# PyGraphistry Documentation

***Release 0.34.10***

**Graphistry, Inc.**

**Oct 08, 2024**

# PYGRAPHISTRY DOCUMENTATION

PyGraphistry is a Python visual graph AI library to extract, transform, query, analyze, model, and visualize big graphs. It includes several notable pieces:

**Graphistry Visualization Client**
: *Graphistry Visualization Client*: Convenience layer for using the optional Graphistry GPU server for accelerated compute and visualization

**Dataframe-native graph manipulation**
: Optimized dataframe-native tabular and graph methods for loading, transforming, analyzing, and visualizing data as graphs

**GFQL (new!)**
: *GFQL (new!)* queries: Home for the GFQL graph dataframe-native query language, with optional GPU support

**Graphistry[AI]**
: Optional methods and integrations for graph autoML, including automatic feature engineering, UMAP, and graph neural networks

**Plugins**
: *Plugins*: Optimized and streamlined integrations for enriching your workflows, including querying databases like Neo4j and Splunk

**Louie.AI (new!)**
: Louie.AI: Use generative AI to talk to your data, including for GFQL queries and Graphistry visualizations

Combined, PyGraphistry reduces your time to graph for going from raw data to visualizations to AI insights in a few lines of code.

> **Note**
>
> Looking where to start? See the quickstart below or the *10 Minutes to PyGraphistry guide*.

CHAPTER

ONE

# QUICKSTART

1. Install:

```
# Install from PyPI
pip install graphistry

# Optionally, get a free GPU account or self-hosted server at https://graphistry.com/get-started
```

2. Start graphing!

```
import graphistry
import pandas as pd

# Load data from any Python data science library or database
df = pd.DataFrame({
  'src': ['A', 'B', 'C'],
  'dst': ['B', 'C', 'A'],
  'nfo': ['X', 'Y', 'Z']
  'abc': [1, 2, 3]
})
g1 = graphistry.edges(df, source="src", destination="dst")

# Server-accelerated GPU visualization
graphistry.register(api=3, server="hub.graphistry.com", username="A", password="B")
g1.plot()

# Use GPUs when available in almost all APIs
import cudf
g2 = g1.edges(cudf.from_pandas(g1._edges))
g2.plot()

# Many local graph wrangling helpers and easy dataframe-native graph manipulation
g3 = g2.materialize_nodes().get_degrees()
print(g3._nodes[['id', 'degree']])

# ML & AI methods
umap_g = graphistry.nodes(df).umap()
umap_g.plot()

# GFQL graph dataframe-native query language with optional GPU support
nearest_neighbors_df = umap_g.chain([ n({'id': 'A'}), e(hops=2), n()])._nodes
```

CHAPTER

TWO

# ARTICLES

We recommend reading the Graphistry blog and github demos. Some useful articles include:

- Graphistry: Visual Graph AI Interactive demo
- PyGraphistry + Databricks
- PyGraphistry + UMAP
- What is Graph Intelligence?

CHAPTER

THREE

# INDICES AND TABLES

## 3.1 10 Minutes to PyGraphistry

Welcome to **PyGraphistry**, the fast and easy platform for graph visualization, querying, analytics, and AI. By the end of this guide, you'll be able to create interactive, GPU-accelerated graph visualizations of your data. If you are already familiar with concepts like dataframes, PyGraphistry will be an easy fit.

PyGraphistry can be used standalone and automatically optimizes for both CPU systems and GPU systems. It is typically used in Python notebooks, dashboards, and web apps. The library includes the GFQL dataframe-native graph query language, official Python bindings for Graphistry GPU visualization & analytics servers, and a variety of graph data science tools.

### 3.1.1 Why Graph Intelligence?

Graphs represent relationships between entities. Whether you're analyzing event logs, social media interactions, security alerts, financial transactions, clickstreams, supply chains, or genomics data, visualizing and analyzing these relationships can reveal patterns and insights that are difficult to detect otherwise.

**Graph visualization and analytics helps you:**

- **Identify Patterns**: Spot clusters, behaviors, progressions, root causes, hubs, and anomalies.
- **Understand Structures**: See how entities are connected and how information flows.
- **Communicate Insights**: Present complex relationships in an understandable way.

As datasets grow larger, traditional tools struggle with performance and complexity, making it challenging for analysts to extract meaningful insights efficiently.

### 3.1.2 What Makes PyGraphistry Special?

**PyGraphistry** is a comprehensive Python library that simplifies working with larger graphs. It is known for:

- **GPU Acceleration**: Work with larger datasets visually or programatically
- **Advanced Visualization**: Rich out-of-the-box visual encodings (e.g., color, size, icon, badges), interactive analysis features (e.g., zooming, cross-filtering, drilldowns, timebars), multiple layout algorithms.
- **Seamless Integration**: Works seamlessly with popular Python data science libraries like Pandas, cuDF, and NetworkX, and integrates easily into notebooks, dashboard tools, web apps, databases, and other tools
- **GFQL dataframe-native graph query language**: Run graph queries and analytics directly on dataframes, with optional GPU acceleration, which gives scalable results without the usual infrastructure overhead.

- **Graphistry[AI]**: With native support for GPU feature engineering, UMAP clustering, and embeddings, quickly perform accelerated graph ETL, analytics, ML/AI, and visualization on large datasets.
- **Multiple Interfaces**: In addition to the PyGraphistry Python bindings, Graphistry provides REST APIs, Node.js and React libraries, and **Louie.AI** for conversational analytics, making it accessible from various platforms and languages.

### 3.1.3 Installation

#### 3.1.3.1 Install PyGraphistry

```
pip install graphistry
```

This performs a minimal installation with dependencies limited to mostly just Pandas and PyArrow.

#### 3.1.3.2 Install cuDF GPU DataFrames (Optional)

For GPU acceleration with DataFrames, install **cuDF** via the NVIDIA RAPIDS Installation Guide.

#### 3.1.3.3 Register with PyGraphistry (Optional)

While most of PyGraphistry can run locally, use with a GPU visualization server requires an account on your own self-hosted Graphistry server or on Graphistry Hub. If you do not have an account yet, create a free GPU account at graphistry.com, or launch your own server.

Then, log in your PyGraphistry client:

```
import graphistry

graphistry.register(api=3, server='hub.graphistry.com', username='YOUR_USERNAME',␣
↪password='YOUR_PASSWORD')
```

Replace with your actual server and credentials.

### 3.1.4 Loading Data Efficiently

The Python data science ecosystem supports connecting to most databases and file type types

Many users start with CSV, JSON, and SQL database. We often see teams adopt formats like **Parquet** and **Apache Arrow**. Graphistry natively leverages these, so loading data with them can often be 10X+ faster than typical libraries.

**Example: Loading Parquet Data**

```
import cudf
import graphistry

# Load the dataset using cuDF
df = cudf.read_parquet('data/honeypot.parquet')

print(df.head())
```

Alternatively, if you don't have a GPU or cuDF, you can use Pandas:

```
import pandas as pd
import graphistry

# Load the dataset using Pandas
df = pd.read_csv('https://raw.githubusercontent.com/graphistry/pygraphistry/master/demos/
↪data/honeypot.csv')

print(df.head())
```

**Sample Data Structure:**

```
attackerIP       victimIP  victimPort         vulnName  count   time(max)   time(min)
0   1.235.32.141  172.31.14.66       139.0  MS08067 (NetAPI)      6  1421433577 ␣
↪1421422669
1  105.157.235.22  172.31.14.66       445.0  MS08067 (NetAPI)      4  1422497735 ␣
↪1422494755
...
```

### 3.1.5 Creating a Basic Visualization

Let's create a simple graph visualization using the honeypot data:

```
g = graphistry.edges(df, 'attackerIP', 'victimIP')
g.plot()  # Make sure you called graphistry.register() above
```

This will render an interactive graph where nodes represent IP addresses, and edges represent attacks.

### 3.1.6 Automatic GPU Acceleration

Note that the `plot()` step uploads the data to the Graphistry server for your server-GPU-accelerated visualization session. This results in smoother interactions and faster rendering, even with large datasets.

Other times, PyGraphistry computes over data locally, such as with GFQL queries. GPU acceleration will be automatically used if your environment supports GPU compute.

### 3.1.7 Adding Visual Encodings

PyGraphistry supports various visual encodings to represent different attributes in your data.

#### 3.1.7.1 Example: Adding Color Encodings

Let's add color encodings based on the vulnerability exploited.

```
# Plot with color encoding
g2 = g1.encode_edge_color(
    'vulnName',
    categorical_mapping={
        'MS08067 (NetAPI)': 'red',
        'OtherVuln': 'blue',
    },
```

(continues on next page)

(continued from previous page)

```
    default_mapping='gray')

g2.plot()
```

Now, edges are colored based on the type of vulnerability, helping you distinguish different attack types.

### 3.1.8 Adjusting Sizes, Labels, Icons, Badges, and More

You can adjust further node and edge settings using data. Sample calls include:

- `bind(point_title=)`: Assign labels to nodes based on a column
- `encode_point_size()`: Adjust node sizes based on a column
- `encode_point_icon()`: Assign different icons to nodes based on a column
- `encode_point_badge()`: Add badges to nodes based on a column
- `encode_point_weight()`: Adjust node weights based on a column
- Equivalent functions for edges: `encode_edge_size()`, `encode_edge_icon()`, `encode_edge_badge()`

Additional settings, such as background colors and logo watermarks, can also be configured.

### 3.1.9 Adding an Interactive Timebar

If your data includes temporal information, you can add a timebar to visualize changes over time.

```
# Ensure column has a datetime dtype
edges['time'] = cudf.to_datetime(df['time(max)'], unit='s')
g = graphistry.edges(edges)

# Plot with time encoding: Graphistry automatically detects Arrow/Parquet native types
g.plot()
```

The timebar appears as soon as the UI detects datetime values, and enables you to interactively explore the graph as it evolves over time.

### 3.1.10 Applying Force-Directed Layout

By default, PyGraphistry uses a force-directed layout. You can adjust its parameters:

```
# Adjust layout settings
g2 = g1.settings(url_params={'play': 7000, 'strongGravity': True, 'edgeInfluence': 2})
g2.plot()
```

### 3.1.11 More Layout Algorithms

PyGraphistry offers additional layout algorithms of its own, and streamlines using layouts from other libraries, so you can display your graph quickly and meaningfully.

For example, GraphViz layouts is known for its high quality for laying out small trees and directed acyclic graphs (DAGs):

```
# pygraphistry handles format conversions behind-the-scenes
g2 = g1.layout_graphviz('dot')
g2.plot()
```

### 3.1.12 Using UMAP for Dimensionality Reduction

For large datasets, you can use UMAP for dimensionality reduction to layout the graph meaningfully. UMAP will identify nodes that are similar across their different attributes.

Special to PyGraphistry, PyGraphistry records and renders the similarity edges between similar entities. We find this to be critical in practice for investigating results and using UMAP in analytical pipelines.

```
# Compute UMAP layout by clustering on some subset of columns
g1 = graphistry.umap(X=['attackerIP', 'victimIP', 'vulnName'])
print('# similarity edges', len(g1._edges))
g1.plot()
```

### 3.1.13 Query graphs with GFQL

GFQL, our dataframe-native graph query language, allows you to run optimized graph queries directly on dataframes without the need for a separate graph database system.

Suppose you want to focus on attacks that started with the "MS08067 (NetAPI)" vulnerability at some specific timestamp, and see everything 2 hops after:

```
g2 = g1.chain([
    n(),
    e(edge_query="vulnName == 'MS08067 (NetAPI)' & `time(max)` > 1421430000"),
    n(),
    e(hops=2)
])

g2.plot()
```

This GFQL query filters the edges based on the vulnerability name and time, then returns the matching nodes and edges for visualization.

### 3.1.14 Utilizing Hypergraphs

PyGraphistry supports hypergraphs, which allow you to quickly visualize complex relationships involving more than two entities.

**Example: Visualizing Attacks as Hyperedges**

```
hg = graphistry.hypergraph(df, ['attackerIP', 'victimIP', 'vulnName', 'victimPort'])

hg['graph'].plot()
```

This will represent each attack as a hyperedge connecting the attacker IP, victim IP, vulnerability name, and port nodes.

### 3.1.15 Embedding Visualizations into Web Apps

You can embed PyGraphistry visualizations in web applications using additional SDKs like **GraphistryJS**.

The JavaScript client comes in two forms and provides further configuration hooks:

- **Vanilla JavaScript**: Use the GraphistryJS library to embed visualizations directly.
- **React**: Use the Graphistry React components for seamless integration.

### 3.1.16 Rendering Options

#### 3.1.16.1 Inline Rendering

In Jupyter notebooks, you can render the visualization inline.

```
g.plot()
```

#### 3.1.16.2 URL Rendering

Alternatively, you can generate a URL to view the visualization in a separate browser tab.

```
url = g.plot(render=False)
print(f"View your visualization at: {url}")
```

### 3.1.17 Next Steps

- *10 Minutes to Graphistry Visualization*: Learn how to create more advanced visualizations.
- *10 Minutes to GFQL*: Use GFQL to query and manipulate your graph data before visualization.
- *Layout guide*: Explore different layouts for your visualizations.
- *Plugins*: Discover more ways to connect to your data and work with your favorite tools.
- *PyGraphistry API Reference*

### 3.1.18 External Resources

- Graphistry UI Guide
- GraphistryJS: Node, React, and vanilla JS clients
- Graphistry REST API: Work from any language
- Graphistry URL settings: Control visualizations via URL parameters`

Happy graphing!

## 3.2 Install

Welcome to the PyGraphistry installation guide. Choose the section that best fits your needs:

### 3.2.1 Installation Guide - Quick Start

This quick start guide will help you install PyGraphistry and its essential dependencies to get you up and running quickly.

#### 3.2.1.1 Minimum System Requirements

Before installing PyGraphistry, ensure your system meets the following minimum requirements:

- **Operating System**: Windows, macOS, Linux, or any Python-capable environment
- **Python Version**: Python 3.8 or higher
- **Hardware**:
  - **CPU**: 1 core
  - **Memory**: 1 GB - in addition to regular OS requirements
  - **GPU**: While optional, we recommend using a browser with WebGL enabled and a GPU, which is most phones and laptops

#### 3.2.1.2 Installing PyGraphistry

##### Basic Installation

Install PyGraphistry using *pip*:

```
pip install graphistry
```

#### Importing and Version Check

Verify the installation by importing PyGraphistry and checking its version:

```
import graphistry
print(graphistry.__version__)
```

#### 3.2.1.3 Log in to a Graphistry GPU Server

To use PyGraphistry's visualization server, you need to connect to a Graphistry GPU server:

- **Get an Account**: Visit the Graphistry Get Started page and choose:
  - **Graphistry Hub**: For immediate access with no installation, use the public Graphistry Hub, which includes free GPU accounts.
  - **Self-Host**: Quick launch on AWS/Azure, or contact staff for on-premises options.
- **Log in**: Once you have an account, register in your Python environment:

  ```
  import graphistry

  graphistry.register(api=3, server='hub.graphistry.com', username='YOUR_USERNAME',␣
  ↪password='YOUR_PASSWORD')
  ```

  Replace *'YOUR_USERNAME'* and *'YOUR_PASSWORD'* with your actual credentials.

  When the command finishes without an exception, you have successfully connected to the server.

  See the authentication guide for additional options such as logging into an organization, SSO, and using API keys.

For additional authentication options, see the Login and Sharing guide.

Happy graphing!

### 3.2.2 Installation Guide - Extended

This extended guide provides detailed instructions for installing PyGraphistry, including optional configurations for enhanced performance and functionality.

#### 3.2.2.1 GPU Mode System Requirements (Optional)

- **Nvidia RAPIDS**: PyGraphistry primarily aligns with Nvidia RAPIDS, so check their requirements for your system:
  - **Volta generation GPUs or newer** are the current Nvidia RAPIDS minimum requirement.
  - **cuDF**: Required.
  - **cuML**, **cuGraph**: Recommended.
- **PyTorch**: PyGraphistry[AI] further aligns with PyTorch for some of its more advanced methods.

#### Core Dependencies (Installed by Default)

PyGraphistry depends on a small set of standard CPU-based Python data science libraries such as pandas, pyarrow, and numpy. If your system is missing these dependencies, they will get installed automatically.

#### 3.2.2.2 Optional Dependencies

PyGraphistry supports a variety of optional dependencies to extend its functionality.

#### GPU Acceleration with RAPIDS

To enable GPU acceleration for DataFrames and graph analytics, install **cuDF**, **cuML**, and **cuGraph** from the NVIDIA RAPIDS suite.

Follow the instructions at the NVIDIA RAPIDS Installation Guide.

#### Additional Optional Dependencies

Many of the following can be used in both CPU mode and GPU mode.

- **AI Libraries**:
  - *torch* (1GB+): PyTorch and related libraries for advanced AI methods in the PyGraphistry AI packages.

    Install with:

    ```
    pip install graphistry[ai]
    ```

- **Graph Libraries**:
  - *networkx*: Integration with NetworkX graphs.

    Install with:

    ```
    pip install graphistry[networkx]
    ```

  - *igraph*: Support for igraph graphs.

    Install with:

    ```
    pip install graphistry[igraph]
    ```

  - *pygraphviz*: Rendering graphs with Graphviz layouts.

    Install with:

    ```
    pip install graphistry[pygraphviz]
    ```

- **Graph Databases and Protocols**:
  - *gremlinpython*: Working with Gremlin graph databases.

    Install with:

    ```
    pip install graphistry[gremlin]
    ```

  - *neo4j*, *neotime*: Connecting to Neo4j via the Bolt protocol.

    Install with:

    ```
    pip install graphistry[bolt]
    ```

- **Data Formats**:
  - *openpyxl*, *xlrd*: Reading NodeXL files.

    Install with:

    ```
    pip install graphistry[nodexl]
    ```

- **Machine Learning and AI**:
  - *umap-learn*, *dirty-cat*, *scikit-learn*: For dimensionality reduction and clustering.

    Install with:

    ```
    pip install graphistry[umap-learn]
    ```

  - *scipy*, *dgl*, *torch<2*, *sentence-transformers*, *faiss-cpu*, *joblib*: Advanced AI functionalities.

    Install with:

    ```
    pip install graphistry[ai]
    ```

- **Jupyter Support**:
  - *ipython*: Enhanced Jupyter notebook integration.

    Install with:

    ```
    pip install graphistry[jupyter]
    ```

#### Installing Multiple Extras

You can install multiple extras by listing them separated by commas:

```
pip install graphistry[networkx,umap-learn]
```

#### Installing All Optional Dependencies

To install all optional dependencies (not generally recommended due to size and potential conflicts):

```
pip install graphistry[all]
```

#### 3.2.2.3 Common Questions

##### Do I Need a Server?

- **No**, you can run GFQL and other PyGraphistry CPU and GPU components locally. To use the full visualization capabilities, you do need access to a Graphistry server.
- **Options**:
  - **Graphistry Hub**: Use the public Graphistry Hub at hub.graphistry.com.
  - **Self-Hosted Server**: Set up your own Graphistry server by following the deployment instructions in the Graphistry CLI Admin Guide.

##### Can I Use PyGraphistry Without GPU Support?

- **Yes**, PyGraphistry can be used without GPU support.
- **GPU Acceleration**: To leverage GPU acceleration, install optional GPU libraries like cuDF and have compatible hardware.

##### What Are the Benefits of Installing Optional Dependencies?

- **Enhanced Functionality**: Support for different graph formats, advanced analytics, machine learning, and integration with various tools and databases. For example, for visualization users needing careful layout of small trees, we recommend *pygraphviz*, while for users of big GFQL workloads, we recommend RAPIDS.
- **Customization**: Install only what you need for your specific use case.

##### How Do I Install Development Dependencies?

For contributors and developers who wish to work on PyGraphistry itself, we recommend using Docker, or for native development:

- **Install with**:

```
pip install graphistry[dev]
```

- **Includes**: Testing tools, documentation tools, and other development dependencies like *flake8*, *pytest*, *sphinx*, etc.

#### 3.2.2.4 References

- **PyGraphistry GitHub Repository**: https://github.com/graphistry/pygraphistry
- **Graphistry Get Started**: https://www.graphistry.com/get-started
- **Graphistry CLI Admin Guide**: https://github.com/graphistry/graphistry-cli
- **NVIDIA RAPIDS Installation Guide**: https://rapids.ai/start.html
- **Graphistry Documentation**: https://hub.graphistry.com/docs/

Happy graphing!

### 3.2.3 Using a Server with PyGraphistry

While PyGraphistry offers robust functionalities out of the box, leveraging a server enhances its capabilities, especially for GPU-accelerated visualizations and remote operations. This guide helps you decide whether to use PyGraphistry without a server or to set up a server using various available options.

#### 3.2.3.1 Using PyGraphistry Without a Server

For most use cases, PyGraphistry can operate seamlessly without the need for a dedicated server. This setup is ideal for:

- **Local Data Visualization**: Create and interact with visualizations directly within your local environment.
- **Basic Graph Analytics**: Perform standard graph operations and analyses without the overhead of server management.
- **Development and Testing**: Ideal for developers building and testing applications that utilize PyGraphistry.

**Note**: Without a server, advanced features like GPU-accelerated visualizations and certain remote capabilities will not be available.

#### 3.2.3.2 Using a Graphistry Server

To unlock the full potential of PyGraphistry, especially for GPU-accelerated visualizations and scalable remote operations, consider setting up a Graphistry server. Below are the available options to get started:

##### Graphistry Hub

**Graphistry Hub** offers a managed solution with the following benefits:

- **Ease of Use**: No installation required; get started immediately.
- **Free Cloud GPU Tier**: Access free GPU resources for accelerated visualizations.
- **Scalability**: Automatically scales with your project needs.

**Getting Started with Graphistry Hub**:

- Visit the Graphistry Get Started page.
- Choose **Graphistry Hub** to create an account and start using the service without any infrastructure setup.

##### Cloud Marketplace Deployments

Deploying Graphistry on cloud platforms like **AWS** and **Azure** provides flexibility and control over your server environment.

### AWS Marketplace

- **Quick Deployment**: Launch Graphistry with pre-configured settings optimized for AWS.
- **Integration**: Seamlessly integrate with other AWS services for enhanced functionality.

**Deploy on AWS**:

- Navigate to the AWS Marketplace and search for "Graphistry."
- Follow the deployment instructions to set up your Graphistry server on AWS.

### Azure Marketplace

- **Azure Integration**: Leverage Azure's robust infrastructure and services.
- **Scalable Resources**: Adjust resources based on your project's demands.

**Deploy on Azure**:

- Visit the Azure Marketplace and search for "Graphistry."
- Follow the provided steps to deploy Graphistry on Azure.

### Kubernetes and Docker-Compose Distributions

For organizations preferring containerized deployments, Graphistry offers support for **Kubernetes** and **Docker-Compose**.

### Kubernetes

- **Orchestration**: Manage containerized applications with Kubernetes for scalability and reliability.
- **Customization**: Tailor the deployment to fit your infrastructure and scaling requirements.

**Deploy with Kubernetes**:

- Access the Kubernetes deployment guides at the Graphistry CLI Admin Guide.
- Follow the instructions to deploy and manage your Graphistry server on a Kubernetes cluster.

### Docker-Compose

- **Simplicity**: Ideal for smaller deployments or development environments.
- **Quick Setup**: Deploy Graphistry using Docker-Compose with minimal configuration.

**Deploy with Docker-Compose**:

- Refer to the Graphistry CLI Admin Guide for Docker-Compose setup instructions.
- Execute the provided Docker-Compose files to launch your Graphistry server locally or on a server.

#### 3.2.3.3 Choosing the Right Option

- **For Beginners or Quick Setup**: Use **Graphistry Hub** for a hassle-free experience.
- **For Enterprise or Scalable Needs**: Deploy via **AWS** or **Azure Marketplace** to leverage cloud infrastructure.
- **For Containerized Environments**: Opt for **Kubernetes** or **Docker-Compose** to integrate with your existing container orchestration workflows.

Happy graphing!

## 3.3 Login and Share

PyGraphistry streamlines working with optional Graphistry server capabilities such as GPU-accelerated visual analytics, sharing visualizations, simplifying graph pipelines, GFQL compute endpoints, and sharing GPU resources.

Server interactions are typically by first logging in (*graphistry.register()*) and then sending data, such as via *g.plot()*. You can set access control policies on all of your uploaded data via *graphistry.privacy()*. Read on for more on both.

### 3.3.1 API authentication to Graphistry servers

*graphistry.register()* is the global method to authenticate your Graphistry client. It sets up your API credentials, specifies the server to connect to, and configures authentication settings. This function should be called before making any Graphistry API calls that use the server such as *.plot()*.

Underneath, it manages use of JWT session tokens over the Graphistry REST API. Likewise, it streamlines using advanced optional modes such as SSO.

#### 3.3.1.1 Basic Usage

To register, import Graphistry and call *graphistry.register()*:

```
import graphistry

# Register with default Graphistry Hub using username/password
graphistry.register(api=3, username="my_username", password="my_password")
```

By default, this connects to **Graphistry Hub** (*hub.graphistry.com*) using the *https* protocol and sets *api=3* for the latest API version. You can override the server, authentication details, and other settings as needed.

#### 3.3.1.2 Core Concepts

##### Personal Accounts vs Organizational Accounts

- **Personal Accounts**: Meant for individual use, typically on Graphistry Hub.
- **Organizational Accounts**: Managed with roles and permissions, often in an enterprise context.

```
user_info = graphistry.user()
print(user_info.get("organization"))  # Returns organization info or None
```

##### Server Configuration

- **Default Server**: By default, *graphistry.register()* connects to the **Graphistry Hub**, including the **free GPU tier** for visual analytics.
- **Custom Server**: If using a private deployment, specify the *server* argument to connect to your custom server.

```
# Connect to a custom server
graphistry.register(
    api=3,
    server="my_custom_graphistry_server.com",
    username="my_username",
    password="my_password"
)
```

##### Protocol Configuration

- **TLS (HTTPS)**: Communication uses *https* by default for secure communication.
- **Non-TLS (HTTP)**: If your server doesn't support TLS, set the *protocol* parameter to *"http"*.

```
# Use HTTP protocol without TLS
graphistry.register(
    api=3,
    protocol="http",
    server="my_custom_graphistry_server.com",
    username="my_username",
    password="my_password"
)
```

##### Authentication Methods

*graphistry.register()* supports several authentication methods:

1. **Username & Password**:

   ```
   graphistry.register(api=3, username="my_username", password="my_password")
   ```

2. **Personal Key ID & Secret** (for scripts or automation):

```
graphistry.register(api=3, personal_key_id="my_key_id", personal_key_secret=
↪"my_key_secret")
```

3. **Single Sign-On (SSO)** (for enterprise users):

```
graphistry.register(api=3, idp_name="my_idp_name", sso_opt_into_type=
↪"browser")
```

SSO authentication options: *sso_opt_into_type* can be *"browser"*, *"display"*, or *None* (default is print).

#### Routing Configuration

- **Server Routing**: By default, server API and browser UI requests route through the same *server*.
- **Custom Browser Routing**: Override browser routing via *client_protocol_hostname*.

```
# Override browser routing
graphistry.register(
    api=3,
    server="my_api_server.com",
    username="my_username",
    password="my_password",
    client_protocol_hostname="https://my_ui_server.com"
)
```

### 3.3.1.3 Advanced Features

#### JWT Session Handling

*graphistry.register()* establishes a **JWT session** after authentication. The session token is managed automatically for future API calls.

#### Retrieving the Current JWT Token

To retrieve the current JWT token, you can use the following command after registering:

```
# Get the current JWT token
current_token = graphistry.api_token()
print(current_token)
```

The token is automatically refreshed as needed during the session.

#### 3.3.1.4 Detailed Parameter Reference

- **username** *(Optional[str])*: Your Graphistry account username.
- **password** *(Optional[str])*: Your Graphistry account password.
- **personal_key_id** *(Optional[str])*: Your personal key ID for secure access.
- **personal_key_secret** *(Optional[str])*: Corresponding personal key secret.
- **server** *(Optional[str])*: The URL of the Graphistry server to connect to (e.g., *hub.graphistry.com* or a custom server).
- **protocol** *(Optional[str])*: The protocol to use (*https* or *http*), defaults to *https*.
- **api** *(Optional[int])*: The API version to use (always set to *3*).
- **client_protocol_hostname** *(Optional[str])*: Overrides the browser protocol/hostname.
- **org_name** *(Optional[str])*: Organization name for SSO authentication.
- **idp_name** *(Optional[str])*: Identity Provider (IdP) for SSO.
- **sso_opt_into_type** *(Optional[str])*: How to display the SSO URL (*"browser"*, *"display"*, or *None*).

#### 3.3.1.5 Examples

##### Register with Username and Password

```
import graphistry

graphistry.register(
    api=3,
    username="my_username",
    password="my_password"
)
```

##### Register with Personal Key ID and Secret

```
import graphistry

graphistry.register(
    api=3,
    personal_key_id="my_key_id",
    personal_key_secret="my_key_secret"
)
```

#### Register with SSO (Organization with Specific IdP)

```
import graphistry

graphistry.register(
    api=3,
    org_name="my_org_name",
    idp_name="my_idp_name",
    sso_opt_into_type="browser"
)
```

#### Register with Custom Server and Protocol

```
import graphistry

graphistry.register(
    api=3,
    protocol="http",
    server="my_custom_server.com",
    username="my_username",
    password="my_password"
)
```

#### Register with Custom Browser Routing

```
import graphistry

graphistry.register(
    api=3,
    server="my_api_server.com",
    username="my_username",
    password="my_password",
    client_protocol_hostname="https://my_ui_server.com"
)
```

---

#### 3.3.1.6 Best Practices

- **Security**: Always use secure protocols (*https*) and validate certificates.
- **Authentication**: Use *personal_key_id* and *personal_key_secret* for automation.
- **SSO**: For organizations, ensure correct *org_name* and, if needed, *idp_name*.
- **Session Management**: The library handles session tokens automatically; ensure safe credential handling when enabling memory storage.

#### 3.3.1.7 Troubleshooting

- **Connection Errors**: Check the *server* and *protocol* parameters and ensure your network allows access.
- **Authentication Failures**: Verify credentials. For SSO, ensure *org_name* and *idp_name* are correct.
- **SSL Issues**: Validate that the server certificate is valid or consider disabling SSL validation (*certificate_validation=False*), though not recommended.

### 3.3.2 Sharing and Access Control

Graphistry provides powerful tools for visualizing and sharing graph data securely. Understanding how to manage privacy settings and share visualizations appropriately is essential for collaborative work and data security. This guide will help you understand how to control privacy settings using the Graphistry API. For more examples, see the Sharing Tutorial Notebook.

#### 3.3.2.1 Overview of Privacy Settings

You have full control over who can view or edit your visualizations. By default, Graphistry visualizations are **public** but **unlisted**, meaning you need to have been given the secret ID of the visualization to know where it is, but do not need to log in to see it. Privacy settings can be adjusted when you create a plot using the *plot()* method.

Key privacy levels include:

- **Private**: Only you can view the visualization.
- **Organization (`"org"`)**: Anyone in your organization can view the visualization.
- **Public** (**unlisted**): Anyone with the link can view the visualization. Graphistry does not make the list of visualizations public, so this os the equivalent of the **unlisted** privacy mode in many platforms.
- **Custom Sharing**: Share with individual users (requires additional configuration).

When sharing with others, you may also configure settings such as *viewer* vs *editor*.

#### 3.3.2.2 Getting Started with Privacy: Public (unlisted)

Before adjusting privacy settings, ensure you have registered with Graphistry:

```
import graphistry

graphistry.register(api=3, username='my_username', password='my_password')
```

By default, any plot you create is public (unlisted), meaning others will not know about your visualization, but if you share a link to it, they can see it without logging in.

#### 3.3.2.3 Creating a Private Visualization

You can set a visualization to a stricter mode by calling *graphistry.privacy()*:

```
graphistry.privacy()


# Sample data
edges = pd.DataFrame({
    'src': ['A', 'B', 'C'],
    'dst': ['B', 'C', 'A']
})

# Create a private plot
plot_url = graphistry.edges(edges, 'src', 'dst').plot(render=False)

print(f"Private visualization URL: {plot_url}")
```

If you are logged into your personal account, only you can access this plot. If you are logged into an organization, the visualization will be private to organization members. When anyone else obtains the URL, they won't be able to view it until you adjust the privacy settings.

#### 3.3.2.4 Sharing Visualizations Within Your Organization

To share a visualization with members of your organization:

```
graphistry.privacy(mode='organization')

# Create an organization-shared plot
plot_url = graphistry.edges(edges, 'src', 'dst').plot(render=False)

print(f"Organization-shared visualization URL: {plot_url}")
```

Now, anyone within your organization who has access to Graphistry can view the plot using the provided URL.

#### 3.3.2.5 Making Visualizations Public

To make a visualization accessible to anyone with the link:

```
graphistry.privacy(mode='public')

# Create a public plot
plot_url = graphistry.edges(edges, 'src', 'dst').plot(render=False)

print(f"Public visualization URL: {plot_url}")
```

This setting is useful when sharing with external collaborators or embedding visualizations in public websites.

#### 3.3.2.6 Controlling Edit Permissions

By default, shared visualizations are editable by same-org members. To allow others to edit or interact with the visualization settings, or set to read-only, you can reconfigure the policy:

```
VIEW = '10'
EDIT = '20'
graphistry.privacy(mode='organization', mode_action=EDIT)

# Allow others to edit the plot
plot_url = graphistry.edges(edges, 'src', 'dst').plot(render=False)

print(f"Editable visualization URL: {plot_url}")
```

#### 3.3.2.7 Understanding Privacy Levels

- **Private**: Only accessible to the creator.
- **Organization (`"org"`)**: Accessible to all users within your Graphistry organization.
- **Public**: Unlisted in any public index, but accessible to anyone with the link. Use cautiously, as this allows broad access.
- **Custom**: Advanced configurations for sharing with specific users.

#### 3.3.2.8 Best Practices for Data Privacy

- **Use Organization Sharing for Internal Collaboration**: Keeps data within your company's control.
- **Limit Public Sharing**: Only make visualizations public if the data is non-sensitive and intended for broad distribution.
- **Regularly Review Shared Visualizations**: Periodically check which visualizations are shared and adjust privacy settings as needed.
- **Use Secure Methods for Sharing Links**: When sharing URLs, use secure channels to prevent unauthorized access.

#### 3.3.2.9 Advanced Features

Look at the documentation and tutorial for individual parameters for more advanced usage modes:

- Invite individual users, including with optional notification emails, using parameters *invited_users* and *notify*
- Use nested privacy settings (*g2 = g1.privacy()*)

#### 3.3.2.10 Additional Resources

For more detailed examples and advanced features, refer to the **Graphistry Sharing Tutorial** available in the official documentation or GitHub repository.

- **Sharing Tutorial Notebook**: *https://github.com/graphistry/pygraphistry/blob/master/demos/more_examples/graphistry_*

This tutorial covers topics such as:

- Creating custom share links
- Embedding visualizations in web applications
- Using access tokens for secure sharing
- Advanced privacy configurations

#### 3.3.2.11 Conclusion

Managing privacy and sharing settings in Graphistry is straightforward and flexible. By understanding and utilizing these features, you can securely collaborate with others while maintaining control over your data.

Remember to:

- Choose the appropriate privacy level for your needs.
- Be cautious when making visualizations public.
- Regularly audit your shared visualizations.
- Use *graphistry.privacy()* to stay informed about your data handling.

## 3.4 Visualize

We recommend getting started with *10 Minutes to PyGraphistry*, *10 Minutes to Graphistry Visualization*, and the *layout guide*

See also:

### 3.4.1 10 Minutes to Graphistry Visualization

This guide covers core visualization topics like the difference between uploading and viewing graphs, how the client/server architecture works, and how to use PyGraphistry's fluent API to create powerful visualizations by combining ideas like encodings, layouts, and settings. Finally, we overview how to embed visualizations into different workflows.

#### 3.4.1.1 Key Concepts

- *Client/Server Architecture*
- *Fluent API Style*
- *Shaping Your Data*
- *Layouts*
- *Node & Edge Encodings*

- *Global URL settings*
- *Plotting: Inline and URL Rendering*
- *Additional Resources*

#### Client/Server Architecture: Uploading vs. Serving vs. Viewing

PyGraphistry uses a **client-server** model. By separating the uploader, server, and viewer, we can achieve better performance, new capabilities, and a variety of usage modes.

- **Upload Client**: In your local environment, you can shape data and call the Graphistry API to upload it to a server (self-hosted or Graphistry Hub).
- **Visualization Server**: The server processes the data using GPU acceleration to handle large graphs.
- **Visualization Client**: The graph is then explored in your browser, where interactions like zooming and filtering are handled smoothly by using local and remote GPU resources as appropriate.

This split architecture allows scalable, high-performance visualization for even the largest datasets.

#### Fluent API Style

PyGraphistry uses a **fluent style** API, which means that methods can be chained together. This allows for concise and readable code without an extensive setup:

```
g1 = graphistry.edges(df, 'src', 'dst')
g2 = g1.nodes(df2, 'n')
g3 = g2.encode_point_size('score')
g3.plot()

# As shorter fluent lines
g = graphistry.edges(df, 'src', 'dst').nodes(df2, 'n')
g.encode_point_size('score').plot()
```

This approach lets you layer operations as needed, keeping code light and intuitive.

### 3.4.1.2 Shaping Your Data

PyGraphistry supports flexible shaping of your graph data:

- **`.edges()` & `.nodes()`**: Define edges between entities and optional node attributes

```
# df[['src', 'dst', ...]]
graphistry.edges(df, 'src', 'dst').plot()

# ... + df2[['n', ...]]
graphistry.edges(df, 'src', 'dst').nodes(df2, 'n').plot()
```

- **Hypergraph**: Use multiple columns for nodes for more complex visualizations

```
# df[['actor', 'event', 'location', ...]]
hg = graphistry.hypergraph(df, ['actor', 'event', 'location'])
hg['graph'].plot()
```

- **UMAP**: Dimensionality reduction & embedding visualization tool based on row similarity

```
# df[['score', 'time', ...]]
graphistry.nodes(df).umap(X=['score', 'time']).plot()
```

These methods ensure you can quickly load & shape data and move into visualizing.

#### 3.4.1.3 Layouts

PyGraphistry's *Layout catalog* provides many options, covering:

- **Live Layout**: Graphistry performs GPU-accelerated force-directed layouts at interaction time. You can adjust settings, such as gravity, edge weight, and initial clustering time:

```
g.settings(url_params={'play': 7000, 'info': True}).plot()
```

- **PyGraphistry Layouts**: PyGraphistry ships with special layouts unavailable elsewhere and that work with the rendering engine's special features:

```
g.time_ring_layout('time_col').plot()
```

- **Plugin Layouts**: Integrated use of external libraries for specific layouts:
  - *Graphviz* for hierarchical and directed layouts such as the *"dot"* engine
  - *cuGraph* for GPU-accelerated FA2, a weaker version of Graphistry's live layout
  - *igraph* for CPU-based layouts, similar to GraphViz and with layouts that focus more on medium-sized social networks
- **External Layouts**: Pass in $x$, $y$ columns, such as from your own edits, external data, or external ML/AI packages:

```
# nodes_df[['x', 'y', 'n', ...]]
g = graphistry.edges(e_df, 's', 'd').nodes(nodes_df, 'n')
g2 = g.settings(url_params={'play': 0}) # skip initial loadtime layout
g2.plot()
```

#### 3.4.1.4 Node & Edge Encodings

You can encode your graph attributes visually using colors, sizes, icons, and more:

- **Direct Encoding**: Set attributes like color directly on nodes or edges.

```
g.encode_point_color('type', categorical_mapping={'A': 'red', 'B': 'blue'}).plot()
```

- **Categorical & Continuous Mappings**: Handle both discrete and continuous data:

```
g.encode_point_color('score', ['blue', 'yellow', 'red'], as_continuous=True).plot()
```

- **Encodings List**: Beyond colors, you can also adjust edge thickness, node icon, and add badges using the following methods:
  - Points:
    * *graphistry.PlotterBase.PlotterBase.encode_point_badge()*
    * *graphistry.PlotterBase.PlotterBase.encode_point_color()*

        * *graphistry.PlotterBase.PlotterBase.encode_point_icon()*
        * *graphistry.PlotterBase.PlotterBase.encode_point_size()*
    - Edges:
        * *graphistry.PlotterBase.PlotterBase.encode_edge_badge()*
        * *graphistry.PlotterBase.PlotterBase.encode_edge_color()*
        * *graphistry.PlotterBase.PlotterBase.encode_edge_icon()*

- **Bind**: Simpler data-driven settings are done through *graphistry.PlotterBase.PlotterBase.bind()*:

```
g.bind(point_title='my_node_title_col')
```

Where:

```
bind(source=None, destination=None, node=None, edge=None,
  edge_title=None, edge_label=None, edge_color=None, edge_weight=None,
  edge_size=None, edge_opacity=None, edge_icon=None,
  edge_source_color=None, edge_destination_color=None,
  point_title=None, point_label=None, point_color=None, point_weight=None,
  point_size=None, point_opacity=None, point_icon=None, point_x=None, point_y=None
)
```

#### 3.4.1.5 Global URL settings

Graphistry visualizations are highly configurable via URL parameters. You can control the look, interaction, and data filters:

```
g.settings(url_params={'play': 7000, 'info': True}).plot()
```

For a complete list of parameters, refer to the official REST URL params page.

#### 3.4.1.6 Plotting: Inline and URL Rendering

Once you're ready to visualize, use *.plot()* to render:

- **Inline Plotting**: Directly embed interactive visualizations in your notebook or Python environment:

```
g.plot()
```

- **URL Rendering**: Get a sharable and embeddable URL to view in the browser:

```
url = g.plot(render=False)
print(f"View your graph at: {url}")
```

You can further control the embeded visualization using URL parameters and JavaScript

#### 3.4.1.7 Next Steps

- *10 Minutes to GFQL*: Use GFQL to query and manipulate your graph data before visualization.
- *Layout guide*: Explore different layouts for your visualizations.
- *Plugins*: Discover more ways to connect to your data and work with your favorite tools.
- *Layout catalog*: Dive deeper into the layout options available in PyGraphistry.
- *PyGraphistry API Reference*

#### 3.4.1.8 External Resources

To dive deeper into graph analytics and visualizations, check out the following resources:

- Graphistry Get Started
- GraphistryJS Clients: NodeJS, React, & Vanilla
- Graphistry GitHub
- Slack Community

Happy graphing!

### 3.4.2 UI Guide

Head over to the Graphistry UI guide for a variety of walkthroughs, including:

- Basics: How to navigate the Graphistry UI
- Filters and exclusions: Drilling down with filters and exclusions
- Histograms: Analyze attributes and manipulate colors, sizes, filters, and more
- Selections: Selections to analyze data and work across tools more easily

### 3.4.3 Quick Guide to PyGraphistry layouts

This guide provides a quick introduction to key layout concepts in PyGraphistry

#### 3.4.3.1 Key Concepts Covered

- *Precomputed Layouts*
- *Internal & Plugin Layouts*
- *Runtime Dynamic Layouts*
- *Runtime Layout Settings*
- Further reading and detailed configuration options for: - *Ring Layout API* - *GIB Layout API* - *Modularity Layout API* - Plugin layouts: *GraphViz*, *cuGraph*, *iGraph*

—

#### 3.4.3.2 Key Concepts

##### Precomputed Layouts

Precomputed layouts involve manually calculating node positions ($x$, $y$ columns) before rendering your graph.

This is useful such as when you need to manually control a layout, or are visualizing externally provided positions such as from embeddings.

```
# Precomputed 'x', 'y' coordinates in a nodes DataFrame
g = graphistry.edges(e_df, 'src', 'dst').nodes(n_df, 'n')
g2 = g.settings(url_params={'play': 0})  # skip initial loadtime layout
g2.plot()
```

Precomputed layouts are ideal for handling complex visualizations where precision is key.

##### Internal & Plugin Layouts

PyGraphistry includes a growing number of built-in layouts.

These help with several scenarios, including:

- Faster performance and greater scale
- Leveraging Graphistry runtime layout features
- Combining layouts

**Graphistry Layouts:**

- **Native Force-Directed Layout:** PyGraphistry's default layout automatically arranges the nodes based on their connectivity.

  ```
  g = graphistry.edges(e_df, 'src', 'dst').plot()
  ```

- **Ring Layout:** Ideal for visualizing sorted, hierarchical, or time-based data.

  ```
  g.time_ring_layout('my_timestamp').plot()
  g.categorical_ring_layout('my_type').plot()
  g.continuous_ring_layout('my_score').plot()
  ```

  For further details, refer to the *Ring Layout API*.

- **Modularity Weighted Layout:** Weights edges based on modularity.

  ```
  # Separate by precomputed modules
  assert 'partition' in g._nodes
  g.modularity_weighted_layout(community_col='partition').plot()

  # Separate by automatically computed modules
  g.modularity_weighted_layout(community_alg='louvain', engine='cudf').plot()
  ```

  Read more in the *Modularity Layout API*.

- **Group-in-a-Box Layout:** Groups nodes into a grid of clusters.

  Popularized by NodeXL for analyzing large social networks, the PyGraphistry version enables quickly working with larger datasets than possible in other packages

```
g.gib_layout().plot()
```

Learn more in the *Group-in-a-Box Layout API*.

**Plugin Layouts:**

- **cuGraph Plugin (GPU-accelerated force layouts):** Ideal for large-scale graphs requiring performance.

```
g.cugraph_force_layout().plot()
```

See the *cuGraph Plugin* for more details.

- **GraphViz Plugin (Hierarchical layouts):** Great for tree-like or hierarchical data.

```
g.graphviz_layout(engine='dot').plot()
```

Find more details in the *GraphViz Plugin*.

- **iGraph Plugin (Kamada-Kawai, Sugiyama, etc.):** Provides classic layout algorithms for a variety of graph types.

```
g.igraph_layout('kamada_kawai').plot()
```

See the *iGraph Plugin* for more information.

#### Runtime Dynamic Layouts

Dynamic layouts allow PyGraphistry to adjust node positions in real-time based on user interactions and graph updates. This provides highly interactive and scalable graph visualizations.

```
# Run the force-directed layout at viz load time for 5 seconds (5,000␣
↪milliseconds)
g = graphistry.edges(e_df, 'src', 'dst')
g.settings(url_params={'play': 5000}).plot()
```

For details on runtime settings and customization, explore the *Layout Settings* page.

---

### 3.4.3.3 Further Reading

Layout in general:

- *Layout Catalog*
- *Layout Settings*

Individaul layouts and plugins:

- *Ring Layout API*
- *GIB Layout API*
- *Modularity Layout API*
- *GraphViz Plugin*
- *cuGraph Plugin*
- *iGraph Plugin*

### 3.4.4 PyGraphistry Layout Catalog

This page provides an overview of the main layouts available in PyGraphistry, including through plugins like graphviz and igraph. Each optimizes for different use cases. Click on a plugin to jump to its section.

- *PyGraphistry Plugin*: GPU-accelerated layouts like ForceAtlas2, modularity-weighted, UMAP, and more.
- *cuGraph Plugin*: Large-scale graph layouts with GPU-optimized ForceAtlas2.
- *Graphviz Plugin*: Hierarchical, directed, and flowchart-like layouts for medium-sized graphs.
- *igraph Plugin*: Versatile 2D/3D layouts including Fruchterman-Reingold, Kamada-Kawai, and more.
- *Custom Layouts*: Manually compute or post-process custom layouts.

#### 3.4.4.1 PyGraphistry Plugins

PyGraphistry supports GPU-accelerated layouts, including ForceAtlas2, modularity-weighted algorithms, and hierarchical ring layouts for large-scale and specialized structures. (*API reference on Graphistry layouts*)

**Supported Layouts**:

- **ForceAtlas2** — Optimized for large, dense graphs. Provides smooth clustering and cluster separation using GPU acceleration. (Implicit: Dynamic load-time run of Graphistry's GPU-accelerated ForceAtlas2)
- **Modularity-Weighted** — Lays out clusters based on modularity, optimizing for visualizing community structures. *API info on modularity-weighted layouts*
- **Group-In-A-Box (GIB)** — Organizes nodes into visually distinct boxes based on their group or cluster for clear structure definition. *API info on group-in-a-box layouts*
- **UMAP** — Reduces high-dimensional data into a 2D layout based on similarity, best for complex datasets needing dimensionality reduction. `API info on UMAP`
- **Hierarchical Ring Layouts** — Creates ring layouts that categorize nodes by time, continuous variables, or categorical properties. *API info on ring layouts*

**Example**:

Visit the *PyGraphistry visualization tutorial*.

```
g.time_ring_layout('time_col').plot()
```

#### 3.4.4.2 cuGraph Plugin

cuGraph provides one GPU-optimized graph layout for scaling large datasets, making it a candidate for massive graphs. (*API reference on cuGraph*)

**Supported Layouts**:

- **ForceAtlas2** — Designed for very large graphs, scaling with GPU acceleration to maintain interactive performance with 100k+ nodes. Less flexible version of the Graphistry ForceAtlas2 GPU algorithm.

```
g.cugraph_layout('force_atlas2').plot()
```

#### 3.4.4.3 Graphviz Plugin

Graphviz specializes in directed and hierarchical layouts, useful for flowcharts, dependency trees, and acyclic graphs (DAGs). (*API reference on graphviz layouts*)

**Supported Layouts**:

- **acyclic** — Removes cycles from directed graphs by reversing edges to make the graph acyclic, useful for processing DAGs.
- **ccomps** — Extracts the connected components from a graph and outputs them as subgraphs.
- **circo** — Circular layout, arranging nodes in a radial fashion, ideal for cycle graphs.
- **dot** — Best for directed acyclic graphs (DAGs) like flowcharts, laying out hierarchies in a top-down manner.
- **fdp** — General force-directed layout, good for smaller undirected graphs.
- **gc** — Used for graph coloring, assigning colors to nodes such that no two adjacent nodes have the same color.
- **gvcolor** — Colorizes graphs based on specific attributes, often used for improving visual distinctions between nodes.
- **gvpr** — Graph pattern scanning and rewriting tool used for scripting changes in a graph, allowing custom manipulation of graph structures.
- **neato** — Force-directed layout for undirected graphs, suitable for smaller networks.
- **nop** — A no-op layout that performs no layout calculations, often used as a placeholder or for manual layout adjustments.
- **osage** — Useful for directed layered graphs with hierarchical structures.
- **patchwork** — Visualizes hierarchical clusters as a nested set of rectangles, similar to a treemap visualization.
- **sccmap** — Finds the strongly connected components in a graph and generates a reduced graph of those components.
- **sfdp** — Force-directed layout optimized for large graphs, providing fast and scalable rendering.
- **tred** — Transitive reduction algorithm that minimizes the number of edges while maintaining reachability between nodes in a directed graph.
- **twopi** — Radial layout that positions nodes in concentric circles, useful for radial hierarchies.
- **unflatten** — Improves readability by adjusting node levels to reduce overlap in hierarchical graphs.

**Example**:

Visit the *API reference on graphviz page* for more examples.

```
g.layout_graphviz('dot').plot()
```

#### 3.4.4.4 igraph Plugin

The igraph plugin offers various layouts forvarious graph types. (*API reference on igraph*)

**Supported Layouts**:

- **auto / automatic** — Automatically chooses the best layout for the given graph based on its structure and size.
- **bipartite** — Positions nodes in two layers, useful for visualizing bipartite graphs (graphs with two distinct sets of nodes).
- **circle / circular** — Positions nodes in a circular layout, suitable for visualizing cycles and small networks.
- **circle_3d / circular_3d** — 3D version of the circular layout, positioning nodes in a 3D circular structure.
- **davidson_harel / dh** — Force-directed layout algorithm with an iterative approach for improving graph aesthetics, especially useful for smaller graphs.
- **drl** — Distributed Recursive Layout, a force-directed layout algorithm optimized for very large graphs.
- **drl_3d** — 3D version of the DRL algorithm, optimized for large graphs in a 3D space.
- **fr / fruchterman_reingold** — Force-directed layout balancing attractive and repulsive forces for clustered yet separated nodes.
- **fr_3d / fruchterman_reingold_3d / fr3d** — 3D version of the Fruchterman-Reingold force-directed layout.
- **grid** — Organizes nodes in a grid structure, useful for matrix-like data.
- **grid_3d** — 3D version of the grid layout, positioning nodes in a 3D grid.
- **graphopt** — Another force-directed layout algorithm, known for its fast convergence on small to medium-sized graphs.
- **kk / kamada_kawai** — Similar to Fruchterman-Reingold, this force-directed layout focuses on preserving geometric distances between nodes.
- **kk_3d / kamada_kawai_3d / kk3d** — 3D version of the Kamada-Kawai algorithm, preserving distances between nodes in a 3D space.
- **lgl / large / large_graph** — Optimized for very large graphs, often used for graphs with thousands of nodes.
- **mds** — Multi-Dimensional Scaling, used for dimensionality reduction and projecting nodes into 2D or 3D space based on similarity.
- **random / random_3d** — Randomly positions nodes in 2D or 3D space, often used for testing or debugging layout algorithms.
- **reingold_tilford / rt / tree** — Specialized for tree structures, arranging nodes hierarchically from top to bottom.
- **reingold_tilford_circular / rt_circular** — Circular version of the Reingold-Tilford tree layout, arranging tree nodes in a radial fashion.
- **sphere / spherical** — 3D layout positioning nodes on the surface of a sphere, useful for 3D graph exploration.
- **star** — Positions nodes in a star configuration, with a central node surrounded by peripheral nodes.
- **sugiyama** — Specialized for hierarchical structures, often used for organizational charts and trees.

Full list: *More Info*

**Example**:

Visit the *API reference on graphviz* for more examples.

```
g.layout_igraph('circle').plot()
```

#### 3.4.4.5 Custom Layouts

Users can manually compute layouts from external sources or post-process the results. This allows flexibility in integrating custom embedding algorithms or other specialized layouts into PyGraphistry. (API reference)

**Example**:

Manually apply a layout and visualize by custom layouts (notebook) .

```
# Input: Precompute some x and y positions
nodes_df : pd.DataFrame = ...
assert 'x' in df.columns and 'y' in df.columns

g2 = (g1
    .nodes(nodes_df)
    .bind(point_x='x', point_y='y')
    .settings(url_params={'play': 0})  # Prevent loadtime layout from running
)
```

#### 3.4.4.6 Further reading

- *PyGraphistry API Reference*: GPU-accelerated layouts such as ForceAtlas2, modularity-weighted, hierarchical rings, UMAP, and group-in-a-box.
- *cuGraph API Reference*: ForceAtlas2 optimized for large-scale graphs using GPU acceleration.
- *Graphviz API Reference*: Best for hierarchical and flowchart/DAG layouts, including options like dot, neato, and circo.
- *igraph API Reference*: Versatile with 2D/3D layouts, including Fruchterman-Reingold, Kamada-Kawai, and Sugiyama.

Visit the respective tutorial links to dive deeper into each plugin's capabilities and usage.

### 3.4.5 Layout Settings & Visualization Embedding

This guide shows how to embed and configure Graphistry visualizations using the PyGraphistry Python API. For users interested in using URL parameters for embedding in HTML, refer to the external documentation.

#### 3.4.5.1 Using PyGraphistry for Customization

You can use the PyGraphistry API to programmatically configure visualizations. Below are some examples of how to use the *g.settings* and *g.addStyle* methods to customize visualizations.

##### Scene Settings

Use *graphistry.PlotterBase.PlotterBase.scene_settings()* to modify the appearance of the graph, including menus, node sizes, and edge opacity:

```
g2 = g.scene_settings(
    # Hide menus
    menu=False,
    info=False,
    # Customize graph
    show_arrows=False,
    point_size=1.0,
    edge_curvature=0.0,
    edge_opacity=0.5,
    point_opacity=0.9
).plot()
```

##### Styling the Background and Foreground

With *graphistry.PlotterBase.PlotterBase.addStyle()*, you can configure background and foreground styles, including colors, gradients, and images:

```
# Set a red background
g.addStyle(bg={'color': 'red'})

# Apply a radial gradient background
g.addStyle(bg={
    'color': '#333',
    'gradient': {
        'kind': 'radial',
        'stops': [
            ["rgba(255,255,255, 0.1)", "10%", "rgba(0,0,0,0)", "20%"]
        ]
    }
})

# Use an image as a background with blend mode
g.addStyle(bg={'image': {'url': 'http://site.com/cool.png', 'blendMode': 'multiply'}})

# Apply blend mode for the foreground
g.addStyle(fg={'blendMode': 'color-burn'})
```

#### Page and Logo Settings

Customize the page title, favicon, and logo using *graphistry.PlotterBase.PlotterBase.addStyle()*, :

```
# Set page title and favicon
g.addStyle(page={'title': 'My Site'})
g.addStyle(page={'favicon': 'http://site.com/favicon.ico'})

# Add a logo
g.addStyle(logo={'url': 'http://www.site.com/transparent_logo.png'})

# Customize logo dimensions and opacity
g.addStyle(logo={
    'url': 'http://www.site.com/transparent_logo.png',
    'dimensions': {'maxHeight': 200, 'maxWidth': 200},
    'style': {'opacity': 0.5}
})
```

For more advanced Python configuration options, refer to the PyGraphistry REST API documentation on URL parameters and Branding metadata.

### 3.4.5.2 HTML/URL-based Configuration

For users interested in configuring Graphistry visualizations through HTML and URL parameters, please refer to the official documentation:

- Graphistry URL Configuration Options

This guide covers how to embed Graphistry visualizations in web pages and configure visualizations via URL parameters like background color, layout settings, and more.

#### IFrame CSS Style Tips

When embedding visualizations in HTML, you can customize the appearance using CSS. Below are some common style tips for *<iframe>* elements:

- **Control the border**:

```
border: 1px solid black;
```

- **Control the size**:

```
width: 100%; height: 80%; min-height: 400px;
```

Refer to the full Graphistry URL Configuration Options for more details.

## 3.5 GFQL: The Dataframe-Native Graph Query Language

Welcome to **GFQL**, the first dataframe-native graph query language with GPU support. GFQL is part of the **PyGraphistry** ecosystem and is designed to make graph analytics easier and faster without the need for external complex infrastructure such as databases. Whether you're working with **CPUs** or leveraging **GPU acceleration** for massive datasets, GFQL integrates seamlessly with your data science workflows through a simple *pip install graphistry*.

**GFQL bridges the gap** between traditional storage-tier graph databases and the modern compute tier, allowing you to perform your favorite high-performance graph queries directly on your dataframes. It's built to be familiar to users of Cypher, other graph query languages, and popular dataframe libraries. By being native to accelerated Python datascience dataframe technologies such as Apache Arrow, Numpy, Nvidia RAPIDS, and Graphistry, it can already do workloads like 100M+ edges in interactive time on a single machine.

See also:

### 3.5.1 10 Minutes to GFQL

Welcome to **GFQL (GraphFrame Query Language)**, the first **dataframe-native graph query language**. GFQL is designed to bring the power of graph queries to your data science workflows without the need for external graph databases or complex infrastructure. It integrates seamlessly with the **PyData**, **Apache Arrow**, and **GPU acceleration** ecosystems, allowing you to process massive graphs efficiently.

In this guide, we'll explore the basics of GFQL in just 10 minutes. You'll learn how to:

- Query and filter nodes and edges.
- Chain multiple hops and apply predicates.
- Leverage automatic GPU acceleration.
- Integrate GFQL into your existing Python workflows.

Let's dive in!

#### 3.5.1.1 Introduction to GFQL

GFQL fills a critical gap in the data community by providing an in-process, high-performance graph query language that operates at the compute tier. Unlike traditional graph databases that couple storage and compute, GFQL allows you to perform graph queries directly on your dataframes, whether they're in-memory or on disk, CPU or GPU.

**Key Benefits:**

- **Dataframe-Native:** Works directly with Pandas, cuDF, and other dataframe libraries.
- **High Performance:** Optimized for both CPU and GPU execution.
- **Ease of Use:** No need for external databases or new infrastructure.
- **Interoperability:** Integrates with the Python data science ecosystem, including PyGraphistry for visualization.

#### 3.5.1.2 Setting Up GFQL

GFQL is part of the open-source *graphistry* library. Install it using pip:

```
pip install graphistry
```

Ensure you have *pandas* or *cudf* installed, depending on whether you want to run on CPU or GPU.

#### 3.5.1.3 Basic Concepts

Before we begin with examples, let's understand some basic concepts:

- **Nodes and Edges:** In GFQL, graphs are represented using dataframes for nodes and edges.
- **Chaining:** GFQL queries are constructed by chaining operations that filter and traverse the graph.
- **Predicates:** Conditions applied to nodes or edges to filter them based on properties.

#### 3.5.1.4 Examples

##### 1. Find Nodes of a Certain Type

You can filter nodes based on their properties using the *n()* function.

**Example: Find all nodes of type "person"**

```
from graphistry.ast import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
print('Number of person nodes:', len(people_nodes_df))
```

**Explanation:**

- *n({"type": "person"})* filters nodes where the *type* property is *"person"*.
- *g.chain([...])* applies the chain of operations to the graph *g*.
- *._nodes* retrieves the resulting nodes dataframe.

##### 2. Find 2-Hop Edge Sequences with an Attribute

Traverse multiple hops and filter edges based on attributes using *e_forward()*.

**Example: Find 2-hop paths where edges are marked as "interesting"**

```
from graphistry.ast import e_forward

g_2_hops = g.chain([ e_forward({"interesting": True}, hops=2) ])
print('Number of edges in 2-hop paths:', len(g_2_hops._edges))
g_2_hops.plot()
```

**Explanation:**

- *e_forward({"interesting": True}, hops=2)* traverses forward edges with *interesting == True* for 2 hops.
- *g_2_hops.plot()* visualizes the resulting subgraph.

### 3. Find Nodes 1-2 Hops Away and Label Each Hop

Label hops in your traversal to analyze specific relationships.

**Example: Find nodes up to 2 hops away from node "a" and label each hop**

```
from graphistry.ast import n, e_undirected

g_2_hops = g.chain([
    n({g._node: "a"}),
    e_undirected(name="hop1"),
    e_undirected(name="hop2")
])
first_hop_edges = g_2_hops._edges[ g_2_hops._edges.hop1 == True ]
print('Number of first-hop edges:', len(first_hop_edges))
```

**Explanation:**

- *n({g._node: "a"})* starts the traversal from node *"a"*.
- *e_undirected(name="hop1")* traverses undirected edges and labels them as *hop1*.
- *e_undirected(name="hop2")* continues traversal and labels edges as *hop2*.
- The labels allow you to filter and analyze edges from specific hops.

### 4. Query for Transaction Nodes Between Risky Nodes

Chain multiple traversals to find patterns between nodes.

**Example: Find transaction nodes between two types of risky nodes**

```
from graphistry.ast import n, e_forward, e_reverse

g_risky = g.chain([
    n({"risk1": True}),
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
hits = g_risky._nodes[ g_risky._nodes.hit == True ]
print('Number of transaction hits:', len(hits))
```

**Explanation:**

- Starts from nodes with *risk1 == True*.
- Traverses forward to transaction nodes, labeling them as *hit*.
- Traverses backward to nodes with *risk2 == True*.
- Identifies transaction nodes connected between two risky nodes.

#### 5. Filter by Multiple Node Types Using *is_in*

Use the *is_in* predicate to filter nodes or edges by multiple values.

**Example: Filter nodes and edges by multiple types**

```
from graphistry.ast import n, e_forward, e_reverse, is_in

g_filtered = g.chain([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
hits = g_filtered._nodes[ g_filtered._nodes.hit == True ]
print('Number of filtered hits:', len(hits))
```

**Explanation:**

- Filters nodes of type *"person"* or *"company"*.
- Traverses forward edges of type *"owns"* or *"reviews"*.
- Filters nodes of type *"transaction"* or *"account"*, labeling them as *hit*.
- Traverses backward to nodes with *risk2 == True*.

#### 3.5.1.5 Leveraging GPU Acceleration

GFQL is optimized for GPU acceleration using *cudf* and *rapids*. When using GPU dataframes, GFQL automatically executes queries on the GPU for massive speedups.

#### 6. Automatic GPU Acceleration

**Example: Run GFQL queries with GPU dataframes**

```
import cudf
import graphistry

# Load data into GPU dataframes
e_gdf = cudf.read_parquet('edges.parquet')
n_gdf = cudf.read_parquet('nodes.parquet')

# Create a graph with GPU dataframes
g_gpu = graphistry.edges(e_gdf, 'src', 'dst').nodes(n_gdf, 'id')

# Run GFQL query (executes on GPU)
g_result = g_gpu.chain([ ... ])
print('Number of resulting edges:', len(g_result._edges))
```

**Explanation:**

- *cudf.read_parquet()* loads data directly into GPU memory.
- GFQL detects *cudf* dataframes and runs the query on the GPU.

- Achieves significant performance improvements on large datasets.

#### 7. Forcing GPU Mode

You can explicitly set the engine to ensure GPU execution.

**Example: Force GFQL to use GPU engine**

```
g_result = g_gpu.chain([ ... ], engine='cudf')
```

**Explanation:**

- *engine='cudf'* forces the use of the GPU-accelerated engine.
- Useful when you want to ensure the query runs on the GPU.

#### 3.5.1.6 Integration with PyData Ecosystem

GFQL integrates seamlessly with the PyData ecosystem, allowing you to combine it with libraries like *pandas*, *networkx*, *igraph*, and *PyTorch*.

#### 8. Combining GFQL with Graph Algorithms

**Example: Compute PageRank on the resulting graph**

```
# Assuming g_result is the result from a GFQL query

# Compute PageRank using cuGraph (GPU)
g_enriched = g_result.compute_cugraph('pagerank')

# View top nodes by PageRank
top_nodes = g_enriched._nodes.sort_values('pagerank', ascending=False).head(5)
print('Top nodes by PageRank:')
print(top_nodes[['id', 'pagerank']])
```

**Explanation:**

- *compute_cugraph('pagerank')* computes the PageRank of nodes using GPU acceleration.
- The enriched graph now contains a *pagerank* column in the nodes dataframe.

#### 9. Visualizing the Graph

Use PyGraphistry's visualization capabilities to explore your graph.

**Example: Visualize high PageRank nodes**

```
# Filter nodes with high PageRank
g_high_pagerank = g_enriched.chain([
    n(query='pagerank > 0.1'),
    e(),
    n(query='pagerank > 0.1')
])
```

(continues on next page)

(continued from previous page)

```
# Plot the subgraph
g_high_pagerank.plot()
```

**Explanation:**

- Filters nodes where *pagerank > 0.1*.
- Visualizes the subgraph consisting of high PageRank nodes.

#### 3.5.1.7 Conclusion and Next Steps

Congratulations! You've covered the basics of GFQL in just 10 minutes. You've learned how to:

- Query and filter nodes and edges using GFQL.
- Chain multiple hops and apply advanced predicates.
- Leverage GPU acceleration for high-performance graph querying.
- Integrate GFQL with graph algorithms and visualization tools.

**Next Steps:**

- **Explore the Documentation:** Dive deeper into GFQL's capabilities by exploring the full documentation.
- **Try GFQL on Your Data:** Apply what you've learned to your datasets and see the benefits firsthand.
- **Leverage PyGraphistry:** Utilize PyGraphistry for advanced visualization and analysis.
- **Join the Community:** Connect with other users and developers in the GFQL community Slack channel.

GFQL opens up new possibilities for graph analysis at scale, without the overhead of managing external databases or infrastructure. With its seamless integration into the Python ecosystem and support for GPU acceleration, GFQL is a powerful tool for modern data science workflows.

Happy graph querying!

### 3.5.2 Overview of GFQL

New to GFQL, the open source dataframe-native graph query language? This article overviews the gaps it fills, special features like GPU accelerations, and where to go next.

#### 3.5.2.1 Why GFQL?

GFQL addresses a critical gap in the data community by providing an in-process graph query language that operates at the compute tier. This means you can:

- **Graph search**: Easily and efficiently query and filter nodes and edges using a familiar syntax.
- **Avoid External Infrastructure**: Avoid calls to external infrastructures and eliminate the need for extra databases.
- **Leverage Existing Workflows**: Integrate with your current Python data science tools and libraries.
- **Achieve High Performance**: Utilize GPU acceleration for massive speedups in graph processing.
- **Simplify Graph Analytics**: Write expressive and concise graph queries in Python.

#### 3.5.2.2 Key Features

- **Dataframe-Native Integration**: Works directly with Pandas, cuDF, and Apache Arrow dataframes.
- **High Performance**: Optimized for both CPU and GPU execution, capable of processing billions of edges.
- **Ease of Use**: Install via *pip* and start querying without the need for external databases.
- **Seamless Visualization**: Integrated with PyGraphistry for GPU-accelerated graph visualization.
- **Flexibility**: Suitable for a wide range of applications, including cybersecurity, fraud detection, financial analysis, and more.

#### 3.5.2.3 Installation Guide

GFQL is built into pygraphistry:

```
pip install graphistry
```

Ensure you have *pandas* or *cudf* installed, depending on whether you want to run on CPU or GPU.

For more information, see *Install* .

#### 3.5.2.4 Key GFQL Concepts

GFQL works on the same graphs as the rest of the PyGraphistry library. The operations run on top of the dataframe engine of your choice, with initial support for Pandas dataframes (CPU) and cuDF dataframes (GPU).

- **Nodes and Edges**: Represented using dataframes, making integration with Pandas and cuDF seamless
- **Functional**: Build queries by layering operations, similar to functional method chaining in Pandas
- **Query**: Run graph pattern matching using method *chain()* in a style similar to the popoular OpenCypher graph query language
- **Predicates**: Apply conditions to filter nodes and edges based on their properties, reusing the optimized native operations of the underlying dataframe engine
- **GPU & CPU vectorization**: GFQL automatically leverages GPU acceleration and in-memory columnar processing for massive speedups on your queries

#### 3.5.2.5 Quick Examples

**Find Nodes of a Certain Type**

Example: Find all nodes where the *type* is *"person"*.

```
from graphistry import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
print('Number of person nodes:', len(people_nodes_df))
```

**Visualize 2-Hop Edge Sequences with an Attribute**

Example: Find 2-hop paths where edges have *"interesting": True*.

```
from graphistry import e_forward

g_2_hops = g.chain([n(), e_forward({"interesting": True}, hops=2) ])
g_2_hops.plot()
```

**Find Nodes 1-2 Hops Away and Label Each Hop**

Example: Find nodes up to 2 hops away from node *"a"* and label each hop.

```
from graphistry import n, e_undirected

g_2_hops = g.chain([
    n({g._node: "a"}),
    e_undirected(name="hop1"),
    e_undirected(name="hop2")
])
first_hop_edges = g_2_hops._edges[ g_2_hops._edges.hop1 == True ]
print('Number of first-hop edges:', len(first_hop_edges))
```

**Query for Transaction Nodes Between Risky Nodes**

Example: Find transaction nodes between two kinds of risky nodes.

```
from graphistry import n, e_forward, e_reverse

g_risky = g.chain([
    n({"risk1": True}),
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
hits = g_risky._nodes[ g_risky._nodes.hit == True ]
print('Number of transaction hits:', len(hits))
```

**Filter by Multiple Node Types Using `is_in`**

Example: Filter nodes and edges by multiple types.

```
from graphistry import n, e_forward, e_reverse, is_in

g_filtered = g.chain([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
hits = g_filtered._nodes[ g_filtered._nodes.hit == True ]
print('Number of filtered hits:', len(hits))
```

#### 3.5.2.6 Leveraging GPU Acceleration

GFQL is optimized to take advantage of GPU acceleration using *cudf* and RAPIDS. When you use GPU dataframes, GFQL automatically executes queries on the GPU for massive speedups.

**Automatic GPU Acceleration**

Example: Run GFQL queries with GPU dataframes.

```
import cudf
import graphistry

# Load data into GPU dataframes
e_gdf = cudf.read_parquet('edges.parquet')
n_gdf = cudf.read_parquet('nodes.parquet')

# Create a graph with GPU dataframes
g_gpu = graphistry.edges(e_gdf, 'src', 'dst').nodes(n_gdf, 'id')

# Run GFQL query (executes on GPU)
g_result = g_gpu.chain([ ... ])  # Your GFQL query here
print('Number of resulting edges:', len(g_result._edges))
```

**Forcing GPU Mode**

Example: Explicitly set the engine to ensure GPU execution.

```
g_result = g_gpu.chain([ ... ], engine='cudf')
```

#### 3.5.2.7 Visualizing GFQL Results

GFQL integrates with PyGraphistry, allowing you to visualize your graphs with GPU-accelerated rendering.

Example: Visualize high PageRank nodes.

```
# Compute PageRank using cuGraph (GPU)
g_enriched = g_result.compute_cugraph('pagerank')

# Filter nodes with high PageRank
g_high_pagerank = g_enriched.chain([
    n(query='pagerank > 0.1'), e(), n(query='pagerank > 0.1')
])

# Plot the subgraph
g_high_pagerank.plot()
```

#### Learn More

Explore the following sections to dive deeper into GFQL's capabilities:

- **10 Minutes to GFQL**: A quickstart guide to get you up and running.
  - *10 Minutes to GFQL*
- **Hop & Chain Quick Reference**: Learn how to chain multiple operations to build complex queries.
  - *GFQL Quick Reference*
- **Predicates Quick Reference**: Apply advanced filtering using predicates.
  - *GFQL Operator Reference*

#### 3.5.2.8 GFQL APIs

Access detailed documentation of GFQL's API:

- **Chain Operations**: Learn how to chain multiple operations to build complex queries.
  - *GFQL Chain*
- **Hop Functions**: Understand how to traverse the graph using hop functions.
  - *GFQL Hop*
- **Predicates**: Apply advanced filtering using predicates.
  - *GFQL Predicates*

### 3.5.3 Translating Between SQL, Pandas, Cypher, and GFQL

This guide provides a comparison between **SQL**, **Pandas**, **Cypher**, and **GFQL**, helping you translate familiar queries into GFQL.

#### 3.5.3.1 Introduction

GFQL (GraphFrame Query Language) is designed to be intuitive for users familiar with SQL, Pandas, or Cypher. By comparing equivalent queries across these languages, you can quickly grasp GFQL's syntax and start utilizing its powerful graph querying capabilities within your Python workflows.

#### 3.5.3.2 Who Is This Guide For?

- **Data Scientists and Analysts**: Familiar with Pandas or SQL, looking to explore graph relationships in their data.
- **Graph Engineers and Developers**: Experienced with Cypher or other graph query languages, interested in integrating graph queries into Python applications.
- **Database Administrators**: Looking to understand how GFQL can complement traditional SQL queries for graph data.
- **Anyone Exploring Graph Analytics**: Interested in learning how to perform graph queries using a dataframe-native approach.

#### 3.5.3.3 Common Graph and Query Tasks

We'll cover a range of common graph and query tasks:

1. **Finding Nodes with Specific Properties**
2. **Exploring Relationships Between Nodes**
3. **Performing Multi-Hop Traversals**
4. **Filtering Edges and Nodes with Conditions**
5. **Aggregations and Grouping**
6. **All Paths and Connectivity**
7. **Community Detection and Clustering**

#### 3.5.3.4 Translation Examples

##### Example 1: Finding Nodes with Specific Properties

**Objective**: Find all nodes where the *type* is *"person"*.

**SQL**

```
SELECT * FROM nodes
WHERE type = 'person';
```

**Pandas**

```
people_nodes_df = nodes_df[ nodes_df['type'] == 'person' ]
```

**Cypher**

```
MATCH (n {type: 'person'})
RETURN n;
```

**GFQL**

```
from graphistry import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
```

**Explanation**:

- **GFQL**: *n({"type": "person"})* filters nodes where *type* is *"person"*. *g.chain([...])* applies this filter to the graph *g*, and *._nodes* retrieves the resulting nodes. The performance is similar to that of Pandas (CPU) or cuDF (GPU).

—

### Example 2: Exploring Relationships Between Nodes

**Objective**: Find all edges connecting nodes of type *"person"* to nodes of type *"company"*.

**SQL**

```
SELECT e.*
FROM edges e
JOIN nodes n1 ON e.src = n1.id
JOIN nodes n2 ON e.dst = n2.id
WHERE n1.type = 'person' AND n2.type = 'company';
```

**Pandas**

```
merged_df = edges_df.merge(
    nodes_df[['id', 'type']], left_on='src', right_on='id', suffixes=('', '_src')
).merge(
    nodes_df[['id', 'type']], left_on='dst', right_on='id', suffixes=('', '_dst')
)

result = merged_df[
    (merged_df['type_src'] == 'person') &
    (merged_df['type_dst'] == 'company')
]
```

**Cypher**

```
MATCH (n1 {type: 'person'})-[e]->(n2 {type: 'company'})
RETURN e;
```

**GFQL**

```
from graphistry import n, e_forward

g_result = g.chain([
    n({"type": "person"}),
    e_forward(),
    n({"type": "company"})
])
edges_df = g_result._edges
```

**Explanation**:

- **GFQL**: Starts from nodes of type *"person"*, traverses forward edges, and reaches nodes of type *"company"*. The resulting edges are stored in *edges_df*.

### Example 3: Performing Multi-Hop Traversals

**Objective**: Find nodes that are two hops away from node *"Alice"*.

**SQL**

```
WITH first_hop AS (
    SELECT e1.dst AS node_id
    FROM edges e1
    WHERE e1.src = 'Alice'
),
second_hop AS (
    SELECT e2.dst AS node_id
    FROM edges e2
    JOIN first_hop fh ON e2.src = fh.node_id
)
SELECT * FROM nodes
WHERE id IN (SELECT node_id FROM second_hop);
```

**Pandas**

```
first_hop = edges_df[ edges_df['src'] == 'Alice' ]['dst']
second_hop = edges_df[ edges_df['src'].isin(first_hop) ]['dst']
result_nodes_df = nodes_df[ nodes_df['id'].isin(second_hop) ]
```

**Cypher**

```
MATCH (n {id: 'Alice'})-->()-->(m)
RETURN m;
```

**GFQL**

```
from graphistry import n, e_forward

g_2_hops = g.chain([
    n({g._node: "Alice"}),
    e_forward(),
    e_forward()
])
nodes_df = g_2_hops._nodes
```

**Explanation**:

- **GFQL**: Starts at node *"Alice"*, performs two forward hops, and obtains nodes two steps away. Results are in *nodes_df*.

### Example 4: Filtering Edges and Nodes with Conditions

**Objective**: Find all edges where the weight is greater than *0.5*.

**SQL**

```
SELECT * FROM edges
WHERE weight > 0.5;
```

**Pandas**

```
filtered_edges_df = edges_df[ edges_df['weight'] > 0.5 ]
```

**Cypher**

```
MATCH ()-[e]->()
WHERE e.weight > 0.5
RETURN e;
```

**GFQL**

```
from graphistry import e_forward

filtered_edges_df = g.chain([ e_forward(edge_query='weight > 0.5') ])._edges
```

**Explanation**:

- **GFQL**: Uses *e_forward({"weight": lambda w: w > 0.5})* to filter edges where *weight > 0.5*.

---

### Example 5: Aggregations and Grouping

**Objective**: Count the number of outgoing edges for each node.

**SQL**

```
SELECT src, COUNT(*) AS out_degree
FROM edges
GROUP BY src;
```

**Pandas**

```
out_degree = edges_df.groupby('src').size().reset_index(name='out_degree')
```

**Cypher**

```
MATCH (n)-[e]->()
RETURN n.id AS node_id, COUNT(e) AS out_degree;
```

**GFQL**

```
out_degree = g._edges.groupby('src').size().reset_index(name='out_degree')
```

**Explanation**:

- **GFQL**: Performs aggregation directly on *g._edges* using standard dataframe operations. Or even shorter, call *g.get_degrees()* to enrich each node with in, out, and total degrees.

—

### Example 6: All Paths and Connectivity

**Objective**: Find all paths between nodes `"Alice"` and `"Bob"`.

**SQL and Pandas**

- Not suitable for path calculations in graphs.

**Cypher**

```
MATCH p = (n {id: 'Alice'})-[*]-(m {id: 'Bob'})
RETURN p;
```

**GFQL**

```
g.chain([ n({g._node: "Alice"}), e(to_fixed_point=True), n({g._node: "Bob"}) ])
```

**Explanation**:

- **GFQL**: Uses *e(to_fixed_point=True)* to find edge sequences of arbitrary length between nodes *"Alice"* and *"Bob"*.

—

### Example 7: Community Detection and Clustering

**Objective**: Identify communities within the graph using the Louvain algorithm.

**SQL and Pandas**

- Not designed for complex graph algorithms like community detection.

**Cypher**

```
CALL algo.louvain.stream() YIELD nodeId, communityId
```

**GFQL**

```
nodes_df = g.compute_cugraph('louvain')._nodes[['id', 'louvain']]
```

**Explanation**:

- **GFQL**: Uses GPU-accelerated algorithms via *compute_cugraph()* for community detection.

—

#### 3.5.3.5 GFQL Functions and Equivalents

**Node Matching**

- **SQL**: `SELECT * FROM nodes WHERE ...`
- **Pandas**: `nodes_df[ condition ]`
- **Cypher**: `MATCH (n {property: value})`
- **GFQL**: `n({ "property": value })`

**Edge Matching**

- **SQL**: `SELECT * FROM edges WHERE ...`
- **Pandas**: `edges_df[ condition ]`
- **Cypher**: `MATCH ()-[e {property: value}]->()`
- **GFQL**: `e_forward({ "property": value })` or `e_reverse({ "property": value })` or `e({ "property": value })`

**Traversal**

- **SQL**: Complex joins or recursive queries
- **Pandas**: Multiple merges; not efficient for deep traversals
- **Cypher**: Patterns like `()-[]->()` for traversal
- **GFQL**: Chains of `n()`, `e_forward()`, `e_reverse()`, and `e()` functions

#### 3.5.3.6 Tips for Users

- **Data Scientists and Analysts**: Use your Pandas knowledge. GFQL operates on dataframes, allowing familiar operations.
- **Engineers and Developers**: Integrate GFQL into Python applications without extra infrastructure.
- **Database Administrators**: Complement SQL queries with GFQL for graph data without changing databases.
- **Graph Enthusiasts**: Start with simple queries and explore complex analytics. Visualize results using PyGraphistry.

#### 3.5.3.7 Additional Resources

- **GFQL Documentation**: Detailed documentation for advanced usage.
- **GFQL Predicates**: Use predicates for complex filtering conditions.
- **PyGraphistry Integration**: Visualize GFQL queries with GPU-accelerated tools.

#### 3.5.3.8 Conclusion

GFQL bridges the gap between traditional querying languages and graph analytics. By translating queries from SQL, Pandas, and Cypher into GFQL, you can leverage powerful graph queries within your Python workflows.

Start exploring GFQL today and unlock new insights from your graph data!

### 3.5.4 GFQL Quick Reference

This quick reference page provides short examples of various parameters and usage patterns.

#### 3.5.4.1 Basic Usage

**Chaining Operations**

```
g.chain([...], engine=EngineAbstract.AUTO)
```

- **`ops`**: Sequence of graph node and edge matchers (*ASTObject* instances).
- **`engine`**: Optional execution engine (*'cudf'* for GPU acceleration).

#### 3.5.4.2 Node Matchers

```
n(filter_dict=None, name=None, query=None)
```

- Filter nodes based on attributes.
- **Parameters**: - *filter_dict*: *{attribute: value}* or *{attribute: condition_function}* - *name*: Optional label; adds a boolean column in the result. - *query*: Custom query string (e.g., *"age > 30 and country == 'USA'"*).

**Examples:**

- Match nodes where *type* is *'person'*:

  ```
  n({"type": "person"})
  ```

- Match nodes with *age* greater than 30:

  ```
  n({"age": lambda x: x > 30})
  ```

- Use a custom query string:

  ```
  n(query="age > 30 and country == 'USA'")
  ```

#### 3.5.4.3 Edge Matchers

```
e_forward(edge_match=None, hops=1, to_fixed_point=False, source_node_match=None,␣
↪destination_node_match=None, source_node_query=None, destination_node_query=None, edge_
↪query=None, name=None)
e_reverse(edge_match=None, hops=1, to_fixed_point=False, source_node_match=None,␣
↪destination_node_match=None, source_node_query=None, destination_node_query=None, edge_
↪query=None, name=None)
e_undirected(edge_match=None, hops=1, to_fixed_point=False, source_node_match=None,␣
↪destination_node_match=None, source_node_query=None, destination_node_query=None, edge_
↪query=None, name=None)

# alias for e_undirected
e(edge_match=None, hops=1, to_fixed_point=False, source_node_match=None, destination_
↪node_match=None, source_node_query=None, destination_node_query=None, edge_query=None,␣
↪name=None)
```

- Traverse edges in the forward direction.
- **Parameters**: - *edge_match*: *{attribute: value}* or *{attribute: condition_function}* - *edge_query*: Custom query string for edge attributes. - *hops*: *int*, number of hops to traverse. - *to_fixed_point*: *bool*, continue traversal until no more matches. - *source_node_match*: Filter for source nodes. - *destination_node_match*: Filter for destination nodes. - *source_node_query*: Custom query string for source nodes. - *destination_node_query*: Custom query string for destination nodes. - *name*: Optional label.

**Examples:**

- Traverse 2 hops forward on edges where *status* is *'active'*:

```
e_forward({"status": "active"}, hops=2)
```

- Use custom edge query strings:

```
e_forward(edge_query="weight > 5 and type == 'connects'")
```

- Filter source and destination nodes with match dictionaries:

```
e_forward(
    source_node_match={"status": "active"},
    destination_node_match={"age": lambda x: x < 30}
)
```

- Filter source and destination nodes with queries:

```
e_forward(
    source_node_query="status == 'active'",
    destination_node_query="age < 30"
)
```

- Label matched edges:

```
e_forward(name="active_edges")
```

**`e_reverse(...)` and `e_undirected(...)`**

- **`e_reverse`**: Same as *e_forward*, but traverses in reverse.

- **`e_undirected`**: Traverses edges regardless of direction.

#### 3.5.4.4 Predicates

**`predicate(values)`**

- Matches using a predicate on entity attributes.

See *GFQL Operator Reference* for more information.

**Example:**

- Match nodes where *category* is *'A'*, *'B'*, or *'C'*:

```
from graphistry import n, is_in

n({"category": is_in(["A", "B", "C"])})
```

#### 3.5.4.5 Combined Examples

- **Find people connected to transactions via active relationships:**

```
g.chain([
    n({"type": "person"}),
    e_forward({"status": "active"}),
    n({"type": "transaction"})
])
```

- **Label nodes and edges during traversal:**

```
g.chain([
    n({"id": "start_node"}, name="start"),
    e_forward(name="edge1"),
    n({"level": 2}, name="middle"),
    e_forward(name="edge2"),
    n({"type": "end_type"}, name="end")
])
```

- **Traverse until no more matches (fixed point):**

```
g.chain([
    n({"status": "infected"}),
    e_forward(to_fixed_point=True),
    n(name="reachable")
])
```

- **Filter by multiple conditions:**

```
g.chain([
    n({"type": is_in(["server", "database"])}),
    e_undirected({"protocol": "TCP"}, hops=3),
    n(query="risk_level >= 8")
])
```

- **Use custom queries in matchers:**

```
g.chain([
    n(query="age > 30 and country == 'USA'"),
    e_forward(edge_query="weight > 5"),
    n(query="status == 'active'")
])
```

#### 3.5.4.6 GPU Acceleration

- **Enable GPU mode:**

```
g.chain([...], engine='cudf')
```

- **Example with cuDF DataFrames:**

```
import cudf

e_gdf = cudf.from_pandas(edge_df)
n_gdf = cudf.from_pandas(node_df)

g = graphistry.nodes(n_gdf, 'node_id').edges(e_gdf, 'src', 'dst')
g.chain([...], engine='cudf')
```

#### 3.5.4.7 Advanced Usage

- **Traversal with source and destination node filters and queries:**

```
e_forward(
    edge_query="type == 'follows' and weight > 2",
    source_node_match={"status": "active"},
    destination_node_query="age < 30",
    hops=2,
    name="social_edges"
)
```

- **Node matcher with all parameters:**

```
n(
    filter_dict={"department": "sales"},
    query="age > 25 and tenure > 2",
    name="experienced_sales"
)
```

- **Edge matcher with all parameters:**

```
e_reverse(
    edge_match={"transaction_type": "refund"},
    edge_query="amount > 100",
    source_node_match={"status": "inactive"},
    destination_node_match={"region": "EMEA"},
    name="large_refunds"
)
```

#### 3.5.4.8 Parameter Summary

- **Common Parameters:** - *filter_dict*: Attribute filters (e.g., *{"status": "active"}*) - *query*: Custom query string (e.g., *"age > 30"*) - *hops*: Number of steps to traverse (*int*, default *1*) - *to_fixed_point*: Continue traversal until no more matches (*bool*, default *False*) - *name*: Label for matchers (*str*) - *source_node_match*, *destination_node_match*: Filters for connected nodes - *source_node_query*, *destination_node_query*: Queries for connected nodes - *edge_match*: Filters for edges - *edge_query*: Query for edges - *engine*: Execution engine (*EngineAbstract.AUTO*, *'cudf'*, etc.)

#### 3.5.4.9 Traversal Directions

- **Forward Traversal:** *e_forward(...)*
- **Reverse Traversal:** *e_reverse(...)*
- **Undirected Traversal:** *e_undirected(...)*

#### 3.5.4.10 Tips and Best Practices

- **Limit hops for performance:** Specify *hops* to control traversal depth.
- **Use naming for analysis:** Apply *name* to label and filter results.
- **Combine filters:** Use *filter_dict* and *query* for precise matching.
- **Leverage GPU acceleration:** Use *engine='cudf'* for large datasets.
- **Avoid infinite loops:** Be cautious with *to_fixed_point=True* in cyclic graphs.

#### 3.5.4.11 Examples at a Glance

- **Find all paths between two nodes:**

```
g.chain([
    n({g._node: "Alice"}),
    e_undirected(hops=3),
    n({g._node: "Bob"})
])
```

- **Match nodes with IDs in a range:**

```
n(query="100 <= id <= 200")
```

- **Traverse edges with specific labels:**

```
e_forward({"label": is_in(["knows", "likes"])})
```

- **Identify subgraphs based on attributes:**

```
g.chain([
    n({"community": "A"}),
    e_undirected(hops=2),
    n({"community": "B"}, name="bridge_nodes")
])
```

- **Custom edge and node queries:**

```
g.chain([
    n(query="age >= 18"),
    e_forward(edge_query="interaction == 'message'"),
    n(query="location == 'NYC'")
])
```

### 3.5.5 GFQL Operator Reference

This reference overviews the operators available in GFQL for constructing predicates in your graph queries. These operators are wrappers around Pandas/cuDF functions, allowing you to express complex filtering conditions intuitively. See the API reference documentation for more details on individual operators.

#### 3.5.5.1 Operators

The following table lists the available operators, their descriptions, and examples of how to use them in GFQL.

**Numeric and Comparison Operators**

| Operator | Description | Example |
|---|---|---|
| `gt(value)` | Greater than `value`. | `n({ "age": gt(18) })` |
| `lt(value)` | Less than `value`. | `n({ "age": lt(65) })` |
| `ge(value)` | Greater than or equal to `value`. | `n({ "score": ge(90) })` |
| `le(value)` | Less than or equal to `value`. | `n({ "score": le(70) })` |
| `eq(value)` | Equal to `value`. | `n({ "status": eq("active") })` |
| `ne(value)` | Not equal to `value`. | `n({ "status": ne("inactive") })` |
| `between(lower, upper)` | Between `lower` and `upper` (inclusive). | `n({ "age": between(18, 65) })` |

**Categorical Operators**

| Operator | Description | Example |
|---|---|---|
| `is_in(values)` | Value is in `values` list. | `n({ "type": is_in(["person", "company"]) })` |
| `is_not_in(values)` | Value is not in `values` list. | `n({ "type": is_not_in(["bot", "spam"]) })` |
| `duplicated(keep='first'` | Marks duplicated values. | `n({ "email": duplicated() })` |

**String Operators**

| Operator | Description | Example |
|---|---|---|
| `contains(pattern)` | String contains `pattern`. | `n({ "name": contains("Smith") })` |
| `startswith(prefix)` | String starts with `prefix`. | `n({ "username": startswith("admin") })` |
| `endswith(suffix)` | String ends with `suffix`. | `n({ "email": endswith("@example.com") })` |
| `match(pattern)` | String matches regex `pattern`. | `n({ "phone": match(r"^\d{3}-\d{4}$") })` |
| `isnumeric()` | String is numeric. | `n({ "code": isnumeric() })` |
| `isalpha()` | String is alphabetic. | `n({ "code": isalpha() })` |
| `isdigit()` | String is digit characters. | `n({ "code": isdigit() })` |
| `islower()` | String is lowercase. | `n({ "tag": islower() })` |
| `isupper()` | String is uppercase. | `n({ "code": isupper() })` |
| `isspace()` | String contains only whitespace. | `n({ "comment": isspace() })` |
| `isalnum()` | String is alphanumeric. | `n({ "code": isalnum() })` |
| `isdecimal()` | String is decimal characters. | `n({ "number": isdecimal() })` |
| `istitle()` | String is title-cased. | `n({ "title": istitle() })` |

**Null and NA Operators**

| Operator | Description | Example |
|---|---|---|
| `isna()` | Value is NA/NaN. | `n({ "email": isna() })` |
| `notna()` | Value is not NA/NaN. | `n({ "email": notna() })` |
| `isnull()` | Alias for `isna()`. | `n({ "email": isnull() })` |
| `notnull()` | Alias for `notna()`. | `n({ "email": notnull() })` |

**Temporal Operators**

| Operator | Description | Example |
|---|---|---|
| `is_month_start()` | Date is the first day of the month. | `n({ "date": is_month_start() })` |
| `is_month_end()` | Date is the last day of the month. | `n({ "date": is_month_end() })` |
| `is_quarter_start()` | Date is the first day of the quarter. | `n({ "date": is_quarter_start() })` |
| `is_quarter_end()` | Date is the last day of the quarter. | `n({ "date": is_quarter_end() })` |
| `is_year_start()` | Date is the first day of the year. | `n({ "date": is_year_start() })` |
| `is_year_end()` | Date is the last day of the year. | `n({ "date": is_year_end() })` |
| `is_leap_year()` | Date is in a leap year. | `n({ "date": is_leap_year() })` |

#### 3.5.5.2 Usage Examples

**Example 1: Filtering Nodes with Numeric Conditions**

```
from graphistry.ast import n, gt, lt

# Find nodes where age is greater than 18 and less than 30
g_filtered = g.chain([
    n({ "age": gt(18) }),
    n({ "age": lt(30) })
])
```

**Example 2: Filtering Nodes by Category**

```
from graphistry.ast import n, is_in

# Find nodes of type 'person' or 'company'
g_filtered = g.chain([
    n({ "type": is_in(["person", "company"]) })
])
```

**Example 3: Filtering Edges with String Conditions**

```
from graphistry.ast import e_forward, contains

# Find edges where the relation contains 'friend'
g_filtered = g.chain([
    e_forward({ "relation": contains("friend") })
])
```

**Example 4: Combining Multiple Predicates**

```
from graphistry.ast import n, is_in, gt

# Find 'person' nodes with age greater than 18
g_filtered = g.chain([
    n({
        "type": eq("person"),
        "age": gt(18)
    })
])
```

#### 3.5.5.3 Additional Notes

- **Lambda Functions**: You can use lambda functions for custom conditions.

  ```
  n({ "score": lambda x: (x > 50) & (x % 2 == 0) })
  ```

- **Importing Operators**: Remember to import the necessary functions.

  ```
  from graphistry.ast import n, e_forward, gt, contains
  ```

- **Combining Conditions**: Use logical operators within lambdas for complex expressions.

  ```
  n({ "age": lambda x: (x > 18) & (x < 65) })
  ```

- **Predicates Module**: Operators are available in the *graphistry.predicates* module.

## 3.6 Plugins

PyGraphistry is frequently used with a variety of external tools such as data providers, compute engines, layout engines, and more.

Users typically prefer to go through PyGraphistry's native dataframe support (Apache Arrow, Pandas, cuDF, ...). That is often an efficient, safe, and easy starting point.

Occasionally, native PyGraphistry plugins streamline common operations, such as with graph databases. We link to the native API integrations below as appropriate.

For more examples, see also the *notebook catalog*.

### 3.6.1 Databases

#### 3.6.1.1 Graph

- Amazon Neptune (*graphistry.gremlin.NeptuneMixin*)
- ArangoDB
- Gremlin (*graphistry.gremlin.GremlinMixin*)
- Memgraph (*graphistry.PlotterBase.PlotterBase.cypher()*)
- Neo4j (*graphistry.PlotterBase.PlotterBase.cypher()*)
- TigerGraph (*graphistry.PlotterBase.PlotterBase.gsql()*)
- Trovares

#### 3.6.1.2 Document, Key-Value, Log, Text, and SIEM

- Amazon DynamoDB
- Azure Cosmos DB (*graphistry.gremlin.CosmosMixin*)
- Azure Data Explorer (Kusto)
- Cassandra
- Elasticsearch
- OpenSearch
- Redis
- Splunk

#### 3.6.1.3 SQL

Typically accessed via dataframe bindings

When available, we recommend exploring for accelerated bindings via ADBC

- Amazon Athena
- Databricks
- OpenSearch

- PostgreSQL
- Amazon Redshift
- BigQuery
- Snowflake
- SQL Server

### 3.6.2 Compute engines

Natively supported in methods such as `.nodes()` and `.edges()`:

- Apache Spark
- Pandas
- cuDF

Partial native support:

- cuML
- Dask
- Dask-cuDF

Accelerated interop via Apache Arrow or Parquet:

- DuckDB
- Polars
- Spark

### 3.6.3 Graph layout and analytics

- *cugraph*: GPU-accelerated graph analytics
- *graphviz*: CPU graph analytics and layouts
- *igraph*: CPU graph analytics and layouts
- *networkx*: CPU graph analytics and layouts

### 3.6.4 Tools

We are constantly experimenting, feel free to add:

- OWASP Amass

### 3.6.5 Storage engines and file formats

GPU-accelerated readers via cuDF (in-memory single-GPU) and Dask-cuDF (bigger-than-memory, multi-GPU):

- Arrow
- CSV
- JSON
- JSONL
- LOG
- ORC
- Parquet
- TXT

Others, often via fsspec:

- Azure blobstore
- GML
- S3
- XLS(X)

## 3.7 Notebook Tutorials

### 3.7.1 Getting Started

### 3.7.2 Visualization

#### 3.7.2.1 Encodings

#### 3.7.2.2 Layout

#### 3.7.2.3 Accounts and Sharing

### 3.7.3 GFQL Graph queries

### 3.7.4 GPU

### 3.7.5 AI

### 3.7.6 Plugins - Data Providers

### 3.7.7 Plugins - Compute & Layout

## 3.8 Python API Reference

See the article on *10 Minutes to PyGraphistry* for a high-level overview of binding data and plotting it.

### 3.8.1 Plotter API Reference

The below Python API reference documentation is for three views of the core graph abstraction, *Plottable*:

- The *graphistry.plotter.Plotter* class that mixes in all layers such as plugins
- The *graphistry.Plottable* abstract interface for the core Graphistry graph object
- The *graphistry.PlotterBase* class implementing it

#### 3.8.1.1 Plotter Class

Main Plotter class for Graphistry.

This class represents a graph in Graphistry and serves as the primary interface for plotting and analyzing graphs. It inherits from multiple mixins, allowing it to extend its functionality with additional graph computation, layouts, conditional formatting, and more.

**Inherits:**

- *graphistry.PlotterBase.PlotterBase*: Base class for plotting graphs.
- `graphistry.compute.ComputeMixin`: Enables computation-related functions like degree calculations.
- `graphistry.layouts.LayoutsMixin`: Provides methods for controlling graph layouts.
- `graphistry.compute.conditional.ConditionalMixin`: Adds support for conditional graph operations.
- `graphistry.feature_utils.FeatureMixin`: Adds feature engineering capabilities.
- `graphistry.umap_utils.UMAPMixin`: Integrates UMAP for dimensionality reduction.
- `graphistry.compute.cluster.ClusterMixin`: Enables clustering-related functionalities.
- `graphistry.dgl_utils.DGLGraphMixin`: Integrates deep graph learning with DGL.
- `graphistry.text_utils.SearchToGraphMixin`: Supports converting search results into graphs.
- `graphistry.embed_utils.HeterographEmbedModuleMixin`: Adds heterograph embedding capabilities.
- *graphistry.gremlin.GremlinMixin*: Provides Gremlin query support for graph databases.
- *graphistry.gremlin.CosmosMixin*: Integrates with Azure Cosmos DB.
- *graphistry.gremlin.NeptuneMixin*: Integrates with AWS Neptune DB.

**Attributes:**

All attributes are inherited from the mixins and base classes.

#### 3.8.1.2 Plottable Interface

`class graphistry.Plottable.Plottable(*args, **kwargs)`

Bases: `object`

`DGL_graph: Any | None`

**bind**(*source=None*, *destination=None*, *node=None*, *edge=None*, *edge_title=None*, *edge_label=None*, *edge_color=None*, *edge_weight=None*, *edge_size=None*, *edge_opacity=None*, *edge_icon=None*, *edge_source_color=None*, *edge_destination_color=None*, *point_title=None*, *point_label=None*, *point_color=None*, *point_weight=None*, *point_size=None*, *point_opacity=None*, *point_icon=None*, *point_x=None*, *point_y=None*)

**chain**(*ops*)

ops is List[ASTObject]

**Parameters**
**ops** (*List[Any]*)

**Return type**
Plottable

**collapse**(*node*, *attribute*, *column*, *self_edges=False*, *unwrap=False*, *verbose=False*)

**Parameters**

- **node** (*str | int*)
- **attribute** (*str | int*)
- **column** (*str | int*)
- **self_edges** (*bool*)
- **unwrap** (*bool*)
- **verbose** (*bool*)

**Return type**
Plottable

**compute_cugraph**(*alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

**Parameters**

- **alg** (*str*)
- **out_col** (*str | None*)
- **params** (*dict*)
- **kind** (*Literal['Graph', 'MultiGraph', 'BiPartiteGraph']*)
- **G** (*Any | None*)

**copy**()

**drop_nodes**(*nodes*)

**Parameters**
**nodes** (*Any*)

**Return type**
Plottable

**edges**(*edges*, *source=None*, *destination=None*, *edge=None*, **args*, ***kwargs*)

**Parameters**

- **edges** (*Callable | Any*)
- **source** (*str | None*)

- **destination** (*str / None*)
- **edge** (*str / None*)

**Return type**
Plottable

**encode_axis**(*rows=[]*)

**Return type**
Plottable

**filter_edges_by_dict**(*filter_dict=None*)

**Parameters**
**filter_dict** (*dict / None*)

**Return type**
Plottable

**filter_nodes_by_dict**(*filter_dict=None*)

**Parameters**
**filter_dict** (*dict / None*)

**Return type**
Plottable

**from_cugraph**(*G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

**Parameters**

- **node_attributes** (*List [str] / None*)
- **edge_attributes** (*List [str] / None*)
- **load_nodes** (*bool*)
- **load_edges** (*bool*)
- **merge_if_existing** (*bool*)

**from_igraph**(*ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

**Parameters**

- **node_attributes** (*List [str] / None*)
- **edge_attributes** (*List [str] / None*)
- **load_nodes** (*bool*)
- **load_edges** (*bool*)
- **merge_if_existing** (*bool*)

**get_degrees**(*col='degree'*, *degree_in='degree_in'*, *degree_out='degree_out'*)

**Parameters**

- **col** (*str*)
- **degree_in** (*str*)
- **degree_out** (*str*)

**Return type**
Plottable

`get_indegrees`(*col='degree_in'*)

**Parameters**
`col` (*str*)

**Return type**
Plottable

`get_outdegrees`(*col='degree_out'*)

**Parameters**
`col` (*str*)

**Return type**
Plottable

`get_topological_levels`(*level_col='level'*, *allow_cycles=True*, *warn_cycles=True*, *remove_self_loops=True*)

**Parameters**

- `level_col` (*str*)
- `allow_cycles` (*bool*)
- `warn_cycles` (*bool*)
- `remove_self_loops` (*bool*)

**Return type**
Plottable

`hop`(*nodes*, *hops=1*, *to_fixed_point=False*, *direction='forward'*, *edge_match=None*, *source_node_match=None*, *destination_node_match=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *return_as_wave_front=False*, *target_wave_front=None*)

**Parameters**

- `nodes` (*DataFrame | None*)
- `hops` (*int | None*)
- `to_fixed_point` (*bool*)
- `direction` (*str*)
- `edge_match` (*dict | None*)
- `source_node_match` (*dict | None*)
- `destination_node_match` (*dict | None*)
- `source_node_query` (*str | None*)
- `destination_node_query` (*str | None*)
- `edge_query` (*str | None*)
- `return_as_wave_front` (*bool*)
- `target_wave_front` (*DataFrame | None*)

**Return type**
Plottable

`keep_nodes`(*nodes*)

**Parameters**
**nodes** (*List | Any*)

**Return type**
Plottable

`layout_cugraph`(*layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

**Parameters**

- **layout** (*str*)
- **params** (*dict*)
- **kind** (*Literal['Graph', 'MultiGraph', 'BiPartiteGraph']*)
- **G** (*Any | None*)
- **bind_position** (*bool*)
- **x_out_col** (*str*)
- **y_out_col** (*str*)
- **play** (*int | None*)

`layout_settings`(*play=None*, *locked_x=None*, *locked_y=None*, *locked_r=None*, *left=None*, *top=None*, *right=None*, *bottom=None*, *lin_log=None*, *strong_gravity=None*, *dissuade_hubs=None*, *edge_influence=None*, *precision_vs_speed=None*, *gravity=None*, *scaling_ratio=None*)

**Parameters**

- **play** (*int | None*)
- **locked_x** (*bool | None*)
- **locked_y** (*bool | None*)
- **locked_r** (*bool | None*)
- **left** (*float | None*)
- **top** (*float | None*)
- **right** (*float | None*)
- **bottom** (*float | None*)
- **lin_log** (*bool | None*)
- **strong_gravity** (*bool | None*)
- **dissuade_hubs** (*bool | None*)
- **edge_influence** (*float | None*)
- **precision_vs_speed** (*float | None*)
- **gravity** (*float | None*)
- **scaling_ratio** (*float | None*)

**materialize_nodes**(*reuse=True*, *engine=EngineAbstract.AUTO*)

**Parameters**

- **reuse** (*bool*)
- **engine** (*EngineAbstract | str*)

**Return type**
Plottable

**nodes**(*nodes*, *node=None*, **args*, ***kwargs*)

**Parameters**

- **nodes** (*Callable | Any*)
- **node** (*str | None*)

**Return type**
Plottable

**pipe**(*graph_transform*, **args*, ***kwargs*)

**Parameters**
**graph_transform** (*Callable*)

**Return type**
Plottable

**prune_self_edges**()

**Return type**
Plottable

**settings**(*height=None*, *url_params={}*, *render=None*)

Specify iframe height and add URL parameter dictionary.

The library takes care of URI component encoding for the dictionary.

**Parameters**

- **height** (*int*) – Height in pixels.
- **url_params** (*dict*) – Dictionary of querystring parameters to append to the URL.
- **render** (*bool*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL

**Return type**
Plottable

**to_cugraph**(*directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

**Parameters**

- **directed** (*bool*)
- **include_nodes** (*bool*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)
- **kind** (*Literal['Graph', 'MultiGraph', 'BiPartiteGraph']*)

**Return type**
*Any*

`to_igraph`(*directed=True*, *use_vids=False*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*)

**Parameters**

- **directed** (*bool*)
- **use_vids** (*bool*)
- **include_nodes** (*bool*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)

**Return type**
*Any*

#### 3.8.1.3 PlotterBase Class

`class graphistry.PlotterBase.PlotterBase`(**args*, ***kwargs*)

Bases: *Plottable*

Graph plotting class.

Created using `Graphistry.bind()`.

Chained calls successively add data and visual encodings, and end with a plot call.

To streamline reuse and replayable notebooks, Plotter manipulations are immutable. Each chained call returns a new instance that derives from the previous one. The old plotter or the new one can then be used to create different graphs.

When using memoization, for .register(api=3) sessions with .plot(memoize=True), Pandas/cudf arrow coercions are memoized, and file uploads are skipped on same-hash dataframes.

The class supports convenience methods for mixing calls across Pandas, NetworkX, and IGraph.

**Parameters**

- **args** (*Any*)
- **kwargs** (*Any*)

`DGL_graph: Any | None`

`addStyle`(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Set general visual styles

See .bind() and .settings(url_params={}) for additional styling options, and style() for another way to set the same attributes.

To facilitate reuse and replayable notebooks, the addStyle() call is chainable. Invocation does not effect the old style: it instead returns a new Plotter instance with the new styles added to the existing ones. Both the old and new styles can then be used for different graphs.

addStyle() will extend the existing style settings, while style() will replace any in the same group

**Parameters**

- **fg** (*dict*) – Dictionary {'blendMode': str} of any valid CSS blend mode

- **bg** (*dict*) – Nested dictionary of page background properties. {'color': str, 'gradient': {'kind': str, 'position': str, 'stops': list }, 'image': { 'url': str, 'width': int, 'height': int, 'blendMode': str }
- **logo** (*dict*) – Nested dictionary of logo properties. { 'url': str, 'autoInvert': bool, 'position': str, 'dimensions': { 'maxWidth': int, 'maxHeight': int }, 'crop': { 'top': int, 'left': int, 'bottom': int, 'right': int }, 'padding': { 'top': int, 'left': int, 'bottom': int, 'right': int}, 'style': str}
- **page** (*dict*) – Dictionary of page metadata settings. { 'favicon': str, 'title': str }

**Returns**
Plotter

**Return type**
Plotter

**Example: Chained merge - results in color, blendMode, and url being set**

```
g2 =  g.addStyle(bg={'color': 'black'}, fg={'blendMode': 'screen'})
g3 = g2.addStyle(bg={'image': {'url': 'http://site.com/watermark.png'}})
```

**Example: Overwrite - results in blendMode multiply**

```
g2 =  g.addStyle(fg={'blendMode': 'screen'})
g3 = g2.addStyle(fg={'blendMode': 'multiply'})
```

**Example: Gradient background**

```
g.addStyle(bg={'gradient': {'kind': 'linear', 'position': 45, 'stops': [[
↪'rgb(0,0,0)', '0%'], ['rgb(255,255,255)', '100%']]}})
```

**Example: Page settings**

```
g.addStyle(page={'title': 'Site - {{ name }}', 'favicon': 'http://site.com/
↪logo.ico'})
```

**bind**(*source=None*, *destination=None*, *node=None*, *edge=None*, *edge_title=None*, *edge_label=None*, *edge_color=None*, *edge_weight=None*, *edge_size=None*, *edge_opacity=None*, *edge_icon=None*, *edge_source_color=None*, *edge_destination_color=None*, *point_title=None*, *point_label=None*, *point_color=None*, *point_weight=None*, *point_size=None*, *point_opacity=None*, *point_icon=None*, *point_x=None*, *point_y=None*)

Relate data attributes to graph structure and visual representation. To facilitate reuse and replayable notebooks, the binding call is chainable. Invocation does not effect the old binding: it instead returns a new Plotter instance with the new bindings added to the existing ones. Both the old and new bindings can then be used for different graphs.

**Parameters**

- **source** (*str*) – Attribute containing an edge's source ID
- **destination** (*str*) – Attribute containing an edge's destination ID
- **node** (*str*) – Attribute containing a node's ID
- **edge** (*str*) – Attribute containing an edge's ID
- **edge_title** (*str*) – Attribute overriding edge's minimized label text. By default, the edge source and destination is used.

- **edge_label** (*str*) – Attribute overriding edge's expanded label text. By default, scrollable list of attribute/value mappings.
- **edge_color** (*str*) – Attribute overriding edge's color. rgba (int64) or int32 palette index, see palette definitions for values. Based on Color Brewer.
- **edge_source_color** (*str*) – Attribute overriding edge's source color if no edge_color, as an rgba int64 value.
- **edge_destination_color** (*str*) – Attribute overriding edge's destination color if no edge_color, as an rgba int64 value.
- **edge_weight** (*str*) – Attribute overriding edge weight. Default is 1. Advanced layout controls will relayout edges based on this value.
- **point_title** (*str*) – Attribute overriding node's minimized label text. By default, the node ID is used.
- **point_label** (*str*) – Attribute overriding node's expanded label text. By default, scrollable list of attribute/value mappings.
- **point_color** (*str*) – Attribute overriding node's color.rgba (int64) or int32 palette index, see palette definitions for values. Based on Color Brewer.
- **point_size** (*str*) – Attribute overriding node's size. By default, uses the node degree. The visualization will normalize point sizes and adjust dynamically using semantic zoom.
- **point_x** (*str*) – Attribute overriding node's initial x position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout
- **point_y** (*str*) – Attribute overriding node's initial y position. Combine with ".settings(url_params={'play': 0}))" to create a custom layout

**Returns**
Plotter

**Return type**
Plotter

**Example: Minimal**

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst')
```

**Example: Node colors**

```
import graphistry
g = graphistry.bind()
g = g.bind(source='src', destination='dst',
           node='id', point_color='color')
```

**Example: Chaining**

```
import graphistry
g = graphistry.bind(source='src', destination='dst', node='id')

g1 = g.bind(point_color='color1', point_size='size1')
```

(continues on next page)

(continued from previous page)

```
g.bind(point_color='color1b')

g2a = g1.bind(point_color='color2a')
g2b = g1.bind(point_color='color2b', point_size='size2b')

g3a = g2a.bind(point_size='size3a')
g3b = g2b.bind(point_size='size3b')
```

In the above **Chaining** example, all bindings use src/dst/id. Colors and sizes bind to:

```
g: default/default
g1: color1/size1
g2a: color2a/size1
g2b: color2b/size2b
g3a: color2a/size3a
g3b: color2b/size3b
```

**bolt**(*driver*)

**chain**(*ops*)

ops is List[ASTObject]

**Parameters**
**ops** (*List [Any]*)

**Return type**
Plottable

**collapse**(*node*, *attribute*, *column*, *self_edges=False*, *unwrap=False*, *verbose=False*)

**Parameters**

- **node** (*str | int*)
- **attribute** (*str | int*)
- **column** (*str | int*)
- **self_edges** (*bool*)
- **unwrap** (*bool*)
- **verbose** (*bool*)

**Return type**
Plottable

**compute_cugraph**(*alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

Run cugraph algorithm on graph. For algorithm parameters, see cuGraph docs.

**Parameters**

- **alg** (*str*) – algorithm name
- **out_col** (*Optional [str]*) – node table output column name, defaults to alg param
- **params** (*dict*) – algorithm parameters passed to cuGraph as kwargs
- **kind** (*CuGraphKind*) – kind of cugraph to use

- **directed** (*bool*) – whether graph is directed
- **G** (*Optional [cugraph.Graph]*) – cugraph graph to use; if None, use self

**Returns**
Plottable

**Return type**
*Plottable*

**Example: Pass params to cugraph**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('betweenness_centrality', params={'k': 2})
assert 'betweenness_centrality' in g2._nodes.columns
```

**Example: Pagerank**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

**Example: Personalized Pagerank**

::
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']}) g = graphistry.edges(edges, 's', 'd') g2 = g.compute_cugraph('pagerank', params={'personalization': cudf.DataFrame({'vertex': ['a'], 'values': [1]})}) assert 'pagerank' in g2._nodes.columns

**Example: Katz centrality with rename**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('katz_centrality', out_col='katz_centrality_renamed')
assert 'katz_centrality_renamed' in g2._nodes.columns
```

**compute_igraph**(*alg*, *out_col=None*, *directed=None*, *use_vids=False*, *params={}*, *stringify_rich_types=True*)

Enrich or replace graph using igraph methods

**Parameters**

- **alg** (*str*) – Name of an igraph.Graph method like *pagerank*
- **out_col** (*Optional [str]*) – For algorithms that generate a node attribute column, *out_col* is the desired output column name. When *None*, use the algorithm's name. (default None)
- **directed** (*Optional [bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – During the to_igraph conversion, whether to interpret IDs as igraph vertex IDs (non-negative integers) or arbitrary values (False, default)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method
- **stringify_rich_types** (*bool*) – When rich types like igraph.Graph are returned, which may be problematic for downstream rendering, coerce them to strings

**Returns**
Plotter

**Return type**
Plotter

**Example: Pagerank**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

**Example: Pagerank with custom name**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', out_col='my_pr')
assert 'my_pr' in g2._nodes.columns
```

**Example: Pagerank on an undirected**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', directed=False)
assert 'pagerank' in g2._nodes.columns
```

**Example: Pagerank with custom parameters**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', params={'damping': 0.85})
assert 'pagerank' in g2._nodes.columns
```

**Example: Personalized Pagerank**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('personalized_pagerank')
assert 'personalized_pagerank' in g2._nodes.columns
```

`copy()`

**Return type**
Plottable

`cypher(`*query*, *params={}*`)`

Execute a Cypher query against a Neo4j, Memgraph, or Amazon Neptune database and retrieve the results.

This method runs a Cypher query on a Neo4j, Memgraph, or Amazon Neptune graph database using a BOLT driver. The query results are transformed into DataFrames for nodes and edges, which are then bound to the current graph visualization context. You can also pass parameters to the Cypher query via the *params* argument.

**Parameters**

- **query** (*str*) – The Cypher query string to execute.
- **params** (*dict, optional*) – Optional dictionary of parameters to pass to the Cypher query.

**Returns**

Plotter with updated nodes and edges based on the query result.

**Return type**

*PlotterBase*

**Raises**

**ValueError** – If no BOLT driver connection is available.

**Example (Simple Neo4j Query)**

```
import graphistry
from neo4j import GraphDatabase

# Register with Neo4j connection details
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))

graphistry.register(bolt=driver)

# Run a basic Cypher query
g = graphistry.cypher('''
    MATCH (node1)-[connection]-(node2)
    RETURN node1, connection, node2;
''')

# Visualize the results
g.plot()
```

**Example (Simple Amazon Neptune Query)**

```
from neo4j import GraphDatabase

# Register with Amazon Neptune connection details
uri = f"bolt://{url}:8182"
driver = GraphDatabase.driver(uri, auth=("ignored", "ignored"), encrypted=True)

graphistry.register(bolt=driver)

# Run a simple Cypher query
g = graphistry.cypher('''
    MATCH (node1)-[connection]-(node2)
    RETURN node1, connection, node2;
''')
```

(continues on next page)

(continued from previous page)

```
# Visualize the results
g.plot()
```

**Example (Simple Memgraph Query)**

```
import graphistry
from neo4j import GraphDatabase

# Register with Memgraph connection details
MEMGRAPH = {
    'uri': "bolt://localhost:7687",
    'auth': (" ", " ")
}

graphistry.register(api=3, username="X", password="Y", bolt=MEMGRAPH)

# Run a simple Cypher query on Memgraph
g = graphistry.cypher('''
    MATCH (node1)-[connection]-(node2)
    RETURN node1, connection, node2;
''')

# Visualize the results
g.plot()
```

**Example (Parameterized Query with Node and Edge Inspection)**

```
import graphistry
from neo4j import GraphDatabase

# Register with Neo4j connection details
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))

graphistry.register(bolt=driver)

# Run a parameterized Cypher query
query = '''
    MATCH (node1)-[connection]-(node2)
    WHERE node1.name = $name
    RETURN node1, connection, node2;
'''
params = {"name": "Alice"}

g = graphistry.cypher(query, params)

# Inspect the resulting nodes and edges DataFrames
print(g._nodes)  # DataFrame with node information
print(g._edges)  # DataFrame with edge information

# Visualize the results
g.plot()
```

This demonstrates how to connect to Neo4j, Memgraph, or Amazon Neptune, run a simple or parameterized Cypher query, inspect query results (nodes and edges), and visualize the graph.

`description`(*description*)

Upload description

**Parameters**
`description` (*str*) – Upload description

`drop_nodes`(*nodes*)

**Parameters**
`nodes` (*Any*)

**Return type**
Plottable

`edges`(*edges*, *source=None*, *destination=None*, *edge=None*, **args*, ***kwargs*)

Specify edge list data and associated edge attribute values. If a callable, will be called with current Plotter and whatever positional+named arguments

**Parameters**
`edges` (*Pandas dataframe, NetworkX graph, or IGraph graph*) – Edges and their attributes, or transform from Plotter to edges

**Returns**
Plotter

**Return type**
Plotter

**Example**

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(df)
    .plot()
```

**Example**

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0], 'id': [0, 1, 2]})
graphistry
    .bind(source='src', destination='dst', edge='id')
    .edges(df)
    .plot()
```

**Example**

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .edges(df, 'src', 'dst')
    .plot()
```

**Example**

```
import graphistry
df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0], 'id': [0, 1, 2]})
graphistry
    .edges(df, 'src', 'dst', 'id')
    .plot()
```

**Example**

```
import graphistry

def sample_edges(g, n):
    return g._edges.sample(n)

df = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})

graphistry
    .edges(df, 'src', 'dst')
    .edges(sample_edges, n=2)
    .edges(sample_edges, None, None, None, 2)  # equivalent
    .plot()
```

**encode_axis**(*rows=[]*)

Render radial and linear axes with optional labels

**Parameters**

**rows** (*List[Dict]*) – List of rows - { label: Optional[str],?r: float, ?x: float, ?y: float, ?internal: true, ?external: true, ?space: true }

**Returns**

Plotter

**Return type**

Plotter

**Example: Several radial axes**

```
g.encode_axis([
  {'r': 14, 'external': True, 'label': 'outermost'},
  {'r': 12, 'external': True},
  {'r': 10, 'space': True},
  {'r': 8, 'space': True},
  {'r': 6, 'internal': True},
  {'r': 4, 'space': True},
  {'r': 2, 'space': True, 'label': 'innermost'}
])
```

**Example: Several horizontal axes**

```
g.encode_axis([
  {"label": "a",  "y": 2, "internal": True },
  {"label": "b",  "y": 40, "external": True, "width": 20, "bounds": {"min":␣
↪40, "max": 400}},
])
```

encode_edge_badge(*column*, *position='TopRight'*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *color=None*, *bg=None*, *fg=None*, *for_current=False*, *for_default=True*, *as_text=None*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

encode_edge_color(*column*, *palette=None*, *as_categorical=None*, *as_continuous=None*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set edge color with more control than bind()

**Parameters**

- **column** (*str*) – Data column name
- **palette** (*Optional[list]*) – Optional list of color-like strings. Ex: [“black, “#FF0”, “rgb(255,255,255)” ]. Used as a gradient for continuous and round-robin for categorical.
- **as_categorical** (*Optional[bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.
- **as_continuous** (*Optional[bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to color-like strings. Ex: {“car”: “red”, “truck”: #000”}
- **default_mapping** (*Optional[str]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=”gray”.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns**
Plotter

**Return type**
Plotter

**Example: See encode_point_color**

encode_edge_icon(*column*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

Set edge icon with more control than bind() Values from Font Awesome 4 such as “laptop”: https://fontawesome.com/v4.7.0/icons/ , image URLs (http://...), and data URIs (data:...). When as_text=True is enabled, values are instead interpreted as raw strings.

**Parameters**

- **column** (*str*) – Data column name
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to icon name strings. Ex: {“toyota”: ‘car’, “ford”: ‘truck’}
- **default_mapping** (*Optional[Union[int, float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.

- **for_default** (*Optional [bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional [bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as_text** (*Optional [bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)

**Returns**
Plotter

**Return type**
Plotter

**Example: Set a string column of icons for the edge icons, same as bind(edge_icon='my_column')**

```
g2a = g.encode_edge_icon('my_icons_column')
```

**Example: Map specific values to specific icons, including with a default**

```
g2a = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'})
g2b = g.encode_edge_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'}, default_mapping='question')
```

**Example: Map countries to abbreviations**

```
g2a = g.encode_edge_icon('country_abbrev', as_text=True)
g2b = g.encode_edge_icon('country', as_text=True, categorical_mapping={
↪'England': 'UK', 'America': 'US'}, default_mapping='')
```

**Example: Border**

```
g2b = g.encode_edge_icon('country', border={'width': 3, color: 'black',
↪'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America': 'US
↪'})
```

**encode_point_badge**(*column*, *position='TopRight'*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *color=None*, *bg=None*, *fg=None*, *for_current=False*, *for_default=True*, *as_text=None*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

**encode_point_color**(*column*, *palette=None*, *as_categorical=None*, *as_continuous=None*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set point color with more control than bind()

**Parameters**

- **column** (*str*) – Data column name
- **palette** (*Optional [list]*) – Optional list of color-like strings. Ex: ["black, "#FF0", "rgb(255,255,255)" ]. Used as a gradient for continuous and round-robin for categorical.
- **as_categorical** (*Optional [bool]*) – Interpret column values as categorical. Ex: Uses palette via round-robin when more values than palette entries.

- **as_continuous** (*Optional [bool]*) – Interpret column values as continuous. Ex: Uses palette for an interpolation gradient when more values than palette entries.
- **categorical_mapping** (*Optional [dict]*) – Mapping from column values to color-like strings. Ex: {"car": "red", "truck": #000"}
- **default_mapping** (*Optional [str]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping="gray".
- **for_default** (*Optional [bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional [bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns**
Plotter

**Return type**
Plotter

**Example: Set a palette-valued column for the color, same as bind(point_color='my_column')**

```
g2a = g.encode_point_color('my_int32_palette_column')
g2b = g.encode_point_color('my_int64_rgb_column')
```

**Example: Set a cold-to-hot gradient of along the spectrum blue, yellow, red**

```
g2 = g.encode_point_color('my_numeric_col', palette=["blue", "yellow", "red
↪"], as_continuous=True)
```

**Example: Round-robin sample from 5 colors in hex format**

```
g2 = g.encode_point_color('my_distinctly_valued_col', palette=["#000", "#00F
↪", "#0F0", "#0FF", "#FFF"], as_categorical=True)
```

**Example: Map specific values to specific colors, including with a default**

```
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪'ford': 'blue'})
g2a = g.encode_point_color('brands', categorical_mapping={'toyota': 'red',
↪'ford': 'blue'}, default_mapping='gray')
```

**encode_point_icon**(*column*, *categorical_mapping=None*, *continuous_binning=None*, *default_mapping=None*, *comparator=None*, *for_default=True*, *for_current=False*, *as_text=False*, *blend_mode=None*, *style=None*, *border=None*, *shape=None*)

Set node icon with more control than bind(). Values from Font Awesome 4 such as "laptop": https://fontawesome.com/v4.7.0/icons/ , image URLs (http://...), and data URIs (data:...). When as_text=True is enabled, values are instead interpreted as raw strings.

**Parameters**

- **column** (*str*) – Data column name
- **categorical_mapping** (*Optional [dict]*) – Mapping from column values to icon name strings. Ex: {"toyota": 'car', "ford": 'truck'}

- **default_mapping** (*Optional[Union[int, float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.
- **as_text** (*Optional[bool]*) – Values should instead be treated as raw strings, instead of icons and images. (Default False.)
- **blend_mode** (*Optional[str]*) – CSS blend mode
- **style** (*Optional[dict]*) – CSS filter properties - opacity, saturation, luminosity, grayscale, and more
- **border** (*Optional[dict]*) – Border properties - 'width', 'color', and 'storke'

**Returns**
Plotter

**Return type**
Plotter

**Example: Set a string column of icons for the point icons, same as bind(point_icon='my_column')**

```
g2a = g.encode_point_icon('my_icons_column')
```

**Example: Map specific values to specific icons, including with a default**

```
g2a = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'})
g2b = g.encode_point_icon('brands', categorical_mapping={'toyota': 'car',
↪'ford': 'truck'}, default_mapping='question')
```

**Example: Map countries to abbreviations**

```
g2b = g.encode_point_icon('country_abbrev', as_text=True)
g2b = g.encode_point_icon('country', as_text=True, categorical_mapping={
↪'England': 'UK', 'America': 'US'}, default_mapping='')
```

**Example: Border**

```
g2b = g.encode_point_icon('country', border={'width': 3, color: 'black',
↪'stroke': 'dashed'}, 'categorical_mapping={'England': 'UK', 'America': 'US
↪'})
```

**encode_point_size**(*column*, *categorical_mapping=None*, *default_mapping=None*, *for_default=True*, *for_current=False*)

Set point size with more control than bind()

**Parameters**

- **column** (*str*) – Data column name
- **categorical_mapping** (*Optional[dict]*) – Mapping from column values to numbers. Ex: {"car": 100, "truck": 200}

- **default_mapping** (*Optional[Union[int, float]]*) – Augment categorical_mapping with mapping for values not in categorical_mapping. Ex: default_mapping=50.
- **for_default** (*Optional[bool]*) – Use encoding for when no user override is set. Default on.
- **for_current** (*Optional[bool]*) – Use encoding as currently active. Clearing the active encoding resets it to default, which may be different. Default on.

**Returns**
Plotter

**Return type**
Plotter

**Example: Set a numerically-valued column for the size, same as bind(point_size='my_column')**

```
g2a = g.encode_point_size('my_numeric_column')
```

**Example: Map specific values to specific colors, including with a default**

```
g2a = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↪'ford': 200})
g2b = g.encode_point_size('brands', categorical_mapping={'toyota': 100,
↪'ford': 200}, default_mapping=50)
```

**filter_edges_by_dict**(*filter_dict=None*)

**Parameters**
**filter_dict** (*dict | None*)

**Return type**
Plottable

**filter_nodes_by_dict**(*filter_dict=None*)

**Parameters**
**filter_dict** (*dict | None*)

**Return type**
Plottable

**from_cugraph**(*G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

If bound IDs, use the same IDs in the returned graph.

If non-empty nodes/edges, instead of returning G's topology, use existing topology and merge in G's attributes

**Parameters**

- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)
- **load_nodes** (*bool*)
- **load_edges** (*bool*)
- **merge_if_existing** (*bool*)

**from_igraph**(*ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Convert igraph object into Plotter

If base g has _node, _source, _destination definitions, use them

When merge_if_existing with preexisting nodes/edges df and shapes match ig, combine attributes

For merge_if_existing to work with edges, must set g._edge and have corresponding edge index attribute in igraph.Graph

**Parameters**

- **ig** (*igraph*) – Source igraph object
- **node_attributes** (*Optional [List [str]]*) – Subset of node attributes to load; None means all (default)
- **edge_attributes** (*Optional [List [str]]*) – Subset of edge attributes to load; None means all (default)
- **load_nodes** (*bool*) – Whether to load nodes dataframe (default True)
- **load_edges** (*bool*) – Whether to load edges dataframe (default True)
- **merge_if_existing** – Whether to merge with existing node/edge dataframes (default True)
- **merge_if_existing** – bool

**Returns**

Plotter

**Example: Convert from igraph, including all node/edge properties**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'],
↪'v': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degrees()
assert 'degree' in g._nodes.columns
g2 = g.from_igraph(g.to_igraph())
assert len(g2._nodes.columns) == len(g._nodes.columns)
```

**Example: Enrich from igraph, but only load in 1 node attribute**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'],
↪'v': [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degree()
assert 'degree' in g._nodes
ig = g.to_igraph(include_nodes=False)
assert 'degree' not in ig.vs
ig.vs['pagerank'] = ig.pagerank()
g2 = g.from_igraph(ig, load_edges=False, node_attributes=[g._node, 'pagerank
↪'])
assert 'pagerank' in g2._nodes
asssert 'degree' in g2._nodes
```

`from_networkx(G)`

Convert a NetworkX graph to a PyGraphistry graph.

This method takes a NetworkX graph and converts it into a format that PyGraphistry can use for visualization. It extracts the node and edge data from the NetworkX graph and binds them to the graph object for further manipulation or visualization using PyGraphistry's API.

**Parameters**
`G` (*networkx.Graph or networkx.DiGraph*) – The NetworkX graph to convert.

**Returns**
A PyGraphistry Plottable object with the node and edge data from the NetworkX graph.

**Return type**
*Plottable*

**Example: Basic NetworkX Conversion**

```
import graphistry
import networkx as nx

# Create a NetworkX graph
G = nx.Graph()
G.add_nodes_from([
    (1, {"v": "one"}),
    (2, {"v": "two"}),
    (3, {"v": "three"}),
    (4, {"v": "four"}),
    (7, {"v": "seven"}),
    (8, {"v": "eight"})
])
G.add_edges_from([
    [2, 3],
    [3, 4],
    [7, 8]
])

# Convert the NetworkX graph to PyGraphistry format
g = from_networkx(G)

g.plot()
```

This example creates a simple NetworkX graph with nodes and edges, converts it using *from_networkx()*, and then plots it with the PyGraphistry API.

**Example: Using Custom Node and Edge Bindings**

```
import graphistry
import networkx as nx

# Create a NetworkX graph with attributes
G = nx.Graph()
G.add_nodes_from([
    (1, {"v": "one"}),
```

(continues on next page)

(continued from previous page)

```
    (2, {"v": "two"}),
    (3, {"v": "three"}),
    (4, {"v": "four"}),
    (7, {"v": "seven"}),
    (8, {"v": "eight"})
])
G.add_edges_from([
    [2, 3],
    [3, 4],
    [7, 8]
])

# Bind custom node and edge names when converting from NetworkX to␣
↪PyGraphistry
g = graphistry.bind(source='src', destination='dst').from_networkx(G)

g.plot()
```

**get_degrees**(*col='degree'*, *degree_in='degree_in'*, *degree_out='degree_out'*)

**Parameters**

- **col** (*str*)
- **degree_in** (*str*)
- **degree_out** (*str*)

**Return type**
Plottable

**get_indegrees**(*col='degree_in'*)

**Parameters**
**col** (*str*)

**Return type**
Plottable

**get_outdegrees**(*col='degree_out'*)

**Parameters**
**col** (*str*)

**Return type**
Plottable

**get_topological_levels**(*level_col='level'*, *allow_cycles=True*, *warn_cycles=True*, *remove_self_loops=True*)

**Parameters**

- **level_col** (*str*)
- **allow_cycles** (*bool*)
- **warn_cycles** (*bool*)
- **remove_self_loops** (*bool*)

**Return type**
Plottable

`graph`(*ig*)

Specify the node and edge data.

**Parameters**
**ig** (*Any*) – NetworkX graph or an IGraph graph with node and edge attributes.

**Returns**
Plotter

**Return type**
Plotter

`gsql`(*query*, *bindings={}*, *dry_run=False*)

Run Tigergraph query in interpreted mode and return transformed Plottable

**param query**
Code to run

**type query**
str

**param bindings**
Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList

**type bindings**
Optional[dict]

**param dry_run**
Return target URL without running

**type dry_run**
bool

**returns**
Plotter

**rtype**
Plotter

**Example: Minimal**

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do
```

(continues on next page)

(continued from previous page)

```
    Start = select t from Start:s-(:e)-:t
    where e.goUpper == TRUE
    accum @@edgeList += e
    having t.type != "Service";
    end;

    print @@edgeList;
}
""").plot()
```

**Example: Full**

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql("""
INTERPRET QUERY () FOR GRAPH Storage {

    OrAccum<BOOL> @@stop;
    ListAccum<EDGE> @@edgeList;
    SetAccum<vertex> @@set;

    @@set += to_vertex("61921", "Pool");

    Start = @@set;

    while Start.size() > 0 and @@stop == false do

    Start = select t from Start:s-(:e)-:t
    where e.goUpper == TRUE
    accum @@edgeList += e
    having t.type != "Service";
    end;

    print @@my_edge_list;
}
""", {'edges': 'my_edge_list'}).plot()
```

**gsql_endpoint**(*method_name*, *args={}*, *bindings={}*, *db=None*, *dry_run=False*)

Invoke Tigergraph stored procedure at a user-definend endpoint and return transformed Plottable

**Parameters**

- **method_name** (*str*) – Stored procedure name
- **args** (*Optional [dict]*) – Named endpoint arguments
- **bindings** (*Optional [dict]*) – Mapping defining names of returned 'edges' and/or 'nodes', defaults to @@nodeList and @@edgeList
- **db** (*Optional [str]*) – Name of the database, defaults to value set in .tigergraph(...)
- **dry_run** (*bool*) – Return target URL without running

**Returns**
Plotter

**Return type**
Plotter

**Example: Minimal**

```
import graphistry
tg = graphistry.tigergraph(db='my_db')
tg.gsql_endpoint('neighbors').plot()
```

**Example: Full**

```
import graphistry
tg = graphistry.tigergraph()
tg.gsql_endpoint('neighbors', {'k': 2}, {'edges': 'my_edge_list'}, 'my_db').
↪plot()
```

**Example: Read data**

```
import graphistry
tg = graphistry.tigergraph()
out = tg.gsql_endpoint('neighbors')
(nodes_df, edges_df) = (out._nodes, out._edges)
```

**hop**(*nodes*, *hops=1*, *to_fixed_point=False*, *direction='forward'*, *edge_match=None*, *source_node_match=None*, *destination_node_match=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *return_as_wave_front=False*, *target_wave_front=None*)

**Parameters**

- **nodes** (*DataFrame | None*)
- **hops** (*int | None*)
- **to_fixed_point** (*bool*)
- **direction** (*str*)
- **edge_match** (*dict | None*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **return_as_wave_front** (*bool*)
- **target_wave_front** (*DataFrame | None*)

**Return type**
Plottable

**hypergraph**(*raw_events*, *entity_types=None*, *opts={}*, *drop_na=True*, *drop_edge_attrs=False*, *verbose=True*, *direct=False*, *engine='pandas'*, *npartitions=None*, *chunksize=None*)

Transform a dataframe into a hypergraph.

**Parameters**

- **raw_events** (*pandas.DataFrame*) – Dataframe to transform (pandas or cudf).
- **entity_types** (*Optional[list]*) – Columns (strings) to turn into nodes, None signifies all
- **opts** (*dict*) – See below
- **drop_edge_attrs** (*bool*) – Whether to include each row's attributes on its edges, defaults to False (include)
- **verbose** (*bool*) – Whether to print size information
- **direct** (*bool*) – Omit hypernode and instead strongly connect nodes in an event
- **engine** (*bool*) – String (pandas, cudf, ...) for engine to use
- **npartitions** (*Optional[int]*) – For distributed engines, how many coarse-grained pieces to split events into
- **chunksize** (*Optional[int]*) – For distributed engines, split events after chunksize rows
- **drop_na** (*bool*)

Create a graph out of the dataframe, and return the graph components as dataframes, and the renderable result Plotter. Hypergraphs reveal relationships between rows and between column values. This transform is useful for lists of events, samples, relationships, and other structured high-dimensional data.

Specify local compute engine by passing *engine='pandas'*, 'cudf', 'dask', 'dask_cudf' (default: 'pandas'). If events are not in that engine's format, they will be converted into it.

The transform creates a node for every unique value in the entity_types columns (default: all columns). If direct=False (default), every row is also turned into a node. Edges are added to connect every table cell to its originating row's node, or if direct=True, to the other nodes from the same row. Nodes are given the attribute 'type' corresponding to the originating column name, or in the case of a row, 'EventID'. Options further control the transform, such column category definitions for controlling whether values reocurring in different columns should be treated as one node, or whether to only draw edges between certain column type pairs.

Consider a list of events. Each row represents a distinct event, and each column some metadata about an event. If multiple events have common metadata, they will be transitively connected through those metadata values. The layout algorithm will try to cluster the events together. Conversely, if an event has unique metadata, the unique metadata will turn into nodes that only have connections to the event node, and the clustering algorithm will cause them to form a ring around the event node.

Best practice is to set EVENTID to a row's unique ID, SKIP to all non-categorical columns (or entity_types to all categorical columns), and CATEGORY to group columns with the same kinds of values.

To prevent creating nodes for null values, set drop_na=True. Some dataframe engines may have undesirable null handling, and recommend replacing None values with np.nan .

The optional `opts={...}` configuration options are:

- 'EVENTID': Column name to inspect for a row ID. By default, uses the row index.

- 'CATEGORIES': Dictionary mapping a category name to inhabiting columns. E.g., {'IP': ['srcAddress', 'dstAddress']}. If the same IP appears in both columns, this makes the transform generate one node for it, instead of one for each column.
- 'DELIM': When creating node IDs, defines the separator used between the column name and node value
- 'SKIP': List of column names to not turn into nodes. For example, dates and numbers are often skipped.
- 'EDGES': For direct=True, instead of making all edges, pick column pairs. E.g., {'a': ['b', 'd'], 'd': ['d']} creates edges between columns a->b and a->d, and self-edges d->d.

**Returns**
{'entities': DF, 'events': DF, 'edges': DF, 'nodes': DF, 'graph': Plotter}

**Return type**
dict

**Parameters**

- **entity_types** (*List[str] / None*)
- **opts** (*dict*)
- **drop_na** (*bool*)
- **drop_edge_attrs** (*bool*)
- **verbose** (*bool*)
- **direct** (*bool*)
- **engine** (*str*)
- **npartitions** (*int / None*)
- **chunksize** (*int / None*)

**Example: Connect user<-row->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df)
g = h['graph'].plot()
```

**Example: Connect user->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph'].plot()
```

**Example: Connect user<->boss**

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True, opts={'EDGES': {'user':␣
↪['boss'], 'boss': ['user']}})
g = h['graph'].plot()
```

**Example: Only consider some columns for nodes**

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, entity_types=['boss'])
g = h['graph'].plot()
```

**Example: Collapse matching user::<id> and boss::<id> nodes into one person::<id> node**

```
import graphistry
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, opts={'CATEGORIES': {'person': ['user
↪', 'boss']}})
g = h['graph'].plot()
```

**Example: Use cudf engine instead of pandas**

```
import cudf, graphistry
users_gdf = cudf.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y
↪']})
h = graphistry.hypergraph(users_gdf, engine='cudf')
g = h['graph'].plot()
```

`igraph2pandas`(*ig*)

Under current bindings, transform an IGraph into a pandas edges dataframe and a nodes dataframe.

Deprecated in favor of *.from_igraph()*

**Example**

```
import graphistry
g = graphistry.bind()

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = g.bind(source='src', destination='dst').edges(es)

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership

(es2, vs2) = g.igraph2pandas(ig)
g.nodes(vs2).bind(point_color='community').plot()
```

`infer_labels`()

**Returns**
Plotter w/neo4j

- Prefers point_title/point_label if available
- Fallback to node id
- Raises exception if no nodes available, no likely candidates, and no matching node id fallback

**Example**

```
import graphistry
g = graphistry.nodes(pd.read_csv('nodes.csv'), 'id_col').infer_labels()
g.plot()
```

`keep_nodes`(*nodes*)

**Parameters**
`nodes` (*List | Any*)

**Return type**
Plottable

`layout_cugraph`(*layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

Layout the grpah using a cuGraph algorithm. For a list of layouts, see cugraph documentation (currently just force_atlas2).

**Parameters**

- `layout` (*str*) – Name of an cugraph layout method like *force_atlas2*
- `params` (*dict*) – Any named parameters to pass to the underlying cugraph method
- `kind` (*CuGraphKind*) – The kind of cugraph Graph
- `directed` (*bool*) – During the to_cugraph conversion, whether to be directed. (default True)
- `G` (*Optional[Any]*) – The cugraph graph (G) to layout. If None, the current graph is used.
- `bind_position` (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- `x_out_col` (*str*) – Attribute to write x position to. (default 'x')
- `y_out_col` (*str*) – Attribute to write x position to. (default 'y')
- `play` (*Optional[str]*) – If defined, set settings(url_params={'play': play}). (default 0)

**Returns**
Plotter

**Return type**
Plotter

**Example: ForceAtlas2 layout**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_cugraph().plot()
```

**Example: Change which column names are generated**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('force_atlas2', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

**Example: Pass parameters to layout methods**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('forceatlas_2', params={'lin_log_mode': True, 'prevent_
↪overlapping': True})
g2.plot()
```

**layout_graphviz**(*prog='dot'*, *args=None*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *skip_styling=False*, *render_to_disk=False*, *path=None*, *format=None*)

Use graphviz for layout, such as hierarchical trees and directed acycle graphs

Requires pygraphviz Python bindings and graphviz native libraries to be installed, see https://pygraphviz.github.io/documentation/stable/install.html

See PROGS for available layout algorithms

To render image to disk, set render=True

**Parameters**

- **self** (Plottable) – Base graph
- **prog** (graphistry.plugins_types.graphviz_types.Prog) – Layout algorithm - "dot", "neato", ...
- **args** (*Optional[str]*) – Additional arguments to pass to the graphviz commandline for layout
- **directed** (*bool*) – Whether the graph is directed (True, default) or undirected (False)
- **strict** (*bool*) – Whether the graph is strict (True) or not (False, default)
- **graph_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.GraphAttr, Any]]) – Graphviz graph attributes, see https://graphviz.org/docs/graph/
- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, Any]]) – Graphviz node attributes, see https://graphviz.org/docs/nodes/
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, Any]]) – Graphviz edge attributes, see https://graphviz.org/docs/edges/
- **skip_styling** (*bool*) – Whether to skip applying default styling (False, default) or not (True)
- **render_to_disk** (*bool*) – Whether to render the graph to disk (False, default) or not (True)
- **path** (*Optional[str]*) – Path to save the rendered image when render_to_disk=True
- **format** (Optional[graphistry.plugins_types.graphviz_types.Format]) – Format of the rendered image when render_to_disk=True
- **drop_unsanitary** (*bool*) – Whether to drop unsanitary attributes (False, default) or not (True), recommended for sensitive settings

**Returns**

Graph with layout and style settings applied, setting x/y

**Return type**
*Plottable*

**Example: Dot layout for rigid hierarchical layout of trees and directed acyclic graphs**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz('dot').plot()
```

**Example: Neato layout for organic layout of small graphs**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz('neato').plot()
```

**Example: Set graphviz attributes at graph level**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz(
    prog='dot',
    graph_attr={
        'ratio': 10
    }
).plot()
```

**Example: Save rendered image to disk as a png**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz(
    'dot',
    render_to_disk=True,
    path='graph.png',
    format='png'
)
```

**Example: Save rendered image to disk as a png with passthrough of rendering styles**

```
import graphistry
edges = pd.DataFrame({
    's': ['a','b','c','d'],
    'd': ['b','c','d','e'],
    'color': ['red', None, None, 'yellow']
})
nodes = pd.DataFrame({
    'n': ['a','b','c','d','e'],
    'shape': ['circle', 'square', None, 'square', 'circle']
})
g = graphistry.edges(edges, 's', 'd')
```

(continues on next page)

(continued from previous page)

```
g.layout_graphviz(
    'dot',
    render_to_disk=True,
    path='graph.png',
    format='png'
)
```

**layout_igraph**(*layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Compute graph layout using igraph algorithm. For a list of layouts, see layout_algs or igraph documentation.

**Parameters**

- **layout** (*str*) – Name of an igraph.Graph.layout method like *sugiyama*
- **directed** (*Optional [bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – Whether to use igraph vertex ids (non-negative integers) or arbitary node ids (False, default)
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional [str]*) – If defined, set settings(url_params={'play': play}). (default 0)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method

**Returns**
Plotter

**Return type**
Plotter

**Example: Sugiyama layout**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama')
assert 'x' in g2._nodes
g2.plot()
```

**Example: Change which column names are generated**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

**Example: Pass parameters to layout methods - Sort nodes by degree**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.get_degrees()
assert 'degree' in g._nodes.columns
g3 = g.layout_igraph('sugiyama', params={'layers': 'degree'})
g3.plot()
```

**layout_settings**(*play=None*, *locked_x=None*, *locked_y=None*, *locked_r=None*, *left=None*, *top=None*, *right=None*, *bottom=None*, *lin_log=None*, *strong_gravity=None*, *dissuade_hubs=None*, *edge_influence=None*, *precision_vs_speed=None*, *gravity=None*, *scaling_ratio=None*)

Set layout options. Additive over previous settings.

Corresponds to options at https://hub.graphistry.com/docs/api/1/rest/url/#urloptions

**Example: Animated radial layout**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'boss': ['c','c','e','e']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1,   1,   2,   3,   4],
    'x': [1,   1,   0,   0,   0],
})
g = (graphistry
    .edges(edges, 's', 'd')
    .nodes(nodes, 'n')
    .layout_settings(locked_r=True, play=2000)
g.plot()
```

**Parameters**

- **play** (*int* / *None*)
- **locked_x** (*bool* / *None*)
- **locked_y** (*bool* / *None*)
- **locked_r** (*bool* / *None*)
- **left** (*float* / *None*)
- **top** (*float* / *None*)
- **right** (*float* / *None*)
- **bottom** (*float* / *None*)
- **lin_log** (*bool* / *None*)
- **strong_gravity** (*bool* / *None*)
- **dissuade_hubs** (*bool* / *None*)
- **edge_influence** (*float* / *None*)
- **precision_vs_speed** (*float* / *None*)

- **gravity** (*float* / *None*)
- **scaling_ratio** (*float* / *None*)

**materialize_nodes**(*reuse=True*, *engine=EngineAbstract.AUTO*)

**Parameters**

- **reuse** (*bool*)
- **engine** (*EngineAbstract* / *str*)

**Return type**
Plottable

**name**(*name*)

Upload name

**Parameters**
**name** (*str*) – Upload name

**networkx2pandas**(*g*)

**networkx_checkoverlap**(*g*)

Raise an error if the node attribute already exists in the graph

**nodes**(*nodes*, *node=None*, **args*, ***kwargs*)

Specify the set of nodes and associated data. If a callable, will be called with current Plotter and whatever positional+named arguments

Must include any nodes referenced in the edge list.

**Parameters**
**nodes** (*Pandas dataframe or Callable*) – Nodes and their attributes.

**Returns**
Plotter

**Return type**
Plotter

**Example**

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry
    .bind(source='src', destination='dst')
    .edges(es)

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.bind(node='v').nodes(vs)

g.plot()
```

**Example**

```
import graphistry

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = graphistry.edges(es, 'src', 'dst')

vs = pandas.DataFrame({'v': [0,1,2], 'lbl': ['a', 'b', 'c']})
g = g.nodes(vs, 'v)

g.plot()
```

**Example**

```
import graphistry

def sample_nodes(g, n):
    return g._nodes.sample(n)

df = pandas.DataFrame({'id': [0,1,2], 'v': [1,2,0]})

graphistry
    .nodes(df, 'id')
    ..nodes(sample_nodes, n=2)
    ..nodes(sample_nodes, None, 2)  # equivalent
    .plot()
```

**nodexl**(*xls_or_url*, *source='default'*, *engine=None*, *verbose=False*)

**pandas2igraph**(*edges*, *directed=True*)

Convert a pandas edge dataframe to an IGraph graph.

Uses current bindings. Defaults to treating edges as directed.

**Example**

```
import graphistry
g = graphistry.bind()

es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
g = g.bind(source='src', destination='dst')

ig = g.pandas2igraph(es)
ig.vs['community'] = ig.community_infomap().membership
g.bind(point_color='community').plot(ig)
```

**pipe**(*graph_transform*, **args*, ***kwargs*)

Create new Plotter derived from current

**Parameters**
**graph_transform** (*Callable*)

**Return type**
Plottable

**Example: Simple**

```
import graphistry

def fill_missing_bindings(g, source='src', destination='dst):
    return g.bind(source=source, destination=destination)

graphistry
    .edges(pandas.DataFrame({'src': [0,1,2], 'd': [1,2,0]}))
    .pipe(fill_missing_bindings, destination='d')  # binds 'src'
    .plot()
```

**plot**(*graph=None*, *nodes=None*, *name=None*, *description=None*, *render=None*, *skip_upload=False*, *as_files=False*, *memoize=True*, *extra_html=''*, *override_html_style=None*, *validate=True*)

Upload data to the Graphistry server and show as an iframe of it.

Uses the currently bound schema structure and visual encodings. Optional parameters override the current bindings.

When used in a notebook environment, will also show an iframe of the visualization.

**Parameters**

- **graph** (*Any*) – Edge table (pandas, arrow, cudf) or graph (NetworkX, IGraph).
- **nodes** (*Any*) – Nodes table (pandas, arrow, cudf)
- **name** (*str*) – Upload name.
- **description** (*str*) – Upload description.
- **render** (*bool*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL
- **skip_upload** (*bool*) – Return node/edge/bindings that would have been uploaded. By default, upload happens.
- **as_files** (*bool*) – Upload distinct node/edge files under the managed Files PI. Default off, will switch to default-on when stable.
- **memoize** (*bool*) – Tries to memoize pandas/cudf->arrow conversion, including skipping upload. Default on.
- **extra_html** (*Optional [str]*) – Allow injecting arbitrary HTML into the visualization iframe.
- **override_html_style** (*Optional [str]*) – Set fully custom style tag.
- **validate** (*Optional [bool]*) – Controls validations, including those for encodings.

**Example: Simple**

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .edges(es)
    .plot()
```

**Example: Shorthand**

```
import graphistry
es = pandas.DataFrame({'src': [0,1,2], 'dst': [1,2,0]})
graphistry
    .bind(source='src', destination='dst')
    .plot(es)
```

**privacy**(*mode=None*, *notify=None*, *invited_users=None*, *message=None*)

Set local sharing mode

**Parameters**

- **mode** (*Optional [Mode]*) – Either "private", "public", or inherit from global privacy()
- **notify** (*Optional [bool]*) – Whether to email the recipient(s) upon upload, defaults to global privacy()
- **invited_users** (*Optional [List]*) – List of recipients, where each is {"email": str, "action": str} and action is "10" (view) or "20" (edit), defaults to global privacy()
- **message** (*str | None*) – Email to send when notify=True

Requires an account with sharing capabilities.

Shared datasets will appear in recipients' galleries.

If mode is set to "private", only accounts in invited_users list can access. Mode "public" permits viewing by any user with the URL.

Action "10" (view) gives read access, while action "20" (edit) gives edit access, like changing the sharing mode.

When notify is true, uploads will trigger notification emails to invitees. Email will use visualization's ".name()"

When settings are not specified, they are inherited from the global graphistry.privacy() defaults

**Example: Limit visualizations to current user**

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.privacy()  # default uploads to mode="private"
g.plot()
```

**Example: Default to publicly viewable visualizations**

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
```

(continues on next page)

(continued from previous page)

```
g = h['graph']
#g = g.privacy(mode="public")  # can skip calling .privacy() for this␣
↪default
g.plot()
```

**Example: Default to sharing with select teammates, and keep notifications opt-in**

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.privacy(
    mode="private",
    invited_users=[
        {"email": "friend1@acme.org", "action": "10"}, # view
        {"email": "friend2@acme.org", "action": "20"}, # edit
    ],
    notify=False)
g.plot()
```

**Example: Keep visualizations public and email notifications upon upload**

```
import graphistry
graphistry.register(api=3, username='myuser', password='mypassword')

#Subsequent uploads default to using .privacy() settings
users_df = pd.DataFrame({'user': ['a','b','x'], 'boss': ['x', 'x', 'y']})
h = graphistry.hypergraph(users_df, direct=True)
g = h['graph']
g = g.name('my cool viz')  # For friendlier invitations
g = g.privacy(
    mode="public",
    invited_users=[
        {"email": "friend1@acme.org", "action": "10"}, # view
        {"email": "friend2@acme.org", "action": "20"}, # edit
    ],
    notify=True)
g.plot()
```

**prune_self_edges**()

**Return type**
Plottable

**reset_caches**()

Reset memoization caches

**scene_settings**(*menu=None*, *info=None*, *show_arrows=None*, *point_size=None*, *edge_curvature=None*, *edge_opacity=None*, *point_opacity=None*)

Set scene options. Additive over previous settings.

Corresponds to options at https://hub.graphistry.com/docs/api/1/rest/url/#urloptions

**Example: Hide arrows and straighten edges**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'boss': ['c','c','e','e']})
nodes = pd.DataFrame({
    'n': ['a', 'b', 'c', 'd', 'e'],
    'y': [1,   1,   2,   3,   4],
    'x': [1,   1,   0,   0,   0],
})
g = (graphistry
    .edges(edges, 's', 'd')
    .nodes(nodes, 'n')
    .scene_settings(show_arrows=False, edge_curvature=0.0)
g.plot()
```

**Parameters**

- **menu** (*bool / None*)
- **info** (*bool / None*)
- **show_arrows** (*bool / None*)
- **point_size** (*float / None*)
- **edge_curvature** (*float / None*)
- **edge_opacity** (*float / None*)
- **point_opacity** (*float / None*)

**settings**(*height=None*, *url_params={}*, *render=None*)

Specify iframe height and add URL parameter dictionary.

The library takes care of URI component encoding for the dictionary.

**Parameters**

- **height** (*int*) – Height in pixels.
- **url_params** (*dict*) – Dictionary of querystring parameters to append to the URL.
- **render** (*bool*) – Whether to render the visualization using the native notebook environment (default True), or return the visualization URL

**style**(*fg=None*, *bg=None*, *page=None*, *logo=None*)

Set general visual styles

See .bind() and .settings(url_params={}) for additional styling options, and addStyle() for another way to set the same attributes.

To facilitate reuse and replayable notebooks, the style() call is chainable. Invocation does not effect the old style: it instead returns a new Plotter instance with the new styles added to the existing ones. Both the old and new styles can then be used for different graphs.

style() will fully replace any defined parameter in the existing style settings, while addStyle() will merge over previous values

**Parameters**

- **fg** (*dict*) – Dictionary {'blendMode': str} of any valid CSS blend mode

- **bg** (*dict*) – Nested dictionary of page background properties. { 'color': str, 'gradient': {'kind': str, 'position': str, 'stops': list }, 'image': { 'url': str, 'width': int, 'height': int, 'blendMode': str }
- **logo** (*dict*) – Nested dictionary of logo properties. { 'url': str, 'autoInvert': bool, 'position': str, 'dimensions': { 'maxWidth': int, 'maxHeight': int }, 'crop': { 'top': int, 'left': int, 'bottom': int, 'right': int }, 'padding': { 'top': int, 'left': int, 'bottom': int, 'right': int}, 'style': str}
- **page** (*dict*) – Dictionary of page metadata settings. { 'favicon': str, 'title': str }

**Returns**
Plotter

**Return type**
Plotter

**Example: Chained merge - results in url and blendMode being set, while color is dropped**

```
g2 =  g.style(bg={'color': 'black'}, fg={'blendMode': 'screen'})
g3 = g2.style(bg={'image': {'url': 'http://site.com/watermark.png'}})
```

**Example: Gradient background**

```
g.style(bg={'gradient': {'kind': 'linear', 'position': 45, 'stops': [['rgb(0,
↪0,0)', '0%'], ['rgb(255,255,255)', '100%']]}})
```

**Example: Page settings**

```
g.style(page={'title': 'Site - {{ name }}', 'favicon': 'http://site.com/logo.
↪ico'})
```

**tigergraph**(*protocol='http'*, *server='localhost'*, *web_port=14240*, *api_port=9000*, *db=None*, *user='tigergraph'*, *pwd='tigergraph'*, *verbose=False*)

Register Tigergraph connection setting defaults

**Parameters**

- **protocol** (*Optional [str]*) – Protocol used to contact the database.
- **server** (*Optional [str]*) – Domain of the database
- **web_port** (*Optional [int]*)
- **api_port** (*Optional [int]*)
- **db** (*Optional [str]*) – Name of the database
- **user** (*Optional [str]*)
- **pwd** (*Optional [str]*)
- **verbose** (*Optional [bool]*) – Whether to print operations

**Returns**
Plotter

**Return type**
Plotter

**Example: Standard**

```
import graphistry
tg = graphistry.tigergraph(protocol='https', server='acme.com', db='my_db',␣
↪user='alice', pwd='tigergraph2')
```

**to_cugraph**(*directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

Convert current graph to a cugraph.Graph object

To assign an edge weight, use *g.bind(edge_weight='some_col').to_cugraph()*

Load from pandas, cudf, or dask_cudf DataFrames

**Parameters**

- **directed** (*bool*)
- **include_nodes** (*bool*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)
- **kind** (*Literal['Graph', 'MultiGraph', 'BiPartiteGraph']*)

**to_igraph**(*directed=True*, *use_vids=False*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*)

Convert current item to igraph Graph . See examples in from_igraph.

**Parameters**

- **directed** (*bool*) – Whether to create a directed graph (default True)
- **include_nodes** (*bool*) – Whether to ingest the nodes table, if it exists (default True)
- **node_attributes** (*Optional[List[str]]*) – Which node attributes to load, None means all (default None)
- **edge_attributes** (*Optional[List[str]]*) – Which edge attributes to load, None means all (default None)
- **use_vids** (*bool*) – Whether to interpret IDs as igraph vertex IDs, which must be non-negative integers (default False)

graphistry.PlotterBase.**maybe_cudf**()

graphistry.PlotterBase.**maybe_dask_cudf**()

graphistry.PlotterBase.**maybe_dask_dataframe**()

graphistry.PlotterBase.**maybe_spark**()

## 3.8.2 GFQL API Reference

### 3.8.2.1 GFQL Chain

**class** graphistry.compute.chain.**Chain**(*chain*)

Bases: ASTSerializable

**Parameters**
**chain** (*List[ASTObject]*)

**classmethod from_json**(*d*)

Convert a JSON AST into a list of ASTObjects

**Parameters**
**d** (*Dict[str, None | bool | str | float | int | List[None | bool | str | float | int | List[JSONVal] | Dict[str, JSONVal]] | Dict[str, None | bool | str | float | int | List[JSONVal] | Dict[str, JSONVal]]]*)

**Return type**
Chain

**to_json**(*validate=True*)

Convert a list of ASTObjects into a JSON AST

**Return type**
*Dict*[str, None | bool | str | float | int | *List*[None | bool | str | float | int | *List*[JSONVal] | *Dict*[str, JSONVal]] | *Dict*[str, None | bool | str | float | int | *List*[JSONVal] | *Dict*[str, JSONVal]]]

**validate**()

**Return type**
None

graphistry.compute.chain.**chain**(*self*, *ops*, *engine=EngineAbstract.AUTO*)

Chain a list of ASTObject (node/edge) traversal operations

Return subgraph of matches according to the list of node & edge matchers If any matchers are named, add a correspondingly named boolean-valued column to the output

For direct calls, exposes convenience *List[ASTObject]*. Internal operational should prefer *Chain*.

Use *engine='cudf'* to force automatic GPU acceleration mode

**Parameters**

- **ops** (*List[ASTObject] | Chain*) – List[ASTObject] Various node and edge matchers
- **self** (Plottable)
- **engine** (*EngineAbstract | str*)

**Returns**
Plotter

**Return type**
Plotter

**Example: Find nodes of some type**

```
from graphistry.ast import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
```

**Example: Find 2-hop edge sequences with some attribute**

```
from graphistry.ast import e_forward

g_2_hops = g.chain([ e_forward({"interesting": True}, hops=2) ])
g_2_hops.plot()
```

**Example: Find any node 1-2 hops out from another node, and label each hop**

```
from graphistry.ast import n, e_undirected

g_2_hops = g.chain([ n({g._node: "a"}), e_undirected(name="hop1"), e_
↪undirected(name="hop2") ])
print('# first-hop edges:', len(g_2_hops._edges[ g_2_hops._edges.hop1 == True ]))
```

**Example: Transaction nodes between two kinds of risky nodes**

```
from graphistry.ast import n, e_forward, e_reverse

g_risky = g.chain([
    n({"risk1": True}),
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

**Example: Filter by multiple node types at each step using is_in**

```
from graphistry.ast import n, e_forward, e_reverse, is_in

g_risky = g.chain([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

**Example: Run with automatic GPU acceleration**

```
import cudf
import graphistry

e_gdf = cudf.from_pandas(df)
g1 = graphistry.edges(e_gdf, 's', 'd')
g2 = g1.chain([ ... ])
```

**Example: Run with automatic GPU acceleration, and force GPU mode**

```
import cudf
import graphistry

e_gdf = cudf.from_pandas(df)
g1 = graphistry.edges(e_gdf, 's', 'd')
g2 = g1.chain([ ... ], engine='cudf')
```

graphistry.compute.chain.**combine_steps**(*g*, *kind*, *steps*, *engine*)

Collect nodes and edges, taking care to deduplicate and tag any names

**Parameters**

- **g** (Plottable)
- **kind** (*str*)
- **steps** (*List[Tuple[ASTObject, Plottable]]*)
- **engine** (*Engine*)

**Return type**

*Any*

1

#### 3.8.2.2 GFQL Hop

graphistry.compute.hop.**hop**(*self*, *nodes=None*, *hops=1*, *to_fixed_point=False*, *direction='forward'*, *edge_match=None*, *source_node_match=None*, *destination_node_match=None*, *source_node_query=None*, *destination_node_query=None*, *edge_query=None*, *return_as_wave_front=False*, *target_wave_front=None*, *engine=EngineAbstract.AUTO*)

Given a graph and some source nodes, return subgraph of all paths within k-hops from the sources

This can be faster than the equivalent chain([...]) call that wraps it with additional steps

See chain() examples for examples of many of the parameters

g: Plotter nodes: dataframe with id column matching g._node. None signifies all nodes (default). hops: consider paths of length 1 to 'hops' steps, if any (default 1). to_fixed_point: keep hopping until no new nodes are found (ignores hops) direction: 'forward', 'reverse', 'undirected' edge_match: dict of kv-pairs to exact match (see also: filter_edges_by_dict) source_node_match: dict of kv-pairs to match nodes before hopping (including intermediate) destination_node_match: dict of kv-pairs to match nodes after hopping (including intermediate) source_node_query: dataframe query to match nodes before hopping (including intermediate) destination_node_query: dataframe query to match nodes after hopping (including intermediate) edge_query: dataframe query to match edges before hopping (including intermediate) return_as_wave_front: Exclude starting node(s) in return, returning only encountered nodes target_wave_front: Only consider these nodes + self._nodes for reachability engine: 'auto', 'pandas', 'cudf' (GPU)

**Parameters**

- **self** (Plottable)
- **nodes** (*Any | None*)
- **hops** (*int | None*)
- **to_fixed_point** (*bool*)

- **direction** (*str*)
- **edge_match** (*dict | None*)
- **source_node_match** (*dict | None*)
- **destination_node_match** (*dict | None*)
- **source_node_query** (*str | None*)
- **destination_node_query** (*str | None*)
- **edge_query** (*str | None*)
- **target_wave_front** (*Any | None*)
- **engine** (*EngineAbstract | str*)

**Return type**
Plottable

graphistry.compute.hop.**query_if_not_none**(*query*, *df*)

**Parameters**

- **query** (*str | None*)
- **df** (*Any*)

**Return type**
*Any*

2

#### 3.8.2.3 GFQL Predicates

##### ASTPredicate

**class** graphistry.compute.predicates.ASTPredicate.**ASTPredicate**

Bases: ASTSerializable

Internal, not intended for use outside of this module. These are fancy columnar predicates used in {k: v, ...} node/edge df matching when going beyond primitive equality

##### Categorical

**class** graphistry.compute.predicates.categorical.**Duplicated**(*keep='first'*)

Bases: ASTPredicate

**Parameters**
**keep** (*Literal['first', 'last', False]*)

**validate**()

**Return type**
None

graphistry.compute.predicates.categorical.**duplicated**(*keep='first'*)

Return whether a given value is duplicated

**Parameters**
**keep** (*Literal['first', 'last', False]*)

**Return type**
*Duplicated*

### Is In

**class** graphistry.compute.predicates.is_in.**IsIn**(*options*)

Bases: ASTPredicate

**Parameters**
**options** (*List[Any]*)

**validate**()

**Return type**
None

graphistry.compute.predicates.is_in.**is_in**(*options*)

**Parameters**
**options** (*List[Any]*)

**Return type**
*IsIn*

### Numeric

**class** graphistry.compute.predicates.numeric.**Between**(*lower*, *upper*, *inclusive=True*)

Bases: ASTPredicate

**Parameters**

- **lower** (*float*)
- **upper** (*float*)
- **inclusive** (*bool*)

**validate**()

**Return type**
None

**class** graphistry.compute.predicates.numeric.**EQ**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

**class** graphistry.compute.predicates.numeric.**GE**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

class graphistry.compute.predicates.numeric.**GT**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

class graphistry.compute.predicates.numeric.**IsNA**

Bases: ASTPredicate

class graphistry.compute.predicates.numeric.**LE**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

class graphistry.compute.predicates.numeric.**LT**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

class graphistry.compute.predicates.numeric.**NE**(*val*)

Bases: NumericASTPredicate

**Parameters**
**val** (*float*)

class graphistry.compute.predicates.numeric.**NotNA**

Bases: ASTPredicate

class graphistry.compute.predicates.numeric.**NumericASTPredicate**(*val*)

Bases: ASTPredicate

**Parameters**
**val** (*int | float*)

**validate**()

**Return type**
None

graphistry.compute.predicates.numeric.**between**(*lower*, *upper*, *inclusive=True*)

Return whether a given value is between a lower and upper threshold

**Parameters**

- **lower** (*float*)
- **upper** (*float*)
- **inclusive** (*bool*)

**Return type**
*Between*

graphistry.compute.predicates.numeric.**eq**(*val*)

Return whether a given value is equal to a threshold

**Parameters**
**val** (*float*)

**Return type**
*EQ*

graphistry.compute.predicates.numeric.**ge**(*val*)

Return whether a given value is greater than or equal to a threshold

**Parameters**
**val** (*float*)

**Return type**
*GE*

graphistry.compute.predicates.numeric.**gt**(*val*)

Return whether a given value is greater than a threshold

**Parameters**
**val** (*float*)

**Return type**
*GT*

graphistry.compute.predicates.numeric.**isna**()

Return whether a given value is NA

**Return type**
*IsNA*

graphistry.compute.predicates.numeric.**le**(*val*)

Return whether a given value is less than or equal to a threshold

**Parameters**
**val** (*float*)

**Return type**
*LE*

graphistry.compute.predicates.numeric.**lt**(*val*)

Return whether a given value is less than a threshold

**Parameters**
**val** (*float*)

**Return type**
*LT*

graphistry.compute.predicates.numeric.**ne**(*val*)

Return whether a given value is not equal to a threshold

**Parameters**
**val** (*float*)

**Return type**
*NE*

graphistry.compute.predicates.numeric.**notna**()

Return whether a given value is not NA

**Return type**
*NotNA*

### String Predicates

class graphistry.compute.predicates.str.**Contains**(*pat*, *case=True*, *flags=0*, *na=None*, *regex=True*)

Bases: ASTPredicate

**Parameters**

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)
- **regex** (*bool*)

validate()

**Return type**
None

class graphistry.compute.predicates.str.**Endswith**(*pat*, *na=None*)

Bases: ASTPredicate

**Parameters**

- **pat** (*str*)
- **na** (*str | None*)

validate()

**Return type**
None

class graphistry.compute.predicates.str.**IsAlnum**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsAlpha**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsDecimal**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsDigit**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsLower**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsNull**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsNumeric**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsSpace**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsTitle**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**IsUpper**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**Match**(*pat*, *case=True*, *flags=0*, *na=None*)

Bases: ASTPredicate

**Parameters**

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)

**validate**()

**Return type**
None

class graphistry.compute.predicates.str.**NotNull**

Bases: ASTPredicate

class graphistry.compute.predicates.str.**Startswith**(*pat*, *na=None*)

Bases: ASTPredicate

**Parameters**

- **pat** (*str*)
- **na** (*str | None*)

**validate**()

**Return type**
None

graphistry.compute.predicates.str.**contains**(*pat*, *case=True*, *flags=0*, *na=None*, *regex=True*)

Return whether a given pattern or regex is contained within a string

**Parameters**

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)
- **regex** (*bool*)

**Return type**
*Contains*

graphistry.compute.predicates.str.**endswith**(*pat*, *na=None*)

**Parameters**

- **pat** (*str*)
- **na** (*str | None*)

**Return type**
*Endswith*

graphistry.compute.predicates.str.**isalnum**()

Return whether a given string is alphanumeric

**Return type**
*IsAlnum*

graphistry.compute.predicates.str.**isalpha**()

Return whether a given string is alphabetic

**Return type**
*IsAlpha*

graphistry.compute.predicates.str.**isdecimal**()

Return whether a given string is decimal

**Return type**
*IsDecimal*

graphistry.compute.predicates.str.**isdigit**()

Return whether a given string is numeric

**Return type**
*IsDigit*

graphistry.compute.predicates.str.**islower**()

Return whether a given string is lowercase

**Return type**
*IsLower*

graphistry.compute.predicates.str.**isnull**()

Return whether a given string is null

**Return type**
*IsNull*

graphistry.compute.predicates.str.**isnumeric**()

Return whether a given string is numeric

**Return type**
*IsNumeric*

graphistry.compute.predicates.str.**isspace**()

Return whether a given string is whitespace

**Return type**
*IsSpace*

graphistry.compute.predicates.str.**istitle**()

Return whether a given string is title case

**Return type**
*IsTitle*

graphistry.compute.predicates.str.**isupper**()

Return whether a given string is uppercase

**Return type**
*IsUpper*

graphistry.compute.predicates.str.**match**(*pat*, *case=True*, *flags=0*, *na=None*)

Return whether a given pattern is at the start of a string

**Parameters**

- **pat** (*str*)
- **case** (*bool*)
- **flags** (*int*)
- **na** (*bool | None*)

**Return type**
*Match*

graphistry.compute.predicates.str.**notnull**()

Return whether a given string is not null

**Return type**
*NotNull*

graphistry.compute.predicates.str.**startswith**(*pat*, *na=None*)

Return whether a given pattern is at the start of a string

**Parameters**

- **pat** (*str*)
- **na** (*str | None*)

**Return type**
*Startswith*

### Temporal

**class** graphistry.compute.predicates.temporal.**IsLeapYear**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsMonthEnd**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsMonthStart**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsQuarterEnd**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsQuarterStart**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsYearEnd**

Bases: ASTPredicate

**class** graphistry.compute.predicates.temporal.**IsYearStart**

Bases: ASTPredicate

graphistry.compute.predicates.temporal.**is_leap_year**()

Return whether a given value is a leap year

**Return type**
*IsLeapYear*

graphistry.compute.predicates.temporal.**is_month_end**()

Return whether a given value is a month end

**Return type**
*IsMonthEnd*

graphistry.compute.predicates.temporal.**is_month_start**()

Return whether a given value is a month start

**Return type**
*IsMonthStart*

graphistry.compute.predicates.temporal.**is_quarter_end**()

Return whether a given value is a quarter end

**Return type**
*IsQuarterEnd*

graphistry.compute.predicates.temporal.**is_quarter_start**()

Return whether a given value is a quarter start

**Return type**
*IsQuarterStart*

graphistry.compute.predicates.temporal.**is_year_end**()

Return whether a given value is a year end

**Return type**
*IsYearEnd*

graphistry.compute.predicates.temporal.**is_year_start**()

Return whether a given value is a year start

**Return type**
*IsYearStart*

### 3.8.3 Compute API Reference

#### 3.8.3.1 ComputeMixin module

**class** graphistry.compute.ComputeMixin.**ComputeMixin**(**args*, ***kwargs*)

Bases: `object`

**chain**(**args*, ***kwargs*)

Chain a list of ASTObject (node/edge) traversal operations

Return subgraph of matches according to the list of node & edge matchers If any matchers are named, add a correspondingly named boolean-valued column to the output

For direct calls, exposes convenience *List[ASTObject]*. Internal operational should prefer *Chain*.

Use *engine='cudf'* to force automatic GPU acceleration mode

**Parameters**
ops – List[ASTObject] Various node and edge matchers

**Returns**
Plotter

**Return type**
Plotter

**Example: Find nodes of some type**

```
from graphistry.ast import n

people_nodes_df = g.chain([ n({"type": "person"}) ])._nodes
```

**Example: Find 2-hop edge sequences with some attribute**

```
from graphistry.ast import e_forward

g_2_hops = g.chain([ e_forward({"interesting": True}, hops=2) ])
g_2_hops.plot()
```

**Example: Find any node 1-2 hops out from another node, and label each hop**

```
from graphistry.ast import n, e_undirected

g_2_hops = g.chain([ n({g._node: "a"}), e_undirected(name="hop1"), e_
↪undirected(name="hop2") ])
print('# first-hop edges:', len(g_2_hops._edges[ g_2_hops._edges.hop1 == True␣
↪]))
```

**Example: Transaction nodes between two kinds of risky nodes**

```
from graphistry.ast import n, e_forward, e_reverse

g_risky = g.chain([
    n({"risk1": True}),
    e_forward(to_fixed=True),
    n({"type": "transaction"}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

**Example: Filter by multiple node types at each step using is_in**

```
from graphistry.ast import n, e_forward, e_reverse, is_in

g_risky = g.chain([
    n({"type": is_in(["person", "company"])}),
    e_forward({"e_type": is_in(["owns", "reviews"])}, to_fixed=True),
    n({"type": is_in(["transaction", "account"])}, name="hit"),
    e_reverse(to_fixed=True),
    n({"risk2": True})
])
print('# hits:', len(g_risky._nodes[ g_risky._nodes.hit ]))
```

**Example: Run with automatic GPU acceleration**

```
import cudf
import graphistry

e_gdf = cudf.from_pandas(df)
g1 = graphistry.edges(e_gdf, 's', 'd')
g2 = g1.chain([ ... ])
```

**Example: Run with automatic GPU acceleration, and force GPU mode**

```
import cudf
import graphistry

e_gdf = cudf.from_pandas(df)
g1 = graphistry.edges(e_gdf, 's', 'd')
g2 = g1.chain([ ... ], engine='cudf')
```

**collapse**(*node*, *attribute*, *column*, *self_edges=False*, *unwrap=False*, *verbose=False*)

Topology-aware collapse by given column attribute starting at *node*

Traverses directed graph from start node *node* and collapses clusters of nodes that share the same property so that topology is preserved.

**Parameters**

- **node** (*str / int*) – start *node* to begin traversal
- **attribute** (*str / int*) – the given *attribute* to collapse over within *column*
- **column** (*str / int*) – the *column* of nodes DataFrame that contains *attribute* to collapse over
- **self_edges** (*bool*) – whether to include self edges in the collapsed graph
- **unwrap** (*bool*) – whether to unwrap the collapsed graph into a single node
- **verbose** (*bool*) – whether to print out collapse summary information

:returns:A new Graphistry instance with nodes and edges DataFrame containing collapsed nodes and edges given by column attribute – nodes and edges DataFrames contain six new columns *collapse_{node / edges}* and *final_{node / edges}*, while original (node, src, dst) columns are left untouched :rtype: Plottable

**drop_nodes**(*nodes*)

return g with any nodes/edges involving the node id series removed

**filter_edges_by_dict**(**args*, ***kwargs*)

filter edges to those that match all values in filter_dict

**filter_nodes_by_dict**(**args*, ***kwargs*)

filter nodes to those that match all values in filter_dict

**get_degrees**(*col='degree'*, *degree_in='degree_in'*, *degree_out='degree_out'*)

Decorate nodes table with degree info

Edges must be dataframe-like: pandas, cudf, ...

Parameters determine generated column names

Warning: Self-cycles are currently double-counted. This may change.

**Example: Generate degree columns**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
print(g._nodes)  # None
g2 = g.get_degrees()
print(g2._nodes)  # pd.DataFrame with 'id', 'degree', 'degree_in',
↪'degree_out'
```

**Parameters**

- **col** (*str*)
- **degree_in** (*str*)
- **degree_out** (*str*)

**get_indegrees**(*col='degree_in'*)

See get_degrees

**Parameters**
**col** (*str*)

**get_outdegrees**(*col='degree_out'*)

See get_degrees

**Parameters**
**col** (*str*)

**get_topological_levels**(*level_col='level'*, *allow_cycles=True*, *warn_cycles=True*, *remove_self_loops=True*)

Label nodes on column level_col based on topological sort depth Supports pandas + cudf, using parallelism within each level computation Options: * allow_cycles: if False and detects a cycle, throw ValueException, else break cycle by picking a lowest-in-degree node * warn_cycles: if True and detects a cycle, proceed with a warning * remove_self_loops: preprocess by removing self-cycles. Avoids allow_cycles=False, warn_cycles=True messages.

Example:

edges_df = gpd.DataFrame({'s': ['a', 'b', 'c', 'd'],'d': ['b', 'c', 'e', 'e']}) g = graphistry.edges(edges_df, 's', 'd') g2 = g.get_topological_levels() g2._nodes.info() # pd.DataFrame with | 'id' , 'level' |

**Parameters**

- **level_col** (*str*)
- **allow_cycles** (*bool*)
- **warn_cycles** (*bool*)
- **remove_self_loops** (*bool*)

**Return type**
Plottable

**hop**(**args*, ***kwargs*)

Given a graph and some source nodes, return subgraph of all paths within k-hops from the sources

This can be faster than the equivalent chain([...]) call that wraps it with additional steps

See chain() examples for examples of many of the parameters

g: Plotter nodes: dataframe with id column matching g._node. None signifies all nodes (default). hops: consider paths of length 1 to 'hops' steps, if any (default 1). to_fixed_point: keep hopping until no new nodes are found (ignores hops) direction: 'forward', 'reverse', 'undirected' edge_match: dict of kv-pairs to exact match (see also: filter_edges_by_dict) source_node_match: dict of kv-pairs to match nodes before hopping (including intermediate) destination_node_match: dict of kv-pairs to match nodes after hopping (including intermediate) source_node_query: dataframe query to match nodes before hopping (including intermediate) destination_node_query: dataframe query to match nodes after hopping (including intermediate) edge_query: dataframe query to match edges before hopping (including intermediate) return_as_wave_front: Exclude starting node(s) in return, returning only encountered nodes target_wave_front: Only consider these nodes + self._nodes for reachability engine: 'auto', 'pandas', 'cudf' (GPU)

**keep_nodes**(*nodes*)

Limit nodes and edges to those selected by parameter nodes For edges, both source and destination must be in nodes Nodes can be a list or series of node IDs, or a dictionary When a dictionary, each key corresponds to a node column, and nodes will be included when all match

**materialize_nodes**(*reuse=True*, *engine=EngineAbstract.AUTO*)

Generate g._nodes based on g._edges

Uses g._node for node id if exists, else 'id'

Edges must be dataframe-like: cudf, pandas, ...

When reuse=True and g._nodes is not None, use it

**Example: Generate nodes**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
print(g._nodes)  # None
g2 = g.materialize_nodes()
print(g2._nodes)  # pd.DataFrame
```

**Parameters**

- **reuse** (*bool*)
- **engine** (*EngineAbstract | str*)

**Return type**
Plottable

**prune_self_edges**()

#### 3.8.3.2 Collapse

graphistry.compute.collapse.**check_default_columns_present_and_coerce_to_string**(*g*)

Helper to set COLLAPSE columns to nodes and edges dataframe, while converting src, dst, node to dtype(str) :param g: graphistry instance

**Returns**
graphistry instance

**Parameters**
**g** (Plottable)

graphistry.compute.collapse.**check_has_set**(*ndf*, *parent*, *child*)

graphistry.compute.collapse.**collapse_algo**(*g*, *child*, *parent*, *attribute*, *column*, *seen*)

Basically candy crush over graph properties in a topology aware manner

Checks to see if child node has desired property from parent, we will need to check if (start_node=parent: has_attribute , children nodes: has_attribute) by case (T, T), (F, T), (T, F) and (F, F),we start recursive collapse (or not) on the children, reassigning nodes and edges.

if (T, T), append children nodes to start_node, re-assign the name of the node, and update the edge table with new name,

if (F, T) start k-(potentially new) super nodes, with k the number of children of start_node. Start node keeps k outgoing edges.

if (T, F) it is the end of the cluster, and we keep new node as is; keep going

if (F, F); keep going

**Parameters**

- **seen** (*dict*)
- **g** (Plottable) – graphistry instance
- **child** (*str / int*) – child node to start traversal, for first traversal, set child=parent or vice versa.
- **parent** (*str / int*) – parent node to start traversal, in main call, this is set to child.
- **attribute** (*str / int*) – attribute to collapse by
- **column** (*str / int*) – column in nodes dataframe to collapse over.

**Returns**

graphistry instance with collapsed nodes.

graphistry.compute.collapse.**collapse_by**(*self*, *parent*, *start_node*, *attribute*, *column*, *seen*, *self_edges=False*, *unwrap=False*, *verbose=True*)

Main call in collapse.py, collapses nodes and edges by attribute, and returns normalized graphistry object.

**Parameters**

- **self** (Plottable) – graphistry instance
- **parent** (*str / int*) – parent node to start traversal, in main call, this is set to child.
- **start_node** (*str / int*)
- **attribute** (*str / int*) – attribute to collapse by
- **column** (*str / int*) – column in nodes dataframe to collapse over.
- **seen** (*dict*) – dict of previously collapsed pairs – {n1, n2) is seen as different from (n2, n1)
- **verbose** (*bool*) – bool, default True
- **self_edges** (*bool*)
- **unwrap** (*bool*)

**Return type**
Plottable

:returns graphistry instance with collapsed and normalized nodes.

graphistry.compute.collapse.**collapse_nodes_and_edges**(*g*, *parent*, *child*)

Asserts that parent and child node in ndf should be collapsed into super node. Sets new ndf with COLLAPSE nodes in graphistry instance g

# this asserts that we SHOULD merge parent and child as super node # outside logic controls when that is the case # for example, it assumes parent is already in cluster keys of COLLAPSE node

**Parameters**

- **g** (Plottable) – graphistry instance
- **parent** (*str / int*) – *node* with *attribute* in *column*
- **child** (*str / int*) – *node* with *attribute* in *column*

**Returns**
graphistry instance

graphistry.compute.collapse.**get_children**(*g*, *node_id*, *hops=1*)

Helper that gets children at k-hops from node *node_id*

:returns graphistry instance of hops

**Parameters**

- **g** (Plottable)
- **node_id** (*str / int*)
- **hops** (*int*)

graphistry.compute.collapse.**get_cluster_store_keys**(*ndf*, *node*)

Main innovation in finding and adding to super node. Checks if node is a segment in any collapse_node in COLLAPSE column of nodes DataFrame

**Parameters**

- **ndf** (*DataFrame*) – node DataFrame
- **node** (*str / int*) – node to find

**Returns**
DataFrame of bools of where *wrap_key(node)* exists in COLLAPSE column

graphistry.compute.collapse.**get_edges_in_out_cluster**(*g*, *node_id*, *attribute*, *column*, *directed=True*)

Traverses children of *node_id* and separates them into incluster and outcluster sets depending if they have *attribute* in node DataFrame *column*

**Parameters**

- **g** (Plottable) – graphistry instance
- **node_id** (*str / int*) – *node* with *attribute* in *column*
- **attribute** (*str / int*) – *attribute* to collapse in *column* over
- **column** (*str / int*) – *column* to collapse over

- **directed** (*bool*)

graphistry.compute.collapse.**get_edges_of_node**(*g*, *node_id*, *outgoing_edges=True*, *hops=1*)

Gets edges of node at k-hops from node

**Parameters**

- **g** (Plottable) – graphistry instance
- **node_id** (*str / int*) – *node* to find edges from
- **outgoing_edges** (*bool*) – bool, if true, finds all outgoing edges of *node*, default True
- **hops** (*int*) – the number of hops from *node* to take, default = 1

**Returns**
DataFrame of edges

graphistry.compute.collapse.**get_new_node_name**(*ndf*, *parent*, *child*)

If child in cluster group, melts name, else makes new parent_name from parent, child

**Parameters**

- **ndf** (*DataFrame*) – node DataFrame
- **parent** (*str / int*) – *node* with *attribute* in *column*
- **child** (*str / int*) – *node* with *attribute* in *column*

**Return type**
str

:returns new_parent_name

graphistry.compute.collapse.**has_edge**(*g*, *n1*, *n2*, *directed=True*)

Checks if *n1* and *n2* share an (directed or not) edge

**Parameters**

- **g** (Plottable) – graphistry instance
- **n1** (*str / int*) – *node* to check if has edge to *n2*
- **n2** (*str / int*) – *node* to check if has edge to *n1*
- **directed** (*bool*) – bool, if True, checks only outgoing edges from *n1*->`n2`, else finds undirected edges

**Returns**
bool, if edge exists between *n1* and *n2*

**Return type**
bool

graphistry.compute.collapse.**has_property**(*g*, *ref_node*, *attribute*, *column*)

Checks if ref_node is in node dataframe in column with attribute :param attribute: :param column: :param g: graphistry instance :param ref_node: *node* to check if it as *attribute* in *column*

**Returns**
bool

**Parameters**

- **g** (Plottable)
- **ref_node** (*str / int*)

- **attribute** (*str / int*)
- **column** (*str / int*)

**Return type**
bool

graphistry.compute.collapse.**in_cluster_store_keys**(*ndf*, *node*)

checks if node is in collapse_node in COLLAPSE column of nodes DataFrame

**Parameters**

- **ndf** (*DataFrame*) – nodes DataFrame
- **node** (*str / int*) – node to find

**Returns**
bool

**Return type**
bool

graphistry.compute.collapse.**melt**(*ndf*, *node*)

Reduces node if in cluster store, otherwise passes it through. ex:

node = "4" will take any sequence from get_cluster_store_keys, "1 2 3", "4 3 6" and returns "1 2 3 4 6" when they have a common entry (3).

:param ndf, node DataFrame :param node: node to melt :returns new_parent_name of super node

**Parameters**

- **ndf** (*DataFrame*)
- **node** (*str / int*)

**Return type**
str

graphistry.compute.collapse.**normalize_graph**(*g*, *self_edges=False*, *unwrap=False*)

Final step after collapse traversals are done, removes duplicates and moves COLLAPSE columns into respective(node, src, dst) columns of node, edges dataframe from Graphistry instance g.

**Parameters**

- **g** (Plottable) – graphistry instance
- **self_edges** (*bool*) – bool, whether to keep duplicates from ndf, edf, default False
- **unwrap** (*bool*) – bool, whether to unwrap node text with ~, default True

**Returns**
final graphistry instance

**Return type**
Plottable

graphistry.compute.collapse.**reduce_key**(*key*)

Takes "1 1 2 1 2 3" -> "1 2 3

**Parameters**
**key** (*str / int*) – node name

**Returns**
new node name with duplicates removed

**Return type**
str

graphistry.compute.collapse.**unpack**(*g*)

Helper method that unpacks graphistry instance

ex:

ndf, edf, src, dst, node = unpack(g)

**Parameters**
**g** (Plottable) – graphistry instance

**Returns**
node DataFrame, edge DataFrame, source column, destination column, node column

graphistry.compute.collapse.**unwrap_key**(*name*)

Unwraps node name: ~name~ -> name

**Parameters**
**name** (*str / int*) – node to unwrap

**Returns**
unwrapped node name

**Return type**
str

graphistry.compute.collapse.**wrap_key**(*name*)

Wraps node name -> ~name~

**Parameters**
**name** (*str / int*) – node name

**Returns**
wrapped node name

**Return type**
str

#### 3.8.3.3 Conditional

**class** graphistry.compute.conditional.**ConditionalMixin**(**args*, ***kwargs*)

Bases: object

**conditional_graph**(*x*, *given*, *kind='nodes'*, **args*, ***kwargs*)

conditional_graph – p(x|given) = p(x, given) / p(given)

Useful for finding the conditional probability of a node or edge attribute

returned dataframe sums to 1 on each column

**Parameters**

- **x** – target column
- **given** – the dependent column
- **kind** – 'nodes' or 'edges'
- **args/kwargs** – additional arguments for g.bind(...)

**Returns**

a graphistry instance with the conditional graph edges weighted by the conditional probability. edges are between *x* and *given*, keep in mind that g._edges.columns = [given, x, _probs]

`conditional_probs`(*x*, *given*, *kind='nodes'*, *how='index'*)

Produces a Dense Matrix of the conditional probability of x given y

**Args:**

x: the column variable of interest given the column y=given given : the variabe to fix constant df pd.DataFrame: dataframe how (str, optional): One of 'column' or 'index'. Defaults to 'index'. kind (str, optional): 'nodes' or 'edges'. Defaults to 'nodes'.

**Returns:**

pd.DataFrame: the conditional probability of x given the column y as dense array like dataframe

`graphistry.compute.conditional.conditional_probability`(*x*, *given*, *df*)

**conditional probability function over categorical variables**

p(x | given) = p(x, given)/p(given)

**Args:**

x: the column variable of interest given the column 'given' given: the variabe to fix constant df: dataframe with columns [given, x]

**Returns:**

pd.DataFrame: the conditional probability of x given the column 'given'

**Parameters**

**df** (*DataFrame*)

`graphistry.compute.conditional.probs`(*x*, *given*, *df*, *how='index'*)

Produces a Dense Matrix of the conditional probability of x given *y=given*

**Args:**

x: the column variable of interest given the column 'y' given : the variabe to fix constant df pd.DataFrame: dataframe how (str, optional): One of 'column' or 'index'. Defaults to 'index'.

**Returns:**

pd.DataFrame: the conditional probability of x given the column 'y' as dense array like dataframe

**Parameters**

**df** (*DataFrame*)

#### 3.8.3.4 Filter by Dictionary

`graphistry.compute.filter_by_dict.filter_by_dict`(*df*, *filter_dict=None*, *engine=EngineAbstract.AUTO*)

return df where rows match all values in filter_dict

**Parameters**

- **df** (*Any*)
- **filter_dict** (*dict | None*)
- **engine** (*EngineAbstract | str*)

**Return type**
*Any*

graphistry.compute.filter_by_dict.**filter_edges_by_dict**(*self*, *filter_dict*, *engine=EngineAbstract.AUTO*)

filter edges to those that match all values in filter_dict

**Parameters**

- **self** (Plottable)
- **filter_dict** (*dict*)
- **engine** (*EngineAbstract | str*)

**Return type**
Plottable

graphistry.compute.filter_by_dict.**filter_nodes_by_dict**(*self*, *filter_dict*, *engine=EngineAbstract.AUTO*)

filter nodes to those that match all values in filter_dict

**Parameters**

- **self** (Plottable)
- **filter_dict** (*dict*)
- **engine** (*EngineAbstract | str*)

**Return type**
Plottable

### 3.8.4 AI

graphistry['ai'] provides a set of utilities for AI and machine learning workflows on graphs, with optional GPU support

#### 3.8.4.1 Featurize

class graphistry.feature_utils.**Embedding**(*df*)

Bases: object

Generates random embeddings of a given dimension that aligns with the index of the dataframe

**Parameters**
**df** (*DataFrame*)

**fit**(*n_dim*)

**Parameters**
**n_dim** (*int*)

**fit_transform**(*n_dim*)

**Parameters**
**n_dim** (*int*)

**transform**(*ids*)

**Return type**
*DataFrame*

**class** graphistry.feature_utils.**FastEncoder**(*df*, *y=None*, *kind='nodes'*)

Bases: object

**fit**(*src=None*, *dst=None*, **args*, ***kwargs*)

**fit_transform**(*src=None*, *dst=None*, **args*, ***kwargs*)

**scale**(*X=None*, *y=None*, *return_pipeline=False*, **args*, ***kwargs*)

Fits new scaling functions on df, y via args-kwargs

**Example:**

```
from graphisty.features import SCALERS, SCALER_OPTIONS
print(SCALERS)
g = graphistry.nodes(df)
# set a scaling strategy for features and targets -- umap uses those and␣
↪produces different results depending.
g2 = g.umap(use_scaler='standard', use_scaler_target=None)

# later if you want to scale new data, you can do so
X, y = g2.transform(df, df, scaled=False)  # unscaled transformer output
# now scale with new settings
X_scaled, y_scaled = g2.scale(X, y, use_scaler='minmax', use_scaler_target=
↪'kbins', n_bins=5)
# fit some other pipeline
clf.fit(X_scaled, y_scaled)
```

args:

```
;X: pd.DataFrame of features
:y: pd.DataFrame of target features
:kind: str, one of 'nodes' or 'edges'
*args, **kwargs: passed to smart_scaler pipeline
```

**returns:**
scaled X, y

**transform**(*df*, *ydf=None*)

Raw transform, no scaling.

**transform_scaled**(*df*, *ydf=None*, *scaling_pipeline=None*, *scaling_pipeline_target=None*)

**class** graphistry.feature_utils.**FastMLB**(*mlb*, *in_column*, *out_columns*)

Bases: object

**fit**(*X*, *y=None*)

**get_feature_names_in**()

**get_feature_names_out**()

**transform**(*df*)

class graphistry.feature_utils.**FeatureMixin**(**args*, ***kwargs*)

Bases: object

FeatureMixin for automatic featurization of nodes and edges DataFrames. Subclasses UMAPMixin for umap-ing of automatic features.

Usage:

```
g = graphistry.nodes(df, 'node_column')
g2 = g.featurize()
```

or for edges,

```
g = graphistry.edges(df, 'src', 'dst')
g2 = g.featurize(kind='edges')
```

or chain them for both nodes and edges,

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node_column')
g2 = g.featurize().featurize(kind='edges')
```

**featurize**(*kind='nodes'*, *X=None*, *y=None*, *use_scaler=None*, *use_scaler_target=None*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=12*, *multilabel=False*, *embedding=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=4.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *impute=True*, *n_quantiles=100*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *similarity=None*, *categories='auto'*, *keep_n_decimals=5*, *remove_node_column=True*, *inplace=False*, *feature_engine='auto'*, *dbscan=False*, *min_dist=0.5*, *min_samples=1*, *memoize=True*, *verbose=False*)

Featurize Nodes or Edges of the underlying nodes/edges DataFrames.

**Parameters**

- **kind** (*str*) – specify whether to featurize *nodes* or *edges*. Edge featurization includes a pairwise src-to-dst feature block using a MultiLabelBinarizer, with any other columns being treated the same way as with *nodes* featurization.
- **X** (*List[str] / str / DataFrame / None*) – Optional input, default None. If symbolic, evaluated against self data based on kind. If None, will featurize all columns of DataFrame
- **y** (*List[str] / str / DataFrame / None*) – Optional Target(s) columns or explicit DataFrame, default None
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] / None*) – selects which scaler (and automatically imputes missing values using mean strategy) to scale the data. Please see scikits-learn documentation https://scikit-learn.org/stable/modules/preprocessing.html Here 'standard' corresponds to 'StandardScaler' in scikits.
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] / None*) – selects which scaler to scale the target
- **cardinality_threshold** (*int*) – dirty_cat threshold on cardinality of categorical labels across columns. If value is greater than threshold, will run GapEncoder (a topic model) on column. If below, will one-hot_encode. Default 40.

- **cardinality_threshold_target** (*int*) – similar to cardinality_threshold, but for target features. Default is set high (400), as targets generally want to be one-hot encoded, but sometimes it can be useful to use GapEncoder (ie, set threshold lower) to create regressive targets, especially when those targets are textual/softly categorical and have semantic meaning across different labels. Eg, suppose a column has fields like ['Application Fraud', 'Other Statuses', 'Lost-Target scaling using/Stolen Fraud', 'Investigation Fraud', ...] the GapEncoder will concentrate the 'Fraud' labels together.
- **n_topics** (*int*) – the number of topics to use in the GapEncoder if cardinality_thresholds is saturated. Default is 42, but good rule of thumb is to consult the Johnson-Lindenstrauss Lemma https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma or use the simplified *random walk* estimate => n_topics_lower_bound ~ (pi/2) * (N-documents)**(1/4)
- **n_topics_target** (*int*) – the number of topics to use in the GapEncoder if cardinality_thresholds_target is saturated for the target(s). Default 12.
- **min_words** (*float*) – sets threshold on how many words to consider in a textual column if it is to be considered in the text processing pipeline. Set this very high if you want any textual columns to bypass the transformer, in favor of GapEncoder (topic modeling). Set to 0 to force all named columns to be encoded as textual (embedding)
- **model_name** (*str*) – Sentence Transformer model to use. Default Paraphrase model makes useful vectors, but at cost of encoding time. If faster encoding is needed, *average_word_embeddings_komninos* is useful and produces less semantically relevant vectors. Please see sentence_transformer (https://www.sbert.net/) library for all available models.
- **multilabel** (*bool*) – if True, will encode a *single* target column composed of lists of lists as multilabel outputs. This only works with y=['a_single_col'], default False
- **embedding** (*bool*) – If True, produces a random node embedding of size *n_topics* default, False. If no node features are provided, will produce random embeddings (for GNN models, for example)
- **use_ngrams** (*bool*) – If True, will encode textual columns as TfIdf Vectors, default, False.
- **ngram_range** (*tuple*) – if use_ngrams=True, can set ngram_range, eg: tuple = (1, 3)
- **max_df** (*float*) – if use_ngrams=True, set max word frequency to consider in vocabulary eg: max_df = 0.2,
- **min_df** (*int*) – if use_ngrams=True, set min word count to consider in vocabulary eg: min_df = 3 or 0.00001
- **categories** (*str / None*) – Optional[str] in ["auto", "k-means", "most_frequent"], decides which category to select in Similarity Encoding, default 'auto'
- **impute** (*bool*) – Whether to impute missing values, default True
- **n_quantiles** (*int*) – if use_scaler = 'quantile', sets the quantile bin size.
- **output_distribution** (*str*) – if use_scaler = 'quantile', can return distribution as ["normal", "uniform"]

- **quantile_range** – if use_scaler = 'robust'|'quantile', sets the quantile range.
- **n_bins** (*int*) – number of bins to use in kbins discretizer, default 10
- **encode** (*str*) – encoding for KBinsDiscretizer, can be one of *onehot*, *onehot-dense*, *ordinal*, default 'ordinal'
- **strategy** (*str*) – strategy for KBinsDiscretizer, can be one of *uniform*, *quantile*, *kmeans*, default 'quantile'
- **n_quantiles** – if use_scaler = "quantile", sets the number of quantiles, default=100
- **output_distribution** – if use_scaler="quantile"|"robust", choose from ["normal", "uniform"]
- **dbscan** (*bool*) – whether to run DBSCAN, default False.
- **min_dist** (*float*) – DBSCAN eps parameter, default 0.5.
- **min_samples** (*int*) – DBSCAN min_samples parameter, default 5.
- **keep_n_decimals** (*int*) – number of decimals to keep
- **remove_node_column** (*bool*) – whether to remove node column so it is not featurized, default True.
- **inplace** (*bool*) – whether to not return new graphistry instance or not, default False.
- **memoize** (*bool*) – whether to store and reuse results across runs, default True.
- **similarity** (*str / None*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch', 'auto']*)
- **verbose** (*bool*)

**Returns**

graphistry instance with new attributes set by the featurization process.

**featurize_or_get_edges_dataframe_if_X_is_None**(*X=None*, *y=None*, *use_scaler=None*, *use_scaler_target=None*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *multilabel=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *feature_engine='pandas'*, *reuse_if_existing=False*, *memoize=True*, *verbose=False*)

helper method gets edge feature and target matrix if X, y are not specified

**Parameters**

- **X** (*List[str] / str / DataFrame / None*) – Data Matrix

- **y** (*List[str] | str | DataFrame | None*) – target, default None
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **cardinality_threshold** (*int*)
- **cardinality_threshold_target** (*int*)
- **n_topics** (*int*)
- **n_topics_target** (*int*)
- **multilabel** (*bool*)
- **use_ngrams** (*bool*)
- **ngram_range** (*tuple*)
- **max_df** (*float*)
- **min_df** (*int*)
- **min_words** (*float*)
- **model_name** (*str*)
- **similarity** (*str | None*)
- **categories** (*str | None*)
- **impute** (*bool*)
- **n_quantiles** (*int*)
- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)
- **memoize** (*bool*)
- **verbose** (*bool*)

**Returns**
data $X$ and $y$

**Return type**
*Tuple*[*DataFrame*, *DataFrame* | None, object]

featurize_or_get_nodes_dataframe_if_X_is_None(*X=None*, *y=None*, *use_scaler=None*, *use_scaler_target=None*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *multilabel=False*, *embedding=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *remove_node_column=True*, *feature_engine='pandas'*, *reuse_if_existing=False*, *memoize=True*, *verbose=False*)

helper method gets node feature and target matrix if X, y are not specified. if X, y are specified will set them as *_node_target* and *_node_target* attributes

**Parameters**

- **X** (*List[str] | str | DataFrame | None*)
- **y** (*List[str] | str | DataFrame | None*)
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **cardinality_threshold** (*int*)
- **cardinality_threshold_target** (*int*)
- **n_topics** (*int*)
- **n_topics_target** (*int*)
- **multilabel** (*bool*)
- **embedding** (*bool*)
- **use_ngrams** (*bool*)
- **ngram_range** (*tuple*)
- **max_df** (*float*)
- **min_df** (*int*)
- **min_words** (*float*)
- **model_name** (*str*)
- **similarity** (*str | None*)
- **categories** (*str | None*)
- **impute** (*bool*)

- **n_quantiles** (*int*)
- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)
- **remove_node_column** (*bool*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)
- **memoize** (*bool*)
- **verbose** (*bool*)

**Return type**
*Tuple*[*DataFrame*, *DataFrame*, object]

**get_matrix**(*columns=None*, *kind='nodes'*, *target=False*)

Returns feature matrix, and if columns are specified, returns matrix with only the columns that contain the string *column_part* in their name.`X = g.get_matrix(['feature1', 'feature2'])` will retrieve a feature matrix with only the columns that contain the string *feature1* or *feature2* in their name. Most useful for topic modeling, where the column names are of the form *topic_0: descriptor*, *topic_1: descriptor*, etc. Can retrieve unique columns in original dataframe, or actual topic features like [ip_part, shoes, preference_x, etc]. Powerful way to retrieve features from a featurized graph by column or (top) features of interest.

**Example:**

```
# get the full feature matrices
X = g.get_matrix()
y = g.get_matrix(target=True)

# get subset of features, or topics, given topic model encoding
X = g2.get_matrix(['172', 'percent'])
X.columns
    => ['ip_172.56.104.67', 'ip_172.58.129.252', 'item_percent']
# or in targets
y = g2.get_matrix(['total', 'percent'], target=True)
y.columns
    => ['basket_price_total', 'conversion_percent', 'CTR_percent', 'CVR_
↪percent']

# not as useful for sbert features.
```

**Caveats:**

- if you have a column name that is a substring of another column name, you may get unexpected results.

**Args:**

**columns (Union[List, str])**
list of column names or a single column name that may exist in columns of the feature matrix. If None, returns original feature matrix

**kind (str, optional)**
Node or Edge features. Defaults to 'nodes'.

**target (bool, optional)**
If True, returns the target matrix. Defaults to False.

**Returns:**
pd.DataFrame: feature matrix with only the columns that contain the string *column_part* in their name.

**Parameters**

- **columns** (*List | str | None*)
- **kind** (*str*)
- **target** (*bool*)

**Return type**
*DataFrame*

**scale**(*df=None*, *y=None*, *kind='nodes'*, *use_scaler=None*, *use_scaler_target=None*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *return_scalers=False*)

Scale data using the same scalers as used in the featurization step.

**Example**

```
g = graphistry.nodes(df)
X, y = g.featurize().scale(kind='nodes', use_scaler='robust', use_scaler_target=
↪'kbins', n_bins=3)

# or
g = graphistry.nodes(df)
# set a scaling strategy for features and targets -- umap uses those and␣
↪produces different results depending.
g2 = g.umap(use_scaler='standard', use_scaler_target=None)

# later if you want to scale new data, you can do so
X, y = g2.transform(df, df, scale=False)
X_scaled, y_scaled = g2.scale(X, y, use_scaler='minmax', use_scaler_target=
↪'kbins', n_bins=5)
# fit some other pipeline
clf.fit(X_scaled, y_scaled)
```

**Args:**

**df**
pd.DataFrame, raw data to transform, if None, will use data from featurization fit

**y**
pd.DataFrame, optional target data

**kind**
str, one of *nodes*, *edges*

**use_scaler**
Scaling transformer

**use_scaler_target**
Scaling transformer on target

**impute**
bool, if True, will impute missing values

**n_quantiles**
int, number of quantiles to use for quantile scaler

**output_distribution**
str, one of *normal*, *uniform*, *lognormal*

**quantile_range**
tuple, range of quantiles to use for quantile scaler

**n_bins**
int, number of bins to use for KBinsDiscretizer

**encode**
str, one of *ordinal*, *onehot*, *onehot-dense*, *binary*

**strategy**
str, one of *uniform*, *quantile*, *kmeans*

**keep_n_decimals**
int, number of decimals to keep after scaling

**return_scalers**
bool, if True, will return the scalers used to scale the data

**Returns:**

(X, y) transformed data if return_graph is False or a graph with inferred edges if return_graph is True, or (X, y, scaler, scaler_target) if return_scalers is True

**Parameters**

- `df` (*DataFrame | None*)
- `y` (*DataFrame | None*)
- `kind` (*str*)
- `use_scaler` (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- `use_scaler_target` (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- `impute` (*bool*)
- `n_quantiles` (*int*)
- `output_distribution` (*str*)
- `n_bins` (*int*)
- `encode` (*str*)
- `strategy` (*str*)
- `keep_n_decimals` (*int*)
- `return_scalers` (*bool*)

`transform`(*df*, *y=None*, *kind='nodes'*, *min_dist='auto'*, *n_neighbors=7*, *merge_policy=False*, *sample=None*, *return_graph=True*, *scaled=True*, *verbose=False*)

Transform new data and append to existing graph, or return dataframes

**args:**

**df**
pd.DataFrame, raw data to transform

**ydf**
pd.DataFrame, optional

**kind**
str # one of *nodes*, *edges*

**return_graph**
bool, if True, will return a graph with inferred edges.

**merge_policy**
bool, if True, adds batch to existing graph nodes via nearest neighbors. If False, will infer edges only between nodes in the batch, default False

**min_dist**
float, if return_graph is True, will use this value in NN search, or 'auto' to infer a good value. min_dist represents the maximum distance between two samples for one to be considered as in the neighborhood of the other.

**sample**
int, if return_graph is True, will use sample edges of existing graph to fill out the new graph

**n_neighbors**
int, if return_graph is True, will use this value for n_neighbors in Nearest Neighbors search

**scaled**
bool, if True, will use scaled transformation of data set during featurization, default True

**verbose**
bool, if True, will print metadata about the graph construction, default False

**Returns:**

X, y: pd.DataFrame, transformed data if return_graph is False or a graphistry Plottable with inferred edges if return_graph is True

**Parameters**

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **kind** (*str*)
- **min_dist** (*str | float | int*)
- **n_neighbors** (*int*)
- **merge_policy** (*bool*)
- **sample** (*int | None*)
- **return_graph** (*bool*)

- **scaled** (*bool*)
- **verbose** (*bool*)

class graphistry.feature_utils.**callThrough**(*x*)

Bases: object

graphistry.feature_utils.**check_if_textual_column**(*df*, *col*, *confidence=0.35*, *min_words=2.5*)

Checks if *col* column of df is textual or not using basic heuristics

**Parameters**

- **df** (*DataFrame*) – DataFrame
- **col** (*str*) – column name
- **confidence** (*float*) – threshold float value between 0 and 1. If column *col* has *confidence* more elements as type *str* it will pass it onto next stage of evaluation. Default 0.35
- **min_words** (*float*) – mean minimum words threshold. If mean words across *col* is greater than this, it is deemed textual. Default 2.5

**Returns**

bool, whether column is textual or not

**Return type**

bool

graphistry.feature_utils.**concat_text**(*df*, *text_cols*)

graphistry.feature_utils.**encode_edges**(*edf*, *src*, *dst*, *mlb*, *fit=False*)

edge encoder – creates multilabelBinarizer on edge pairs.

**Args:**

edf (pd.DataFrame): edge dataframe src (string): source column dst (string): destination column mlb (sklearn): multilabelBinarizer fit (bool, optional): If true, fits multilabelBinarizer. Defaults to False.

**Returns**

tuple: pd.DataFrame, multilabelBinarizer

graphistry.feature_utils.**encode_multi_target**(*ydf*, *mlb=None*)

graphistry.feature_utils.**encode_textual**(*df*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*)

**Parameters**

- **df** (*DataFrame*)
- **min_words** (*float*)
- **model_name** (*str*)
- **use_ngrams** (*bool*)
- **ngram_range** (*tuple*)
- **max_df** (*float*)
- **min_df** (*int*)

**Return type**
*Tuple*[*DataFrame*, *List*, *Any*]

graphistry.feature_utils.**features_without_target**(*df*, *y=None*)

Checks if y DataFrame column name is in df, and removes it from df if so

**Parameters**

- **df** (*DataFrame*) – model DataFrame
- **y** (*List | str | DataFrame | None*) – target DataFrame

**Returns**
DataFrames of model and target

**Return type**
*DataFrame*

graphistry.feature_utils.**find_bad_set_columns**(*df*, *bad_set=['[]']*)

Finds columns that if not coerced to strings, will break processors.

**Parameters**

- **df** (*DataFrame*) – DataFrame
- **bad_set** (*List*) – List of strings to look for.

**Returns**
list

graphistry.feature_utils.**fit_pipeline**(*X*, *transformer*, *keep_n_decimals=5*)

Helper to fit DataFrame over transformer pipeline. Rounds resulting matrix X by keep_n_digits if not 0, which helps for when transformer pipeline is scaling or imputer which sometime introduce small negative numbers, and umap metrics like Hellinger need to be positive :param X: DataFrame to transform. :param transformer: Pipeline object to fit and transform :param keep_n_decimals: Int of how many decimal places to keep in rounded transformed data

**Parameters**

- **X** (*DataFrame*)
- **keep_n_decimals** (*int*)

**Return type**
*DataFrame*

graphistry.feature_utils.**get_cardinality_ratio**(*df*)

Calculates the ratio of unique values to total number of rows of DataFrame

**Parameters**
**df** (*DataFrame*) – DataFrame

graphistry.feature_utils.**get_dataframe_by_column_dtype**(*df*, *include=None*, *exclude=None*)

graphistry.feature_utils.**get_matrix_by_column_part**(*X*, *column_part*)

Get the feature matrix by column part existing in column names.

**Parameters**

- **X** (*DataFrame*)
- **column_part** (*str*)

**Return type**
*DataFrame*

graphistry.feature_utils.**get_matrix_by_column_parts**(*X*, *column_parts*)

Get the feature matrix by column parts list existing in column names.

**Parameters**

- **X** (*DataFrame*)
- **column_parts** (*list / str / None*)

**Return type**
*DataFrame*

graphistry.feature_utils.**get_numeric_transformers**(*ndf*, *y=None*)

graphistry.feature_utils.**get_preprocessing_pipeline**(*use_scaler='robust'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='quantile'*)

Helper function for imputing and scaling np.ndarray data using different scaling transformers.

**Parameters**

- **X** – np.ndarray
- **impute** (*bool*) – whether to run imputing or not
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*) – Selects scaling transformer
- **n_quantiles** (*int*) – if use_scaler = 'quantile', sets the quantile bin size.
- **output_distribution** (*str*) – if use_scaler = 'quantile', can return distribution as ["normal", "uniform"]
- **quantile_range** – if use_scaler = 'robust'/'quantile', sets the quantile range.
- **n_bins** (*int*) – number of bins to use in kbins discretizer
- **encode** (*str*) – encoding for KBinsDiscretizer, can be one of *onehot*, *onehot-dense*, *ordinal*, default 'ordinal'
- **strategy** (*str*) – strategy for KBinsDiscretizer, can be one of *uniform*, *quantile*, *kmeans*, default 'quantile'

**Returns**
scaled array, imputer instances or None, scaler instance or None

**Return type**
*Any*

graphistry.feature_utils.**get_text_preprocessor**(*ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*)

graphistry.feature_utils.**get_textual_columns**(*df*, *min_words=2.5*)

Collects columns from df that it deems are textual.

**Parameters**

- **df** (*DataFrame*) – DataFrame
- **min_words** (*float*)

**Returns**
list of columns names

**Return type**
*List*

graphistry.feature_utils.**group_columns_by_dtypes**(*df*, *verbose=True*)

**Parameters**

- **df** (*DataFrame*)
- **verbose** (*bool*)

**Return type**
*Dict*

graphistry.feature_utils.**identity**(*x*)

graphistry.feature_utils.**impute_and_scale_df**(*df*, *use_scaler='robust'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*)

**Parameters**

- **df** (*DataFrame*)
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*)
- **impute** (*bool*)
- **n_quantiles** (*int*)
- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)

**Return type**
*Tuple*[*DataFrame*, *Any*]

graphistry.feature_utils.**is_cudf_df**(*df*)

**Parameters**
**df** (*Any*)

**Return type**
bool

graphistry.feature_utils.**is_cudf_s**(*s*)

**Parameters**
**s** (*Any*)

**Return type**
bool

graphistry.feature_utils.**is_dataframe_all_numeric**(*df*)

**Parameters**
**df** (*DataFrame*)

**Return type**
bool

graphistry.feature_utils.**make_array**(*X*)

graphistry.feature_utils.**passthrough_df_cols**(*df*, *columns*)

graphistry.feature_utils.**process_dirty_dataframes**(*ndf*, *y*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *similarity=None*, *categories='auto'*, *multilabel=False*, *feature_engine='pandas'*)

Dirty_Cat encoder for record level data. Will automatically turn inhomogeneous dataframe into matrix using smart conversion tricks.

**Parameters**

- **ndf** (*DataFrame*) – node DataFrame
- **y** (*DataFrame | None*) – target DataFrame or series
- **cardinality_threshold** (*int*) – For ndf columns, below this threshold, encoder is OneHot, above, it is GapEncoder
- **cardinality_threshold_target** (*int*) – For target columns, below this threshold, encoder is OneHot, above, it is GapEncoder
- **n_topics** (*int*) – number of topics for GapEncoder, default 42
- **similarity** (*str | None*) – one of 'ngram', 'levenshtein-ratio', 'jaro', or'jaro-winkler'}) – The type of pairwise string similarity to use. If None or False, uses a SuperVectorizer
- **n_topics_target** (*int*)
- **categories** (*str | None*)
- **multilabel** (*bool*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)

**Returns**
Encoded data matrix and target (if not None), the data encoder, and the label encoder.

**Return type**
*Tuple*[*DataFrame*, *DataFrame* | None, *Any*, *Any*]

graphistry.feature_utils.**process_edge_dataframes**(*edf*, *y*, *src*, *dst*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *use_scaler=None*, *use_scaler_target=None*, *multilabel=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *feature_engine='pandas'*)

Custom Edge-record encoder. Uses a MultiLabelBinarizer to generate a src/dst vector and then process_textual_or_other_dataframes that encodes any other data present in edf, textual or not.

**Parameters**

- **edf** (*DataFrame*) – pandas DataFrame of edge features
- **y** (*DataFrame*) – pandas DataFrame of edge labels
- **src** (*str*) – source column to select in edf
- **dst** (*str*) – destination column to select in edf
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*) – Scaling transformer
- **use_scaler_target'** – Scaling transformer for target
- **cardinality_threshold** (*int*)
- **cardinality_threshold_target** (*int*)
- **n_topics** (*int*)
- **n_topics_target** (*int*)
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **multilabel** (*bool*)
- **use_ngrams** (*bool*)
- **ngram_range** (*tuple*)
- **max_df** (*float*)
- **min_df** (*int*)
- **min_words** (*float*)
- **model_name** (*str*)
- **similarity** (*str | None*)
- **categories** (*str | None*)
- **impute** (*bool*)
- **n_quantiles** (*int*)

- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)

**Returns**
Encoded data matrix and target (if not None), the data encoders, and the label encoder.

**Return type**
*Tuple*[*DataFrame*, *DataFrame*, *DataFrame*, *DataFrame*, *List*[*Any*], *Any*, *Any* | None, *Any* | None, *Any*, *List*[str]]

graphistry.feature_utils.**process_nodes_dataframes**(*df*, *y*, *cardinality_threshold=40*, *cardinality_threshold_target=400*, *n_topics=42*, *n_topics_target=7*, *use_scaler='robust'*, *use_scaler_target='kbins'*, *multilabel=False*, *embedding=False*, *use_ngrams=False*, *ngram_range=(1, 3)*, *max_df=0.2*, *min_df=3*, *min_words=2.5*, *model_name='paraphrase-MiniLM-L6-v2'*, *similarity=None*, *categories='auto'*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*, *feature_engine='pandas'*)

Automatic Deep Learning Embedding/ngrams of Textual Features, with the rest of the columns taken care of by dirty_cat

**Parameters**

- **df** (*DataFrame*) – pandas DataFrame of data
- **y** (*DataFrame*) – pandas DataFrame of targets
- **n_topics** (*int*) – number of topics in Gap Encoder
- **n_topics_target** (*int*) – number of topics in Gap Encoder for target
- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*) – Scaling transformer
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*) – Scaling transformer for target
- **confidence** – Number between 0 and 1, will pass column for textual processing if total entries are string like in a column and above this relative threshold.
- **min_words** (*float*) – Sets the threshold for average number of words to include column for textual sentence encoding. Lower values means that columns will be labeled textual and sent to sentence-encoder. Set to 0 to force named columns as textual.
- **model_name** (*str*) – SentenceTransformer model name. See available list at https://www.sbert.net/docs/pretrained_models. html#sentence-embedding-models

- **cardinality_threshold** (*int*)
- **cardinality_threshold_target** (*int*)
- **multilabel** (*bool*)
- **embedding** (*bool*)
- **use_ngrams** (*bool*)
- **ngram_range** (*tuple*)
- **max_df** (*float*)
- **min_df** (*int*)
- **similarity** (*str | None*)
- **categories** (*str | None*)
- **impute** (*bool*)
- **n_quantiles** (*int*)
- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)

**Returns**
X_enc, y_enc, data_encoder, label_encoder, scaling_pipeline, scaling_pipeline_target, text_model, text_cols,

**Return type**
*Tuple*[*DataFrame*, *Any*, *DataFrame*, *Any*, *Any*, *Any*, *Any* | None, *Any* | None, *Any*, *List*[str]]

graphistry.feature_utils.**remove_internal_namespace_if_present**(*df*)

Some tranformations below add columns to the DataFrame, this method removes them before featurization Will not drop if suffix is added during UMAP-ing

**Parameters**
**df** (*DataFrame*) – DataFrame

**Returns**
DataFrame with dropped columns in reserved namespace

graphistry.feature_utils.**remove_node_column_from_symbolic**(*X_symbolic*, *node*)

graphistry.feature_utils.**resolve_X**(*df*, *X*)

**Parameters**

- **df** (*DataFrame | None*)
- **X** (*List[str] | str | DataFrame | None*)

**Return type**
*DataFrame*

graphistry.feature_utils.**resolve_feature_engine**(*feature_engine*)

**Parameters**
**feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch', 'auto']*)

**Return type**
*Literal*['none', 'pandas', 'dirty_cat', 'torch']

graphistry.feature_utils.**resolve_scaler**(*use_scaler*, *feature_engine*)

**Parameters**

- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)

**Return type**
*Literal*['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']

graphistry.feature_utils.**resolve_scaler_target**(*use_scaler_target*, *feature_engine*, *multilabel*)

**Parameters**

- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile'] | None*)
- **feature_engine** (*Literal['none', 'pandas', 'dirty_cat', 'torch']*)
- **multilabel** (*bool*)

**Return type**
*Literal*['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']

graphistry.feature_utils.**resolve_y**(*df*, *y*)

**Parameters**

- **df** (*DataFrame | None*)
- **y** (*List[str] | str | DataFrame | None*)

**Return type**
*DataFrame*

graphistry.feature_utils.**reuse_featurization**(*g*, *memoize*, *metadata*)

**Parameters**

- **g** (Plottable)
- **memoize** (*bool*)
- **metadata** (*Any*)

graphistry.feature_utils.**safe_divide**(*a*, *b*)

graphistry.feature_utils.**set_currency_to_float**(*df*, *col*, *return_float=True*)

**Parameters**

- **df** (*DataFrame*)
- **col** (*str*)
- **return_float** (*bool*)

graphistry.feature_utils.**set_to_bool**(*df*, *col*, *value*)

**Parameters**

- **df** (*DataFrame*)
- **col** (*str*)
- **value** (*Any*)

graphistry.feature_utils.**set_to_datetime**(*df*, *cols*, *new_col*)

**Parameters**

- **df** (*DataFrame*)
- **cols** (*List*)
- **new_col** (*str*)

graphistry.feature_utils.**set_to_numeric**(*df*, *cols*, *fill_value=0.0*)

**Parameters**

- **df** (*DataFrame*)
- **cols** (*List*)
- **fill_value** (*float*)

graphistry.feature_utils.**smart_scaler**(*X_enc*, *y_enc*, *use_scaler*, *use_scaler_target*, *impute=True*, *n_quantiles=10*, *output_distribution='normal'*, *quantile_range=(25, 75)*, *n_bins=10*, *encode='ordinal'*, *strategy='uniform'*, *keep_n_decimals=5*)

**Parameters**

- **use_scaler** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*)
- **use_scaler_target** (*Literal['none', 'kbins', 'standard', 'robust', 'minmax', 'quantile']*)
- **impute** (*bool*)
- **n_quantiles** (*int*)
- **output_distribution** (*str*)
- **n_bins** (*int*)
- **encode** (*str*)
- **strategy** (*str*)
- **keep_n_decimals** (*int*)

graphistry.feature_utils.**transform**(*df*, *ydf*, *res*, *kind*, *src*, *dst*)

**Parameters**

- **df** (*DataFrame*)
- **ydf** (*DataFrame*)
- **res** (*List*)
- **kind** (*str*)

**Return type**
*Tuple*[*DataFrame*, *DataFrame*]

graphistry.feature_utils.**transform_dirty**(*df*, *data_encoder*, *name=''*)

**Parameters**

- **df** (*DataFrame*)
- **data_encoder** (*Any*)
- **name** (*str*)

**Return type**
*DataFrame*

graphistry.feature_utils.**transform_text**(*df*, *text_model*, *text_cols*)

**Parameters**

- **df** (*DataFrame*)
- **text_model** (*Any*)
- **text_cols** (*List / str*)

**Return type**
*DataFrame*

graphistry.feature_utils.**where_is_currency_column**(*df*, *col*)

**Parameters**

- **df** (*DataFrame*)
- **col** (*str*)

#### 3.8.4.2 UMAP

class graphistry.umap_utils.**UMAPMixin**(**args*, ***kwargs*)

Bases: object

UMAP Mixin for automagic UMAPing

**filter_weighted_edges**(*scale=1.0*, *index_to_nodes_dict=None*, *inplace=False*, *kind='nodes'*)

Filter edges based on _weighted_edges_df (ex: from .umap())

**Parameters**

- **scale** (*float*)
- **index_to_nodes_dict** (*Dict / None*)
- **inplace** (*bool*)
- **kind** (*str*)

**transform_umap**(*df*, *y=None*, *kind='nodes'*, *min_dist='auto'*, *n_neighbors=7*, *merge_policy=False*, *sample=None*, *return_graph=True*, *fit_umap_embedding=True*, *umap_transform_kwargs={}*)

Transforms data into UMAP embedding

**Args:**

**df**
Dataframe to transform

**y**
Target column

**kind**
One of *nodes* or *edges*

**min_dist**
Epsilon for including neighbors in infer_graph

**n_neighbors**
Number of neighbors to use for contextualization

**merge_policy**
if True, use previous graph, adding new batch to existing graph's neighbors useful to contextualize new data against existing graph. If False, *sample* is irrelevant.

sample: Sample number of existing graph's neighbors to use for contextualization – helps make denser graphs return_graph: Whether to return a graph or just the embeddings fit_umap_embedding: Whether to infer graph from the UMAP embedding on the new data, default True

**Parameters**

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **kind** (*str*)
- **min_dist** (*str | float | int*)
- **n_neighbors** (*int*)
- **merge_policy** (*bool*)
- **sample** (*int | None*)
- **return_graph** (*bool*)
- **fit_umap_embedding** (*bool*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

**Return type**
*Tuple*[*DataFrame*, *DataFrame*, *DataFrame*] | Plottable

**umap**(*X=None*, *y=None*, *kind='nodes'*, *scale=1.0*, *n_neighbors=12*, *min_dist=0.1*, *spread=0.5*, *local_connectivity=1*, *repulsion_strength=1*, *negative_sample_rate=5*, *n_components=2*, *metric='euclidean'*, *suffix=''*, *play=0*, *encode_position=True*, *encode_weight=True*, *dbscan=False*, *engine='auto'*, *feature_engine='auto'*, *inplace=False*, *memoize=True*, *umap_kwargs={}*, *umap_fit_kwargs={}*, *umap_transform_kwargs={}*, ***featurize_kwargs*)

UMAP the featurized nodes or edges data, or pass in your own X, y (optional) dataframes of values

Example

```
>>> import graphistry
>>> g = graphistry.nodes(pd.DataFrame({'node': [0,1,2], 'data': [1,2,3], 'meta':
↪ ['a', 'b', 'c']}))
```

(continues on next page)

(continued from previous page)

```
>>> g2 = g.umap(n_components=3, spread=1.0, min_dist=0.1, n_neighbors=12,␣
↪negative_sample_rate=5, local_connectivity=1, repulsion_strength=1.0, metric=
↪'euclidean', suffix='', play=0, encode_position=True, encode_weight=True,␣
↪dbscan=False, engine='auto', feature_engine='auto', inplace=False,␣
↪memoize=True)
>>> g2.plot()
```

Parameters

**X**
either a dataframe ndarray of features, or column names to featurize

**y**
either an dataframe ndarray of targets, or column names to featurize targets

**kind**
*nodes* or *edges* or None. If None, expects explicit X, y (optional) matrices, and will Not associate them to nodes or edges. If X, y (optional) is given, with kind = [nodes, edges], it will associate new matrices to nodes or edges attributes.

**scale**
multiplicative scale for pruning weighted edge DataFrame gotten from UMAP, between [0, ..) with high end meaning keep all edges

**n_neighbors**
UMAP number of nearest neighbors to include for UMAP connectivity, lower makes more compact layouts. Minimum 2

**min_dist**
UMAP float between 0 and 1, lower makes more compact layouts.

**spread**
UMAP spread of values for relaxation

**local_connectivity**
UMAP connectivity parameter

**repulsion_strength**
UMAP repulsion strength

**negative_sample_rate**
UMAP negative sampling rate

**n_components**
number of components in the UMAP projection, default 2

**metric**
UMAP metric, default 'euclidean'. see (UMAP-LEARN)[https://umap-learn.readthedocs.io/ en/latest/parameters.html] documentation for more.

**suffix**
optional suffix to add to x, y attributes of umap.

**play**
Graphistry play parameter, default 0, how much to evolve the network during clustering. 0 preserves the original UMAP layout.

**encode_weight**
if True, will set new edges_df from implicit UMAP, default True.

**encode_position**
whether to set default plotting bindings – positions x,y from umap for .plot(), default True

**dbscan**
whether to run DBSCAN on the UMAP embedding, default False.

**engine**
selects which engine to use to calculate UMAP: default "auto" will use cuML if available, otherwise UMAP-LEARN.

**feature_engine**
How to encode data ("none", "auto", "pandas", "dirty_cat", "torch")

**inplace**
bool = False, whether to modify the current object, default False. when False, returns a new object, useful for chaining in a functional paradigm.

**memoize**
whether to memoize the results of this method, default True.

**umap_kwargs**
Optional kwargs to pass to underlying UMAP library constructor

**umap_fit_kwargs**
Optional kwargs to pass to underlying UMAP fit method, including fit part of fit_transform

**umap_transform_kwargs**
Optional kwargs to pass to underlying UMAP transform method, including transform part of fit_transform

**featurize_kwargs**
Optional kwargs to pass to .featurize()

**Returns**
self, with attributes set with new data

**Parameters**

- `X` (*List[str] | str | DataFrame | None*)
- `y` (*List[str] | str | DataFrame | None*)
- `kind` (*str*)
- `scale` (*float*)
- `n_neighbors` (*int*)
- `min_dist` (*float*)
- `spread` (*float*)
- `local_connectivity` (*int*)
- `repulsion_strength` (*float*)
- `negative_sample_rate` (*int*)
- `n_components` (*int*)
- `metric` (*str*)
- `suffix` (*str*)

- **play** (*int | None*)
- **encode_position** (*bool*)
- **encode_weight** (*bool*)
- **dbscan** (*bool*)
- **engine** (*Literal['cuml', 'umap_learn', 'auto']*)
- **feature_engine** (*str*)
- **inplace** (*bool*)
- **memoize** (*bool*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

**umap_fit**(*X*, *y=None*, *umap_fit_kwargs={}*)

**Parameters**

- **X** (*DataFrame*)
- **y** (*DataFrame | None*)
- **umap_fit_kwargs** (*Dict[str, Any]*)

**umap_lazy_init**(*res*, *n_neighbors=12*, *min_dist=0.1*, *spread=0.5*, *local_connectivity=1*, *repulsion_strength=1*, *negative_sample_rate=5*, *n_components=2*, *metric='euclidean'*, *engine='auto'*, *suffix=''*, *umap_kwargs={}*, *umap_fit_kwargs={}*, *umap_transform_kwargs={}*)

**Parameters**

- **n_neighbors** (*int*)
- **min_dist** (*float*)
- **spread** (*float*)
- **local_connectivity** (*int*)
- **repulsion_strength** (*float*)
- **negative_sample_rate** (*int*)
- **n_components** (*int*)
- **metric** (*str*)
- **engine** (*Literal['cuml', 'umap_learn', 'auto']*)
- **suffix** (*str*)
- **umap_kwargs** (*Dict[str, Any]*)
- **umap_fit_kwargs** (*Dict[str, Any]*)
- **umap_transform_kwargs** (*Dict[str, Any]*)

graphistry.umap_utils.**assert_imported**()

graphistry.umap_utils.**assert_imported_cuml**()

graphistry.umap_utils.**is_legacy_cuml**()

graphistry.umap_utils.**make_safe_gpu_dataframes**(*X*, *y*, *engine*)

graphistry.umap_utils.**prune_weighted_edges_df_and_relabel_nodes**(*wdf*, *scale=0.1*, *index_to_nodes_dict=None*)

Prune the weighted edge DataFrame so to return high fidelity similarity scores.

**Parameters**

- **wdf** (*DataFrame | Any*) – weighted edge DataFrame gotten via UMAP
- **scale** (*float*) – lower values means less edges > (max - scale * std)
- **index_to_nodes_dict** (*Dict | None*) – dict of index to node name; remap src/dst values if provided

**Returns**
pd.DataFrame

**Return type**
*DataFrame*

graphistry.umap_utils.**resolve_umap_engine**(*engine*)

**Parameters**
**engine** (*Literal['cuml', 'umap_learn', 'auto']*)

**Return type**
*Literal*['cuml', 'umap_learn']

graphistry.umap_utils.**reuse_umap**(*g*, *memoize*, *metadata*)

**Parameters**

- **g** (Plottable)
- **memoize** (*bool*)
- **metadata** (*Any*)

graphistry.umap_utils.**umap_graph_to_weighted_edges**(*umap_graph*, *engine*, *is_legacy*, *cfg=<module 'graphistry.constants' from '/home/docs/checkouts/readthedocs.org/user_builds/pygraphistry/checkouts/0.34.10/graphistry*

**Parameters**
**engine** (*Literal['cuml', 'umap_learn']*)

#### 3.8.4.3 Semantic Search

class graphistry.text_utils.**SearchToGraphMixin**(**args*, ***kwargs*)

Bases: object

**assert_features_line_up_with_nodes**()

**assert_fitted**()

**build_index**(*angular=False*, *n_trees=None*)

**classmethod load_search_instance**(*savepath*)

**save_search_instance**(*savepath*)

**search**(*query*, *cols=None*, *thresh=5000*, *fuzzy=True*, *top_n=10*)

Natural language query over nodes that returns a dataframe of results sorted by relevance column "distance".

If node data is not yet feature-encoded (and explicit edges are given), run automatic feature engineering:

```
g2 = g.featurize(kind='nodes', X=['text_col_1', ..],
min_words=0 # forces all named columns are textually encoded
)
```

If edges do not yet exist, generate them via

```
g2 = g.umap(kind='nodes', X=['text_col_1', ..],
min_words=0 # forces all named columns are textually encoded
)
```

If an index is not yet built, it is generated *g2.build_index()* on the fly at search time. Otherwise, can set *g2.build_index()* to build it ahead of time.

**Args:**

**query (str)**
natural language query.

**cols (list or str, optional)**
if fuzzy=False, select which column to query. Defaults to None since fuzzy=True by defaul.

**thresh (float, optional)**
distance threshold from query vector to returned results. Defaults to 5000, set large just in case, but could be as low as 10.

**fuzzy (bool, optional)**
if True, uses embedding + annoy index for recall, otherwise does string matching over given *cols* Defaults to True.

**top_n (int, optional)**
how many results to return. Defaults to 100.

**Returns:**

**pd.DataFrame, vector_encoding_of_query:** rank ordered dataframe of results matching query

vector encoding of query via given transformer/ngrams model if fuzzy=True else None

**Parameters**

- **query** (*str*)
- **thresh** (*float*)
- **fuzzy** (*bool*)
- **top_n** (*int*)

**search_graph**(*query*, *scale=0.5*, *top_n=100*, *thresh=5000*, *broader=False*, *inplace=False*)

**Input a natural language query and return a graph of results.**
See help(g.search) for more information

**Args:**

**query (str)**
query input eg "coding best practices"

**scale (float, optional)**
edge weigh threshold, Defaults to 0.5.

**top_n (int, optional)**
how many results to return. Defaults to 100.

**thresh (float, optional)**
distance threshold from query vector to returned results. Defaults to 5000, set large just in case, but could be as low as 10.

**broader (bool, optional)**
if True, will retrieve entities connected via an edge that were not necessarily bubbled up in the results_dataframe. Defaults to False.

**inplace (bool, optional)**
whether to return new instance (default) or mutate self. Defaults to False.

**Returns:**
graphistry Instance: g

**Parameters**

- **query** (*str*)
- **scale** (*float*)
- **top_n** (*int*)
- **thresh** (*float*)
- **broader** (*bool*)
- **inplace** (*bool*)

#### 3.8.4.4 DBSCAN

**class** graphistry.compute.cluster.**ClusterMixin**(**args*, ***kwargs*)

Bases: object

**dbscan**(*min_dist=0.2*, *min_samples=1*, *cols=None*, *kind='nodes'*, *fit_umap_embedding=True*, *target=False*, *verbose=False*, *engine_dbscan='sklearn'*, **args*, ***kwargs*)

**DBSCAN clustering on cpu or gpu infered automatically. Adds a *_dbscan* column to nodes or edges.**
NOTE: g.transform_dbscan(..) currently unsupported on GPU.

Examples:

```
g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')

# cluster by UMAP embeddings
```

(continues on next page)

(continued from previous page)

```
kind = 'nodes' | 'edges'
g2 = g.umap(kind=kind).dbscan(kind=kind)
print(g2._nodes['_dbscan']) | print(g2._edges['_dbscan'])

# dbscan in umap or featurize API
g2 = g.umap(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)
# or, here dbscan is infered from features, not umap embeddings
g2 = g.featurize(dbscan=True, min_dist=1.2, min_samples=2, **kwargs)

# and via chaining,
g2 = g.umap().dbscan(min_dist=1.2, min_samples=2, **kwargs)

# cluster by feature embeddings
g2 = g.featurize().dbscan(**kwargs)

# cluster by a given set of feature column attributes, or with target=True
g2 = g.featurize().dbscan(cols=['ip_172', 'location', 'alert'], target=False,␣
↪**kwargs)

# equivalent to above (ie, cols != None and umap=True will still use features␣
↪dataframe, rather than UMAP embeddings)
g2 = g.umap().dbscan(cols=['ip_172', 'location', 'alert'], umap=True | False,␣
↪**kwargs)

g2.plot() # color by `_dbscan` column
```

**Useful:**

Enriching the graph with cluster labels from UMAP is useful for visualizing clusters in the graph by color, size, etc, as well as assessing metrics per cluster, e.g. https://github.com/graphistry/pygraphistry/blob/master/demos/ai/cyber/cyber-redteam-umap-demo.ipynb

**Args:**

**min_dist float**
The maximum distance between two samples for them to be considered as in the same neighborhood.

**kind str**
'nodes' or 'edges'

**cols**
list of columns to use for clustering given *g.featurize* has been run, nice way to slice features or targets by fragments of interest, e.g. ['ip_172', 'location', 'ssh', 'warnings']

**fit_umap_embedding bool**
whether to use UMAP embeddings or features dataframe to cluster DBSCAN

**min_samples**
The number of samples in a neighborhood for a point to be considered as a core point. This includes the point itself.

**target**
whether to use the target column as the clustering feature

**Parameters**

- **min_dist** (*float*)
- **min_samples** (*int*)
- **cols** (*List / str / None*)
- **kind** (*str*)
- **fit_umap_embedding** (*bool*)
- **target** (*bool*)
- **verbose** (*bool*)
- **engine_dbscan** (*str*)

**transform_dbscan**(*df*, *y=None*, *min_dist='auto'*, *infer_umap_embedding=False*, *sample=None*, *n_neighbors=None*, *kind='nodes'*, *return_graph=True*, *verbose=False*)

Transforms a minibatch dataframe to one with a new column '_dbscan' containing the DBSCAN cluster labels on the minibatch and generates a graph with the minibatch and the original graph, with edges between the minibatch and the original graph inferred from the umap embedding or features dataframe. Graph nodes | edges will be colored by '_dbscan' column.

Examples:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.featurize().dbscan()

predict:
::

    emb, X, _, ndf = g2.transform_dbscan(ndf, return_graph=False)
    # or
    g3 = g2.transform_dbscan(ndf, return_graph=True)
    g3.plot()
```

likewise for umap:

```
fit:
    g = graphistry.edges(edf, 'src', 'dst').nodes(ndf, 'node')
    g2 = g.umap(X=.., y=..).dbscan()

predict:
::

    emb, X, y, ndf = g2.transform_dbscan(ndf, ndf, return_graph=False)
    # or
    g3 = g2.transform_dbscan(ndf, ndf, return_graph=True)
    g3.plot()
```

**Args:**

**df**
dataframe to transform

**y**
optional labels dataframe

**min_dist**
The maximum distance between two samples for them to be considered as in the same neighborhood. smaller values will result in less edges between the minibatch and the original graph. Default 'auto', infers min_dist from the mean distance and std of new points to the original graph

**fit_umap_embedding**
whether to use UMAP embeddings or features dataframe when inferring edges between the minibatch and the original graph. Default False, uses the features dataframe

**sample**
number of samples to use when inferring edges between the minibatch and the original graph, if None, will only use closest point to the minibatch. If greater than 0, will sample the closest *sample* points in existing graph to pull in more edges. Default None

**kind**
'nodes' or 'edges'

**return_graph**
whether to return a graph or the (emb, X, y, minibatch df enriched with DBSCAN labels), default True infered graph supports kind='nodes' only.

**verbose**
whether to print out progress, default False

**Parameters**

- **df** (*DataFrame*)
- **y** (*DataFrame | None*)
- **min_dist** (*float | str*)
- **infer_umap_embedding** (*bool*)
- **sample** (*int | None*)
- **n_neighbors** (*int | None*)
- **kind** (*str*)
- **return_graph** (*bool*)
- **verbose** (*bool*)

graphistry.compute.cluster.**dbscan_fit**(*g*, *dbscan*, *kind='nodes'*, *cols=None*, *use_umap_embedding=True*, *target=False*, *verbose=False*)

**Fits clustering on UMAP embeddings if umap is True, otherwise on the features dataframe**
or target dataframe if target is True.

**Args:**

**g**
graphistry graph

**kind**
'nodes' or 'edges'

**cols**
list of columns to use for clustering given *g.featurize* has been run

**use_umap_embedding**
whether to use UMAP embeddings or features dataframe for clustering (default: True)

**Parameters**

- **g** (*Any*)
- **dbscan** (*Any*)
- **kind** (*str*)
- **cols** (*List | str | None*)
- **use_umap_embedding** (*bool*)
- **target** (*bool*)
- **verbose** (*bool*)

graphistry.compute.cluster.**dbscan_predict**(*X*, *model*)

DBSCAN has no predict per se, so we reverse engineer one here from https://stackoverflow.com/questions/27822752/scikit-learn-predicting-new-points-with-dbscan

**Parameters**

- **X** (*DataFrame*)
- **model** (*Any*)

graphistry.compute.cluster.**get_model_matrix**(*g*, *kind*, *cols*, *umap*, *target*)

Allows for a single function to get the model matrix for both nodes and edges as well as targets, embeddings, and features

**Args:**

**g**
graphistry graph

**kind**
'nodes' or 'edges'

**cols**
list of columns to use for clustering given *g.featurize* has been run

**umap**
whether to use UMAP embeddings or features dataframe

**target**
whether to use the target dataframe or features dataframe

**Returns:**
pd.DataFrame: dataframe of model matrix given the inputs

**Parameters**

- **kind** (*str*)
- **cols** (*List | str | None*)

graphistry.compute.cluster.**make_safe_gpu_dataframes**(*X*, *y*, *engine*)

helper method to coerce a dataframe to the correct type (pd vs cudf)

graphistry.compute.cluster.**resolve_cpu_gpu_engine**(*engine*)

**Parameters**
**engine** (*Literal['cuml', 'umap_learn', 'auto']*)

**Return type**
*Literal*['cuml', 'umap_learn']

## 3.8.5 Utilities

### 3.8.5.1 Arrow uploader Module

class graphistry.arrow_uploader.**ArrowUploader**(*server_base_path='http://nginx'*, *view_base_path='http://localhost'*, *name=None*, *description=None*, *edges=None*, *nodes=None*, *node_encodings=None*, *edge_encodings=None*, *token=None*, *dataset_id=None*, *metadata=None*, *certificate_validation=True*, *org_name=None*)

Bases: object

**Parameters**

- **edges** (*Table | None*)
- **nodes** (*Table | None*)
- **org_name** (*str | None*)

**arrow_to_buffer**(*table*)

**Parameters**
**table** (*Table*)

**cascade_privacy_settings**(*mode=None*, *notify=None*, *invited_users=None*, *mode_action=None*, *message=None*)

**Cascade:**

- local (passed in)
- global
- hard-coded

**Parameters**

- **mode** (*Literal['private', 'organization', 'public'] | None*)
- **notify** (*bool | None*)
- **invited_users** (*List[str] | None*)
- **mode_action** (*str | None*)
- **message** (*str | None*)

**property certificate_validation**

`create_dataset`(*json*, *validate=True*)

**Parameters**
`validate` (*bool*)

`property dataset_id: str`

`property description: str`

`property edge_encodings`

`property edges: Table | None`

`g_to_edge_bindings`(*g*)

`g_to_edge_encodings`(*g*)

`g_to_node_bindings`(*g*)

`g_to_node_encodings`(*g*)

`login`(*username*, *password*, *org_name=None*)

`maybe_bindings`(*g*, *bindings*, *base={}*)

`maybe_post_share_link`(*g*)

Skip if never called .privacy() Return True/False based on whether called

**Return type**
bool

`property metadata`

`property name: str`

`property node_encodings`

`property nodes: Table | None`

`property org_name: str | None`

`pkey_login`(*personal_key_id*, *personal_key_secret*, *org_name=None*)

`post`(*as_files=True*, *memoize=True*, *validate=True*)

Note: likely want to pair with self.maybe_post_share_link(g)

**Parameters**

- `as_files` (*bool*)
- `memoize` (*bool*)
- `validate` (*bool*)

`post_arrow`(*arr*, *graph_type*, *opts=''*)

**Parameters**

- `arr` (*Table*)
- `graph_type` (*str*)
- `opts` (*str*)

`post_arrow_generic`(*sub_path*, *tok*, *arr*, *opts=''*)

**Parameters**

- **sub_path** (*str*)
- **tok** (*str*)
- **arr** (*Table*)

**Return type**
*Response*

`post_edges_arrow`(*arr=None*, *opts=''*)

**Parameters**
**arr** (*Table | None*)

`post_edges_file`(*file_path*, *file_type='csv'*)

`post_file`(*file_path*, *graph_type='edges'*, *file_type='csv'*)

`post_g`(*g*, *name=None*, *description=None*)
Warning: main post() does not call this

`post_nodes_arrow`(*arr=None*, *opts=''*)

**Parameters**
**arr** (*Table | None*)

`post_nodes_file`(*file_path*, *file_type='csv'*)

`post_share_link`(*obj_pk*, *obj_type='dataset'*, *privacy=None*)
Set sharing settings. Any settings not passed here will cascade from PyGraphistry or defaults

**Parameters**

- **obj_pk** (*str*)
- **obj_type** (*str*)
- **privacy** (*Privacy | None*)

`refresh`(*token=None*)

`property server_base_path: str`

`sso_get_token`(*state*)
Koa, 04 May 2022 Use state to get token

`sso_login`(*org_name=None*, *idp_name=None*)
Koa, 04 May 2022 Get SSO login auth_url or token

`property token: str`

`verify`(*token=None*)

**Return type**
bool

`property view_base_path: str`

#### 3.8.5.2 Arrow File Uploader Module

class graphistry.ArrowFileUploader.**ArrowFileUploader**(*uploader*)

Bases: object

Implement file API with focus on Arrow support

Memoization in this class is based on reference equality, while plotter is based on hash. That means the plotter resolves different-identity value matches, so by the time ArrowFileUploader compares, identities are unified for faster reference-based checks.

**Example: Upload files with per-session memoization**

uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)

file1_id = afu.create_and_post_file(arr)[0] file2_id = afu.create_and_post_file(arr)[0]

assert file1_id == file2_id # memoizes by default (memory-safe: weak refs)

**Example: Explicitly create a file and upload data for it**

uploader : ArrowUploader arr : pa.Table afu = ArrowFileUploader(uploader)

file1_id = afu.create_file() afu.post_arrow(arr, file_id)

file2_id = afu.create_file() afu.post_arrow(arr, file_id)

assert file1_id != file2_id

**Parameters**

**uploader** (*Any*)

**create_and_post_file**(*arr*, *file_id=None*, *file_opts={}*, *upload_url_opts='erase=true'*, *memoize=True*)

Create file and upload data for it.

Default upload_url_opts='erase=true' throws exceptions on parse errors and deletes upload.

Default memoize=True skips uploading 'arr' when previously uploaded in current session

See File REST API for file_opts (file create) and upload_url_opts (file upload)

**Parameters**

- **arr** (*Table*)
- **file_id** (*str | None*)
- **file_opts** (*dict*)
- **upload_url_opts** (*str*)
- **memoize** (*bool*)

**Return type**

*Tuple*[str, dict]

**create_file**(*file_opts={}*)

Creates File and returns file_id str.

**Defauls:**

- file_type: 'arrow'

See File REST API for file_opts

**Parameters**
**file_opts** (*dict*)

**Return type**
str

**post_arrow**(*arr*, *file_id*, *url_opts='erase=true'*)

Upload new data to existing file id

Default url_opts='erase=true' throws exceptions on parse errors and deletes upload.

See File REST API for url_opts (file upload)

**Parameters**

- **arr** (*Table*)
- **file_id** (*str*)
- **url_opts** (*str*)

**Return type**
dict

**uploader: Any = None**

graphistry.ArrowFileUploader.**DF_TO_FILE_ID_CACHE: WeakKeyDictionary = <WeakKeyDictionary>**

**NOTE: Will switch to pa.Table -> ... when RAPIDS upgrades from pyarrow,**
which adds weakref support

class graphistry.ArrowFileUploader.**MemoizedFileUpload**(*file_id*, *output*)

Bases: object

**Parameters**

- **file_id** (*str*)
- **output** (*dict*)

**file_id: str**

**output: dict**

class graphistry.ArrowFileUploader.**WrappedTable**(*arr*)

Bases: object

**Parameters**
**arr** (*Table*)

**arr: Table**

graphistry.ArrowFileUploader.**cache_arr**(*arr*)

Hold reference to most recent memoization entries Hack until RAPIDS supports Arrow 2.0, when pa.Table becomes weakly referenceable

#### 3.8.5.3 Validation

graphistry.validate.validate_encodings.**cascade_encoding**(*base_encoding*, *encoding*)

graphistry.validate.validate_encodings.**validate_complex**(*encodings*, *kind*, *attributes=None*)

**Parameters**
**attributes** (*List / None*)

graphistry.validate.validate_encodings.**validate_complex_encoding**(*kind*, *mode*, *name*, *enc*, *attributes=None*)

**Parameters**
**attributes** (*List / None*)

graphistry.validate.validate_encodings.**validate_complex_encoding_badge**(*kind*, *mode*, *name*, *badge*)

graphistry.validate.validate_encodings.**validate_complex_encoding_color**(*base_path*, *kind*, *mode*, *name*, *enc*)

graphistry.validate.validate_encodings.**validate_complex_encoding_icon**(*kind*, *mode*, *name*, *enc*)

graphistry.validate.validate_encodings.**validate_edge_encodings**(*encodings*, *edge_attributes=None*)

**Parameters**
**edge_attributes** (*List / None*)

graphistry.validate.validate_encodings.**validate_encodings**(*node_encodings*, *edge_encodings*, *node_attributes=None*, *edge_attributes=None*)

Validate node and edge encodings for compatibility with the given attributes.

This function processes and validates the *node_encodings* and *edge_encodings* against the provided node and edge attributes, ensuring they follow the expected format. If any encoding is invalid, a *ValueError* is raised with details. It is a subset of what the server checks, and run by the uploader.

**Parameters**

- **node_encodings** (*dict*) – Encodings for the nodes in the graph.
- **edge_encodings** (*dict*) – Encodings for the edges in the graph.
- **node_attributes** (*Optional [List]*) – List of node attributes to validate encodings against.
- **edge_attributes** (*Optional [List]*) – List of edge attributes to validate encodings against.

**Returns**
A dictionary containing the validated encodings for nodes and edges, in the form:

**Return type**
dict

**Example:**
node_encodings = {'color': 'blue', 'size': 5} edge_encodings = {'weight': 0.2} result = validate_encodings(node_encodings, edge_encodings) # {'node_encodings': {'color': 'blue', 'size': 5}, 'edge_encodings': {'weight': 0.2}}

graphistry.validate.validate_encodings.**validate_encodings_generic**(*encodings*, *kind*, *required_bindings*)

graphistry.validate.validate_encodings.**validate_mapping**(*mapping*, *base_path*)

graphistry.validate.validate_encodings.**validate_node_encodings**(*encodings*, *node_attributes=None*)

**Parameters**
**node_attributes** (*List / None*)

graphistry.validate.validate_encodings.**validate_style**(*base_path*, *enc*)

#### 3.8.5.4 Versioneer

Git implementation of _version.py.

**exception** graphistry._version.**NotThisMethod**

Bases: Exception

Exception raised if a method is not valid for the current scenario.

**class** graphistry._version.**VersioneerConfig**

Bases: object

Container for Versioneer configuration parameters.

graphistry._version.**get_config**()

Create, populate and return the VersioneerConfig() object.

graphistry._version.**get_keywords**()

Get the keywords needed to look up the version information.

graphistry._version.**get_versions**()

Get version information or return default if unable to do so.

graphistry._version.**git_get_keywords**(*versionfile_abs*)

Extract version information from the given file.

graphistry._version.**git_pieces_from_vcs**(*tag_prefix*, *root*, *verbose*, *run_command=<function run_command>*)

Get version from 'git describe' in the root of the source tree.

This only gets called if the git-archive 'subst' keywords were *not* expanded, and _version.py hasn't already been rewritten with a short version string, meaning we're inside a checked out source tree.

graphistry._version.**git_versions_from_keywords**(*keywords*, *tag_prefix*, *verbose*)

Get version information from git keywords.

graphistry._version.**plus_or_dot**(*pieces*)

Return a + if we don't already have one, else return a .

graphistry._version.**register_vcs_handler**(*vcs*, *method*)

Create decorator to mark a method as the handler of a VCS.

graphistry._version.**render**(*pieces*, *style*)

Render the given version pieces into the requested style.

graphistry._version.**render_git_describe**(*pieces*)

TAG[-DISTANCE-gHEX][-dirty].

Like 'git describe –tags –dirty –always'.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

graphistry._version.**render_git_describe_long**(*pieces*)

TAG-DISTANCE-gHEX[-dirty].

Like 'git describe –tags –dirty –always -long'. The distance/hash is unconditional.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

graphistry._version.**render_pep440**(*pieces*)

Build up version string, with post-release "local version identifier".

Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if you get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty

Exceptions: 1: no tags. git_describe was just HEX. 0+untagged.DISTANCE.gHEX[.dirty]

graphistry._version.**render_pep440_old**(*pieces*)

TAG[.postDISTANCE[.dev0]] .

The ".dev0" means dirty.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

graphistry._version.**render_pep440_post**(*pieces*)

TAG[.postDISTANCE[.dev0]+gHEX] .

The ".dev0" means dirty. Note that .dev0 sorts backwards (a dirty tree will appear "older" than the corresponding clean one), but you shouldn't be releasing software with -dirty anyways.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

graphistry._version.**render_pep440_pre**(*pieces*)

TAG[.post0.devDISTANCE] – No -dirty.

Exceptions: 1: no tags. 0.post0.devDISTANCE

graphistry._version.**run_command**(*commands*, *args*, *cwd=None*, *verbose=False*, *hide_stderr=False*, *env=None*)

Call the given command(s).

graphistry._version.**versions_from_parentdir**(*parentdir_prefix*, *root*, *verbose*)

Try to determine the version from the parent directory name.

Source tarballs conventionally unpack into a directory that includes both the project name and a version string. We will also support searching up two directory levels for an appropriately named parent directory

#### Modules

#### Plugins

#### Plugin Types

#### Graphistry Validate Module

This module contains functions related to the validation of node and edge encodings.

### 3.8.6 Layouts

Native layout engines within Graphistry.

We recommend using the various plugins for additional layouts, such as for tree and hierarchical data diagramming.

#### 3.8.6.1 Group-in-a-Box Layout

graphistry.layout.gib.gib.**group_in_a_box_layout**(*self*, *partition_alg=None*, *partition_params=None*, *layout_alg=None*, *layout_params=None*, *x=0*, *y=0*, *w=None*, *h=None*, *encode_colors=True*, *colors=None*, *partition_key=None*, *engine='auto'*)

Perform a group-in-a-box layout on a graph, supporting both CPU and GPU execution modes.

The layout algorithm groups nodes into clusters and organizes them within rectangular bounding boxes, optionally applying color encoding based on a partitioning scheme.

**Args:**

partition_alg (Optional[str]): The algorithm to use for partitioning the graph nodes. partition_params (Optional[dict]): Parameters for the partition algorithm. layout_alg (Optional[str]): The layout algorithm to arrange nodes within each partition. layout_params (Optional[dict]): Parameters for the layout algorithm. x (int, optional): The x-coordinate for the top-left corner of the layout. Default is 0. y (int, optional): The y-coordinate for the top-left corner of the layout. Default is 0. w (Optional[int]): The width of the layout. If None, automatically determined. h (Optional[int]): The height of the layout. If None, automatically determined. encode_colors (bool, optional): Whether to apply color encoding to nodes based on partitions. Default is True. colors (Optional[List[str]]): List of colors to use for the partitions. partition_key (Optional[str]): The key for partitioning nodes. Default is None. engine (Union[Engine, Literal["auto"]], optional): Execution engine for the layout, either "auto", CPU, or GPU. Default is "auto".

**Returns:**

Plottable: An object representing the layout that can be plotted or visualized.

**Parameters**

- **colors** (*List[str] | None*)
- **partition_key** (*str | None*)
- **engine** (*Engine | Literal['auto']*)

**Return type**

Plottable

graphistry.layout.gib.gib.**resolve_partition_key**(*g*, *partition_key=None*)

#### 3.8.6.2 Modularity Weighted Layout

##### Submodules

##### Module contents

graphistry.layout.modularity_weighted.modularity_weighted.**modularity_weighted_layout**(*g*, *community_col=None*, *community_alg=None*, *community_params=None*, *same_community_wei*
*cross_community_wei*
*edge_influence=2.0*, *engine=EngineAbstract.A*

Compute a modularity-weighted layout, where edges are weighted based on whether they connect nodes in the same community or different communities.

Computes the community if not provided, including with GPU acceleration, using Louvain

**Parameters**

- **g** (Plottable) – input graph
- **community_col** (*str | None*) – column in nodes with community labels
- **community_alg** (*str | None*) – community detection algorithm, e.g., 'louvain' or 'community_multilevel'
- **community_params** (*Dict[str, Any] | None*) – parameters for community detection algorithm
- **same_community_weight** (*float*) – weight for edges connecting nodes in the same community
- **cross_community_weight** (*float*) – weight for edges connecting nodes in different communities
- **edge_influence** (*float*) – influence of edge weights on layout
- **engine** (*EngineAbstract*) – graph engine, e.g., 'pandas', 'cudf', 'auto'. CPU uses igraph algorithms, and GPU, cugraph

**Returns**
graph with layout

**Return type**
Plottable

**Example: Basic**

```
g = g.modularity_weighted_layout()
g.plot()
```

**Example: Use existing community labels**

```
assert 'my_community' in g._nodes.columns
g = g.modularity_weighted_layout(community_col='my_community')
g.plot()
```

**Example: Use GPU-accelerated Louvain algorithm**

```
g = g.modularity_weighted_layout(community_alg='louvain', engine='cudf')
g = g.modularity_weighted_layout(community_alg='community_multilevel',␣
↪engine='pandas')
```

**Example: Use custom layout settings**

```
g = g.modularity_weighted_layout(
    community_col='community',
    same_community_weight=2.0,
    cross_community_weight=0.3,
    edge_influence=2.0
)
g.plot()
```

#### 3.8.6.3 Ring Layouts: Categorical, Continuous, Time

graphistry.layout.ring.categorical.**find_first_numeric_column**(*df*)

**Parameters**
**df** (*Any*)

**Return type**
str

graphistry.layout.ring.categorical.**gen_axis**(*order*, *val_to_r*, *unhandled*, *combine_unhandled*, *append_unhandled*, *axis*, *label*, *reverse*)

**Parameters**

- **order** (*List[str]*)
- **val_to_r** (*Dict[Any, float]*)
- **unhandled** (*Set[Any]*)
- **combine_unhandled** (*bool*)
- **append_unhandled** (*bool*)
- **axis** (*Dict[Any, str] | None*)
- **label** (*Callable[[Any, int, float], str] | None*)
- **reverse** (*bool*)

**Return type**
*List*[*Dict*]

graphistry.layout.ring.categorical.**ring_categorical**(*g*, *ring_col*, *order=None*, *drop_empty=True*, *combine_unhandled=False*, *append_unhandled=True*, *min_r=100*, *max_r=1000*, *axis=None*, *format_axis=None*, *format_labels=None*, *reverse=False*, *play_ms=0*, *engine=EngineAbstract.AUTO*)

Radial graph layout where nodes are positioned based on a categorical column ring_col

Uses GPU when cudf nodes are used, otherwise pandas

min_r, max_r are the first/last axis positions

**G**
Plottable

**Ring_col**
Optional[str] Column name of nodes numerica-typed column; defaults to first numeric node column

**Order**
Optional[List[Any]] Order of axis specified in category values

**Drop_empty**
bool (default True) Whether to drop axis when no values populating them

**Combine_unhandled**
bool (default False) Whether to collapse all unexpected values into one ring or one-per-unique-value

**Append_unhandled**
bool (default True) Whether to append or prepend the unexpected items axis

**Min_r**
float Minimum radius, default 100

**Max_r**
float Maximum radius, default 1000

**Ring_step**
Optional[float] Distance between rings in terms of pixels

**Axis**
Optional[Dict[Any, str]], Set to provide labels for each ring by mapping from the categorical input domain values. Requires all values to be mapped.

**Format_axis**
Optional[Callable[[List[Dict]], List[Dict]]] Optional transform function to format axis

**Format_label**
Optional[Callable[[Any, int, float], str]] Optional transform function to format axis label text based on axis value, ring number, and ring position

**Reverse**
bool Reverse the direction of the rings

**Play_ms**
int initial layout time in milliseconds, default 2000

**Engine**
Union[EngineAbstract, str], default EngineAbstract.AUTO, pick CPU vs GPU engine via 'auto', 'pandas', 'cudf'

**Returns**
Plotter

**Return type**
Plotter

**Parameters**

- **g** (Plottable)
- **ring_col** (*str*)
- **order** (*List[Any] | None*)
- **drop_empty** (*bool*)
- **combine_unhandled** (*bool*)
- **append_unhandled** (*bool*)
- **min_r** (*float*)
- **max_r** (*float*)
- **axis** (*Dict[Any, str] | None*)
- **format_axis** (*Callable[[List[Dict]], List[Dict]] | None*)
- **format_labels** (*Callable[[Any, int, float], str] | None*)
- **reverse** (*bool*)
- **play_ms** (*int*)
- **engine** (*EngineAbstract | str*)

**Example: Minimal categorical ring layout**

```
assert 'a_cat_node_column' in g._nodes
g.ring_categorical_layout('a_cat_node_column').plot()
```

**Example: Categorical ring layout with a few rings, and rest as Other**

```
g2 = g.ring_categorical_layout('a_cat_node_column', order=['a', 'b', 'c'], combine_
↪unhandled=True)
g2.plot()
```

**Example: Categorical ring layout with relabeled axis rings**

```
g2 = g.ring_categorical_layout(
    'a_cat_node_column',
    axis={
        'a': 'ring a',
        'b': 'ring b',
        'c': 'ring c'
    }
)
g2.plot()
```

**Example: Categorical ring layout without labels**

```
EMPTY_AXIS_LIST = []
g2 = g.ring_categorical_layout('a_cat_node_column', format_labels=lambda axis:␣
↪EMPTY_AXIS_LIST)
```

**Example: Categorical ring layout with specific first and last ring positions**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_categorical_layout(
    'a_cat_node_column',
    min_r=400,
    max_r=1000,
)
g2.plot()
```

**Example: Categorical ring layout in reverse order**

```
g2 = g.ring_categorical_layout('a_cat_node_column', order=['a', 'b', 'c'],␣
↪reverse=True)
g2.plot()
```

graphistry.layout.ring.continuous.**find_first_numeric_column**(*df*)

**Parameters**
**df** (*Any*)

**Return type**
str

graphistry.layout.ring.continuous.**gen_axis**(*axis_input*, *num_rings*, *v_start*, *v_step*, *r_start*, *step_r*, *label=None*, *reverse=False*)

**Parameters**

- **axis_input** (*Dict[float, str] | List[str] | None*)
- **num_rings** (*int*)
- **v_start** (*float*)
- **v_step** (*float*)
- **r_start** (*float*)
- **step_r** (*float*)
- **label** (*Callable[[float, int, float], str] | None*)
- **reverse** (*bool*)

**Return type**
*List*[*Dict*]

graphistry.layout.ring.continuous.**ring_continuous**(*g*, *ring_col=None*, *v_start=None*, *v_end=None*, *v_step=None*, *min_r=100*, *max_r=1000*, *normalize_ring_col=True*, *num_rings=None*, *ring_step=None*, *axis=None*, *format_axis=None*, *format_labels=None*, *reverse=False*, *play_ms=0*, *engine=EngineAbstract.AUTO*)

Radial graph layout where nodes are positioned based on a numeric-typed column ring_col

Uses GPU when cudf nodes are used, otherwise pandas

min_r, max_r are the first/last axis positions

**optional v_start, v_end are used to line up the input value domain to the axis:**

- v_start: corresponds to the first axis at min_r, defaulting to g._nodes[ring_col].min()
- v_end: corresponds to the last axis at max_r, defaulting to g._nodes[ring_col].max()

**G**
Plottable

**Ring_col**
Optional[str] Column name of nodes numerica-typed column; defaults to first numeric node column

**V_start**
Optional[float] Value at innermost axis (at min_r), defaults to g._nodes[ring_col].min()

**V_end**
Optional[float] Value at outermost axis (at max_r), defaults to g._nodes[ring_col].max()

**V_step**
Optional[float] Distance between rings in terms of ring column value domain

**Min_r**
float Minimum radius, default 100

**Max_r**
float Maximum radius, default 1000

**Normalize_ring_col**
bool, default True, Whether to recale to min/max r, or pass through existing values

**Num_rings**
Optional[int] Number of rings

**Ring_step**
Optional[float] Distance between rings in terms of pixels

**Axis**
Optional[Union[Dict[float,str],List[str]]], Set to provide labels for each ring, and in dict mode, also specify radius for each

**Format_axis**
Optional[Callable[[List[Dict]], List[Dict]]] Optional transform function to format axis

**Format_label**
Optional[Callable[[float, int, float], str]] Optional transform function to format axis label text based on axis value, ring number, and ring width

**Reverse**
bool Reverse the direction of the rings

**Play_ms**
int initial layout time in milliseconds, default 2000

**Engine**
Union[EngineAbstract, str], default EngineAbstract.AUTO, pick CPU vs GPU engine via 'auto', 'pandas', 'cudf'

**Returns**
Plotter

**Return type**
Plotter

**Parameters**

- **g** (Plottable)
- **ring_col** (*str | None*)
- **v_start** (*float | None*)
- **v_end** (*float | None*)
- **v_step** (*float | None*)
- **min_r** (*float | None*)
- **max_r** (*float | None*)
- **normalize_ring_col** (*bool*)
- **num_rings** (*int | None*)
- **ring_step** (*float | None*)
- **axis** (*Dict[float, str] | List[str] | None*)
- **format_axis** (*Callable[[List[Dict]], List[Dict]] | None*)
- **format_labels** (*Callable[[float, int, float], str] | None*)
- **reverse** (*bool*)
- **play_ms** (*int*)
- **engine** (*EngineAbstract | str*)

**Example: Minimal continuous ring layout**

```
g.ring_continuous_layout().plot()
```

**Example: Continuous ring layout**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_continuous_layout('my_numeric_col')
g2.plot()
```

**Example: Continuous ring layout with 7 rings**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_continuous_layout('my_numeric_col', num_rings=7)
g2.plot()
```

**Example: Continuous ring layout using small steps**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_continuous_layout('my_numeric_col', ring_step=20.0)
g2.plot()
```

**Example: Continuous ring layout without labels**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
EMPTY_AXIS_LIST = []
g2 = g.ring_continuous_layout('my_numeric_col', format_labels=lambda axis: EMPTY_
↪AXIS_LIST)
```

**Example: Continuous ring layout with specific first and last ring positions**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_continuous_layout(
    'my_numeric_col',
    min_r=200,
    max_r=2000,
    v_start=32,  # corresponding column value at first axis radius at pixel radius␣
↪200
    v_end=83,  # corresponding column value at last axius radius at pixel radius␣
↪2000
)
g2.plot()
```

**Example: Continuous ring layout in reverse order**

```
assert 'float' in g._nodes.my_numeric_col.dtype.name
g2 = g.ring_continuous_layout('my_numeric_col', reverse=True)
g2.plot()
```

graphistry.layout.ring.time.**TimeUnit**

Time unit for axis labels

- 's': seconds
- 'm': minutes
- 'h': hours
- 'D': days
- 'W': weeks
- 'M': months
- 'Y': years
- 'C': centuries

alias of `Literal`['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C']

graphistry.layout.ring.time.**find_round_bin_width**(*duration*, *time_unit=None*)

**Parameters**

- **duration** (*timedelta64*)
- **time_unit** (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C'] | None*)

**Return type**

*Tuple*[*Literal*['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C'], ~pandas._libs.tslibs.offsets.DateOffset, ~numpy.timedelta64]

graphistry.layout.ring.time.**gen_axis**(*num_rings*, *time_start*, *step_dur*, *rounded_set_offset*, *round_unit*, *r_start*, *r_end*, *scalar*, *label=None*, *reverse=False*)

**Parameters**

- **num_rings** (*int*)
- **time_start** (*datetime64*)
- **step_dur** (*timedelta64*)
- **rounded_set_offset** (*DateOffset*)
- **round_unit** (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C']*)
- **r_start** (*float*)
- **r_end** (*float*)
- **scalar** (*float*)
- **label** (*Callable[[datetime64, int, timedelta64], str] | None*)
- **reverse** (*bool*)

**Return type**
*List*[*Dict*]

graphistry.layout.ring.time.**pretty_print_time**(*time*, *round_unit*)

**Parameters**

- **time** (*datetime64*)
- **round_unit** (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C']*)

**Return type**
str

graphistry.layout.ring.time.**round_to_nearest**(*time*, *unit*)

**Parameters**

- **time** (*datetime64*)
- **unit** (*timedelta64*)

**Return type**
*datetime64*

graphistry.layout.ring.time.**time_ring**(*g*, *time_col=None*, *num_rings=None*, *time_start=None*, *time_end=None*, *time_unit=None*, *min_r=100*, *max_r=1000*, *reverse=False*, *format_axis=None*, *format_label=None*, *play_ms=2000*, *engine=EngineAbstract.AUTO*)

Radial graph layout where nodes are positioned based on a datetime64-typed column time_col

Uses GPU when cudf nodes are used, otherwise pandas with custom start and end times

**G**
Plottable

**Time_col**
Optional[str] Column name of nodes datetime64-typed column; defaults to first node datetime64 column

**Num_rings**
Optional[int] Number of rings

**Time_start**
Optional[numpy.datetime64] First ring and axis label

**Time_end**
Optional[numpy.datetime64] Last ring and axis label

**Time_unit**
Optional[TimeUnit] Time unit for axis labels

**Min_r**
float Minimum radius, default 100

**Max_r**
float Maximum radius, default 1000

**Reverse**
bool Reverse the direction of the rings in terms of time

**Format_axis**
Optional[Callable[[List[Dict]], List[Dict]]] Optional transform function to format axis

**Format_label**
Optional[Callable[[numpy.datetime64, int, numpy.timedelta64], str]] Optional transform function to format axis label text based on axis time, ring number, and ring duration width

**Play_ms**
int initial layout time in milliseconds, default 2000

**Engine**
Union[EngineAbstract, str], default EngineAbstract.AUTO, pick CPU vs GPU engine via 'auto', 'pandas', 'cudf'

**Returns**
Plotter

**Return type**
Plotter

**Parameters**

- `g` (*Plottable*)
- `time_col` (*str | None*)
- `num_rings` (*int | None*)
- `time_start` (*datetime64 | None*)
- `time_end` (*datetime64 | None*)
- `time_unit` (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C'] | None*)
- `min_r` (*float*)
- `max_r` (*float*)
- `reverse` (*bool*)
- `format_axis` (*Callable[[List[Dict]], List[Dict]] | None*)

- **format_label** (*Callable[[datetime64, int, timedelta64], str] | None*)
- **play_ms** (*int*)
- **engine** (*EngineAbstract | str*)

**Example: Minimal time ring layout**

```
g.time_ring_layout().plot()
```

**Example: Time ring layout**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
g2 = g.time_ring_layout('my_time_col')
g2.plot()
```

**Example: Time ring layout with 7 rings**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
g2 = g.time_ring_layout('my_time_col', num_rings=7)
g2.plot()
```

**Example: Time ring layout using days**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
g2 = g.time_ring_layout('my_time_col', time_unit='D')
g2.plot()
```

**Example: Time ring layout without labels**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
EMPTY_AXIS_LIST = []
g2 = g.time_ring_layout('my_time_col', format_labels=lambda axis: EMPTY_AXIS_LIST)
```

**Example: Time ring layout with specific first and last ring positions**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
g2 = g.time_ring_layout('my_time_col', min_r=200, max_r=2000)
g2.plot()
```

**Example: Time ring layout in reverse order**

```
assert 'datetime64' in g._nodes.my_time_col.dtype.name
g2 = g.time_ring_layout('my_time_col', reverse=True)
g2.plot()
```

graphistry.layout.ring.time.**time_stats**(*s*, *num_rings=20*, *time_start=None*, *time_end=None*, *time_unit=None*)

**Parameters**

- **s** (*Series*)
- **num_rings** (*int | None*)
- **time_start** (*datetime64 | None*)
- **time_end** (*datetime64 | None*)

- **time_unit** (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C'] | None*)

**Return type**
*Tuple*[*Literal*['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C'], ~numpy.timedelta64, ~pandas._libs.tslibs.offsets.DateOffset, ~numpy.datetime64, ~numpy.datetime64, int]

graphistry.layout.ring.time.**unit_to_timedelta**(*unit*)

**Parameters**
**unit** (*Literal['s', 'm', 'h', 'D', 'W', 'M', 'Y', 'C']*)

**Return type**
*timedelta64*

#### 3.8.6.4 Sugiyama Layout

We recommend using plugin versions of this layout instead, such as from *igraph*

##### graphistry.layout.sugiyama.sugiyamaLayout module

class graphistry.layout.sugiyama.sugiyamaLayout.**SugiyamaLayout**(*g*)

Bases: object

The classic Sugiyama layout aka layered layout.

- See https://en.wikipedia.org/wiki/Layered_graph_drawing
- Excellent explanation: https://www.youtube.com/watch?v=Z0RGCWxvCxA

**Attributes**

- **dirvh (int): the current aligment state for alignment policy:**
  dirvh=0 -> dirh=+1, dirv=-1: leftmost upper dirvh=1 -> dirh=-1, dirv=-1: rightmost upper dirvh=2 -> dirh=+1, dirv=+1: leftmost lower dirvh=3 -> dirh=-1, dirv=+1: rightmost lower
- order_iter (int): the default number of layer placement iterations
- order_attr (str): set attribute name used for layer ordering
- xspace (int): horizontal space between vertices in a layer
- yspace (int): vertical space between layers
- dw (int): default width of a vertex
- dh (int): default height of a vertex
- g (GraphBase): the graph component reference
- layers (list[sugiyama.layer.Layer]): the list of layers
- layoutVertices (dict): associate vertex (possibly dummy) with their sugiyama attributes
- ctrls (dict): associate edge with all its vertices (including dummies)
- dag (bool): the current acyclic state
- init_done (bool): True if things were initialized

**Example**

```
g = nx.generators.connected_watts_strogatz_graph(1000, 2, 0.3)
# render
SugiyamaLayout.draw(g)
# positions
positions_dictionary = SugiyamaLayout.arrange(g)
```

**Parameters**
**g** (GraphBase)

**static arrange**(*obj*, *iteration_count=1.5*, *source_column='source'*, *target_column='target'*, *layout_direction=0*, *topological_coordinates=False*, *root=None*, *include_levels=False*)

Returns the positions from a Sugiyama layout iteration.

**Parameters**

- **layout_direction** –
  - 0: top-to-bottom
  - 1: right-to-left
  - 2: bottom-to-top
  - 3: left-to-right
- **obj** (*DataFrame* / Graph) – can be a Sugiyama graph or a Pandas frame.
- **iteration_count** – increase the value for diminished crossings
- **source_column** – if a Pandas frame is given, the name of the column with the source of the edges
- **target_column** – if a Pandas frame is given, the name of the column with the target of the edges
- **topological_coordinates** – whether to use coordinates with the x-values in the [0,1] range and the y-value equal to the layer index.
- **include_levels** – whether the tree-level is included together with the coordinates. If so, you get a triple (x,y,level).
- **root** – optional list of roots.

**Returns**
a dictionary of positions.

**create_dummies**(*e*)

Creates and defines all dummy vertices for edge e.

**ctrls: Dict[*Vertex*, *LayoutVertex*]**

**property dirh**

**property dirv**

**property dirvh**

`draw_step()`

Iterator that computes all vertices coordinates and edge routing after just one step (one layer after the other from top to bottom to top). Use it only for "animation" or debugging purpose.

`dummyctrl(r, control_vertices)`

Creates a DummyVertex at layer r inserted in the ctrl dict of the associated edge and layer.

**Arguments**

- r (int): layer value
- ctrl (dict): the edge's control vertices

**Returns**

sugiyama.DummyVertex : the created DummyVertex.

`static ensure_root_is_vertex(g, root)`

Turns the given list of roots (names or data) to actual vertices in the given graph.

**Parameters**

- **g** (Graph) – the graph wherein the given roots names are supposed to be
- **root** (*object*) – the data or the vertex

**Returns**

the list of vertices to use as roots

`find_nearest_layer(start_vertex)`

`static graph_from_pandas(df, source_column='source', target_column='target')`

`static has_cycles(obj, source_column='source', target_column='target')`

**Parameters**

**obj** (*DataFrame* | Graph)

`initialize(root=None)`

Initializes the layout algorithm.

**Parameters:**

- root (Vertex): a vertex to be used as root

`layers: List[Layer]`

`layout(iteration_count=1.5, topological_coordinates=False, layout_direction=0)`

Compute every node coordinates after converging to optimal ordering by N rounds, and finally perform the edge routing.

**Parameters**

**topological_coordinates** – whether to use ( [0,1], layer index) coordinates

`layoutVertices`

The map from vertex to LayoutVertex.

`layout_edges()`

Basic edge routing applied only for edges with dummy points. Enhanced edge routing can be performed by using the appropriate

**ordering_step**(*oneway=False*)

iterator that computes all vertices ordering in their layers (one layer after the other from top to bottom, to top again unless oneway is True).

**set_coordinates**()

Computes all vertex coordinates using Brandes & Kopf algorithm. See https://www.semanticscholar.org/paper/Fast-and-Simple-Horizontal-Coordinate-Assignment-Brandes-Köpf/69cb129a8963b21775d6382d15b0b447b01eb1f8

**set_topological_coordinates**(*layout_direction=0*)

**xspace: int**

**yspace: int**

### Module contents

#### 3.8.6.5 Utils

### Submodules

### graphistry.layout.utils.dummyVertex module

**class** graphistry.layout.utils.dummyVertex.**DummyVertex**(*r=None*)

Bases: *LayoutVertex*

A DummyVertex is used for edges that span over several layers, it's inserted in every inner layer.

**Attributes**

- view (viewclass): since a DummyVertex is acting as a Vertex, it must have a view.
- ctrl (list[_sugiyama_attr]): the list of associated dummy vertices.

**inner**(*direction*)

True if a neighbor in the given direction is *dummy*.

**neighbors**(*direction*)

Reflect the Vertex method and returns the list of adjacent vertices (possibly dummy) in the given direction. :param direction: +1 for the next layer (children) and -1 (parents) for the previous

**Parameters**
**direction** (*int*)

### graphistry.layout.utils.geometry module

graphistry.layout.utils.geometry.**angle_between_vectors**(*p1*, *p2*)

graphistry.layout.utils.geometry.**lines_intersection**(*xy1*, *xy2*, *xy3*, *xy4*)

Returns the intersection of two lines.

graphistry.layout.utils.geometry.**new_point_at_distance**(*pt*, *distance*, *angle*)

graphistry.layout.utils.geometry.**rectangle_point_intersection**(*rec*, *p*)

Returns the intersection point between the Rectangle (w,h) that characterize the rec object and the line that goes from the recs' object center to the 'p' point.

graphistry.layout.utils.geometry.**set_round_corner**(*e*, *pts*)

graphistry.layout.utils.geometry.**setcurve**(*e*, *pts*, *tgs=None*)

Returns the spline curve that path through the list of points P. The spline curve is a list of cubic bezier curves (nurbs) that have matching tangents at their extreme points. The method considered here is taken from "The NURBS book" (Les A. Piegl, Wayne Tiller, Springer, 1997) and implements a local interpolation rather than a global interpolation.

**Args:**

e: pts: tgs:

Returns:

graphistry.layout.utils.geometry.**size_median**(*recs*)

graphistry.layout.utils.geometry.**tangents**(*P*, *n*)

### graphistry.layout.utils.layer module

**class** graphistry.layout.utils.layer.**Layer**(*iterable=()*, */*)

Bases: `list`

Layer is where Sugiyama layout organises vertices in hierarchical lists. The placement of a vertex is done by the Sugiyama class, but it highly relies on the *ordering* of vertices in each layer to reduce crossings. This ordering depends on the neighbors found in the upper or lower layers.

**Attributes:**

layout (SugiyamaLayout): a reference to the sugiyama layout instance that contains this layer upper (Layer): a reference to the *upper* layer (layer-1) lower (Layer): a reference to the *lower* layer (layer+1) crossings (int) : number of crossings detected in this layer

**Methods:**

setup (layout): set initial attributes values from provided layout nextlayer(): returns *next* layer in the current layout's direction parameter. prevlayer(): returns *previous* layer in the current layout's direction parameter. order(): compute *optimal* ordering of vertices within the layer.

**crossings = None**

**layout = None**

**lower = None**

**neighbors**(*v*)

neighbors refer to upper/lower adjacent nodes. Note that v.neighbors() provides neighbors of v in the graph, while this method provides the Vertex and DummyVertex adjacent to v in the upper or lower layer (depending on layout.dirv state).

**nextlayer()**

**order()**

**prevlayer()**

**setup**(*layout*)

**upper = None**

### graphistry.layout.utils.layoutVertex module

**class** graphistry.layout.utils.layoutVertex.**LayoutVertex**(*layer=None*, *is_dummy=0*)

Bases: object

The Sugiyama layout adds new attributes to vertices. These attributes are stored in an internal _sugimyama_vertex_attr object.

**Attributes:**

layer (int): layer number dummy (0/1): whether the vertex is a dummy pos (int): the index of the vertex within the layer x (list(float)): the list of computed horizontal coordinates of the vertex bar (float): the current barycenter of the vertex

**Parameters**

**layer** (*int | None*)

### graphistry.layout.utils.poset module

**class** graphistry.layout.utils.poset.**Poset**(*collection=[]*)

Bases: object

Poset class implements a set but allows to integrate over the elements in a deterministic way and to get specific objects in the set. Membership operator defaults to comparing ___hash___ of objects but Poset allows to check for ___cmp___/___eq___ membership by using contains___cmp___(obj)

**add**(*obj*)

**contains__cmp__**(*obj*)

**copy**()

**deepcopy**()

**difference**(**args*)

**get**(*obj*)

**index**(*obj*)

**intersection**(**args*)

**issubset**(*other*)

**issuperset**(*other*)

**remove**(*obj*)

**symmetric_difference**(**args*)

**union**(*other*)

**update**(*other*)

### graphistry.layout.utils.rectangle module

class graphistry.layout.utils.rectangle.**Rectangle**(*w=1*, *h=1*)

Bases: object

Rectangular region.

### graphistry.layout.utils.routing module

class graphistry.layout.utils.routing.**EdgeViewer**

Bases: object

**setpath**(*pts*)

graphistry.layout.utils.routing.**route_with_lines**(*e*, *pts*)

Basic edge routing with lines. The layout pass has already provided to list of points through which the edge shall be drawn. We just compute the position where to adjust the tail and head.

graphistry.layout.utils.routing.**route_with_rounded_corners**(*e*, *pts*)

graphistry.layout.utils.routing.**route_with_splines**(*e*, *pts*)

Enhanced edge routing where 'corners' of the above polyline route are rounded with a Bezier curve.

### Module contents

### graphistry.layout.graph.edge module

class graphistry.layout.graph.edge.**Edge**(*x*, *y*, *w=1*, *data=None*, *connect=False*)

Bases: *EdgeBase*

A graph edge.

**Attributes**

- data (object): an optional payload
- w (int): an optional weight associated with the edge (default 1) used by Dijkstra to find min-flow paths.
- feedback (bool): whether the Tarjan algorithm has inverted this edge to de-cycle the graph.

**Parameters**

- **w** (*int*)
- **data** (*object*)

**attach**()

Attach this edge to the edge collections of the vertices.

**data: object**

**detach**()

Removes this edge from the edge collections of the vertices.

**feedback: bool**

**w: int**

### graphistry.layout.graph.edgeBase module

**class** graphistry.layout.graph.edgeBase.**EdgeBase**(*x*, *y*)

Bases: object

Base class for edges.

**Attributes**

- degree (int): degree of the edge (number of unique vertices).
- v (list[Vertex]): list of vertices associated with this edge.

**degree: int**

Is 0 if a loop, otherwise 1.

### graphistry.layout.graph.graph module

**class** graphistry.layout.graph.graph.**Graph**(*vertices=None*, *edges=None*, *directed=True*)

Bases: object

**N**(*v*, *f_io=0*)

**add_edge**(*e*)

add edge e and its vertices into the Graph possibly merging the associated graph_core components

**add_edges**(*edges*)

**Parameters**
**edges** (*List*)

**add_vertex**(*v*)

add vertex v into the Graph as a new component

**component_class**

alias of *GraphBase*

**connected**()

returns the list of components

**deg_avg**()

the average degree of vertices

**deg_max**()

the maximum degree of vertices

**deg_min**()

the minimum degree of vertices

**edges**()

**eps**()

the graph epsilon value (norm/order), average number of edges per vertex.

`get_vertex_from_data`(*data*)

`get_vertices_count`()

`norm`()
the norm of the graph (number of edges)

`order`()
the order of the graph (number of vertices)

`path`(*x*, *y*, *f_io=0*, *hook=None*)

`remove_edge`(*e*)
remove edge e possibly spawning two new cores if the graph_core that contained e gets disconnected.

`remove_vertex`(*x*)
remove vertex v and all its edges.

`vertices`()
see graph_core

### graphistry.layout.graph.graphBase module

`class graphistry.layout.graph.graphBase.GraphBase`(*vertices=None*, *edges=None*, *directed=True*)
Bases: `object`

A connected graph of Vertex/Edge objects. A GraphBase is a *component* of a Graph that contains a connected set of Vertex and Edges.

**Attributes:**
verticesPoset (Poset[Vertex]): the partially ordered set of vertices of the graph. edgesPoset (Poset[Edge]): the partially ordered set of edges of the graph. loops (set[Edge]): the set of *loop* edges (of degree 0). directed (bool): indicates if the graph is considered *oriented* or not.

`N`(*v*, *f_io=0*)

`add_edge`(*e*)
add edge e. At least one of its vertex must belong to the graph, the other being added automatically.

`add_single_vertex`(*v*)
allow a GraphBase to hold a single vertex.

`complement`(*G*)

`constant_function`(*value*)

`contract`(*e*)

`deg_avg`()
the average degree of vertices

`deg_max`()
the maximum degree of vertices

`deg_min`()
the minimum degree of vertices

**dft**(*start_vertex=None*)

**dijkstra**(*x*, *f_io=0*, *hook=None*)

shortest weighted-edges paths between x and all other vertices by dijkstra's algorithm with heap used as priority queue.

**edges**(*cond=None*)

generates an iterator over edges, with optional filter

**eps**()

the graph epsilon value (norm/order), average number of edges per vertex.

**get_scs_with_feedback**(*roots=None*)

Minimum FAS algorithm (feedback arc set) creating a DAG. Returns the set of strongly connected components ("scs") by using Tarjan algorithm. These are maximal sets of vertices such that there is a path from each vertex to every other vertex. The algorithm performs a DFS from the provided list of root vertices. A cycle is of course a strongly connected component,but a strongly connected component can include several cycles. The Feedback Acyclic Set of edge to be removed/reversed is provided by marking the edges with a "feedback" flag. Complexity is O(V+E).

**Parameters**
**roots**

**Returns**

**leaves**()

returns the list of *leaves* (vertices with no outward edges).

**matrix**(*cond=None*)

This associativity matrix is like the adjacency matrix but antisymmetric. Returns the associativity matrix of the graph component

**Parameters**
**cond** – same a the condition function in vertices().

**Returns**
array

**norm**()

The size of the edge poset (number of edges).

**order**()

the order of the graph (number of vertices)

**partition**()

**path**(*x*, *y*, *f_io=0*, *hook=None*)

shortest path between vertices x and y by breadth-first descent, contrained by f_io direction if provided. The path is returned as a list of Vertex objects. If a *hook* function is provided, it is called at every vertex added to the path, passing the vertex object as argument.

**remove_edge**(*e*)

remove Edge e, asserting that the resulting graph is still connex.

**remove_vertex**(*x*)

remove Vertex x and all associated edges.

**roots**()

returns the list of *roots* (vertices with no inward edges).

**spans**(*vertices*)

**union_update**(*G*)

**vertices**(*cond=None*)

generates an iterator over vertices, with optional filter

### graphistry.layout.graph.vertex module

**class** graphistry.layout.graph.vertex.**Vertex**(*data=None*)

Bases: *VertexBase*

Vertex class enhancing a VertexBase with graph-related features.

**Attributes**

component (GraphBase): the component of connected vertices that contains this vertex. By default, a vertex belongs no component but when it is added in a graph, c points to the connected component in this graph. data (object) : an object associated with the vertex.

**property index**

### graphistry.layout.graph.vertexBase module

**class** graphistry.layout.graph.vertexBase.**VertexBase**

Bases: object

Base class for vertices.

**Attributes**

e (list[Edge]): list of edges associated with this vertex.

**degree**()

degree() : degree of the vertex (number of edges).

**detach**()

removes this vertex from all its edges and returns this list of edges.

**e_dir**(*dir*)

either e_in, e_out or all edges depending on provided direction parameter (>0 means outward).

**e_from**(*x*)

returns the Edge from vertex v directed toward this vertex.

**e_in**()

e_in() : list of edges directed toward this vertex.

**e_out**()

e_out(): list of edges directed outward this vertex.

`e_to`(*y*)

returns the Edge from this vertex directed toward vertex v.

`e_with`(*v*)

return the Edge with both this vertex and vertex v

`neighbors`(*direction=0*)

Returns the neighbors of this vertex. List of neighbor vertices in all directions (default) or in filtered f_io direction (>0 means outward).

**Parameters**
`direction` –

- 0: parent and children
- -1: parents
- +1: children

**Returns**
list of vertices

#### Module contents

#### 3.8.6.6 Layout plugins: igraph, graphviz, and more

Several plugins provide a large variety of additional layouts:

- *cuGraph* : GPU-accelerated FA2, a naive version of Graphistry's layout engine
- *graphviz*: Especially strong at tree and hierarchical data diagramming such as the dot engine
- *igraph* : A variety of layouts, including Sugiyama, Fruchterman-Reingold, and Kamada-Kawai
- NetworkX: A variety of layouts

### 3.8.7 Plugins

#### 3.8.7.1 Data Providers

PyGraphistry supports a variety of data providers natively

#### Azure Cosmos DB for Apache Gremlin

Azure Cosmos DB supports Gremlin graph queries

`class graphistry.gremlin.CosmosMixin`(**args*, ***kwargs*)

Bases: `object`

`cosmos`(*COSMOS_ACCOUNT=None*, *COSMOS_DB=None*, *COSMOS_CONTAINER=None*, *COSMOS_PRIMARY_KEY=None*, *gremlin_client=None*)

Provide credentials as arguments, as environment variables, or by providing a gremlin-python client Environment variable names are the same as the constructor argument names If no client provided, create (connect)

**Example: Login and plot**

```
import graphistry
(graphistry
    .cosmos(
        COSMOS_ACCOUNT='a',
        COSMOS_DB='b',
        COSMOS_CONTAINER='c',
        COSMOS_PRIMARY_KEY='d')
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Parameters**

- **COSMOS_ACCOUNT** (*str* / *None*)
- **COSMOS_DB** (*str* / *None*)
- **COSMOS_CONTAINER** (*str* / *None*)
- **COSMOS_PRIMARY_KEY** (*str* / *None*)
- **gremlin_client** (*Any* / *None*)

### Gremlin - Apache ThinkerPop

Gremlin is a graph traversal language that is part of the Apache TinkerPop graph computing framework.

As an open source technology, multiple databases support it.

**class** graphistry.gremlin.**GremlinMixin**(**args*, *gremlin_client=None*, ***kwargs*)

Bases: object

Universal Gremlin<>pandas/graphistry functionality across Gremlin connectors

Currently serializes queries as strings instead of bytecode in order to support cosmosdb

**Parameters**

**gremlin_client** (*Any* / *None*)

**connect**()

Use previously provided credentials to connect. Disconnect any preexisting clients.

**drop_graph**()

Remove all graph nodes and edges from the database

**fetch_edges**(*batch_size=1000*, *dry_run=False*, *verbose=False*, *ignore_errors=False*)

Enrich edges by matching g._edges to gremlin edges

**Return type**

Plottable | *List*[str]

**fetch_nodes**(*batch_size=1000*, *dry_run=False*, *verbose=False*, *ignore_errors=False*)

Enrich nodes by matching g._node to gremlin nodes If no g._nodes table available, first synthesize g._nodes from g._edges

**Return type**

Plottable | *List*[str]

**gremlin**(*queries*)

Run one or more gremlin queries and get back the result as a graph object To support cosmosdb, sends as strings

**Example: Login and plot **

```
import graphistry
(graphistry
    .gremlin_client(my_gremlin_client)
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Parameters**
**queries** (*str | Iterable[str]*)

**Return type**
Plottable

**gremlin_client**(*gremlin_client*)

Set the Gremlin client to interact with the Gremlin-enabled database.

This method allows you to pass a custom Gremlin Python client to interact with databases like CosmosDB using Gremlin queries. It stores the client for subsequent queries in the Graphistry workflow.

**Parameters**
**gremlin_client** (*gremlin_python.driver.client.Client*) – Instance of the Gremlin Python client.

**Returns**
The instance of Graphistry, updated with the Gremlin client.

**Return type**
*GremlinMixin*

**Example: Login and plot**

```
import graphistry
from gremlin_python.driver.client import Client

my_gremlin_client = Client(
    f'wss://MY_ACCOUNT.gremlin.cosmosdb.azure.com:443/',
    'g',
    username=f"/dbs/MY_DB/colls/{self.COSMOS_CONTAINER}",
    password=self.COSMOS_PRIMARY_KEY,
    message_serializer=GraphSONSerializersV2d0())

g = (graphistry
    .gremlin_client(my_gremlin_client)
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
)

g.plot()
```

**gremlin_run**(*queries*, *throw=False*)

**Parameters**
**queries** (*Iterable[str]*)

**Return type**
*Any*

**resultset_to_g**(*resultsets*, *mode='infer'*, *verbose=False*, *ignore_errors=False*)

Convert traversal results to graphistry object with ._nodes, ._edges If only received nodes or edges, populate that field For custom src/dst/node bindings, passing in a Graphistry instance with .bind(source=.., destination=..., node=...) Otherwise, will do src/dst/id For dict results (ex: valueMap/elementMap), specify mode='nodes' ('edges'), else will inspect field 'type'

**Parameters**

- **resultsets** (*Any | Iterable[Any]*)
- **mode** (*str*)

**Return type**
Plottable

### Amazon Neptune

Amazon Neptune is a managed graph database by Amazon. It supports OpenCypher, RDF, Gremlin, and various analytical capabilities.

**class** graphistry.gremlin.**NeptuneMixin**(**args*, ***kwargs*)

Bases: object

**neptune**(*NEPTUNE_READER_HOST=None*, *NEPTUNE_READER_PORT=None*, *NEPTUNE_READER_PROTOCOL=None*, *endpoint=None*, *gremlin_client=None*)

Provide credentials as arguments, as environment variables, or by providing a gremlin-python client Environment variable names are the same as the constructor argument names If endpoint provided, do not need host/port/protocol If no client provided, create (connect)

**Example: Login and plot via parrams**

```
import graphistry
(graphistry
    .neptune(
        NEPTUNE_READER_PROTOCOL='wss'
        NEPTUNE_READER_HOST='neptunedbcluster-xyz.cluster-ro-abc.us-east-
↪1.neptune.amazonaws.com'
        NEPTUNE_READER_PORT='8182'
    )
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Example: Login and plot via env vars**

```
import graphistry
(graphistry
    .neptune()
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Example: Login and plot via endpoint**

```
import graphistry
(graphistry
    .neptune(endpoint='wss://neptunedbcluster-xyz.cluster-ro-abc.us-east-
↪1.neptune.amazonaws.com:8182/gremlin')
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Example: Login and plot via client**

```
import graphistry
(graphistry
    .neptune(gremlin_client=client)
    .gremlin('g.E().sample(10)')
    .fetch_nodes()  # Fetch properties for nodes
    .plot())
```

**Parameters**

- **NEPTUNE_READER_HOST** (*str* / *None*)
- **NEPTUNE_READER_PORT** (*str* / *None*)
- **NEPTUNE_READER_PROTOCOL** (*str* / *None*)
- **endpoint** (*str* / *None*)
- **gremlin_client** (*Any* / *None*)

#### 3.8.7.2 Compute

PyGraphistry supports a variety of frameworks for graph tasks like analytics and layout

##### cuGraph

cuGraph is a GPU-accelerated graph library that leverages the Nvidia RAPIDS ecosystem. PyGraphistry provides a more fluent interface to enrich and transform your data with cuGraph methods without the boilerplate.

graphistry.plugins.cugraph.**compute_cugraph**(*self*, *alg*, *out_col=None*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*)

Run cugraph algorithm on graph. For algorithm parameters, see cuGraph docs.

**Parameters**

- **alg** (*str*) – algorithm name

- **out_col** (*Optional [str]*) – node table output column name, defaults to alg param
- **params** (*dict*) – algorithm parameters passed to cuGraph as kwargs
- **kind** (*CuGraphKind*) – kind of cugraph to use
- **directed** (*bool*) – whether graph is directed
- **G** (*Optional [cugraph.Graph]*) – cugraph graph to use; if None, use self
- **self** (Plottable)

**Returns**
Plottable

**Return type**
*Plottable*

**Example: Pass params to cugraph**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('betweenness_centrality', params={'k': 2})
assert 'betweenness_centrality' in g2._nodes.columns
```

**Example: Pagerank**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

**Example: Personalized Pagerank**

::
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']}) g = graphistry.edges(edges, 's', 'd') g2 = g.compute_cugraph('pagerank', params={'personalization': cudf.DataFrame({'vertex': ['a'], 'values': [1]})}) assert 'pagerank' in g2._nodes.columns

**Example: Katz centrality with rename**

```
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.compute_cugraph('katz_centrality', out_col='katz_centrality_renamed')
assert 'katz_centrality_renamed' in g2._nodes.columns
```

graphistry.plugins.cugraph.**df_to_gdf**(*df*)

**Parameters**
**df** (*Any*)

graphistry.plugins.cugraph.**from_cugraph**(*self*, *G*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

If bound IDs, use the same IDs in the returned graph.

If non-empty nodes/edges, instead of returning G's topology, use existing topology and merge in G's attributes

**Parameters**

- **node_attributes** (*List [ str ] | None*)
- **edge_attributes** (*List [ str ] | None*)
- **load_nodes** (*bool*)
- **load_edges** (*bool*)
- **merge_if_existing** (*bool*)

**Return type**
Plottable

graphistry.plugins.cugraph.**layout_cugraph**(*self*, *layout='force_atlas2'*, *params={}*, *kind='Graph'*, *directed=True*, *G=None*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*)

Layout the grpah using a cuGraph algorithm. For a list of layouts, see cugraph documentation (currently just force_atlas2).

**Parameters**

- **layout** (*str*) – Name of an cugraph layout method like *force_atlas2*
- **params** (*dict*) – Any named parameters to pass to the underlying cugraph method
- **kind** (*CuGraphKind*) – The kind of cugraph Graph
- **directed** (*bool*) – During the to_cugraph conversion, whether to be directed. (default True)
- **G** (*Optional [ Any ]*) – The cugraph graph (G) to layout. If None, the current graph is used.
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)
- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional [ str ]*) – If defined, set settings(url_params={'play': play}). (default 0)
- **self** (Plottable)

**Returns**
Plotter

**Return type**
Plotter

**Example: ForceAtlas2 layout**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_cugraph().plot()
```

**Example: Change which column names are generated**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('force_atlas2', x_out_col='my_x', y_out_col='my_y')
```

(continues on next page)

(continued from previous page)

```
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

**Example: Pass parameters to layout methods**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_cugraph('forceatlas_2', params={'lin_log_mode': True, 'prevent_
↪overlapping': True})
g2.plot()
```

graphistry.plugins.cugraph.**to_cugraph**(*self*, *directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *kind='Graph'*)

Convert current graph to a cugraph.Graph object

To assign an edge weight, use *g.bind(edge_weight='some_col').to_cugraph()*

Load from pandas, cudf, or dask_cudf DataFrames

**Parameters**

- **self** (Plottable)
- **directed** (*bool*)
- **include_nodes** (*bool*)
- **node_attributes** (*List[str] | None*)
- **edge_attributes** (*List[str] | None*)
- **kind** (*Literal['Graph', 'MultiGraph', 'BiPartiteGraph']*)

## Constants

graphistry.plugins.cugraph.**compute_algs: List[str] = ['betweenness_centrality', 'katz_centrality', 'ecg', 'leiden', 'louvain', 'spectralBalancedCutClustering', 'spectralModularityMaximizationClustering', 'connected_components', 'strongly_connected_components', 'core_number', 'hits', 'pagerank', 'bfs', 'bfs_edges', 'sssp', 'shortest_path', 'shortest_path_length', 'batched_ego_graphs', 'edge_betweenness_centrality', 'jaccard', 'jaccard_w', 'overlap', 'overlap_coefficient', 'overlap_w', 'sorensen', 'sorensen_coefficient', 'sorensen_w', 'ego_graph', 'k_core', 'minimum_spanning_tree']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins.cugraph.**layout_algs: List[str] = ['force_atlas2']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins_types.cugraph_types.**CuGraphKind**

alias of `Literal`['Graph', 'MultiGraph', 'BiPartiteGraph']

### graphviz

graphviz is a popular graph visualization library that PyGraphistry can interface with. This allows you to leverage graphviz's powerful layout algorithms, and optionally, static picture renderer. It is especially well-known for its "dot" layout algorithm for hierarchical and tree layouts of graphs with less than 10,000 nodes and edges.

graphistry.plugins.graphviz.**g_to_pgv**(*g*, *directed=True*, *strict=False*, *drop_unsanitary=False*)

**Parameters**

- **g** (Plottable)
- **directed** (*bool*)
- **strict** (*bool*)
- **drop_unsanitary** (*bool*)

**Return type**
None

graphistry.plugins.graphviz.**g_with_pgv_layout**(*g*, *graph*)

**Parameters**

- **g** (Plottable)
- **graph** (*None*)

**Return type**
Plottable

graphistry.plugins.graphviz.**layout_graphviz**(*self*, *prog='dot'*, *args=None*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *skip_styling=False*, *render_to_disk=False*, *path=None*, *format=None*, *drop_unsanitary=False*)

Use graphviz for layout, such as hierarchical trees and directed acycle graphs

Requires pygraphviz Python bindings and graphviz native libraries to be installed, see https://pygraphviz.github.io/documentation/stable/install.html

See PROGS for available layout algorithms

To render image to disk, set render=True

**Parameters**

- **self** (Plottable) – Base graph
- **prog** (graphistry.plugins_types.graphviz_types.Prog) – Layout algorithm - "dot", "neato", ...
- **args** (*Optional[str]*) – Additional arguments to pass to the graphviz commandline for layout
- **directed** (*bool*) – Whether the graph is directed (True, default) or undirected (False)
- **strict** (*bool*) – Whether the graph is strict (True) or not (False, default)
- **graph_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.GraphAttr, Any]]) – Graphviz graph attributes, see https://graphviz.org/docs/graph/

- **node_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.NodeAttr, Any]]) – Graphviz node attributes, see https://graphviz.org/docs/nodes/
- **edge_attr** (Optional[Dict[graphistry.plugins_types.graphviz_types.EdgeAttr, Any]]) – Graphviz edge attributes, see https://graphviz.org/docs/edges/
- **skip_styling** (*bool*) – Whether to skip applying default styling (False, default) or not (True)
- **render_to_disk** (*bool*) – Whether to render the graph to disk (False, default) or not (True)
- **path** (*Optional[str]*) – Path to save the rendered image when render_to_disk=True
- **format** (Optional[graphistry.plugins_types.graphviz_types.Format]) – Format of the rendered image when render_to_disk=True
- **drop_unsanitary** (*bool*) – Whether to drop unsanitary attributes (False, default) or not (True), recommended for sensitive settings

**Returns**

Graph with layout and style settings applied, setting x/y

**Return type**

*Plottable*

**Example: Dot layout for rigid hierarchical layout of trees and directed acyclic graphs**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz('dot').plot()
```

**Example: Neato layout for organic layout of small graphs**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz('neato').plot()
```

**Example: Set graphviz attributes at graph level**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz(
    prog='dot',
    graph_attr={
        'ratio': 10
    }
).plot()
```

**Example: Save rendered image to disk as a png**

```
import graphistry
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
```

(continues on next page)

(continued from previous page)

```
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz(
    'dot',
    render_to_disk=True,
    path='graph.png',
    format='png'
)
```

**Example: Save rendered image to disk as a png with passthrough of rendering styles**

```
import graphistry
edges = pd.DataFrame({
    's': ['a','b','c','d'],
    'd': ['b','c','d','e'],
    'color': ['red', None, None, 'yellow']
})
nodes = pd.DataFrame({
    'n': ['a','b','c','d','e'],
    'shape': ['circle', 'square', None, 'square', 'circle']
})
g = graphistry.edges(edges, 's', 'd')
g.layout_graphviz(
    'dot',
    render_to_disk=True,
    path='graph.png',
    format='png'
)
```

graphistry.plugins.graphviz.**layout_graphviz_core**(*g*, *prog='dot'*, *args=None*, *directed=True*, *strict=False*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *drop_unsanitary=False*)

**Parameters**

- **g** (Plottable)
- **prog** (*Literal['acyclic', 'ccomps', 'circo', 'dot', 'fdp', 'gc', 'gvcolor', 'gvpr', 'neato', 'nop', 'osage', 'patchwork', 'sccmap', 'sfdp', 'tred', 'twopi', 'unflatten']*)
- **args** (*str | None*)
- **directed** (*bool*)
- **strict** (*bool*)
- **graph_attr** (*Dict[Literal['_background', 'bb', 'beautify', 'bgcolor', 'center', 'charset', 'class', 'clusterrank', 'colorscheme', 'comment', 'compound', 'concentrate', 'Damping', 'defaultdist', 'dim', 'dimen', 'diredgeconstraints', 'dpi', 'epsilon', 'esep', 'fontcolor', 'fontname', 'fontnames', 'fontpath', 'fontsize', 'forcelabels', 'gradientangle', 'href', 'id', 'imagepath', 'inputscale', 'K', 'label', 'label_scheme', 'labeljust', 'labelloc', 'landscape', 'layerlistsep', 'layers', 'layerselect', 'layersep', 'layout', 'levels', 'levelsgap',*

*'lheight', 'linelength', 'lp', 'lwidth', 'margin', 'maxiter', 'mclimit', 'mindist', 'mode', 'model', 'newrank', 'nodesep', 'nojustify', 'normalize', 'notranslate', 'nslimit', 'nslimit1', 'oneblock', 'ordering', 'orientation', 'outputorder', 'overlap', 'overlap_scaling', 'overlap_shrink', 'pack', 'packmode', 'pad', 'page', 'pagedir', 'quadtree', 'quantum', 'rankdir', 'ranksep', 'ratio', 'remincross', 'repulsiveforce', 'resolution', 'root', 'rotate', 'rotation', 'scale', 'searchsize', 'sep', 'showboxes', 'size', 'smoothing', 'sortv', 'splines', 'start', 'style', 'stylesheet', 'target', 'TBbalance', 'tooltip', 'truecolor', 'URL', 'viewport', 'voro_margin', 'xdotversion'], ~typing.Any] | None*)

- **node_attr** (*Dict[Literal['area', 'class', 'color', 'colorscheme', 'comment', 'distortion', 'fillcolor', 'fixedsize', 'fontcolor', 'fontname', 'fontsize', 'gradientangle', 'group', 'height', 'href', 'id', 'image', 'imagepos', 'imagescale', 'label', 'labelloc', 'layer', 'margin', 'nojustify', 'ordering', 'orientation', 'penwidth', 'peripheries', 'pin', 'pos', 'rects', 'regular', 'root', 'samplepoints', 'shape', 'shapefile', 'showboxes', 'sides', 'skew', 'sortv', 'style', 'target', 'tooltip', 'URL', 'vertices', 'width', 'xlabel', 'xlp', 'z'], ~typing.Any] | None*)
- **edge_attr** (*Dict[Literal['arrowhead', 'arrowsize', 'arrowtail', 'class', 'color', 'colorscheme', 'comment', 'constraint', 'decorate', 'dir', 'edgehref', 'edgetarget', 'edgetooltip', 'edgeURL', 'fillcolor', 'fontcolor', 'fontname', 'fontsize', 'head_lp', 'headclip', 'headhref', 'headlabel', 'headport', 'headtarget', 'headtooltip', 'headURL', 'href', 'id', 'label', 'labelangle', 'labeldistance', 'labelfloat', 'labelfontcolor', 'labelfontname', 'labelfontsize', 'labelhref', 'labeltarget', 'labeltooltip', 'labelURL', 'layer', 'len', 'lhead', 'lp', 'ltail', 'minlen', 'nojustify', 'penwidth', 'pos', 'samehead', 'sametail', 'showboxes', 'style', 'tail_lp', 'tailclip', 'tailhref', 'taillabel', 'tailport', 'tailtarget', 'tailtooltip', 'tailURL', 'target', 'tooltip', 'URL', 'weight', 'xlabel', 'xlp'], ~typing.Any] | None*)
- **drop_unsanitary** (*bool*)

**Return type**
None

graphistry.plugins.graphviz.**pgv_styling**(*g*)

**Parameters**
**g** (Plottable)

**Return type**
Plottable

### Constants

graphistry.plugins_types.graphviz_types.**EdgeAttr**

alias of `Literal`['arrowhead', 'arrowsize', 'arrowtail', 'class', 'color', 'colorscheme', 'comment', 'constraint', 'decorate', 'dir', 'edgehref', 'edgetarget', 'edgetooltip', 'edgeURL', 'fillcolor', 'fontcolor', 'fontname', 'fontsize', 'head_lp', 'headclip', 'headhref', 'headlabel', 'headport', 'headtarget', 'headtooltip', 'headURL', 'href', 'id', 'label', 'labelangle', 'labeldistance', 'labelfloat', 'labelfontcolor', 'labelfontname', 'labelfontsize', 'labelhref', 'labeltarget', 'labeltooltip', 'labelURL', 'layer', 'len', 'lhead', 'lp', 'ltail', 'minlen', 'nojustify', 'penwidth', 'pos', 'samehead', 'sametail', 'showboxes', 'style', 'tail_lp', 'tailclip', 'tailhref', 'taillabel', 'tailport', 'tailtarget', 'tailtooltip', 'tailURL', 'target', 'tooltip', 'URL', 'weight', 'xlabel', 'xlp']

graphistry.plugins_types.graphviz_types.**Format**

alias of `Literal`['canon', 'cmap', 'cmapx', 'cmapx_np', 'dia', 'dot', 'fig', 'gd', 'gd2', 'gif', 'hpgl', 'imap', 'imap_np', 'ismap', 'jpe', 'jpeg', 'jpg', 'mif', 'mp', 'pcl', 'pdf', 'pic', 'plain', 'plain-ext', 'png', 'ps', 'ps2', 'svg', 'svgz', 'vml', 'vmlz', 'vrml', 'vtx', 'wbmp', 'xdot', 'xlib']

graphistry.plugins_types.graphviz_types.**GraphAttr**

alias of `Literal`['_background', 'bb', 'beautify', 'bgcolor', 'center', 'charset', 'class', 'clusterrank', 'colorscheme', 'comment', 'compound', 'concentrate', 'Damping', 'defaultdist', 'dim', 'dimen', 'diredgeconstraints', 'dpi', 'epsilon', 'esep', 'fontcolor', 'fontname', 'fontnames', 'fontpath', 'fontsize', 'forcelabels', 'gradientangle', 'href', 'id', 'imagepath', 'inputscale', 'K', 'label', 'label_scheme', 'labeljust', 'labelloc', 'landscape', 'layerlistsep', 'layers', 'layerselect', 'layersep', 'layout', 'levels', 'levelsgap', 'lheight', 'linelength', 'lp', 'lwidth', 'margin', 'maxiter', 'mclimit', 'mindist', 'mode', 'model', 'newrank', 'nodesep', 'nojustify', 'normalize', 'notranslate', 'nslimit', 'nslimit1', 'oneblock', 'ordering', 'orientation', 'outputorder', 'overlap', 'overlap_scaling', 'overlap_shrink', 'pack', 'packmode', 'pad', 'page', 'pagedir', 'quadtree', 'quantum', 'rankdir', 'ranksep', 'ratio', 'remincross', 'repulsiveforce', 'resolution', 'root', 'rotate', 'rotation', 'scale', 'searchsize', 'sep', 'showboxes', 'size', 'smoothing', 'sortv', 'splines', 'start', 'style', 'stylesheet', 'target', 'TBbalance', 'tooltip', 'truecolor', 'URL', 'viewport', 'voro_margin', 'xdotversion']

graphistry.plugins_types.graphviz_types.**NodeAttr**

alias of `Literal`['area', 'class', 'color', 'colorscheme', 'comment', 'distortion', 'fillcolor', 'fixedsize', 'fontcolor', 'fontname', 'fontsize', 'gradientangle', 'group', 'height', 'href', 'id', 'image', 'imagepos', 'imagescale', 'label', 'labelloc', 'layer', 'margin', 'nojustify', 'ordering', 'orientation', 'penwidth', 'peripheries', 'pin', 'pos', 'rects', 'regular', 'root', 'samplepoints', 'shape', 'shapefile', 'showboxes', 'sides', 'skew', 'sortv', 'style', 'target', 'tooltip', 'URL', 'vertices', 'width', 'xlabel', 'xlp', 'z']

graphistry.plugins_types.graphviz_types.**Prog**

alias of `Literal`['acyclic', 'ccomps', 'circo', 'dot', 'fdp', 'gc', 'gvcolor', 'gvpr', 'neato', 'nop', 'osage', 'patchwork', 'sccmap', 'sfdp', 'tred', 'twopi', 'unflatten']

graphistry.plugins_types.graphviz_types.**EDGE_ATTRS**
**typing.List[graphistry.plugins_types.graphviz_types.EdgeAttr]**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins_types.graphviz_types.**FORMATS**
**typing.List[graphistry.plugins_types.graphviz_types.Format]**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins_types.graphviz_types.GRAPH_ATTRS
typing.List[graphistry.plugins_types.graphviz_types.GraphAttr]

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins_types.graphviz_types.NODE_ATTRS
typing.List[graphistry.plugins_types.graphviz_types.NodeAttr]

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins_types.graphviz_types.PROGS
typing.List[graphistry.plugins_types.graphviz_types.Prog]

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

### igraph

igraph is a popular graph library that PyGraphistry can interface with. This allows you to leverage igraph's layout algorithms, and optionally, algorithmic enrichments. It is CPU-based and can generally handle small/medium-sized graphs.

graphistry.plugins.igraph.compute_igraph(*self*, *alg*, *out_col=None*, *directed=None*, *use_vids=False*, *params={}*, *stringify_rich_types=True*)

Enrich or replace graph using igraph methods

**Parameters**

- **alg** (*str*) – Name of an igraph.Graph method like *pagerank*
- **out_col** (*Optional[str]*) – For algorithms that generate a node attribute column, *out_col* is the desired output column name. When *None*, use the algorithm's name. (default None)
- **directed** (*Optional[bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – During the to_igraph conversion, whether to interpret IDs as igraph vertex IDs (non-negative integers) or arbitrary values (False, default)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method
- **stringify_rich_types** (*bool*) – When rich types like igraph.Graph are returned, which may be problematic for downstream rendering, coerce them to strings
- **self** (Plottable)

**Returns**
Plotter

**Return type**
Plotter

**Example: Pagerank**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank')
assert 'pagerank' in g2._nodes.columns
```

**Example: Pagerank with custom name**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', out_col='my_pr')
assert 'my_pr' in g2._nodes.columns
```

**Example: Pagerank on an undirected**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', directed=False)
assert 'pagerank' in g2._nodes.columns
```

**Example: Pagerank with custom parameters**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('pagerank', params={'damping': 0.85})
assert 'pagerank' in g2._nodes.columns
```

**Example: Personalized Pagerank**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['c','c','e','e']})
g = graphistry.edges(edges, 's', 'd').materialize_nodes()
g2 = g.compute_igraph('personalized_pagerank')
assert 'personalized_pagerank' in g2._nodes.columns
```

graphistry.plugins.igraph.**from_igraph**(*self*, *ig*, *node_attributes=None*, *edge_attributes=None*, *load_nodes=True*, *load_edges=True*, *merge_if_existing=True*)

Convert igraph object into Plotter

If base g has _node, _source, _destination definitions, use them

When merge_if_existing with preexisting nodes/edges df and shapes match ig, combine attributes

For merge_if_existing to work with edges, must set g._edge and have corresponding edge index attribute in igraph.Graph

**Parameters**

- **ig** (*igraph*) – Source igraph object
- **node_attributes** (*Optional [List [str]]*) – Subset of node attributes to load; None means all (default)

- **edge_attributes** (*Optional [List [str] ]*) – Subset of edge attributes to load; None means all (default)
- **load_nodes** (*bool*) – Whether to load nodes dataframe (default True)
- **load_edges** (*bool*) – Whether to load edges dataframe (default True)
- **merge_if_existing** (*bool*) – Whether to merge with existing node/edge dataframes (default True)
- **merge_if_existing** – bool

**Returns**
Plotter

**Return type**
Plottable

**Example: Convert from igraph, including all node/edge properties**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'], 'v':
↪ [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degrees()
assert 'degree' in g._nodes.columns
g2 = g.from_igraph(g.to_igraph())
assert len(g2._nodes.columns) == len(g._nodes.columns)
```

**Example: Enrich from igraph, but only load in 1 node attribute**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a', 'b', 'c', 'd'], 'd': ['b', 'c', 'd', 'e'], 'v':
↪ [101, 102, 103, 104]})
g = graphistry.edges(edges, 's', 'd').materialize_nodes().get_degree()
assert 'degree' in g._nodes
ig = g.to_igraph(include_nodes=False)
assert 'degree' not in ig.vs
ig.vs['pagerank'] = ig.pagerank()
g2 = g.from_igraph(ig, load_edges=False, node_attributes=[g._node, 'pagerank'])
assert 'pagerank' in g2._nodes
asssert 'degree' in g2._nodes
```

graphistry.plugins.igraph.**layout_igraph**(*self*, *layout*, *directed=None*, *use_vids=False*, *bind_position=True*, *x_out_col='x'*, *y_out_col='y'*, *play=0*, *params={}*)

Compute graph layout using igraph algorithm. For a list of layouts, see layout_algs or igraph documentation.

**Parameters**

- **layout** (*str*) – Name of an igraph.Graph.layout method like *sugiyama*
- **directed** (*Optional [bool]*) – During the to_igraph conversion, whether to be directed. If None, try directed and then undirected. (default None)
- **use_vids** (*bool*) – Whether to use igraph vertex ids (non-negative integers) or arbitary node ids (False, default)
- **bind_position** (*bool*) – Whether to call bind(point_x=, point_y=) (default True)

- **x_out_col** (*str*) – Attribute to write x position to. (default 'x')
- **y_out_col** (*str*) – Attribute to write x position to. (default 'y')
- **play** (*Optional [str]*) – If defined, set settings(url_params={'play': play}). (default 0)
- **params** (*dict*) – Any named parameters to pass to the underlying igraph method
- **self** (Plottable)

**Returns**
Plotter

**Return type**
Plotter

**Example: Sugiyama layout**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama')
assert 'x' in g2._nodes
g2.plot()
```

**Example: Change which column names are generated**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.layout_igraph('sugiyama', x_out_col='my_x', y_out_col='my_y')
assert 'my_x' in g2._nodes
assert g2._point_x == 'my_x'
g2.plot()
```

**Example: Pass parameters to layout methods - Sort nodes by degree**

```
import graphistry, pandas as pd
edges = pd.DataFrame({'s': ['a','b','c','d'], 'd': ['b','c','d','e']})
g = graphistry.edges(edges, 's', 'd')
g2 = g.get_degrees()
assert 'degree' in g._nodes.columns
g3 = g.layout_igraph('sugiyama', params={'layers': 'degree'})
g3.plot()
```

graphistry.plugins.igraph.**to_igraph**(*self*, *directed=True*, *include_nodes=True*, *node_attributes=None*, *edge_attributes=None*, *use_vids=False*)

Convert current item to igraph Graph . See examples in from_igraph.

**Parameters**

- **directed** (*bool*) – Whether to create a directed graph (default True)
- **include_nodes** (*bool*) – Whether to ingest the nodes table, if it exists (default True)
- **node_attributes** (*Optional [List [str]]*) – Which node attributes to load, None means all (default None)

- **edge_attributes** (*Optional [List [str]]*) – Which edge attributes to load, None means all (default None)
- **use_vids** (*bool*) – Whether to interpret IDs as igraph vertex IDs, which must be non-negative integers (default False)
- **self** (Plottable)

### Constants

graphistry.plugins.igraph.compute_algs: List[str] = ['articulation_points', 'authority_score', 'betweenness', 'bibcoupling', 'harmonic_centrality', 'closeness', 'clusters', 'cocitation', 'community_edge_betweenness', 'community_fastgreedy', 'community_infomap', 'community_label_propagation', 'community_leading_eigenvector', 'community_leiden', 'community_multilevel', 'community_optimal_modularity', 'community_spinglass', 'community_walktrap', 'constraint', 'coreness', 'gomory_hu_tree', 'harmonic_centrality', 'hub_score', 'eccentricity', 'eigenvector_centrality', 'k_core', 'pagerank', 'personalized_pagerank', 'spanning_tree']

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

graphistry.plugins.igraph.layout_algs: List[str] = ['auto', 'automatic', 'bipartite', 'circle', 'circular', 'dh', 'davidson_harel', 'drl', 'drl_3d', 'fr', 'fruchterman_reingold', 'fr_3d', 'fr3d', 'fruchterman_reingold_3d', 'grid', 'grid_3d', 'graphopt', 'kk', 'kamada_kawai', 'kk_3d', 'kk3d', 'kamada_kawai_3d', 'lgl', 'large', 'large_graph', 'mds', 'random', 'random_3d', 'rt', 'tree', 'reingold_tilford', 'rt_circular', 'reingold_tilford_circular', 'sphere', 'spherical', 'circle_3d', 'circular_3d', 'star', 'sugiyama']

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

### NetworkX Methods

The following methods are provided for converting and managing NetworkX graph data within PyGraphistry.

graphistry.PlotterBase.PlotterBase.**from_networkx**(*self*, *G*)

Convert a NetworkX graph to a PyGraphistry graph.

This method takes a NetworkX graph and converts it into a format that PyGraphistry can use for visualization. It extracts the node and edge data from the NetworkX graph and binds them to the graph object for further manipulation or visualization using PyGraphistry's API.

**Parameters**
G (*networkx.Graph or networkx.DiGraph*) – The NetworkX graph to convert.

**Returns**
A PyGraphistry Plottable object with the node and edge data from the NetworkX graph.

**Return type**
*Plottable*

**Example: Basic NetworkX Conversion**

```
import graphistry
import networkx as nx

# Create a NetworkX graph
G = nx.Graph()
G.add_nodes_from([
    (1, {"v": "one"}),
    (2, {"v": "two"}),
    (3, {"v": "three"}),
    (4, {"v": "four"}),
    (7, {"v": "seven"}),
    (8, {"v": "eight"})
])
G.add_edges_from([
    [2, 3],
    [3, 4],
    [7, 8]
])

# Convert the NetworkX graph to PyGraphistry format
g = from_networkx(G)

g.plot()
```

This example creates a simple NetworkX graph with nodes and edges, converts it using *from_networkx()*, and then plots it with the PyGraphistry API.

**Example: Using Custom Node and Edge Bindings**

```
import graphistry
import networkx as nx

# Create a NetworkX graph with attributes
G = nx.Graph()
G.add_nodes_from([
    (1, {"v": "one"}),
    (2, {"v": "two"}),
    (3, {"v": "three"}),
    (4, {"v": "four"}),
    (7, {"v": "seven"}),
    (8, {"v": "eight"})
])
G.add_edges_from([
    [2, 3],
    [3, 4],
    [7, 8]
])

# Bind custom node and edge names when converting from NetworkX to PyGraphistry
g = graphistry.bind(source='src', destination='dst').from_networkx(G)

g.plot()
```

graphistry.PlotterBase.PlotterBase.**networkx2pandas**(*self*, *g*)

graphistry.PlotterBase.PlotterBase.**networkx_checkoverlap**(*self*, *g*)

Raise an error if the node attribute already exists in the graph

## 3.9 Join the Community

The Graphistry team is active in a few places, so come join us:

- Blog
- Slack Channel
- GitHub
- Twitter
- LinkedIn

## 3.10 Support

Stuck or thinking about a new project? Let's chat!

- Get Started
- Blog
- Slack Channel
- Zendesk Support
- GitHub
- Twitter
- LinkedIn

## 3.11 Graphistry Ecosystem and Louie.AI

The Graphistry community of projects, open source, and partners has grown over the years:

### 3.11.1 Graphistry Core

- REST API
- *JS APIs (github) <https://github.com/graphistry/graphistry-js>_*: Node, React, and vanilla JS
- graph-app-kit (github): Python dashboarding with Graphistry and Streamlit

### 3.11.2 GFQL: Dataframe-native Graph Query Language

Our open-source graph query language GFQL with optional GPU support

The Graphistry team created GFQL to fill the gap between pandas/cudf and cypher. This project has been years in the making, and is built out of need from our experiences in working with graphs of all sizes in the compute and visualization tiers.

### 3.11.3 Graphistry Louie.AI

Louie.AI is the new genAI-native experience for Graphistry and your favorite databases

Louie.AI features:

- genAI-native notebooks: Talk to your data & databases and get back answers, visualizations, and more
- genAI-native dashboards: Build and share dashboards with your data, AI, and Graphistry
- API: Use Louie.AI's API to integrate genAI-native experiences into your own apps, both visual and headless
- Real-time AI Knowledge Graph Database: Target data, transform into preintegrated genAI-friendly indexes, then talk to it or trigger workflows

Check out the Louie.AI homepage for more information and early access.

### 3.11.4 Graphistry cu_cat

Automatic feature engineering is an important way pygraphistry[ai] streamlines ML and AI workflows. To make that fast, we have been adding GPU acceleration through a GPU-first port of dirty_cat (skrub).

Head over to the cu-cat homepage (github)

### 3.11.5 Community

Graphistry works with a variety of partners and projects, some of which include:

**GPU dataframes:**

- Nvidia RAPIDS: cuDF, cuGraph, cuML
- Apache Arrow: Python, JS, and more

**Graph**

- Neo4j
- Trovares
- Tigergraph
- ArangoDB
- JanusGraph
- Memgraph

**Log databases and SIEMs:**

- Elasticsearch
- Microsoft Kusto

- Microsoft Msticpy
- OpenSearch
- Splunk

**Graph analytics:**

- igraph

**Python data science ecosystem:**

- Streamlit
- Jupyter
- Pandas
- Dask
- genindex
- modindex
- search