type ([str](#), *optional*) – The optional string which will be used as a node type
- secondary\_index ([int](#), *optional*) – The optional integer which will be used as a secondary index

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
add_properties(timestamp, properties, secondary_index=None)
```

Adds properties to the graph.

**Parameters:**

- timestamp ([TimeInput](#)) – The timestamp of the temporal property.
- properties ([dict](#)) – The temporal properties of the graph.
- secondary\_index ([int](#), *optional*) – The optional integer which will be used as a secondary index

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

## **cache (path)**

Write PersistentGraph to cache file and initialise the cache.

Future updates are tracked. Use write\_updates to persist them to the cache file. If the file already exists its contents are overwritten.

### **Parameters:**

path ([str](#)) – The path to the cache file

### **Return type:**

[None](#)

## **create\_node (timestamp, id, properties=None, node\_type=None, secondary\_index=None)**

Creates a new node with the given id and properties to the graph. It fails if the node already exists.

### **Parameters:**

- timestamp ([TimeInput](#)) – The timestamp of the node.
- id ([str](#) | [int](#)) – The id of the node.
- properties ([PropInput](#), *optional*) – The properties of the node.
- node\_type ([str](#), *optional*) – The optional string which will be used as a node type
- secondary\_index ([int](#), *optional*) – The optional integer which will be used as a secondary index

### **Returns:**

the newly created node.

### **Return type:**

[MutableNode](#)

### **Raises:**

GraphError – If the operation fails.

## **delete\_edge (timestamp, src, dst, layer=None, secondary\_index=None)**

Deletes an edge given the timestamp, src and dst nodes and layer (optional)

### **Parameters:**

- timestamp ([int](#)) – The timestamp of the edge.
- src ([str](#) | [int](#)) – The id of the source node.
- dst ([str](#) | [int](#)) – The id of the destination node.
- layer ([str](#), *optional*) – The layer of the edge.
- secondary\_index ([int](#), *optional*) – The optional integer which will be used as a secondary index.

**Returns:**

The deleted edge

**Return type:**

[MutableEdge](#)

**Raises:**

`GraphError` – If the operation fails.

`static deserialise(bytes)`

Load PersistentGraph from serialised bytes.

**Parameters:**

`bytes (bytes)` – The serialised bytes to decode

**Return type:**

[PersistentGraph](#)

`edge(src, dst)`

Gets the edge with the specified source and destination nodes

**Parameters:**

- `src (str | int)` – the source node id
- `dst (str | int)` – the destination node id

**Returns:**

The edge with the specified source and destination nodes, or `None` if the edge does not exist

**Return type:**

[Optional\[MutableEdge\]](#)

`event_graph()`

Get event graph

**Returns:**

the graph with event semantics applied

**Return type:**

[Graph](#)

`get_all_node_types()`

Returns all the node types in the graph.

**Returns:**

A list of node types

**Return type:**

[list\[str\]](#)

```
import_edge(edge, merge=False)
```

Import a single edge into the graph.

This function takes an edge object and an optional boolean flag. If the flag is set to true, the function will merge the import of the edge even if it already exists in the graph.

**Parameters:**

- `edge` ([Edge](#)) – An edge object representing the edge to be imported.
- `merge` ([bool](#)) – An optional boolean flag indicating whether to merge the import of the edge. Defaults to False.

**Returns:**

The imported edge.

**Return type:**

[Edge](#)

**Raises:**

`GraphError` – If the operation fails.

```
import_edge_as(edge, new_id, merge=False)
```

Import a single edge into the graph with new id.

This function takes a edge object, a new edge id and an optional boolean flag. If the flag is set to true, the function will merge the import of the edge even if it already exists in the graph.

**Parameters:**

- `edge` ([Edge](#)) – A edge object representing the edge to be imported.
- `new_id` ([tuple](#)) – The ID of the new edge. It's a tuple of the source and destination node ids.
- `merge` ([bool](#)) – An optional boolean flag indicating whether to merge the import of the edge. Defaults to False.

**Returns:**

The imported edge.

**Return type:**

[Edge](#)

**Raises:**

`GraphError` – If the operation fails.

```
import_edges(edges, merge=False)
```

Import multiple edges into the graph.

This function takes a vector of edge objects and an optional boolean flag. If the flag is set to true, the function will merge the import of the edges even if they already exist in the graph.

#### Parameters:

- edges (`List[Edge]`) – A vector of edge objects representing the edges to be imported.
- merge (`bool`) – An optional boolean flag indicating whether to merge the import of the edges. Defaults to False.

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

`None`

#### Raises:

`GraphError` – If the operation fails.

`import_edges_as(edges, new_ids, merge=False)`

Import multiple edges into the graph with new ids.

This function takes a vector of edge objects, a list of new edge ids and an optional boolean flag. If the flag is set to true, the function will merge the import of the edges even if they already exist in the graph.

#### Parameters:

- edges (`List[Edge]`) – A vector of edge objects representing the edges to be imported.
- new\_ids (`list[Tuple[GID, GID]]`) – The new edge ids
- merge (`bool`) – An optional boolean flag indicating whether to merge the import of the edges. Defaults to False.

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

`None`

#### Raises:

`GraphError` – If the operation fails.

`import_node(node, merge=False)`

Import a single node into the graph.

This function takes a node object and an optional boolean flag. If the flag is set to true, the function will merge the import of the node even if it already exists in the graph.

**Parameters:**

- node ([Node](#)) – A node object representing the node to be imported.
- merge ([bool](#)) – An optional boolean flag indicating whether to merge the import of the node. Defaults to False.

**Returns:**

A Node object if the node was successfully imported, and an error otherwise.

**Return type:**

[Node](#)

**Raises:**

GraphError – If the operation fails.

`import_node_as(node, new_id, merge=False)`

Import a single node into the graph with new id.

This function takes a node object, a new node id and an optional boolean flag. If the flag is set to true, the function will merge the import of the node even if it already exists in the graph.

**Parameters:**

- node ([Node](#)) – A node object representing the node to be imported.
- new\_id ([str|int](#)) – The new node id.
- merge ([bool](#)) – An optional boolean flag indicating whether to merge the import of the node. Defaults to False.

**Returns:**

A Node object if the node was successfully imported, and an error otherwise.

**Return type:**

[Node](#)

**Raises:**

GraphError – If the operation fails.

`import_nodes(nodes, merge=False)`

Import multiple nodes into the graph.

This function takes a vector of node objects and an optional boolean flag. If the flag is set to true, the function will merge the import of the nodes even if they already exist in the graph.

**Parameters:**

- nodes ([List\[Node\]](#)) – A vector of node objects representing the nodes to be imported.

- `merge (bool)` – An optional boolean flag indicating whether to merge the import of the nodes. Defaults to False.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

`import_nodes_as(nodes, new_ids, merge=False)`

Import multiple nodes into the graph with new ids.

This function takes a vector of node objects, a list of new node ids and an optional boolean flag. If the flag is set to true, the function will merge the import of the nodes even if they already exist in the graph.

**Parameters:**

- `nodes (List\[Node\])` – A vector of node objects representing the nodes to be imported.
- `new_ids (List\[str|int\])` – A list of node IDs to use for the imported nodes.
- `merge (bool)` – An optional boolean flag indicating whether to merge the import of the nodes. Defaults to False.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

`static load_cached(path)`

Load PersistentGraph from a file and initialise it as a cache file.

Future updates are tracked. Use `write_updates` to persist them to the cache file.

**Parameters:**

`path (str)` – The path to the cache file

**Returns:**

the loaded graph with initialised cache

**Return type:**

[PersistentGraph](#)

```
load_edge_deletions_from_pandas(df, time, src, dst,
layer=None, layer_col=None)
```

Load edges deletions from a Pandas DataFrame into the graph.

#### Parameters:

- df ([DataFrame](#)) – The Pandas DataFrame containing the edges.
- time ([str](#)) – The column name for the update timestamps.
- src ([str](#)) – The column name for the source node ids.
- dst ([str](#)) – The column name for the destination node ids.
- layer ([str, optional](#)) – A constant value to use as the layer for all edges.  
Defaults to None. (cannot be used in combination with layer\_col)
- layer\_col ([str, optional](#)) – The edge layer col name in dataframe. Defaults to None. (cannot be used in combination with layer)

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

[None](#)

#### Raises:

GraphError – If the operation fails.

```
load_edge_deletions_from_parquet(parquet_path, time, src,
dst, layer=None, layer_col=None)
```

Load edges deletions from a Parquet file into the graph.

#### Parameters:

- parquet\_path ([str](#)) – Parquet file or directory of Parquet files path containing node information.
- src ([str](#)) – The column name for the source node ids.
- dst ([str](#)) – The column name for the destination node ids.
- time ([str](#)) – The column name for the update timestamps.
- layer ([str, optional](#)) – A constant value to use as the layer for all edges.  
Defaults to None. (cannot be used in combination with layer\_col)
- layer\_col ([str, optional](#)) – The edge layer col name in dataframe. Defaults to None. (cannot be used in combination with layer)

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

[None](#)

#### Raises:

GraphError – If the operation fails.

```
load_edge_props_from_pandas(df, src, dst,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edge properties from a Pandas DataFrame.

#### Parameters:

- df ([DataFrame](#)) – The Pandas DataFrame containing edge information.
- src ([str](#)) – The column name for the source node.
- dst ([str](#)) – The column name for the destination node.
- constant\_properties ([List\[str\]](#), *optional*) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput](#), *optional*) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str](#), *optional*) – The edge layer name. Defaults to None.
- layer\_col ([str](#), *optional*) – The edge layer col name in dataframe. Defaults to None.

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

[None](#)

#### Raises:

[GraphError](#) – If the operation fails.

```
load_edge_props_from_parquet(parquet_path, src, dst,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edge properties from parquet file

#### Parameters:

- parquet\_path ([str](#)) – Parquet file or directory of Parquet files path containing edge information.
- src ([str](#)) – The column name for the source node.
- dst ([str](#)) – The column name for the destination node.
- constant\_properties ([List\[str\]](#), *optional*) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput](#), *optional*) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str](#), *optional*) – The edge layer name. Defaults to None.
- layer\_col ([str](#), *optional*) – The edge layer col name in dataframe. Defaults to None.

#### Returns:

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_edges_from_pandas(df, time, src, dst, properties=None,
constant_properties=None, shared_constant_properties=None,
layer=None, layer_col=None)
```

Load edges from a Pandas DataFrame into the graph.

**Parameters:**

- df ([DataFrame](#)) – The Pandas DataFrame containing the edges.
- time ([str](#)) – The column name for the update timestamps.
- src ([str](#)) – The column name for the source node ids.
- dst ([str](#)) – The column name for the destination node ids.
- properties ([List\[str\], optional](#)) – List of edge property column names. Defaults to None.
- constant\_properties ([List\[str\], optional](#)) – List of constant edge property column names. Defaults to None.
- shared\_constant\_properties ([PropInput, optional](#)) – A dictionary of constant properties that will be added to every edge. Defaults to None.
- layer ([str, optional](#)) – A constant value to use as the layer for all edges. Defaults to None. (cannot be used in combination with layer\_col)
- layer\_col ([str, optional](#)) – The edge layer col name in dataframe. Defaults to None. (cannot be used in combination with layer)

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_edges_from_parquet(parquet_path, time, src, dst,
properties=None, constant_properties=None,
shared_constant_properties=None, layer=None, layer_col=None)
```

Load edges from a Parquet file into the graph.

**Parameters:**

- parquet\_path ([str](#)) – Parquet file or directory of Parquet files path containing edges
- time ([str](#)) – The column name for the update timestamps.
- src ([str](#)) – The column name for the source node ids.

- `dst (str)` – The column name for the destination node ids.
- `properties (List[str], optional)` – List of edge property column names. Defaults to `None`.
- `constant_properties (List[str], optional)` – List of constant edge property column names. Defaults to `None`.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every edge. Defaults to `None`.
- `layer (str, optional)` – A constant value to use as the layer for all edges. Defaults to `None`. (cannot be used in combination with `layer_col`)
- `layer_col (str, optional)` – The edge layer col name in dataframe. Defaults to `None`. (cannot be used in combination with `layer`)

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

**static load\_from\_file(path)**

Load PersistentGraph from a file.

**Parameters:**

`path (str)` – The path to the file.

**Return type:**

[PersistentGraph](#)

**load\_node\_props\_from\_pandas(df, id, node\_type=None, node\_type\_col=None, constant\_properties=None, shared\_constant\_properties=None)**

Load node properties from a Pandas DataFrame.

**Parameters:**

- `df (DataFrame)` – The Pandas DataFrame containing node information.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to `None`. (cannot be used in combination with `node_type_col`)
- `node_type_col (str, optional)` – The node type col name in dataframe. Defaults to `None`. (cannot be used in combination with `node_type`)
- `constant_properties (List[str], optional)` – List of constant node property column names. Defaults to `None`.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every node. Defaults to `None`.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_node_props_from_parquet(parquet_path, id,
node_type=None, node_type_col=None, constant_properties=None,
shared_constant_properties=None)
```

Load node properties from a parquet file.

**Parameters:**

- `parquet_path (str)` – Parquet file or directory of Parquet files path containing node information.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to None. (cannot be used in combination with `node_type_col`)
- `node_type_col (str, optional)` – The node type col name in dataframe. Defaults to None. (cannot be used in combination with `node_type`)
- `constant_properties (List\[str, optional\])` – List of constant node property column names. Defaults to None.
- `shared_constant_properties (PropInput, optional)` – A dictionary of constant properties that will be added to every node. Defaults to None.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

```
load_nodes_from_pandas(df, time, id, node_type=None,
node_type_col=None, properties=None, constant_properties=None,
shared_constant_properties=None)
```

Load nodes from a Pandas DataFrame into the graph.

**Parameters:**

- `df (DataFrame)` – The Pandas DataFrame containing the nodes.
- `time (str)` – The column name for the timestamps.
- `id (str)` – The column name for the node IDs.
- `node_type (str, optional)` – A constant value to use as the node type for all nodes. Defaults to None. (cannot be used in combination with `node_type_col`)

- `node_type_col` (`str, optional`) – The node type col name in dataframe. Defaults to `None`. (cannot be used in combination with `node_type`)
- `properties` (`List[str, optional]`) – List of node property column names. Defaults to `None`.
- `constant_properties` (`List[str, optional]`) – List of constant node property column names. Defaults to `None`.
- `shared_constant_properties` (`PropInput, optional`) – A dictionary of constant properties that will be added to every node. Defaults to `None`.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

```
load_nodes_from_parquet(parquet_path, time, id,
node_type=None, node_type_col=None, properties=None,
constant_properties=None, shared_constant_properties=None)
```

Load nodes from a Parquet file into the graph.

**Parameters:**

- `parquet_path` (`str`) – Parquet file or directory of Parquet files containing the nodes
- `time` (`str`) – The column name for the timestamps.
- `id` (`str`) – The column name for the node IDs.
- `node_type` (`str, optional`) – A constant value to use as the node type for all nodes. Defaults to `None`. (cannot be used in combination with `node_type_col`)
- `node_type_col` (`str, optional`) – The node type col name in dataframe. Defaults to `None`. (cannot be used in combination with `node_type`)
- `properties` (`List[str, optional]`) – List of node property column names. Defaults to `None`.
- `constant_properties` (`List[str, optional]`) – List of constant node property column names. Defaults to `None`.
- `shared_constant_properties` (`PropInput, optional`) – A dictionary of constant properties that will be added to every node. Defaults to `None`.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

`None`

**Raises:**

`GraphError` – If the operation fails.

**node(*id*)**

Gets the node with the specified id

**Parameters:**

*id* ([str](#) | [int](#)) – the node id

**Returns:**

The node with the specified id, or None if the node does not exist

**Return type:**

[Optional\[MutableNode\]](#)

**persistent\_graph()**

Get persistent graph

**Returns:**

the graph with persistent semantics applied

**Return type:**

[PersistentGraph](#)

**save\_to\_file(*path*)**

Saves the PersistentGraph to the given path.

**Parameters:**

*path* ([str](#)) – The path to the file.

**Return type:**

[None](#)

**save\_to\_zip(*path*)**

Saves the PersistentGraph to the given path.

**Parameters:**

*path* ([str](#)) – The path to the file.

**Return type:**

[None](#)

**serialise()**

Serialise PersistentGraph to bytes.

**Return type:**

[bytes](#)

**update\_constant\_properties(*properties*)**

Updates static properties to the graph.

**Parameters:**

properties (*dict*) – The static properties of the graph.

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

GraphError – If the operation fails.

**write\_updates()**

Persist the new updates by appending them to the cache file.

**Return type:**

[None](#)

# Node

`class Node`

Bases: [object](#)

A node (or node) in the graph.

**Methods:**

[`after`\(start\)](#)

Create a view of the Node including all events after start (exclusive).

[`at`\(time\)](#)

Create a view of the Node including all events at time.

[`before`\(end\)](#)

Create a view of the Node including all events before end (exclusive).

[`default\_layer\(\)`](#)

Return a view of Node containing only the default edge layer  
:returns: The layered view  
:rtype: Node

<code><u>degree</u>()</code>	Get the degree of this node (i.e., the number of edges that are incident to it).
<code><u>exclude_layer</u>(name)</code>	Return a view of Node containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of Node containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of Node containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of Node containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>filter_edges</u>(filter)</code>	Return a filtered view that only includes edges that satisfy the filter
<code><u>filter_exploded_edges</u>(filter)</code>	Return a filtered view that only includes exploded edges that satisfy the filter
<code><u>filter_nodes</u>(filter)</code>	Return a filtered view that only includes nodes that satisfy the filter
<code><u>has_layer</u>(name)</code>	Check if Node has the layer "name"
<code><u>history</u>()</code>	Returns the history of a node, including node additions and changes made to node.

---

**history\_date\_time()** Returns the history of a node, including node additions and changes made to node.

---

**in\_degree()** Get the in-degree of this node (i.e., the number of edges that are incident to it from other nodes).

---

**is\_active()** Check if the node is active, i.e., it's history is not empty

---

**latest()** Create a view of the Node including all events at the latest time.

---

**layer(name)** Return a view of Node containing the layer "name" Errors if the layer does not exist

---

**layers(names)** Return a view of Node containing all layers names Errors if any of the layers do not exist.

---

**out\_degree()** Get the out-degree of this node (i.e., the number of edges that are incident to it from this node).

---

**rolling(window[, step])** Creates a WindowSet with the given window size and optional step using a rolling window.

---

**shrink\_end(end)** Set the end of the window to the smaller of end and self.end()

---

**shrink\_start(start)** Set the start of the window to the larger of start and self.start()

---

**shrink\_window**(start, end)  
Shrink both the start and end of the window  
(same as calling shrink\_start followed by  
shrink\_end but more efficient)

---

**snapshot\_at**(time)  
Create a view of the Node including all events  
that have not been explicitly deleted at time.

---

**snapshot\_latest**()  
Create a view of the Node including all events  
that have not been explicitly deleted at the  
latest time.

---

**valid\_layers**(names)  
Return a view of Node containing all layers  
names Any layers that do not exist are  
ignored

---

**window**(start, end)  
Create a view of the Node including all events  
between start (inclusive) and end (exclusive)

---

#### Attributes:

---

**earliest\_date\_time** Returns the earliest datetime that the node exists.

---

**earliest\_time** Returns the earliest time that the node exists.

---

**edges** Get the edges that are incident to this node.

---

**end** Gets the latest time that this Node is valid.

---

**end\_date\_time** Gets the latest datetime that this Node is valid

---

**id** Returns the id of the node.

---

**in\_edges** Get the edges that point into this node.

---

<b><code>in_neighbours</code></b>	Get the neighbours of this node that point into this node.
<b><code>latest_date_time</code></b>	Returns the latest datetime that the node exists.
<b><code>latest_time</code></b>	Returns the latest time that the node exists.
<b><code>name</code></b>	Returns the name of the node.
<b><code>neighbours</code></b>	Get the neighbours of this node.
<b><code>node_type</code></b>	Returns the type of node
<b><code>out_edges</code></b>	Get the edges that point out of this node.
<b><code>out_neighbours</code></b>	Get the neighbours of this node that point out of this node.
<b><code>properties</code></b>	The properties of the node
<b><code>start</code></b>	Gets the start time for rolling and expanding windows for this Node
<b><code>start_date_time</code></b>	Gets the earliest datetime that this Node is valid
<b><code>window_size</code></b>	Get the window size (difference between start and end) for this Node
<b><code>after(start)</code></b>	Create a view of the Node including all events after start (exclusive).
<b>Parameters:</b>	
<b><code>start (<i>TimeInput</i>)</code></b> – The start time of the window.	
<b>Return type:</b>	
<b><code>Node</code></b>	
<b><code>at(<i>time</i>)</code></b>	Create a view of the Node including all events at time.

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[Node](#)

**before (end)**

Create a view of the Node including all events before end (exclusive).

**Parameters:**

end ([TimeInput](#)) – The end time of the window.

**Return type:**

[Node](#)

**default\_layer()**

Return a view of Node containing only the default edge layer :returns: The layered

view :rtype: Node

**degree ()**

Get the degree of this node (i.e., the number of edges that are incident to it).

**Returns:**

The degree of this node.

**Return type:**

[int](#)

**earliest\_date\_time**

Returns the earliest datetime that the node exists.

**Returns:**

The earliest datetime that the node exists as a Datetime.

**Return type:**

[datetime](#)

**earliest\_time**

Returns the earliest time that the node exists.

**Returns:**

The earliest time that the node exists as an integer.

**Return type:**

[int](#)

**edges**

Get the edges that are incident to this node.

**Returns:**

The incident edges.

**Return type:**

[Edges](#)

**end**

Gets the latest time that this Node is valid.

**Returns:**

The latest time that this Node is valid or None if the Node is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this Node is valid

**Returns:**

The latest datetime that this Node is valid or None if the Node is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of Node containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[Node](#)

**exclude\_layers(names)**

Return a view of Node containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[Node](#)

**`exclude_valid_layer(name)`**

Return a view of Node containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[Node](#)

**`exclude_valid_layers(names)`**

Return a view of Node containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[Node](#)

**`expanding(step)`**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**`filter_edges(filter)`**

Return a filtered view that only includes edges that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[Node](#)

**`filter_exploded_edges(filter)`**

Return a filtered view that only includes exploded edges that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the exploded edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

`Node`

`filter_nodes(filter)`

Return a filtered view that only includes nodes that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the node properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

`Node`

`has_layer(name)`

Check if Node has the layer “name”

**Parameters:**

`name (str)` – the name of the layer to check

**Return type:**

`bool`

`history()`

Returns the history of a node, including node additions and changes made to node.

**Returns:**

A list of unix timestamps of the event history of node.

**Return type:**

`List[int]`

`history_date_time()`

Returns the history of a node, including node additions and changes made to node.

**Returns:**

A list of timestamps of the event history of node.

**Return type:**

List[datetime]

`id`

Returns the id of the node. This is a unique identifier for the node.

**Returns:**

The id of the node.

**Return type:**

(str|int)

`in_degree()`

Get the in-degree of this node (i.e., the number of edges that are incident to it from other nodes).

**Returns:**

The in-degree of this node.

**Return type:**

int

`in_edges`

Get the edges that point into this node.

**Returns:**

The inbound edges.

**Return type:**

Edges

`in_neighbours`

Get the neighbours of this node that point into this node.

**Returns:**

The in-neighbours.

**Return type:**

PathFromNode

`is_active()`

Check if the node is active, i.e., its history is not empty

**Return type:**

bool

`latest()`

Create a view of the Node including all events at the latest time.

**Return type:**

Node

**latest\_date\_time**

Returns the latest datetime that the node exists.

**Returns:**

The latest datetime that the node exists as a Datetime.

**Return type:**

datetime

**latest\_time**

Returns the latest time that the node exists.

**Returns:**

The latest time that the node exists as an integer.

**Return type:**

int

**layer(name)**

Return a view of Node containing the layer “name” Errors if the layer does not exist

**Parameters:**

name (str) – then name of the layer.

**Returns:**

The layered view

**Return type:**

Node

**layers(names)**

Return a view of Node containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names (list[str]) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

Node

**name**

Returns the name of the node.

**Returns:**

The id of the node as a string.

**Return type:**

[str](#)

**neighbours**

Get the neighbours of this node.

**Returns:**

The neighbours (both inbound and outbound).

**Return type:**

[PathFromNode](#)

**node\_type**

Returns the type of node

**Returns:**

The node type if it is set or None otherwise.

**Return type:**

[Optional\[str\]](#)

**out\_degree()**

Get the out-degree of this node (i.e., the number of edges that are incident to it from this node).

**Returns:**

The out-degree of this node.

**Return type:**

[int](#)

**out\_edges**

Get the edges that point out of this node.

**Returns:**

The outbound edges.

**Return type:**

[Edges](#)

**out\_neighbours**

Get the neighbours of this node that point out of this node.

**Returns:**

The out-neighbours.

**Return type:**

[PathFromNode](#)

**properties**

The properties of the node

**Returns:**

A list of properties.

**Return type:**

[Properties](#)

**rolling(window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

`end (TimeInput)` – the new end time of the window

**Return type:**

[Node](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start (TimeInput)` – the new start time of the window

**Return type:**

[Node](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start` ([TimeInput](#)) – the new start time for the window
- `end` ([TimeInput](#)) – the new end time for the window

**Return type:**

[Node](#)

`snapshot_at(time)`

Create a view of the Node including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[Node](#)

`snapshot_latest()`

Create a view of the Node including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[Node](#)

`start`

Gets the start time for rolling and expanding windows for this Node

**Returns:**

The earliest time that this Node is valid or None if the Node is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this Node is valid

**Returns:**

The earliest datetime that this Node is valid or None if the Node is valid for all times.

**Return type:**

`Optional[datetime]`

`valid_layers(names)`

Return a view of Node containing all layers names Any layers that do not exist are ignored

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

`Node`

`window(start, end)`

Create a view of the Node including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start (TimelInput | None)` – The start time of the window (unbounded if None).
- `end (TimelInput | None)` – The end time of the window (unbounded if None).

**Return type:**

`Node`

`window_size`

Get the window size (difference between start and end) for this Node

**Return type:**

`Optional[int]`

# Nodes

`class Nodes`

Bases: `object`

A list of nodes that can be iterated over.

**Methods:**

`after(start)`

Create a view of the Nodes including all events after start (exclusive).

---

<code>at(time)</code>	Create a view of the Nodes including all events at time.
<code>before(end)</code>	Create a view of the Nodes including all events before end (exclusive).
<code>collect()</code>	Collect all nodes into a list
<code>default_layer()</code>	Return a view of Nodes containing only the default edge layer :returns: The layered view :rtype: Nodes
<code>degree()</code>	Returns the number of edges of the nodes
<code>exclude_layer(name)</code>	Return a view of Nodes containing all layers except the excluded name Errors if any of the layers do not exist.
<code>exclude_layers(names)</code>	Return a view of Nodes containing all layers except the excluded names Errors if any of the layers do not exist.
<code>exclude_valid_layer(name)</code>	Return a view of Nodes containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code>exclude_valid_layers(names)</code>	Return a view of Nodes containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

---

---

<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>filter_edges</u>(filter)</code>	Return a filtered view that only includes edges that satisfy the filter
<code><u>filter_exploded_edges</u>(filter)</code>	Return a filtered view that only includes exploded edges that satisfy the filter
<code><u>filter_nodes</u>(filter)</code>	Return a filtered view that only includes nodes that satisfy the filter
<code><u>has_layer</u>(name)</code>	Check if Nodes has the layer "name"
<code><u>history</u>()</code>	Returns all timestamps of nodes, when a node is added or change to a node is made.
<code><u>history_date_time</u>()</code>	Returns all timestamps of nodes, when a node is added or change to a node is made.
<code><u>in_degree</u>()</code>	Returns the number of in edges of the nodes
<code><u>latest</u>()</code>	Create a view of the Nodes including all events at the latest time.
<code><u>layer</u>(name)</code>	Return a view of Nodes containing the layer "name" Errors if the layer does not exist
<code><u>layers</u>(names)</code>	Return a view of Nodes containing all layers names Errors if any of the layers do not exist.

---

---

<code><u>out_degree()</u></code>	Returns the number of out edges of the nodes
<code><u>rolling</u>(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><u>shrink_end</u>(end)</code>	Set the end of the window to the smaller of end and self.end()
<code><u>shrink_start</u>(start)</code>	Set the start of the window to the larger of start and self.start()
<code><u>shrink_window</u>(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code><u>snapshot_at</u>(time)</code>	Create a view of the Nodes including all events that have not been explicitly deleted at time.
<code><u>snapshot_latest</u>()</code>	Create a view of the Nodes including all events that have not been explicitly deleted at the latest time.
<code><u>to_df</u>([include_property_history, ...])</code>	Converts the graph's nodes into a Pandas DataFrame.
<code><u>type_filter</u>(node_types)</code>	Filter nodes by node type
<code><u>valid_layers</u>(names)</code>	Return a view of Nodes containing all layers names Any layers that do not exist are ignored

---

---

<code>window(start, end)</code>	Create a view of the Nodes including all events between start (inclusive) and end (exclusive)
---------------------------------	---

---

#### Attributes:

<code>earliest_date_time</code>	The earliest time nodes are active as datetime objects
---------------------------------	--

---

<code>earliest_time</code>	The earliest times nodes are active
----------------------------	-------------------------------------

---

<code>edges</code>	Get the edges that are incident to this node.
--------------------	---

---

<code>end</code>	Gets the latest time that this Nodes is valid.
------------------	--

---

<code>end_date_time</code>	Gets the latest datetime that this Nodes is valid
----------------------------	---

---

<code>id</code>	The node ids
-----------------	--------------

---

<code>in_edges</code>	Get the edges that point into this node.
-----------------------	--

---

<code>in_neighbours</code>	Get the neighbours of this node that point into this node.
----------------------------	--

---

<code>latest_date_time</code>	The latest time nodes are active as datetime objects
-------------------------------	--

---

<code>latest_time</code>	The latest time nodes are active
--------------------------	----------------------------------

---

<code>name</code>	The node names
-------------------	----------------

---

<code>neighbours</code>	Get the neighbours of this node.
-------------------------	----------------------------------

---

<code>node_type</code>	The node types
------------------------	----------------

---

<code>out_edges</code>	Get the edges that point out of this node.
------------------------	--

---

---

**`out_neighbours`** Get the neighbours of this node that point out of this node.

---

**`properties`** The properties of the node

---

**`start`** Gets the start time for rolling and expanding windows for this Nodes

---

**`start_date_time`** Gets the earliest datetime that this Nodes is valid

---

**`window_size`** Get the window size (difference between start and end) for this Nodes

---

**`after(start)`**

Create a view of the Nodes including all events after start (exclusive).

**Parameters:**

`start` (*TimeInput*) – The start time of the window.

**Return type:**

**`Nodes`**

**`at(time)`**

Create a view of the Nodes including all events at time.

**Parameters:**

`time` (*TimeInput*) – The time of the window.

**Return type:**

**`Nodes`**

**`before(end)`**

Create a view of the Nodes including all events before end (exclusive).

**Parameters:**

`end` (*TimeInput*) – The end time of the window.

**Return type:**

**`Nodes`**

**`collect()`**

Collect all nodes into a list

**Returns:**

the list of nodes

**Return type:**

[list\[Node\]](#)

**default\_layer()**

Return a view of Nodes containing only the default edge layer :returns: The layered

view :rtype: Nodes

**degree()**

Returns the number of edges of the nodes

**Returns:**

a view of the undirected node degrees

**Return type:**

[DegreeView](#)

**earliest\_date\_time**

The earliest time nodes are active as datetime objects

**Returns:**

a view of the earliest active times.

**Return type:**

[EarliestDateTimeView](#)

**earliest\_time**

The earliest times nodes are active

**Returns:**

a view of the earliest active times

**Return type:**

[EarliestTimeView](#)

**edges**

Get the edges that are incident to this node.

**Returns:**

The incident edges.

**Return type:**

[NestedEdges](#)

**end**

Gets the latest time that this Nodes is valid.

**Returns:**

The latest time that this Nodes is valid or None if the Nodes is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this Nodes is valid

**Returns:**

The latest datetime that this Nodes is valid or None if the Nodes is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of Nodes containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[Nodes](#)

**exclude\_layers(names)**

Return a view of Nodes containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[Nodes](#)

**exclude\_valid\_layer(name)**

Return a view of Nodes containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[Nodes](#)

### **exclude\_valid\_layers(names)**

Return a view of Nodes containing all layers except the excluded names :param  
names: list of layer names that are excluded for the new view :type names: list[str]

#### **Returns:**

The layered view

#### **Return type:**

Nodes

### **expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

#### **Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

#### **Returns:**

A WindowSet object.

#### **Return type:**

WindowSet

### **filter\_edges(filter)**

Return a filtered view that only includes edges that satisfy the filter

#### **Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the edge properties. Construct a filter using Prop.

#### **Returns:**

The filtered view

#### **Return type:**

Nodes

### **filter\_exploded\_edges(filter)**

Return a filtered view that only includes exploded edges that satisfy the filter

#### **Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the exploded edge properties. Construct a filter using Prop.

#### **Returns:**

The filtered view

#### **Return type:**

**Nodes****filter\_nodes(filter)**

Return a filtered view that only includes nodes that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the node properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:****Nodes****has\_layer(name)**

Check if Nodes has the layer “name”

**Parameters:**

`name (str)` – the name of the layer to check

**Return type:****bool****history()**

Returns all timestamps of nodes, when a node is added or change to a node is made.

**Returns:**

a view of the node histories

**Return type:****HistoryView****history\_date\_time()**

Returns all timestamps of nodes, when a node is added or change to a node is made.

**Returns:**

a view of the node histories as datetime objects.

**Return type:****HistoryDateTimeView****id**

The node ids

**Returns:**

a view of the node ids

**Return type:**

[IdView](#)

`in_degree()`

Returns the number of in edges of the nodes

**Returns:**

a view of the in-degrees of the nodes

**Return type:**

[DegreeView](#)

`in_edges`

Get the edges that point into this node.

**Returns:**

The inbound edges.

**Return type:**

[NestedEdges](#)

`in_neighbours`

Get the neighbours of this node that point into this node.

**Returns:**

The in-neighbours.

**Return type:**

[PathFromGraph](#)

`latest()`

Create a view of the Nodes including all events at the latest time.

**Return type:**

[Nodes](#)

`latest_date_time`

The latest time nodes are active as datetime objects

**Returns:**

a view of the latest active times

**Return type:**

[LatestDateTimeView](#)

`latest_time`

The latest time nodes are active

**Returns:**

a view of the latest active times

**Return type:**

[LatestTimeView](#)

**layer (name)**

Return a view of Nodes containing the layer “name” Errors if the layer does not exist

**Parameters:**

name (*str*) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[Nodes](#)

**layers (names)**

Return a view of Nodes containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names (*list[str]*) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[Nodes](#)

**name**

The node names

**Returns:**

a view of the node names

**Return type:**

[NameView](#)

**neighbours**

Get the neighbours of this node.

**Returns:**

The neighbours (both inbound and outbound).

**Return type:**

[PathFromGraph](#)

## **node\_type**

The node types

### **Returns:**

a view of the node types

### **Return type:**

[NodeTypeView](#)

## **out\_degree()**

Returns the number of out edges of the nodes

### **Returns:**

a view of the out-degrees of the nodes

### **Return type:**

[DegreeView](#)

## **out\_edges**

Get the edges that point out of this node.

### **Returns:**

The outbound edges.

### **Return type:**

[NestedEdges](#)

## **out\_neighbours**

Get the neighbours of this node that point out of this node.

### **Returns:**

The out-neighbours.

### **Return type:**

[PathFromGraph](#)

## **properties**

The properties of the node

### **Returns:**

A view of the node properties

### **Return type:**

[PropertiesView](#)

## **rolling(window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window` (`int` | `str`) – The size of the window.
- `step` (`int` | `str` | `None`) – The step size of the window. `step` defaults to `window`.

**Returns:**

A `WindowSet` object.

**Return type:**

`WindowSet`

`shrink_end(end)`

Set the end of the window to the smaller of `end` and `self.end()`

**Parameters:**

`end` (`TimeInput`) – the new end time of the window

**Return type:**

`Nodes`

`shrink_start(start)`

Set the start of the window to the larger of `start` and `self.start()`

**Parameters:**

`start` (`TimeInput`) – the new start time of the window

**Return type:**

`Nodes`

`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling `shrink_start` followed by `shrink_end` but more efficient)

**Parameters:**

- `start` (`TimeInput`) – the new start time for the window
- `end` (`TimeInput`) – the new end time for the window

**Return type:**

`Nodes`

`snapshot_at(time)`

Create a view of the `Nodes` including all events that have not been explicitly deleted at `time`.

This is equivalent to `before(time + 1)` for `Graph` and `at(time)` for `PersistentGraph`

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[Nodes](#)

`snapshot_latest()`

Create a view of the Nodes including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[Nodes](#)

`start`

Gets the start time for rolling and expanding windows for this Nodes

**Returns:**

The earliest time that this Nodes is valid or None if the Nodes is valid for all times.

**Return type:**

[Optional\[int\]](#)

`start_date_time`

Gets the earliest datetime that this Nodes is valid

**Returns:**

The earliest datetime that this Nodes is valid or None if the Nodes is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

`to_df(include_property_history=False, convert_datetime=False)`

Converts the graph's nodes into a Pandas DataFrame.

This method will create a DataFrame with the following columns: - “name”: The name of the node. - “properties”: The properties of the node. - “update\_history”: The update history of the node.

**Parameters:**

- `include_property_history` ([bool](#)) – A boolean, if set to True, the history of each property is included, if False, only the latest value is shown. Defaults to False.
- `convert_datetime` ([bool](#)) – A boolean, if set to True will convert the timestamp to python datetimes. Defaults to False.

**Returns:**

the view of the node data as a pandas Dataframe

**Return type:**

[DataFrame](#)

`type_filter(node_types)`

Filter nodes by node type

**Parameters:**

`node_types (list[str])` – the list of node types to keep

**Returns:**

the filtered view of the nodes

**Return type:**

[Nodes](#)

`valid_layers(names)`

Return a view of Nodes containing all layers names Any layers that do not exist are ignored

**Parameters:**

`names (list[str])` – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[Nodes](#)

`window(start, end)`

Create a view of the Nodes including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start (TimeInput | None)` – The start time of the window (unbounded if None).
- `end (TimeInput | None)` – The end time of the window (unbounded if None).

**Return type:**

[Nodes](#)

`window_size`

Get the window size (difference between start and end) for this Nodes

**Return type:**

[Optional\[int\]](#)

# PathFromNode

**class PathFromNode**

Bases: [object](#)

Methods:

[\*\*after\*\*](#)(start)

Create a view of the PathFromNode including all events after start (exclusive).

[\*\*at\*\*](#)(time)

Create a view of the PathFromNode including all events at time.

[\*\*before\*\*](#)(end)

Create a view of the PathFromNode including all events before end (exclusive).

[\*\*collect\*\*](#)()

Collect all nodes into a list

[\*\*default\\_layer\*\*](#)()

Return a view of PathFromNode containing only the default edge layer :returns: The layered view :rtype: PathFromNode

[\*\*degree\*\*](#)()

the node degrees

[\*\*exclude\\_layer\*\*](#)(name)

Return a view of PathFromNode containing all layers except the excluded name Errors if any of the layers do not exist.

[\*\*exclude\\_layers\*\*](#)(names)

Return a view of PathFromNode containing all layers except the excluded names Errors if any of the layers do not exist.

[\*\*exclude\\_valid\\_layer\*\*](#)(name)

Return a view of PathFromNode containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

---

**exclude\_valid\_layers**(names) Return a view of PathFromNode containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

---

**expanding**(step) Creates a WindowSet with the given step size using an expanding window.

---

**filter\_edges**(filter) Return a filtered view that only includes edges that satisfy the filter

---

**filter\_exploded\_edges**(filter) Return a filtered view that only includes exploded edges that satisfy the filter

---

**filter\_nodes**(filter) Return a filtered view that only includes nodes that satisfy the filter

---

**has\_layer**(name) Check if PathFromNode has the layer "name"

---

**in\_degree**() the node in-degrees

---

**latest**() Create a view of the PathFromNode including all events at the latest time.

---

**layer**(name) Return a view of PathFromNode containing the layer "name" Errors if the layer does not exist

---

**layers**(names) Return a view of PathFromNode containing all layers names Errors if any of the layers do not exist.

---

**out\_degree**() the node out-degrees

---

<code>rolling(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
--------------------------------------	--

---

<code>shrink_end(end)</code>	Set the end of the window to the smaller of end and self.end()
------------------------------	--

---

<code>shrink_start(start)</code>	Set the start of the window to the larger of start and self.start()
----------------------------------	---

---

<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
--	--

---

<code>snapshot_at(time)</code>	Create a view of the PathFromNode including all events that have not been explicitly deleted at time.
--------------------------------	---

---

<code>snapshot_latest()</code>	Create a view of the PathFromNode including all events that have not been explicitly deleted at the latest time.
--------------------------------	--

---

<code>type_filter(node_types)</code>	filter nodes by type
--------------------------------------	----------------------

---

<code>valid_layers(names)</code>	Return a view of PathFromNode containing all layers names Any layers that do not exist are ignored
----------------------------------	--

---

<code>window(start, end)</code>	Create a view of the PathFromNode including all events between start (inclusive) and end (exclusive)
---------------------------------	--

---

#### Attributes:

<code>earliest_time</code>	the node earliest times
----------------------------	-------------------------

---

---

<code>edges</code>	Get the edges that are incident to this node.
<code>end</code>	Gets the latest time that this PathFromNode is valid.
<code>end_date_time</code>	Gets the latest datetime that this PathFromNode is valid
<code>id</code>	the node ids
<code>in_edges</code>	Get the edges that point into this node.
<code>in_neighbours</code>	Get the neighbours of this node that point into this node.
<code>latest_time</code>	the node latest times
<code>name</code>	the node names
<code>neighbours</code>	Get the neighbours of this node.
<code>node_type</code>	the node types
<code>out_edges</code>	Get the edges that point out of this node.
<code>out_neighbours</code>	Get the neighbours of this node that point out of this node.
<code>properties</code>	the node properties
<code>start</code>	Gets the start time for rolling and expanding windows for this PathFromNode
<code>start_date_time</code>	Gets the earliest datetime that this PathFromNode is valid
<code>window_size</code>	Get the window size (difference between start and end) for this PathFromNode

---

**`after(start)`**

Create a view of the PathFromNode including all events after start (exclusive).

**Parameters:**

`start (TimeInput)` – The start time of the window.

**Return type:**

[PathFromNode](#)

**`at(time)`**

Create a view of the PathFromNode including all events at time.

**Parameters:**

`time (TimeInput)` – The time of the window.

**Return type:**

[PathFromNode](#)

**`before(end)`**

Create a view of the PathFromNode including all events before end (exclusive).

**Parameters:**

`end (TimeInput)` – The end time of the window.

**Return type:**

[PathFromNode](#)

**`collect()`**

Collect all nodes into a list

**Returns:**

the list of nodes

**Return type:**

[list\[Node\]](#)

**`default_layer()`**

Return a view of PathFromNode containing only the default edge layer :returns: The layered view :rtype: PathFromNode

**`degree()`**

the node degrees

**`earliest_time`**

the node earliest times

**`edges`**

Get the edges that are incident to this node.

**Returns:**

The incident edges.

**Return type:**

[Edges](#)

**end**

Gets the latest time that this PathFromNode is valid.

**Returns:**

The latest time that this PathFromNode is valid or None if the PathFromNode is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this PathFromNode is valid

**Returns:**

The latest datetime that this PathFromNode is valid or None if the PathFromNode is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of PathFromNode containing all layers except the excluded name  
Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

**exclude\_layers(names)**

Return a view of PathFromNode containing all layers except the excluded names  
Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

**exclude\_valid\_layer(name)**

Return a view of PathFromNode containing all layers except the excluded name

:param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

**exclude\_valid\_layers(names)**

Return a view of PathFromNode containing all layers except the excluded names

:param names: list of layer names that are excluded for the new view :type names:

list[str]

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**filter\_edges(filter)**

Return a filtered view that only includes edges that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[PathFromNode](#)

`filter_exploded_edges(filter)`

Return a filtered view that only includes exploded edges that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the exploded edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[PathFromNode](#)

`filter_nodes(filter)`

Return a filtered view that only includes nodes that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the node properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[PathFromNode](#)

`has_layer(name)`

Check if PathFromNode has the layer “name”

**Parameters:**

`name (str)` – the name of the layer to check

**Return type:**

`bool`

`id`

the node ids

`in_degree()`

the node in-degrees

`in_edges`

Get the edges that point into this node.

**Returns:**

The inbound edges.

**Return type:**

[Edges](#)

[in\\_neighbours](#)

Get the neighbours of this node that point into this node.

**Returns:**

The in-neighbours.

**Return type:**

[PathFromNode](#)

[latest\(\)](#)

Create a view of the PathFromNode including all events at the latest time.

**Return type:**

[PathFromNode](#)

[latest\\_time](#)

the node latest times

[layer\(name\)](#)

Return a view of PathFromNode containing the layer “name” Errors if the layer does not exist

**Parameters:**

name (*str*) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

[layers\(names\)](#)

Return a view of PathFromNode containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names (*list[str]*) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[PathFromNode](#)

[name](#)

the node names

## **neighbours**

Get the neighbours of this node.

### **Returns:**

The neighbours (both inbound and outbound).

### **Return type:**

[PathFromNode](#)

**node\_type**

the node types

**out\_degree()**

the node out-degrees

**out\_edges**

Get the edges that point out of this node.

### **Returns:**

The outbound edges.

### **Return type:**

[Edges](#)

**out\_neighbours**

Get the neighbours of this node that point out of this node.

### **Returns:**

The out-neighbours.

### **Return type:**

[PathFromNode](#)

**properties**

the node properties

**rolling(window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

### **Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

### **Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[PathFromNode](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

start ([TimeInput](#)) – the new start time of the window

**Return type:**

[PathFromNode](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start ([TimeInput](#)) – the new start time for the window
- end ([TimeInput](#)) – the new end time for the window

**Return type:**

[PathFromNode](#)

**snapshot\_at(time)**

Create a view of the PathFromNode including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[PathFromNode](#)

**snapshot\_latest()**

Create a view of the PathFromNode including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[PathFromNode](#)

**start**

Gets the start time for rolling and expanding windows for this PathFromNode

**Returns:**

The earliest time that this PathFromNode is valid or None if the PathFromNode is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this PathFromNode is valid

**Returns:**

The earliest datetime that this PathFromNode is valid or None if the PathFromNode is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**type\_filter(node\_types)**

filter nodes by type

**Parameters:**

node\_types ([list\[str\]](#)) – the node types to keep

**Returns:**

the filtered view

**Return type:**

[PathFromNode](#)

**valid\_layers(names)**

Return a view of PathFromNode containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

## [PathFromNode](#)

`window(start, end)`

Create a view of the PathFromNode including all events between start (inclusive) and end (exclusive)

### Parameters:

- `start` ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- `end` ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

### Return type:

[PathFromNode](#)

`window_size`

Get the window size (difference between start and end) for this PathFromNode

### Return type:

[Optional\[int\]](#)

# PathFromGraph

`class PathFromGraph`

Bases: [object](#)

### Methods:

[after](#)(start)

Create a view of the PathFromGraph including all events after start (exclusive).

[at](#)(time)

Create a view of the PathFromGraph including all events at time.

[before](#)(end)

Create a view of the PathFromGraph including all events before end (exclusive).

[collect](#)()

Collect all nodes into a list

[default\\_layer](#)()

Return a view of PathFromGraph containing only the default edge layer :returns: The layered view :rtype: PathFromGraph

---

<code><u>degree</u>()</code>	the node degrees
<code><u>exclude_layer</u>(name)</code>	Return a view of PathFromGraph containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of PathFromGraph containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of PathFromGraph containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of PathFromGraph containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>filter_edges</u>(filter)</code>	Return a filtered view that only includes edges that satisfy the filter
<code><u>filter_exploded_edges</u>(filter)</code>	Return a filtered view that only includes exploded edges that satisfy the filter
<code><u>filter_nodes</u>(filter)</code>	Return a filtered view that only includes nodes that satisfy the filter
<code><u>has_layer</u>(name)</code>	Check if PathFromGraph has the layer "name"

---

---

<b><a href="#">history()</a></b>	Returns all timestamps of nodes, when an node is added or change to an node is made.
<b><a href="#">history_date_time()</a></b>	Returns all timestamps of nodes, when an node is added or change to an node is made.
<b><a href="#">in_degree()</a></b>	the node in-degrees
<b><a href="#">latest()</a></b>	Create a view of the PathFromGraph including all events at the latest time.
<b><a href="#">layer(name)</a></b>	Return a view of PathFromGraph containing the layer "name" Errors if the layer does not exist
<b><a href="#">layers(names)</a></b>	Return a view of PathFromGraph containing all layers names Errors if any of the layers do not exist.
<b><a href="#">out_degree()</a></b>	the node out-degrees
<b><a href="#">rolling(window[, step])</a></b>	Creates a WindowSet with the given window size and optional step using a rolling window.
<b><a href="#">shrink_end(end)</a></b>	Set the end of the window to the smaller of end and self.end()
<b><a href="#">shrink_start(start)</a></b>	Set the start of the window to the larger of start and self.start()
<b><a href="#">shrink_window(start, end)</a></b>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)

---

---

<code><u>snapshot_at</u>(time)</code>	Create a view of the PathFromGraph including all events that have not been explicitly deleted at time.
<code><u>snapshot_latest</u>()</code>	Create a view of the PathFromGraph including all events that have not been explicitly deleted at the latest time.
<code><u>type_filter</u>(node_types)</code>	filter nodes by type
<code><u>valid_layers</u>(names)</code>	Return a view of PathFromGraph containing all layers names Any layers that do not exist are ignored
<code><u>window</u>(start, end)</code>	Create a view of the PathFromGraph including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

---

<code><u>earliest_date_time</u></code>	Returns the earliest date time of the nodes.
<code><u>earliest_time</u></code>	the node earliest times
<code><u>edges</u></code>	Get the edges that are incident to this node.
<code><u>end</u></code>	Gets the latest time that this PathFromGraph is valid.
<code><u>end_date_time</u></code>	Gets the latest datetime that this PathFromGraph is valid
<code><u>id</u></code>	the node ids
<code><u>in_edges</u></code>	Get the edges that point into this node.

---

---

<b><code>in_neighbours</code></b>	Get the neighbours of this node that point into this node.
<b><code>latest_date_time</code></b>	Returns the latest date time of the nodes.
<b><code>latest_time</code></b>	the node latest times
<b><code>name</code></b>	the node names
<b><code>neighbours</code></b>	Get the neighbours of this node.
<b><code>node_type</code></b>	the node types
<b><code>out_edges</code></b>	Get the edges that point out of this node.
<b><code>out_neighbours</code></b>	Get the neighbours of this node that point out of this node.
<b><code>properties</code></b>	the node properties
<b><code>start</code></b>	Gets the start time for rolling and expanding windows for this PathFromGraph
<b><code>start_date_time</code></b>	Gets the earliest datetime that this PathFromGraph is valid
<b><code>window_size</code></b>	Get the window size (difference between start and end) for this PathFromGraph
<b><code>after(start)</code></b>	Create a view of the PathFromGraph including all events after start (exclusive).
<b>Parameters:</b>	
<b><code>start (<i>TimeInput</i>)</code></b> – The start time of the window.	
<b>Return type:</b>	
<b><code>PathFromGraph</code></b>	
<b><code>at(<i>time</i>)</code></b>	

Create a view of the PathFromGraph including all events at time.

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[PathFromGraph](#)

**before (end)**

Create a view of the PathFromGraph including all events before end (exclusive).

**Parameters:**

end ([TimeInput](#)) – The end time of the window.

**Return type:**

[PathFromGraph](#)

**collect()**

Collect all nodes into a list

**Returns:**

the list of nodes

**Return type:**

[list\[list\[Node\]\]](#)

**default\_layer()**

Return a view of PathFromGraph containing only the default edge layer .returns: The layered view :rtype: PathFromGraph

**degree()**

the node degrees

**earliest\_date\_time**

Returns the earliest date time of the nodes.

**earliest\_time**

the node earliest times

**edges**

Get the edges that are incident to this node.

**Returns:**

The incident edges.

**Return type:**

[NestedEdges](#)

**end**

Gets the latest time that this PathFromGraph is valid.

**Returns:**

The latest time that this PathFromGraph is valid or None if the PathFromGraph is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this PathFromGraph is valid

**Returns:**

The latest datetime that this PathFromGraph is valid or None if the PathFromGraph is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of PathFromGraph containing all layers except the excluded name  
Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**exclude\_layers(names)**

Return a view of PathFromGraph containing all layers except the excluded names  
Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**exclude\_valid\_layer(name)**

Return a view of PathFromGraph containing all layers except the excluded name

:param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**exclude\_valid\_layers(names)**

Return a view of PathFromGraph containing all layers except the excluded names

:param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**filter\_edges(filter)**

Return a filtered view that only includes edges that satisfy the filter

**Parameters:**

filter ([PropertyFilter](#)) – The filter to apply to the edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

[PathFromGraph](#)

**filter\_exploded\_edges(filter)**

Return a filtered view that only includes exploded edges that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the exploded edge properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

`PathFromGraph`

`filter_nodes(filter)`

Return a filtered view that only includes nodes that satisfy the filter

**Parameters:**

`filter (PropertyFilter)` – The filter to apply to the node properties. Construct a filter using Prop.

**Returns:**

The filtered view

**Return type:**

`PathFromGraph`

`has_layer(name)`

Check if PathFromGraph has the layer “name”

**Parameters:**

`name (str)` – the name of the layer to check

**Return type:**

`bool`

`history()`

Returns all timestamps of nodes, when an node is added or change to an node is made.

`history_date_time()`

Returns all timestamps of nodes, when an node is added or change to an node is made.

`id`

the node ids

`in_degree()`

the node in-degrees

`in_edges`

Get the edges that point into this node.

**Returns:**

The inbound edges.

**Return type:**

[NestedEdges](#)

**in\_neighbours**

Get the neighbours of this node that point into this node.

**Returns:**

The in-neighbours.

**Return type:**

[PathFromGraph](#)

**latest()**

Create a view of the PathFromGraph including all events at the latest time.

**Return type:**

[PathFromGraph](#)

**latest\_date\_time**

Returns the latest date time of the nodes.

**latest\_time**

the node latest times

**layer(name)**

Return a view of PathFromGraph containing the layer “name” Errors if the layer does not exist

**Parameters:**

name (*str*) – then name of the layer.

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**layers(names)**

Return a view of PathFromGraph containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names (*list[str]*) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

**name**

the node names

**neighbours**

Get the neighbours of this node.

**Returns:**

The neighbours (both inbound and outbound).

**Return type:**[PathFromGraph](#)**node\_type**

the node types

**out\_degree()**

the node out-degrees

**out\_edges**

Get the edges that point out of this node.

**Returns:**

The outbound edges.

**Return type:**[NestedEdges](#)**out\_neighbours**

Get the neighbours of this node that point out of this node.

**Returns:**

The out-neighbours.

**Return type:**[PathFromGraph](#)**properties**

the node properties

**rolling(window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- `window (int | str)` – The size of the window.
- `step (int | str | None)` – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[PathFromGraph](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

start ([TimeInput](#)) – the new start time of the window

**Return type:**

[PathFromGraph](#)

**shrink\_window(start, end)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- start ([TimeInput](#)) – the new start time for the window
- end ([TimeInput](#)) – the new end time for the window

**Return type:**

[PathFromGraph](#)

**snapshot\_at(time)**

Create a view of the PathFromGraph including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

time ([TimeInput](#)) – The time of the window.

**Return type:**

[PathFromGraph](#)

**snapshot\_latest()**

Create a view of the PathFromGraph including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[PathFromGraph](#)

**start**

Gets the start time for rolling and expanding windows for this PathFromGraph

**Returns:**

The earliest time that this PathFromGraph is valid or None if the PathFromGraph is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this PathFromGraph is valid

**Returns:**

The earliest datetime that this PathFromGraph is valid or None if the PathFromGraph is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**type\_filter(node\_types)**

filter nodes by type

**Parameters:**

node\_types ([list\[str\]](#)) – the node types to keep

**Returns:**

the filtered view

**Return type:**

[PathFromGraph](#)

**valid\_layers(names)**

Return a view of PathFromGraph containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[PathFromGraph](#)

`window(start, end)`

Create a view of the PathFromGraph including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start` ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- `end` ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

[PathFromGraph](#)

`window_size`

Get the window size (difference between start and end) for this PathFromGraph

**Return type:**

[Optional\[int\]](#)

## MutableNode

`class MutableNode`

Bases: [Node](#)

Methods:

[`add\_constant\_properties\(properties\)`](#)

Add constant properties to a node in the graph.

[`add\_updates\(t\[, properties, secondary\_index\]\)`](#)

Add updates to a node in the graph at a specified time.

[`set\_node\_type\(new\_type\)`](#)

Set the type on the node.

[`update\_constant\_properties\(properties\)`](#)

Update constant properties of a node in the graph overwriting existing values.

[`add\_constant\_properties\(properties\)`](#)

Add constant properties to a node in the graph. This function is used to add properties to a node that remain constant and do not change over time. These properties are fundamental attributes of the node.

#### Parameters:

properties (*PropInput*) – A dictionary of properties to be added to the node. Each key is a string representing the property name, and each value is of type Prop representing the property value.

**add\_updates(t, properties=None, secondary\_index=None)**

Add updates to a node in the graph at a specified time. This function allows for the addition of property updates to a node within the graph. The updates are time-stamped, meaning they are applied at the specified time.

#### Parameters:

- t (*TimeInput*) – The timestamp at which the updates should be applied.
- properties (*PropInput, optional*) – A dictionary of properties to update. Each key is a string representing the property name, and each value is of type Prop representing the property value. If None, no properties are updated.
- secondary\_index (*int, optional*) – The optional integer which will be used as a secondary index

#### Returns:

This function does not return a value, if the operation is successful.

#### Return type:

*None*

#### Raises:

GraphError – If the operation fails.

**set\_node\_type(new\_type)**

Set the type on the node. This only works if the type has not been previously set, otherwise will throw an error

#### Parameters:

new\_type (*str*) – The new type to be set

**update\_constant\_properties(properties)**

Update constant properties of a node in the graph overwriting existing values. This function is used to add properties to a node that remain constant and do not change over time. These properties are fundamental attributes of the node.

#### Parameters:

**properties** (*PropInput*) – A dictionary of properties to be added to the node. Each key is a string representing the property name, and each value is of type Prop representing the property value.

# Edge

**class Edge**

Bases: [object](#)

PyEdge is a Python class that represents an edge in the graph. An edge is a directed connection between two nodes.

Methods:

---

[\*\*after\*\*](#)(start)

Create a view of the Edge including all events after start (exclusive).

---

[\*\*at\*\*](#)(time)

Create a view of the Edge including all events at time.

---

[\*\*before\*\*](#)(end)

Create a view of the Edge including all events before end (exclusive).

---

[\*\*default\\_layer\*\*](#)()

Return a view of Edge containing only the default edge layer  
:returns: The layered view  
:rtype: Edge

---

[\*\*deletions\*\*](#)()

Returns a list of timestamps of when an edge is deleted

---

[\*\*deletions\\_data\\_time\*\*](#)()

Returns a list of timestamps of when an edge is deleted

---

[\*\*exclude\\_layer\*\*](#)(name)

Return a view of Edge containing all layers except the excluded name Errors if any of the layers do not exist.

---

<code><u>exclude_layers</u>(names)</code>	Return a view of Edge containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of Edge containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of Edge containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>explode</u>()</code>	Explodes returns an edge object for each update within the original edge.
<code><u>explode_layers</u>()</code>	Explode layers returns an edge object for each layer within the original edge.
<code><u>has_layer</u>(name)</code>	Check if Edge has the layer "name"
<code><u>history</u>()</code>	Returns a list of timestamps of when an edge is added or change to an edge is made.
<code><u>history_counts</u>()</code>	Returns the number of times an edge is added or change to an edge is made.
<code><u>history_date_time</u>()</code>	Returns a list of timestamps of when an edge is added or change to an edge is made.

---

**`is_active()`** Check if the edge is currently active (i.e., has at least one update within this period) :rtype: bool

---

**`is_deleted()`** Check if the edge is currently deleted :rtype: bool

---

**`is_self_loop()`** Check if the edge is on the same node :rtype: bool

---

**`is_valid()`** Check if the edge is currently valid (i.e., not deleted) :rtype: bool

---

**`latest()`** Create a view of the Edge including all events at the latest time.

---

**`layer(name)`** Return a view of Edge containing the layer "name" Errors if the layer does not exist

---

**`layers(names)`** Return a view of Edge containing all layers names Errors if any of the layers do not exist.

---

**`rolling(window[, step])`** Creates a WindowSet with the given window size and optional step using a rolling window.

---

**`shrink_end(end)`** Set the end of the window to the smaller of end and self.end()

---

**`shrink_start(start)`** Set the start of the window to the larger of start and self.start()

---

**shrink\_window**(start, end)  
Shrink both the start and end of the window  
(same as calling shrink\_start followed by  
shrink\_end but more efficient)

---

**snapshot\_at**(time)  
Create a view of the Edge including all events  
that have not been explicitly deleted at time.

---

**snapshot\_latest**()  
Create a view of the Edge including all events  
that have not been explicitly deleted at the  
latest time.

---

**valid\_layers**(names)  
Return a view of Edge containing all layers  
names Any layers that do not exist are  
ignored

---

**window**(start, end)  
Create a view of the Edge including all events  
between start (inclusive) and end (exclusive)

---

#### Attributes:

---

**date\_time** Gets the datetime of an exploded edge.

---

**dst** Returns the destination node of the edge.

---

**earliest\_date\_time** Gets of earliest datetime of an edge.

---

**earliest\_time** Gets the earliest time of an edge.

---

**end** Gets the latest time that this Edge is valid.

---

**end\_date\_time** Gets the latest datetime that this Edge is valid

---

**id** The id of the edge.

---

<code>latest_date_time</code>	Gets of latest datetime of an edge.
<code>latest_time</code>	Gets the latest time of an edge.
<code>layer_name</code>	Gets the name of the layer this edge belongs to - assuming it only belongs to one layer
<code>layer_names</code>	Gets the names of the layers this edge belongs to
<code>nbr</code>	Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)
<code>properties</code>	Returns a view of the properties of the edge.
<code>src</code>	Returns the source node of the edge.
<code>start</code>	Gets the start time for rolling and expanding windows for this Edge
<code>start_date_time</code>	Gets the earliest datetime that this Edge is valid
<code>time</code>	Gets the time of an exploded edge.
<code>window_size</code>	Get the window size (difference between start and end) for this Edge

---

#### `after(start)`

Create a view of the Edge including all events after start (exclusive).

#### **Parameters:**

`start (TimeInput)` – The start time of the window.

#### **Return type:**

`Edge`

#### `at(time)`

Create a view of the Edge including all events at time.

#### **Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[Edge](#)

`before(end)`

Create a view of the Edge including all events before end (exclusive).

**Parameters:**

`end` ([TimeInput](#)) – The end time of the window.

**Return type:**

[Edge](#)

`date_time`

Gets the datetime of an exploded edge.

**Returns:**

the datetime of an exploded edge

**Return type:**

[datetime](#)

`default_layer()`

Return a view of Edge containing only the default edge layer :returns: The layered view :rtype: Edge

`deletions()`

Returns a list of timestamps of when an edge is deleted

**Returns:**

A list of unix timestamps

**Return type:**

[List\[int\]](#)

`deletions_data_time()`

Returns a list of timestamps of when an edge is deleted

**Returns:**

`List[datetime]`

`dst`

Returns the destination node of the edge.

`earliest_date_time`

Gets of earliest datetime of an edge.

**Returns:**

the earliest datetime of an edge

**Return type:**

[datetime](#)

**earliest\_time**

Gets the earliest time of an edge.

**Returns:**

The earliest time of an edge

**Return type:**

[int](#)

**end**

Gets the latest time that this Edge is valid.

**Returns:**

The latest time that this Edge is valid or None if the Edge is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this Edge is valid

**Returns:**

The latest datetime that this Edge is valid or None if the Edge is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of Edge containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[Edge](#)

**exclude\_layers(names)**

Return a view of Edge containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[Edge](#)

**exclude\_valid\_layer(name)**

Return a view of Edge containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[Edge](#)

**exclude\_valid\_layers(names)**

Return a view of Edge containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[Edge](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**explode()**

Explodes returns an edge object for each update within the original edge.

**explode\_layers()**

Explode layers returns an edge object for each layer within the original edge. These new edge object contains only updates from respective layers.

**has\_layer(name)**

Check if Edge has the layer “name”

**Parameters:**

`name (str)` – the name of the layer to check

**Return type:**

`bool`

**history()**

Returns a list of timestamps of when an edge is added or change to an edge is made.

**Returns:**

A list of unix timestamps.

**Return type:**

`List[int]`

**history\_counts()**

Returns the number of times an edge is added or change to an edge is made.

**Returns:**

The number of times an edge is added or change to an edge is made.

**Return type:**

`int`

**history\_date\_time()**

Returns a list of timestamps of when an edge is added or change to an edge is made.

**Returns:**

`List[datetime]`

**id**

The id of the edge.

**is\_active()**

Check if the edge is currently active (i.e., has at least one update within this period)

:rtype: bool

**is\_deleted()**

Check if the edge is currently deleted :rtype: bool

**is\_self\_loop()**

Check if the edge is on the same node :rtype: bool

**is\_valid()**

Check if the edge is currently valid (i.e., not deleted) :rtype: bool

**latest()**

Create a view of the Edge including all events at the latest time.

**Return type:**

[Edge](#)

**latest\_date\_time**

Gets of latest datetime of an edge.

**Returns:**

the latest datetime of an edge

**Return type:**

[datetime](#)

**latest\_time**

Gets the latest time of an edge.

**Returns:**

The latest time of an edge

**Return type:**

[int](#)

**layer(name)**

Return a view of Edge containing the layer “name” Errors if the layer does not exist

**Parameters:**

`name (str) – then name of the layer.`

**Returns:**

The layered view

**Return type:**

[Edge](#)

**layer\_name**

Gets the name of the layer this edge belongs to - assuming it only belongs to one layer

**Returns:**

The name of the layer

**Return type:**

[str](#)

**layer\_names**

Gets the names of the layers this edge belongs to

**Returns:**

List[str]- The name of the layer

**layers (names)**

Return a view of Edge containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[Edge](#)

**nbr**

Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)

**properties**

Returns a view of the properties of the edge.

**Returns:**

Properties on the Edge.

**rolling (window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- window ([int](#) | [str](#)) – The size of the window.
- step ([int](#) | [str](#) | [None](#)) – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

`end` ([TimeInput](#)) – the new end time of the window

**Return type:**

[Edge](#)

`shrink_start(start)`

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start` ([TimeInput](#)) – the new start time of the window

**Return type:**

[Edge](#)

`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start` ([TimeInput](#)) – the new start time for the window
- `end` ([TimeInput](#)) – the new end time for the window

**Return type:**

[Edge](#)

`snapshot_at(time)`

Create a view of the Edge including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[Edge](#)

`snapshot_latest()`

Create a view of the Edge including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**

[Edge](#)

`src`

Returns the source node of the edge.

**start**

Gets the start time for rolling and expanding windows for this Edge

**Returns:**

The earliest time that this Edge is valid or None if the Edge is valid for all times.

**Return type:**

[Optional\[int\]](#)

**start\_date\_time**

Gets the earliest datetime that this Edge is valid

**Returns:**

The earliest datetime that this Edge is valid or None if the Edge is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**time**

Gets the time of an exploded edge.

**Returns:**

The time of an exploded edge

**Return type:**

[int](#)

**valid\_layers (names)**

Return a view of Edge containing all layers names Any layers that do not exist are ignored

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[Edge](#)

**window (start, end)**

Create a view of the Edge including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start ([TimeInput](#) | [None](#)) – The start time of the window (unbounded if None).
- end ([TimeInput](#) | [None](#)) – The end time of the window (unbounded if None).

**Return type:**

[Edge](#)

**window\_size**

Get the window size (difference between start and end) for this Edge

**Return type:**

[Optional\[int\]](#)

# Edges

**class Edges**

Bases: [object](#)

A list of edges that can be iterated over.

Methods:

[after](#)(start)

Create a view of the Edges including all events after start (exclusive).

[at](#)(time)

Create a view of the Edges including all events at time.

[before](#)(end)

Create a view of the Edges including all events before end (exclusive).

[collect](#)()

Collect all edges into a list

[count](#)()

Returns the number of edges

[default\\_layer](#)()

Return a view of Edges containing only the default edge layer :returns: The layered view :rtype: Edges

[deletions](#)()

Returns all timestamps of edges where an edge is deleted

---

<code><u>deletions_date_time</u>()</code>	Returns all timestamps of edges where an edge is deleted
<code><u>exclude_layer</u>(name)</code>	Return a view of Edges containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of Edges containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of Edges containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of Edges containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>explode</u>()</code>	Explodes returns an edge object for each update within the original edge.
<code><u>explode_layers</u>()</code>	Explode layers returns an edge object for each layer within the original edge.
<code><u>has_layer</u>(name)</code>	Check if Edges has the layer "name"

---

<code><u>history</u>()</code>	Returns all timestamps of edges, when an edge is added or change to an edge is made.
<code><u>history_counts</u>()</code>	
<code><u>history_date_time</u>()</code>	Returns all timestamps of edges, when an edge is added or change to an edge is made.
<code><u>is_active</u>()</code>	
<code><u>is_deleted</u>()</code>	Check if the edges are deleted
<code><u>is_self_loop</u>()</code>	Check if the edges are on the same node
<code><u>is_valid</u>()</code>	Check if the edges are valid (i.e. not deleted).
<code><u>latest</u>()</code>	Create a view of the Edges including all events at the latest time.
<code><u>layer</u>(name)</code>	Return a view of Edges containing the layer "name" Errors if the layer does not exist
<code><u>layers</u>(names)</code>	Return a view of Edges containing all layers names Errors if any of the layers do not exist.
<code><u>rolling</u>(window[, step])</code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><u>shrink_end</u>(end)</code>	Set the end of the window to the smaller of end and self.end()

---

<code>shrink_start(start)</code>	Set the start of the window to the larger of start and self.start()
<code>shrink_window(start, end)</code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)
<code>snapshot_at(time)</code>	Create a view of the Edges including all events that have not been explicitly deleted at time.
<code>snapshot_latest()</code>	Create a view of the Edges including all events that have not been explicitly deleted at the latest time.
<code>to_df([include_property_history, ...])</code>	Converts the graph's edges into a Pandas DataFrame.
<code>valid_layers(names)</code>	Return a view of Edges containing all layers names Any layers that do not exist are ignored
<code>window(start, end)</code>	Create a view of the Edges including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

---

<code>date_time</code>	Returns the date times of exploded edges
<code>dst</code>	Returns the destination node of the edge.
<code>earliest_date_time</code>	Returns the earliest date time of the edges.
<code>earliest_time</code>	Returns the earliest time of the edges.

---

---

<code>end</code>	Gets the latest time that this Edges is valid.
<code>end_date_time</code>	Gets the latest datetime that this Edges is valid
<code>id</code>	Returns all ids of the edges.
<code>latest_date_time</code>	Returns the latest date time of the edges.
<code>latest_time</code>	Returns the latest time of the edges.
<code>layer_name</code>	Get the layer name that all edges belong to - assuming they only belong to one layer
<code>layer_names</code>	Get the layer names that all edges belong to - assuming they only belong to one layer
<code>nbr</code>	Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)
<code>properties</code>	Returns all properties of the edges
<code>src</code>	Returns the source node of the edge.
<code>start</code>	Gets the start time for rolling and expanding windows for this Edges
<code>start_date_time</code>	Gets the earliest datetime that this Edges is valid
<code>time</code>	Returns the times of exploded edges
<code>window_size</code>	Get the window size (difference between start and end) for this Edges
<code>after(start)</code>	Create a view of the Edges including all events after start (exclusive).

**Parameters:**

`start` ([TimeInput](#)) – The start time of the window.

**Return type:**

[Edges](#)

`at(time)`

Create a view of the Edges including all events at time.

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[Edges](#)

`before(end)`

Create a view of the Edges including all events before end (exclusive).

**Parameters:**

`end` ([TimeInput](#)) – The end time of the window.

**Return type:**

[Edges](#)

`collect()`

Collect all edges into a list

**Returns:**

the list of edges

**Return type:**

[list\[Edge\]](#)

`count()`

Returns the number of edges

`date_time`

Returns the date times of exploded edges

**Returns:**

A list of date times.

`default_layer()`

Return a view of Edges containing only the default edge layer :returns: The layered view :rtype: Edges

`deletions()`

Returns all timestamps of edges where an edge is deleted

**Returns:**

A list of lists of unix timestamps

`deletions_date_time()`

Returns all timestamps of edges where an edge is deleted

**Returns:**

A list of lists of DateTime objects

`dst`

Returns the destination node of the edge.

`earliest_date_time`

Returns the earliest date time of the edges.

**Returns:**

Earliest date time of the edges.

`earliest_time`

Returns the earliest time of the edges.

Returns: Earliest time of the edges.

`end`

Gets the latest time that this Edges is valid.

**Returns:**

The latest time that this Edges is valid or None if the Edges is valid for all times.

**Return type:**

`Optional[int]`

`end_date_time`

Gets the latest datetime that this Edges is valid

**Returns:**

The latest datetime that this Edges is valid or None if the Edges is valid for all times.

**Return type:**

`Optional[datetime]`

`exclude_layer(name)`

Return a view of Edges containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

`name (str)` – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[Edges](#)

**exclude\_layers(names)**

Return a view of Edges containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[Edges](#)

**exclude\_valid\_layer(name)**

Return a view of Edges containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str

**Returns:**

The layered view

**Return type:**

[Edges](#)

**exclude\_valid\_layers(names)**

Return a view of Edges containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[Edges](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**explode()**

Explodes returns an edge object for each update within the original edge.

**explode\_layers()**

Explode layers returns an edge object for each layer within the original edge. These new edge object contains only updates from respective layers.

**has\_layer(name)**

Check if Edges has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**history()**

Returns all timestamps of edges, when an edge is added or change to an edge is made.

**Returns:**

A list of lists unix timestamps.

**history\_counts()**

**history\_date\_time()**

Returns all timestamps of edges, when an edge is added or change to an edge is made.

**Returns:**

A list of lists of timestamps.

**id**

Returns all ids of the edges.

**is\_active()**

**is\_deleted()**

Check if the edges are deleted

**is\_self\_loop()**

Check if the edges are on the same node

**is\_valid()**

Check if the edges are valid (i.e. not deleted)

**latest()**

Create a view of the Edges including all events at the latest time.

**Return type:**

Edges

**latest\_date\_time**

Returns the latest date time of the edges.

**Returns:**

Latest date time of the edges.

**latest\_time**

Returns the latest time of the edges.

**Returns:**

Latest time of the edges.

**layer(name)**

Return a view of Edges containing the layer “name” Errors if the layer does not exist

**Parameters:**

**name (*str*) – then name of the layer.**

**Returns:**

The layered view

**Return type:**

Edges

**layer\_name**

Get the layer name that all edges belong to - assuming they only belong to one layer

**Returns:**

The name of the layer

**layer\_names**

Get the layer names that all edges belong to - assuming they only belong to one layer

**Returns:**

A list of layer names

**layers(names)**

Return a view of Edges containing all layers names Errors if any of the layers do not exist.

**Parameters:**

**names (*list[str]*) – list of layer names for the new view**

**Returns:**

**The layered view**

**Return type:**

[Edges](#)

**nbr**

Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)

**properties**

Returns all properties of the edges

**rolling(*window*, *step*=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- *window* ([int](#) | [str](#)) – The size of the window.
- *step* ([int](#) | [str](#) | [None](#)) – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(*end*)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

*end* ([TimeInput](#)) – the new end time of the window

**Return type:**

[Edges](#)

**shrink\_start(*start*)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

*start* ([TimeInput](#)) – the new start time of the window

**Return type:**

[Edges](#)

**shrink\_window(*start*, *end*)**

Shrink both the start and end of the window (same as calling shrink\_start followed by shrink\_end but more efficient)

**Parameters:**

- `start` ([TimeInput](#)) – the new start time for the window
- `end` ([TimeInput](#)) – the new end time for the window

**Return type:**[Edges](#)`snapshot_at(time)`

Create a view of the Edges including all events that have not been explicitly deleted at time.

This is equivalent to before(time + 1) for Graph and at(time) for PersistentGraph

**Parameters:**`time` ([TimeInput](#)) – The time of the window.**Return type:**[Edges](#)`snapshot_latest()`

Create a view of the Edges including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for Graph and latest() for PersistentGraph

**Return type:**[Edges](#)`src`

Returns the source node of the edge.

`start`

Gets the start time for rolling and expanding windows for this Edges

**Returns:**

The earliest time that this Edges is valid or None if the Edges is valid for all times.

**Return type:**[Optional\[int\]](#)`start_date_time`

Gets the earliest datetime that this Edges is valid

**Returns:**

The earliest datetime that this Edges is valid or None if the Edges is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

### time

Returns the times of exploded edges

### Returns:

Time of edge

```
to_df(include_property_history=True, convert_datetime=False,  
explode=False)
```

Converts the graph's edges into a Pandas DataFrame.

This method will create a DataFrame with the following columns: - “src”: The source node of the edge. - “dst”: The destination node of the edge. - “layer”: The layer of the edge. - “properties”: The properties of the edge. - “update\_history”: The update history of the edge. This column will be included if include\_update\_history is set to true.

### Parameters:

- `include_property_history (bool)` – A boolean, if set to True, the history of each property is included, if False, only the latest value is shown. Ignored if exploded. Defaults to True.
- `convert_datetime (bool)` – A boolean, if set to True will convert the timestamp to python datetimes. Defaults to False.
- `explode (bool)` – A boolean, if set to True, will explode each edge update into its own row. Defaults to False.

### Returns:

If successful, this PyObject will be a Pandas DataFrame.

### Return type:

[DataFrame](#)

```
valid_layers(names)
```

Return a view of Edges containing all layers names Any layers that do not exist are ignored

### Parameters:

`names (list[str])` – list of layer names for the new view

### Returns:

The layered view

### Return type:

[Edges](#)

```
window(start, end)
```

Create a view of the Edges including all events between start (inclusive) and end (exclusive)

**Parameters:**

- `start` (`TimeInput` | `None`) – The start time of the window (unbounded if None).
- `end` (`TimeInput` | `None`) – The end time of the window (unbounded if None).

**Return type:**

`Edges`

`window_size`

Get the window size (difference between start and end) for this Edges

**Return type:**

`Optional[int]`

# NestedEdges

`class NestedEdges`

Bases: `object`

**Methods:**

`after(start)`

Create a view of the NestedEdges including all events after start (exclusive).

`at(time)`

Create a view of the NestedEdges including all events at time.

`before(end)`

Create a view of the NestedEdges including all events before end (exclusive).

`collect()`

Collect all edges into a list

`default_layer()`

Return a view of NestedEdges containing only the default edge layer  
:returns: The layered view  
:rtype: NestedEdges

`deletions()`

Returns all timestamps of edges, where an edge is deleted

---

<code><u>deletions_date_time</u>()</code>	Returns all timestamps of edges, where an edge is deleted
<code><u>exclude_layer</u>(name)</code>	Return a view of NestedEdges containing all layers except the excluded name Errors if any of the layers do not exist.
<code><u>exclude_layers</u>(names)</code>	Return a view of NestedEdges containing all layers except the excluded names Errors if any of the layers do not exist.
<code><u>exclude_valid_layer</u>(name)</code>	Return a view of NestedEdges containing all layers except the excluded name :param name: layer name that is excluded for the new view :type name: str
<code><u>exclude_valid_layers</u>(names)</code>	Return a view of NestedEdges containing all layers except the excluded names :param names: list of layer names that are excluded for the new view :type names: list[str]
<code><u>expanding</u>(step)</code>	Creates a WindowSet with the given step size using an expanding window.
<code><u>explode</u>()</code>	Explodes returns an edge object for each update within the original edge.
<code><u>explode_layers</u>()</code>	Explode layers returns an edge object for each layer within the original edge.
<code><u>has_layer</u>(name)</code>	Check if NestedEdges has the layer "name"
<code><u>history</u>()</code>	Returns all timestamps of edges, when an edge is added or change to an edge is made.

---

---

<code><u>history_date_time()</u></code>	Returns all timestamps of edges, when an edge is added or change to an edge is made.
<code><u>is_active()</u></code>	
<code><u>is_deleted()</u></code>	Check if edges are deleted
<code><u>is_self_loop()</u></code>	Check if the edges are on the same node
<code><u>is_valid()</u></code>	Check if edges are valid (i.e., not deleted)
<code><u>latest()</u></code>	Create a view of the NestedEdges including all events at the latest time.
<code><u>layer(name)</u></code>	Return a view of NestedEdges containing the layer "name" Errors if the layer does not exist
<code><u>layers(names)</u></code>	Return a view of NestedEdges containing all layers names Errors if any of the layers do not exist.
<code><u>rolling(window[, step])</u></code>	Creates a WindowSet with the given window size and optional step using a rolling window.
<code><u>shrink_end(end)</u></code>	Set the end of the window to the smaller of end and self.end()
<code><u>shrink_start(start)</u></code>	Set the start of the window to the larger of start and self.start()
<code><u>shrink_window(start, end)</u></code>	Shrink both the start and end of the window (same as calling shrink_start followed by shrink_end but more efficient)

---

---

**`snapshot_at`(time)** Create a view of the NestedEdges including all events that have not been explicitly deleted at time.

---

**`snapshot_latest()`** Create a view of the NestedEdges including all events that have not been explicitly deleted at the latest time.

---

**`valid_layers`(names)** Return a view of NestedEdges containing all layers names Any layers that do not exist are ignored

---

**`window`(start, end)** Create a view of the NestedEdges including all events between start (inclusive) and end (exclusive)

---

#### Attributes:

---

**`date_time`** Get the date times of exploded edges

---

**`dst`** Returns the destination node of the edge.

---

**`earliest_date_time`** Returns the earliest date time of the edges.

---

**`earliest_time`** Returns the earliest time of the edges.

---

**`end`** Gets the latest time that this NestedEdges is valid.

---

**`end_date_time`** Gets the latest datetime that this NestedEdges is valid

---

**`id`** Returns all ids of the edges.

---

**`latest_date_time`** Returns the latest date time of the edges.

---

**`latest_time`** Returns the latest time of the edges.

---

<code>layer_name</code>	Returns the name of the layer the edges belong to - assuming they only belong to one layer
<code>layer_names</code>	Returns the names of the layers the edges belong to
<code>nbr</code>	Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)
<code>properties</code>	Returns all properties of the edges
<code>src</code>	Returns the source node of the edge.
<code>start</code>	Gets the start time for rolling and expanding windows for this NestedEdges
<code>start_date_time</code>	Gets the earliest datetime that this NestedEdges is valid
<code>time</code>	Returns the times of exploded edges
<code>window_size</code>	Get the window size (difference between start and end) for this NestedEdges
<code>after(start)</code>	Create a view of the NestedEdges including all events after start (exclusive).
<b>Parameters:</b>	
<code>start</code> ( <i>TimeInput</i> ) – The start time of the window.	
<b>Return type:</b>	
<code>NestedEdges</code>	
<code>at(time)</code>	Create a view of the NestedEdges including all events at time.
<b>Parameters:</b>	
<code>time</code> ( <i>TimeInput</i> ) – The time of the window.	
<b>Return type:</b>	
<code>NestedEdges</code>	

**`before(end)`**

Create a view of the NestedEdges including all events before end (exclusive).

**Parameters:**

`end` (*TimeInput*) – The end time of the window.

**Return type:**

[NestedEdges](#)

**`collect()`**

Collect all edges into a list

**>Returns:**

the list of edges

**Return type:**

[list\[list\[Edges\]\]](#)

**`date_time`**

Get the date times of exploded edges

**`default_layer()`**

Return a view of NestedEdges containing only the default edge layer :returns: The layered view :rtype: NestedEdges

**`deletions()`**

Returns all timestamps of edges, where an edge is deleted

**>Returns:**

A list of lists of lists of unix timestamps

**`deletions_date_time()`**

Returns all timestamps of edges, where an edge is deleted

**>Returns:**

A list of lists of lists of DateTime objects

**`dst`**

Returns the destination node of the edge.

**`earliest_date_time`**

Returns the earliest date time of the edges.

**`earliest_time`**

Returns the earliest time of the edges.

**`end`**

Gets the latest time that this NestedEdges is valid.

**Returns:**

The latest time that this NestedEdges is valid or None if the NestedEdges is valid for all times.

**Return type:**

[Optional\[int\]](#)

**end\_date\_time**

Gets the latest datetime that this NestedEdges is valid

**Returns:**

The latest datetime that this NestedEdges is valid or None if the NestedEdges is valid for all times.

**Return type:**

[Optional\[datetime\]](#)

**exclude\_layer(name)**

Return a view of NestedEdges containing all layers except the excluded name Errors if any of the layers do not exist.

**Parameters:**

name ([str](#)) – layer name that is excluded for the new view

**Returns:**

The layered view

**Return type:**

[NestedEdges](#)

**exclude\_layers(names)**

Return a view of NestedEdges containing all layers except the excluded names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names that are excluded for the new view

**Returns:**

The layered view

**Return type:**

[NestedEdges](#)

**exclude\_valid\_layer(name)**

Return a view of NestedEdges containing all layers except the excluded name

:param name: layer name that is excluded for the new view :type name: str

**Returns:**

**The layered view**

**Return type:**

[NestedEdges](#)

**exclude\_valid\_layers(names)**

Return a view of NestedEdges containing all layers except the excluded names

:param names: list of layer names that are excluded for the new view :type names: list[str]

**Returns:**

The layered view

**Return type:**

[NestedEdges](#)

**expanding(step)**

Creates a WindowSet with the given step size using an expanding window.

An expanding window is a window that grows by step size at each iteration.

**Parameters:**

step ([int](#) | [str](#)) – The step size of the window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**explode()**

Explodes returns an edge object for each update within the original edge.

**explode\_layers()**

Explode layers returns an edge object for each layer within the original edge. These new edge object contains only updates from respective layers.

**has\_layer(name)**

Check if NestedEdges has the layer “name”

**Parameters:**

name ([str](#)) – the name of the layer to check

**Return type:**

[bool](#)

**history()**

Returns all timestamps of edges, when an edge is added or change to an edge is made.

**history\_date\_time()**

Returns all timestamps of edges, when an edge is added or change to an edge is made.

**id**

Returns all ids of the edges.

**is\_active()**

**is\_deleted()**

Check if edges are deleted

**is\_self\_loop()**

Check if the edges are on the same node

**is\_valid()**

Check if edges are valid (i.e., not deleted)

**latest()**

Create a view of the NestedEdges including all events at the latest time.

**Return type:**

[NestedEdges](#)

**latest\_date\_time**

Returns the latest date time of the edges.

**latest\_time**

Returns the latest time of the edges.

**layer(name)**

Return a view of NestedEdges containing the layer “name” Errors if the layer does not exist

**Parameters:**

**name (*str*) – then name of the layer.**

**Returns:**

The layered view

**Return type:**

[NestedEdges](#)

**layer\_name**

Returns the name of the layer the edges belong to - assuming they only belong to one layer

**layer\_names**

Returns the names of the layers the edges belong to

**layers(names)**

**Return** a view of NestedEdges containing all layers names Errors if any of the layers do not exist.

**Parameters:**

names ([list\[str\]](#)) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

[NestedEdges](#)

**nbr**

Returns the node at the other end of the edge (same as dst() for out-edges and src() for in-edges)

**properties**

Returns all properties of the edges

**rolling(window, step=None)**

Creates a WindowSet with the given window size and optional step using a rolling window.

A rolling window is a window that moves forward by step size at each iteration.

**Parameters:**

- window ([int](#) | [str](#)) – The size of the window.
- step ([int](#) | [str](#) | [None](#)) – The step size of the window. step defaults to window.

**Returns:**

A WindowSet object.

**Return type:**

[WindowSet](#)

**shrink\_end(end)**

Set the end of the window to the smaller of end and self.end()

**Parameters:**

end ([TimeInput](#)) – the new end time of the window

**Return type:**

[NestedEdges](#)

**shrink\_start(start)**

Set the start of the window to the larger of start and self.start()

**Parameters:**

`start` ([TimeInput](#)) – the new start time of the window

**Return type:**

[NestedEdges](#)

`shrink_window(start, end)`

Shrink both the start and end of the window (same as calling `shrink_start` followed by `shrink_end` but more efficient)

**Parameters:**

- `start` ([TimeInput](#)) – the new start time for the window
- `end` ([TimeInput](#)) – the new end time for the window

**Return type:**

[NestedEdges](#)

`snapshot_at(time)`

Create a view of the `NestedEdges` including all events that have not been explicitly deleted at time.

This is equivalent to `before(time + 1)` for `Graph` and `at(time)` for `PersistentGraph`

**Parameters:**

`time` ([TimeInput](#)) – The time of the window.

**Return type:**

[NestedEdges](#)

`snapshot_latest()`

Create a view of the `NestedEdges` including all events that have not been explicitly deleted at the latest time.

This is equivalent to a no-op for `Graph` and `latest()` for `PersistentGraph`

**Return type:**

[NestedEdges](#)

`src`

Returns the source node of the edge.

`start`

Gets the start time for rolling and expanding windows for this `NestedEdges`

**Returns:**

The earliest time that this `NestedEdges` is valid or `None` if the `NestedEdges` is valid for all times.

**Return type:**

Optional[int]

**start\_date\_time**

Gets the earliest datetime that this NestedEdges is valid

**Returns:**

The earliest datetime that this NestedEdges is valid or None if the NestedEdges is valid for all times.

**Return type:**

Optional[datetime]

**time**

Returns the times of exploded edges

**valid\_layers(names)**

Return a view of NestedEdges containing all layers names Any layers that do not exist are ignored

**Parameters:**

names (list[str]) – list of layer names for the new view

**Returns:**

The layered view

**Return type:**

NestedEdges

**window(start, end)**

Create a view of the NestedEdges including all events between start (inclusive) and end (exclusive)

**Parameters:**

- start (TimeInput | None) – The start time of the window (unbounded if None).
- end (TimeInput | None) – The end time of the window (unbounded if None).

**Return type:**

NestedEdges

**window\_size**

Get the window size (difference between start and end) for this NestedEdges

**Return type:**

Optional[int]

# MutableEdge

```
class MutableEdge
```

Bases: [Edge](#)

Methods:

---

```
add_constant_properties(properties[, layer])
```

Add constant properties to an edge in the graph.

---

```
add_updates(t[, properties, layer, ...])
```

Add updates to an edge in the graph at a specified time.

---

```
delete(t[, layer])
```

Mark the edge as deleted at the specified time.

---

```
update_constant_properties(properties[, layer])
```

Update constant properties of an edge in the graph overwriting existing values.

---

```
add_constant_properties(properties, layer=None)
```

Add constant properties to an edge in the graph. This function is used to add properties to an edge that remain constant and do not change over time. These properties are fundamental attributes of the edge.

Parameters:

- properties ([PropInput](#)) – A dictionary of properties to be added to the edge.
- layer ([str, optional](#)) – The layer you want these properties to be added on to.

```
add_updates(t, properties=None, layer=None,  
secondary_index=None)
```

Add updates to an edge in the graph at a specified time. This function allows for the addition of property updates to an edge within the graph. The updates are time-stamped, meaning they are applied at the specified time.

Parameters:

- t ([TimeInput](#)) – The timestamp at which the updates should be applied.
- properties ([PropInput, optional](#)) – A dictionary of properties to update.
- layer ([str, optional](#)) – The layer you want these properties to be added on to.
- secondary\_index ([int, optional](#)) – The optional integer which will be used as a secondary index

**Returns:**

This function does not return a value, if the operation is successful.

**Return type:**

[None](#)

**Raises:**

`GraphError` – If the operation fails.

`delete(t, layer=None)`

Mark the edge as deleted at the specified time.

**Parameters:**

- `t` ([TimeInput](#)) – The timestamp at which the deletion should be applied.
- `layer` ([str, optional](#)) – The layer you want the deletion applied to .

`update_constant_properties(properties, layer=None)`

Update constant properties of an edge in the graph overwriting existing values. This function is used to add properties to an edge that remains constant and does not change over time. These properties are fundamental attributes of the edge.

**Parameters:**

- `properties` ([PropInput](#)) – A dictionary of properties to be added to the edge.
- `layer` ([str, optional](#)) – The layer you want these properties to be added on to.

# Properties

`class Properties`

Bases: [object](#)

A view of the properties of an entity

**Methods:**

---

`as_dict()` Convert properties view to a dict

---

`get(key)` Get property value.

---

`items()` Get a list of key-value pairs

---

`keys()` Get the names for all properties (includes temporal and static properties)

---

`values\(\)` Get the values of the properties

Attributes:

`constant` Get a view of the constant properties (meta-data) only.

`temporal` Get a view of the temporal properties only.

`as\_dict\(\)`

Convert properties view to a dict

`constant`

Get a view of the constant properties (meta-data) only.

`get\(key\)`

Get property value.

First searches temporal properties and returns latest value if it exists. If not, it falls back to static properties.

`items\(\)`

Get a list of key-value pairs

`keys\(\)`

Get the names for all properties (includes temporal and static properties)

`temporal`

Get a view of the temporal properties only.

`values\(\)`

Get the values of the properties

If a property exists as both temporal and static, temporal properties take priority with fallback to the static property if the temporal value does not exist.

## ConstantProperties

`class ConstantProperties`

Bases: `object`

A view of constant properties of an entity

Methods:

---

`as\_dict\(\)` convert the properties view to a python dict

---

<code>get(key)</code>	get property value by key
<code>items()</code>	lists the property keys together with the corresponding value
<code>keys()</code>	lists the available property keys
<code>values()</code>	lists the property values
<code>as_dict()</code>	convert the properties view to a python dict

**Return type:**

`dict[str, PropValue]`

`get(key)`

get property value by key

**Parameters:**

key (`str`) – the name of the property

**Returns:**

the property value or None if value for key does not exist

**Return type:**

`PropValue | None`

`items()`

lists the property keys together with the corresponding value

**Returns:**

the property keys with corresponding values

**Return type:**

`list[Tuple[str, PropValue]]`

`keys()`

lists the available property keys

**Returns:**

the property keys

**Return type:**

`list[str]`

`values()`

lists the property values

**Returns:**

the property values

**Return type:**

[list](#) | Array

# TemporalProperties

**class TemporalProperties**

Bases: [object](#)

A view of the temporal properties of an entity

**Methods:**

---

[\*\*get\*\*\(key\)](#) Get property value for key if it exists

---

[\*\*histories\*\*\(\)](#) Get the histories of all properties

---

[\*\*histories\\_date\\_time\*\*\(\)](#) Get the histories of all properties

---

[\*\*items\*\*\(\)](#) List the property keys together with the corresponding values

---

[\*\*keys\*\*\(\)](#) List the available property keys

---

[\*\*latest\*\*\(\)](#) Get the latest value of all properties

---

[\*\*values\*\*\(\)](#) List the values of the properties

---

**get(key)**

Get property value for key if it exists

**Returns:**

the property view if it exists, otherwise None

**histories()**

Get the histories of all properties

**Returns:**

the mapping of property keys to histories

**Return type:**

`dict[str, list[Tuple[int, PropValue]]]`

`histories_date_time()`

Get the histories of all properties

**Returns:**

the mapping of property keys to histories

**Return type:**

`dict[str, list[Tuple[datetime, PropValue]]]`

`items()`

List the property keys together with the corresponding values

`keys()`

List the available property keys

`latest()`

Get the latest value of all properties

**Returns:**

the mapping of property keys to latest values

**Return type:**

`dict[str, Any]`

`values()`

List the values of the properties

**Returns:**

the list of property views

**Return type:**

`list[TemporalProp]`

# PropertiesView

`class PropertiesView`

Bases: `object`

Methods:

---

<code>as_dict()</code>	Convert properties view to a dict
<code>get(key)</code>	Get property value.
<code>items()</code>	Get a list of key-value pairs
<code>keys()</code>	Get the names for all properties (includes temporal and constant properties)

---

<code>values()</code>	Get the values of the properties
-----------------------	----------------------------------

#### Attributes:

---

<code>constant</code>	Get a view of the constant properties (meta-data) only.
<code>temporal</code>	Get a view of the temporal properties only.
<code>as_dict()</code>	Convert properties view to a dict
<code>constant</code>	Get a view of the constant properties (meta-data) only.
<code>get(key)</code>	Get property value.
<code>items()</code>	First searches temporal properties and returns latest value if it exists. If not, it falls back to constant properties.
<code>keys()</code>	Get the names for all properties (includes temporal and constant properties)
<code>temporal</code>	Get a view of the temporal properties only.
<code>values()</code>	Get the values of the properties

---

If a property exists as both temporal and constant, temporal properties take priority with fallback to the constant property if the temporal value does not exist.

# TemporalProp

`class TemporalProp`

Bases: `object`

A view of a temporal property

Methods:

<code>at(t)</code>	Get the value of the property at time t
<code>average()</code>	Compute the average of all property values.
<code>count()</code>	Count the number of properties.
<code>history()</code>	Get the timestamps at which the property was updated
<code>history_date_time()</code>	Get the timestamps at which the property was updated
<code>items()</code>	List update timestamps and corresponding property values
<code>items_date_time()</code>	List update timestamps and corresponding property values
<code>max()</code>	Find the maximum property value and its associated time.
<code>mean()</code>	Compute the mean of all property values.
<code>median()</code>	Compute the median of all property values.
<code>min()</code>	Find the minimum property value and its associated time.
<code>ordered_dedupe(latest_time)</code>	
<code>sum()</code>	Compute the sum of all property values.
<code>unique()</code>	

---

`value()` Get the latest value of the property

---

`values()` Get the property values for each update

---

`at(t)` Get the value of the property at time t

`average()` Compute the average of all property values. Alias for mean().

**Returns:**

The average of each property values, or None if count is zero.

**Return type:**

`Prop`

`count()` Count the number of properties.

**Returns:**

The number of properties.

**Return type:**

`int`

`history()` Get the timestamps at which the property was updated

`history_date_time()` Get the timestamps at which the property was updated

`items()` List update timestamps and corresponding property values

`items_date_time()` List update timestamps and corresponding property values

`max()` Find the maximum property value and its associated time.

**Returns:**

A tuple containing the time and the maximum property value.

**Return type:**

`(i64, Prop)`

`mean()` Compute the mean of all property values. Alias for mean().

**Returns:**

The mean of each property values, or None if count is zero.

**Return type:**

[Prop](#)

`median()`

Compute the median of all property values.

**Returns:**

A tuple containing the time and the median property value, or None if empty

**Return type:**

(i64, [Prop](#))

`min()`

Find the minimum property value and its associated time.

**Returns:**

A tuple containing the time and the minimum property value.

**Return type:**

(i64, [Prop](#))

`ordered_dedupe(latest_time)`

`sum()`

Compute the sum of all property values.

**Returns:**

The sum of all property values.

**Return type:**

[Prop](#)

`unique()`

`value()`

Get the latest value of the property

`values()`

Get the property values for each update

# Prop

`class Prop(name)`

Bases: [object](#)

A reference to a property used for constructing filters

Use ==, !=, <, <=, >, >= to filter based on property value (these filters always exclude entities that do not have the property) or use one of the methods to construct other kinds of filters.

**Parameters:**name ([str](#)) – the name of the property**Methods:**

<a href="#">any</a> (values)	Create a filter that keeps entities if their property value is in the set
------------------------------	---

---

<a href="#">is_none</a> ()	Create a filter that only keeps entities that do not have the property
----------------------------	--

---

<a href="#">is_some</a> ()	Create a filter that only keeps entities if they have the property
----------------------------	--

---

<a href="#">not_any</a> (values)	Create a filter that keeps entities if their property value is not in the set or if they don't have the property
----------------------------------	--

**any (values)**

Create a filter that keeps entities if their property value is in the set

**Parameters:**values ([set\[PropValue\]](#)) – the set of values to match**Returns:**

the property filter

**Return type:**[PropertyFilter](#)**is\_none ()**

Create a filter that only keeps entities that do not have the property

**Returns:**

the property filter

**Return type:**[PropertyFilter](#)**is\_some ()**

Create a filter that only keeps entities if they have the property

**Returns:**

the property filter

**Return type:**[PropertyFilter](#)**not\_any (values)**

Create a filter that keeps entities if their property value is not in the set or if they don't have the property

**Parameters:**

values (`set[PropValue]`) – the set of values to exclude

**Returns:**

the property filter

**Return type:**

`PropertyFilter`

# PropertyFilter

`class PropertyFilter`

Bases: `object`

# WindowSet

`class WindowSet`

Bases: `object`

**Methods:**

`time_index([center])` Returns the time index of this window set

---

`time_index(center=False)`

Returns the time index of this window set

It uses the last time of each window as the reference or the center of each if center is set to True

**Parameters:**

center (`bool`) – if True time indexes are centered. Defaults to False

**Returns:**

the time index”

**Return type:**

`Iterable`

# typing

## Type Aliases

```
type PropValue = bool | int | float | datetime | str | Graph
| PersistentGraph | Document | list\[PropValue\] | dict\[str, PropValue\]
type GID = int | str
type PropInput = Mapping\[str, PropValue\]
type NodeInput = int | str | Node
type TimeInput = int | str | float | datetime
type Direction = Literal\['in', 'out', 'both'\]
```